

Peter Y. K. Cheung
George A. Constantinides
Jose T. de Sousa (Eds.)

LNCS 2778

Field-Programmable Logic and Applications

13th International Conference, FPL 2003
Lisbon, Portugal, September 2003
Proceedings



Springer

Lecture Notes in Computer Science

2778

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Peter Y. K. Cheung George A. Constantinides
Jose T. de Sousa (Eds.)

Field-Programmable Logic and Applications

13th International Conference, FPL 2003
Lisbon, Portugal, September 1-3, 2003
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Peter Y. K. Cheung
George A. Constantinides
Imperial College of Science, Technology, and Medicine
Dept. of Electrical and Electronic Engineering
Exhibition Road, London SW7 2 BT, UK
E-mail: p.cheung@ic.ac.uk; george.constantinides@ieee.org

Jose T. de Sousa
Technical University of Lisbon
INESC-ID/IST
R. Alves Redol, 9, Apartido 13069, 1000-029, Lisboa, Portugal
E-mail: jts@inesc-id.pt

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at <<http://dnb.ddb.de>>.

CR Subject Classification (1998): B.6-7, C.2, J.6

ISSN 0302-9743
ISBN 3-540-40822-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Steingräber Satztechnik GmbH
Printed on acid-free paper SPIN 10931431 06/3142 5 4 3 2 1 0

Preface

This book contains the papers presented at the 13th International Workshop on Field Programmable Logic and Applications (FPL) held on September 1–3, 2003. The conference was hosted by the Institute for Systems and Computer Engineering-Research and Development of Lisbon (INESC-ID) and the Department of Electrical and Computer Engineering of the IST-Technical University of Lisbon, Portugal.

The FPL series of conferences was founded in 1991 at Oxford University (UK), and has been held annually since: in Oxford (3 times), Vienna, Prague, Darmstadt, London, Tallinn, Glasgow, Villach, Belfast and Montpellier. It brings together academic researchers, industrial experts, users and newcomers in an informal, welcoming atmosphere that encourages productive exchange of ideas and knowledge between delegates.

Exciting advances in field programmable logic show no sign of slowing down. New grounds have been broken in architectures, design techniques, run-time re-configuration, and applications of field programmable devices in several different areas. Many of these innovations are reported in this volume.

The size of FPL conferences has grown significantly over the years. FPL in 2002 saw 214 papers submitted, representing an increase of 83% when compared to the year before. The interest and support for FPL in the programmable logic community continued this year with 216 papers submitted. The technical program was assembled from 90 selected regular papers and 56 posters, resulting in this volume of proceedings. The program also included three invited plenary keynote presentations from LSI Logic, Xilinx and Cadence, and three industrial tutorials from Altera, Mentor Graphics and Dafca.

Due to the inclusive tradition of the conference, FPL continues to attract submissions from all over the world. The accepted contributions were submitted by researchers from 32 different countries:

USA	42	Belgium	6	Brazil	2	Estonia	1
Spain	33	Netherlands	6	Canada	2	Norway	1
UK	29	Mexico	5	Hungary	2	India	1
Germany	14	Greece	4	Iran	2	Slovakia	1
Japan	13	Poland	4	Korea	2	Slovenia	1
Portugal	12	Switzerland	4	Romania	2		
Italy	9	Australia	3	Singapore	2		
Czech Rep.	8	Ireland	3	Austria	1		
France	7	China	2	Egypt	1		

We would like to thank all the authors for submitting their first versions of the papers and the final versions of the accepted papers. We also gratefully acknowledge the reviewing work done by the Program Committee members and many additional reviewers who contributed their time and expertise towards the compilation of this volume. The members of our Program Committee and all other reviewers are listed on the following pages. We are particularly pleased that of the 1029 reviews sought, 95% were completed.

We would like to thank QuickSilver Technology for their sponsorship of the Michal Servit Award, Celoxica for sponsoring the official FPL website www.fpl.org, Xilinx and Synplicity for their early support of the conference, and Coreworks for help in registration processing. We are indebted to Richard van de Stadt, the author of CyberChair. This excellent free software made our task of managing the submission and reviewing process much easier. We are grateful for the help and advice received from Wayne Luk and Horácio Neto. In addition, we acknowledge the help of the following research students from Imperial College London in checking the integrity of the manuscripts: Christos Bouganis, Wim Melis, Gareth Morris, Andy Royal, Pete Sedcole, Nalin Sidhao, and Theerayod Wiangtong.

We are grateful to Springer-Verlag, particularly Alfred Hofmann and Anna Kramer, for their work in publishing this book.

June 2003

Peter Y.K. Cheung
George A. Constantinides
Jose T. de Sousa

Organization

Organizing Committee

Program Chair	Peter Y.K. Cheung, Imperial College London, UK
Program Co-chair	George A. Constantinides, Imperial College London, UK
General Chair	Jose T. de Sousa, INESC-ID/IST, Technical University of Lisbon, Portugal
Publicity Chair	Reiner Hartenstein, University of Kaiserslautern, Germany
Local Chair	Horácio Neto, INESC-ID/IST, Technical University of Lisbon, Portugal
Finance Chair	Fernando Gonçalves, INESC-ID/IST, Technical University of Lisbon, Portugal
Exhibition Chair	João Cardoso, INESC-ID/UA, University of Algarve, Portugal

Program Committee

Nazeeh Aranki	Jet Propulsion Laboratory, USA
Jeff Arnold	Stretch, Inc., USA
Peter Athanas	Virginia Tech, USA
Neil Bergmann	Queensland University of Technology, Australia
Dinesh Bhatia	University of Texas, USA
Eduardo Boemo	University of Madrid, Spain
Gordon Brebner	Xilinx, Inc., USA
Andrew Brown	University of Southampton, UK
Klaus Buchenrieder	Infineon Technologies AG, Germany
Charles Chiang	Synopsys, Inc., USA
Peter Cheung	Imperial College London, UK
George Constantinides	Imperial College London, UK
Andre DeHon	California Institute of Technology, USA
Jose T. de Sousa	Technical University of Lisbon, Portugal
Carl Ebeling	University of Washington, USA
Hossam ElGindy	University of New South Wales, Australia
Manfred Glesner	Darmstadt University of Technology, Germany
Fernando Goncalves	Technical University of Lisbon, Portugal
Steven Guccione	Quicksilver Technology, USA
Reiner Hartenstein	University of Kaiserslautern, Germany
Scott Hauck	University of Washington, USA
Brad Hutchings	Brigham Young University, USA
Tom Kean	Algotronix Consulting, UK
Andreas Koch	University of Braunschweig, Germany
Dominique Lavenier	University of Montpellier II, France
Philip Leong	Chinese University of Hong Kong, China
Wayne Luk	Imperial College London, UK
Patrick Lysaght	Xilinx, Inc., USA
Bill Mangione-Smith	University of California at Los Angeles, USA
Reinhard Männer	University of Mannheim, Germany
Oskar Mencer	Bell Labs, USA
George Milne	University of Western Australia
Toshiyaki Miyazaki	NTT Network Innovation Labs, Japan
Fernando Moraes	PUCRS, Brazil
Horacio Neto	Technical University of Lisbon, Portugal
Sebastien Pillement	ENSSAT, France
Dhiraj Pradhan	University of Bristol, UK
Viktor Prasanna	University of Southern California, USA
Michel Renovell	University of Montpellier II, France
Jonathan Rose	University of Toronto, Canada
Zoran Salcic	University of Auckland, New Zealand
Hartmut Schmeck	University of Karlsruhe, Germany
Rainer Spallek	Dresden University of Technology, Germany

Adrian Stoica	Jet Propulsion Laboratory, USA
Jürgen Teich	University of Paderborn, Germany
Lothar Thiele	ETH Zürich, Switzerland
Liones Torres	University of Montpellier II, France
Stephen Trimberger	Xilinx, Inc., USA
Milan Vasilko	Bournemouth University, UK
Ranga Vemuri	University of Cincinnati, USA
Roger Woods	Queen's University Belfast, UK

Steering Committee

Jose T. de Sousa	Technical University of Lisbon, Portugal
Manfred Glesner	Darmstadt University of Technology, Germany
John Gray	Independent Consultant, UK
Herbert Grünbacher	Carinthia Technical Institute, Austria
Reiner Hartenstein	University of Kaiserslautern, Germany
Andres Keevallik	Tallinn Technical University, Estonia
Wayne Luk	Imperial College London, UK
Patrick Lysaght	Xilinx, Inc., USA
Michel Renovell	University of Montpellier II, France
Roger Woods	Queen's University Belfast, UK

Additional Reviewers

Anuradha Agarwal	Francisco Barat
Ali Ahmadinia	Jorge Barreiros
Seong-Yong Ahn	Marcus Bednara
Rui Aguiar	Peter Bellows
Miguel A. Aguirre	Mohammed Benaissa
Bashir Al-Hashimi	AbdSamad BenKrid
Ferhat Alim	Khaled BenKrid
Jose Alves	Pascal Benoit
Hideharu Amano	Manuel Berenguel
Jose Nelson Amaral	Daniel Berg
David Antos	Paul Berube
António José Araújo	Jean-Luc Beuchat
Miguel Arias-Estrada	Rajarshee Bharadwaj
Rubén Arteaga	Unai Bidarte
Armando Astarloa	Bob Blake
José Augusto	Brandon Blodget
Shailendra Aulakh	Jose A. Boluda
Vicente Baena	Vanderlei Bonato
Zachary Baker	Andrea Boni
Jonathan Ballagh	Marcos R. Boschetti
Sergio Bampi	Ignacio Bravo

Ney Calazans	Toshihito Fujiwara
Danna Cao	Rafael Gadea-Girones
Francisco Cardells-Tormo	Altaf Abdul Gaffar
Joao Cardoso	Federico Garcia
Dylan Carline	Alberto Garcia-Ortiz
Luigi Carro	Ester M. Garzon
Nicholas Carter	Manjunath Gangadhar
Gregorio Cappuccino	Antonio Gentile
Joaquín Cerdà	Raul Mateos Gil
Abhijeet Chakraborty	Rafael Gadea-Girones
François Charot	Federico Garcia
Seonil Choi	Jörn Gause
Bobda Christophe	Fahmi Ghozzi
Alessandro Cilardo	Guy Gogniat
Christopher Clark	Richard Aderbal Gonçalves
John Cochran	Gokul Govindu
James Cohoon	Gail Gray
Stuart Colsell	Jong-Ru Guo
Katherine Compton	Manish Handa
Pasquale Corsonello	Frank Hannig
Tom Van Court	Jim Harkin
Octavian Cret	Martin Herbordt
Damian Dalton	Antonin Hermanek
Alan Daly	Fabiano Hessel
Martin Danek	Teruo Higashino
Klaus Danne	Roland Höller
Eric Debes	Renqiu Huang
Martin Delvai	Ashraf Hussein
Daniel Denning	Shuichi Ichikawa
Arturo Diaz-Perez	José Luis Imaña
Pedro Diniz	Minoru Inamori
Peiliang Dong	Preston Jackson
Cillian O'Driscoll	Kamakoti
Mark E. Dunham	Parivallal Kannan
Alireza Ejlali	Irwin Kennedy
Tarek El-Ghazawi	Tim Kerins
Peeter Ellerjee	Jawad Khan
Wilfried Elmenreich	Sami Khawam
Rolf Enzler	Daniel Kirschner
Ken Erickson	Tomoyoshi Kobori
Roberto Esper-Chaín Falcón	Fatih Kocan
Béla Fehér	Dirk Koch
Michael Ferguson	Zbigniew Kokosinski
Marcio Merino Fernandes	Andrzej Krasniewski
Viktor Fischer	Rohini Krishnan

Georgi Kuzmanov	Allen Michalski
Soonhak Kwon	Maria Jose Moura
David Lacasa	John Nestor
John Lach	Jiri Novotny
Jesus Lazaro	John Oliver
Barry Lee	Eva M. Ortigosa
Dong-U Lee	Fernando Ortiz
Gareth Lee	Damjan Oseli
Jirong Liao	Jingzhao Ou
Valentino Liberali	Marcio Oyamada
Bossuet Lilian	Chris Papachristou
Fernanda Lima	Fernando Pardo
John Lockwood	Stavros Paschalakis
Andrea Lodi	Kolin Paul
Robert Lorenz	Cong Vinh Phan
Michael G. Lorenz	Juan Manuel Sanchez Perez
David Rodriguez Lozano	Stefania Perri
Shih-Lien Lu	Mihail Petrov
Martin Ma	Thilo Pionteck
Usama Malik	Marco Platzner
Cesar Augusto Marcon	Jüri Pöldre
Theodore Marescaux	Dionisios N. Pnevmatikatos
Eduardo Marques	Kara Poon
L.J. McDaid	Juan Antonio Gomez Pulido
Paul McHardy	Federico Quaglio
Maire McLoone	Senthil Rajamani
Bingfeng Mei	Javier Ramirez
Mahmoud Meribout	Juergen Reichardt
Uwe Meyer-Baese	Javier Resano
Yosuke Miyajima	Fernando Rincón
Sumit Mohanty	Francisco Rodríguez-Henríquez
Gareth Morris	Nuno Roma
Elena Moscu	Eduardo Ros
Francisco Moya-Fernandez	Gaël Rouvroy
Madhubanti Mukherjee	Andrew Royal
Tudor Murgan	Giacinto Paolo Saggese
Ciaron Murphy	Marcelino Santos
Takahiro Murooka	Gilles Sassatelli
Kouichi Nagami	Toshinori Sato
Ulrich Nageldinger	Sergei Sawitzki
Jeff Namkung	Pascal Scalart
Ángel Grediaga Olivo	Bernd Scheuermann
Pilar Martinez Ortigosa	Jan Schier
Selene Maya-Rueda	Clemens Schlachta
Wim Melis	Klaus Schleisiek

XII Organization

Herman Schmit	Yann Thoma
David Schuehler	Tim Todman
Ronald Scrofano	Jon Tombs
Pete Sedcole	Cesar Torres-Huitzil
Peter-Michael Seidel	Kuen Tsoi
Shay Seng	Marek Tudruj
Sakir Sezer	Richard Turner
Naoki Shibata	Fabrizio Vacca
Tsunemichi Shiozawa	Sudhir Vaka
Nalin Sidahao	Eduardo do Valle Simoes
Reetinder Sidhu	József Vásárhelyi
Valery Sklyarov	Joerg Velten
Ioulia Skliarova	Felip Vicedo
Gerard Smit	Tanya Vladimirova
Raphael Some	Markus Weinhardt
Ioannis Sourdis	Theerayod Wiangtong
Lionel Sousa	Juan Manuel Xicotencatl
Ludovico de Souza	Andy Yan
François-Xavier Standaert	Keiichi Yasumoto
Henry Styles	Pavel Zemcik
Qing Su	Xiaoyang Zeng
Vijay Sundaresan	Yumin Zhang
Noriyuki Takahashi	Jihan Zhu
Shigeyuki Takano	Ling Zhuo
Kalle Tammemäe	Peter Zipf
Konstantinos Tatas	Claudiu Zissulescu-Ianculescu
Raoul Tawel	Mark Zwolinski
John Teifel	

Table of Contents

Technologies and Trends

- Reconfigurable Circuits Using Hybrid Hall Effect Devices 1
S. Ferrera, N.P. Carter

- Gigahertz FPGA by SiGe BiCMOS Technology
for Low Power, High Speed Computing with 3-D Memory 11
*C. You, J.-R. Guo, R.P. Kraft, M. Chu, R. Heikaus, O. Erdogan,
P. Curran, B. Goda, K. Zhou, J.F. McDonald*

Communications Applications

- Implementing an OFDM Receiver
on the RaPiD Reconfigurable Architecture 21
C. Ebeling, C. Fisher, G. Xing, M. Shen, H. Liu

- Symbol Timing Synchronization in FPGA-Based Software Radios:
Application to DVB-S 31
F. Cardells-Tormo, J. Valls-Coquillat, V. Almenar-Terre

High Level Design Tools 1

- An Algorithm Designer's Workbench for Platform FPGAs..... 41
S. Mohanty, V.K. Prasanna

- Prototyping for the Concurrent Development
of an IEEE 802.11 Wireless LAN Chipset 51
L. de Souza, P. Ryan, J. Crawford, K. Wong, G. Zyner, T. McDermott

Reconfigurable Architectures

- ADRES: An Architecture with Tightly Coupled VLIW Processor
and Coarse-Grained Reconfigurable Matrix 61
B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins

- Inter-processor Connection Reconfiguration
Based on Dynamic Look-Ahead Control of Multiple Crossbar Switches.... 71
E. Laskowski, M. Tudruj

- Arbitrating Instructions in an $\rho\mu$ -Coded CCM 81
G. Kuzmanov, S. Vassiliadis

Cryptographic Applications 1

How Secure Are FPGAs in Cryptographic Applications?	91
<i>T. Wollinger, C. Paar</i>	
FPGA Implementations of the RC6 Block Cipher	101
<i>J.-L. Beuchat</i>	
Very High Speed 17 Gbps SHACAL Encryption Architecture	111
<i>M. McLoone, J.V. McCanny</i>	

Place and Route Tools

Track Placement: Orchestrating Routing Structures to Maximize Routability	121
<i>K. Compton, S. Hauck</i>	
Quark Routing	131
<i>S.T. McCulloch, J.P. Cohoon</i>	
Global Routing for Lookup-Table Based FPGAs Using Genetic Algorithms	141
<i>J. Barreiros, E. Costa</i>	

Multi-context FPGAs

Virtualizing Hardware with Multi-context Reconfigurable Arrays	151
<i>R. Enzler, C. Plessl, M. Platzner</i>	
A Dynamically Adaptive Switching Fabric on a Multicontext Reconfigurable Device	161
<i>H. Amano, A. Jouraku, K. Anjo</i>	
Reducing the Configuration Loading Time of a Coarse Grain Multicontext Reconfigurable Device	171
<i>T. Kitaoka, H. Amano, K. Anjo</i>	

Cryptographic Applications 2

Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES	181
<i>G. Rovroy, F.-X. Standaert, J.-J. Quisquater, J.-D. Legat</i>	
Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm	194
<i>I. Gonzalez, S. Lopez-Buedo, F.J. Gomez, J. Martinez</i>	

An Implementation Comparison of an IDEA Encryption Cryptosystem on Two General-Purpose Reconfigurable Computers	204
<i>A. Michalski, K. Gaj, T. El-Ghazawi</i>	

Low-Power Issues 1

Data Processing System with Self-reconfigurable Architecture, for Low Cost, Low Power Applications	220
<i>M.G. Lorenz, L. Mengibar, L. Entrena, R. Sánchez-Reillo</i>	
Low Power Coarse-Grained Reconfigurable Instruction Set Processor	230
<i>F. Barat, M. Jayapala, T.V. Aa, R. Lauwereins, G. Deconinck, H. Corporaal</i>	
Encoded-Low Swing Technique for Ultra Low Power Interconnect	240
<i>R. Krishnan, J. Pineda de Gyvez, H.J.M. Veendrick</i>	

Run-Time Configurations

Building Run-Time Reconfigurable Systems from Tiles	252
<i>G. Lee, G. Milne</i>	
Exploiting Redundancy to Speedup Reconfiguration of an FPGA	262
<i>I. Kennedy</i>	
Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration	272
<i>K. Danne, C. Bobda, H. Kalte</i>	

Cryptographic Applications 3

Efficient Modular-Pipelined AES Implementation in Counter Mode on ALTERA FPGA	282
<i>F. Charot, E. Yahya, C. Wagner</i>	
An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm	292
<i>G.P. Saggese, A. Mazzeo, N. Mazzocca, A.G.M. Strollo</i>	
Two Approaches for a Single-Chip FPGA Implementation of an Encryptor/Decryptor AES Core	303
<i>N.A. Saqib, F. Rodríguez-Henríquez, A. Díaz-Pérez</i>	

Compilation Tools

Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations	313
<i>K.R.S. Shayee, J. Park, P.C. Diniz</i>	

Branch Optimisation Techniques for Hardware Compilation	324
<i>H. Styles, W. Luk</i>	

A Model for Hardware Realization of Kernel Loops.....	334
<i>J. Liao, W.-F. Wong, T. Mitra</i>	

Asynchronous Techniques

Programmable Asynchronous Pipeline Arrays.....	345
<i>J. Teifel, R. Manohar</i>	

Globally Asynchronous Locally Synchronous FPGA Architectures	355
<i>A. Royal, P.Y.K. Cheung</i>	

Biology-Related Applications

Case Study of a Functional Genomics Application for an FPGA-Based Coprocessor	365
<i>T. Van Court, M.C. Herbordt, R.J. Barton</i>	

A Smith-Waterman Systolic Cell	375
<i>C.W. Yu, K.H. Kwong, K.H. Lee, P.H.W. Leong</i>	

Codesign

Software Decelerators.....	385
<i>E. Keller, G. Brebner, P. James-Roxby</i>	

A Unified Codesign Run-Time Environment for the UltraSONIC Reconfigurable Computer	396
<i>T. Wiangtong, P.Y.K. Cheung, W. Luk</i>	

Reconfigurable Fabrics

Extra-dimensional Island-Style FPGAs	406
<i>H. Schmit</i>	

Using Multiplexers for Control and Data in D-Fabrix	416
<i>T. Stansfield</i>	

Heterogeneous Logic Block Architectures for Via-Patterned Programmable Fabrics	426
<i>A. Koorapaty, L. Pileggi, H. Schmit</i>	

Image Processing Applications

A Real-Time Visualization System for PIV	437
<i>T. Fujiwara, K. Fujimoto, T. Maruyama</i>	
A Real-Time Stereo Vision System with FPGA	448
<i>Y. Miyajima, T. Maruyama</i>	
Synthesizing on a Reconfigurable Chip an Autonomous Robot Image Processing System	458
<i>J.A. Boluda, F. Pardo</i>	

SAT Techniques

Reconfigurable Hardware SAT Solvers: A Survey of Systems	468
<i>I. Skliarova, A.B. Ferrari</i>	
Fault Tolerance Analysis of Distributed Reconfigurable Systems Using SAT-Based Techniques	478
<i>R. Feldmann, C. Haubelt, B. Monien, J. Teich</i>	
Hardware Implementations of Real-Time Reconfigurable WSAT Variants	488
<i>R.H.C. Yap, S.Z.Q. Wang, M.J. Henz</i>	

Application-Specific Architectures

Core-Based Reusable Architecture for Slave Circuits with Extensive Data Exchange Requirements	497
<i>U. Bidarte, A. Astarloa, A. Zuloaga, J. Jimenez, I. Martínez de Alegría</i>	
Time and Energy Efficient Matrix Factorization Using FPGAs	507
<i>S. Choi, V.K. Prasanna</i>	
Improving DSP Performance with a Small Amount of Field Programmable Logic	520
<i>J. Oliver, V. Akella</i>	

DSP Applications

Fully Parameterized Discrete Wavelet Packet Transform Architecture Oriented to FPGA	533
<i>G. Payá, M.M. Peiró, F. Ballester, F. Mora</i>	

XVIII Table of Contents

An FPGA System for the High Speed Extraction, Normalization and Classification of Moment Descriptors	543
<i>S. Paschalakis, P. Lee, M. Bober</i>	

Design and Implementation of a Novel FIR Filter Architecture with Boundary Handling on Xilinx VIRTEX FPGAs	553
<i>A. Benkrid, K. Benkrid, D. Crookes</i>	

Dynamic Reconfiguration

A Self-reconfiguring Platform	565
<i>B. Blodget, P. James-Roxby, E. Keller, S. McMillan, P. Sundararajan</i>	

Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices	575
<i>C. Steiger, H. Walder, M. Platzner</i>	

Run-Time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems	585
<i>J. Resano, D. Mozos, D. Verkest, S. Vernalde, F. Catthoor</i>	

SoC Architectures

Networks on Chip as Hardware Components of an OS for Reconfigurable Systems	595
<i>T. Marescaux, J.-Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, R. Lauwereins</i>	

A Reconfigurable Platform for Real-Time Embedded Video Image Processing	606
<i>N.P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, W. Luk</i>	

Emulation

Emulation-Based Analysis of Soft Errors in Deep Sub-micron Circuits	616
<i>M.S. Reorda, M. Violante</i>	

HW-Driven Emulation with Automatic Interface Generation	627
<i>M. Çakir, E. Grimpe, W. Nebel</i>	

Cache Design

Implementation of HW\$im – A Real-Time Configurable Cache Simulator	638
<i>S.-L. Lu, K. Lai</i>	

The Bank Nth Chance Replacement Policy for FPGA-Based CAMs	648
<i>P. Berube, A. Zinyk, J.N. Amaral, M. MacGregor</i>	

Arithmetic 1

Variable Precision Multipliers for FPGA-Based Reconfigurable Computing Systems	661
<i>P. Corsonello, S. Perri, M.A. Iachino, G. Cocorullo</i>	
A New Arithmetic Unit in GF(2^m) for Reconfigurable Hardware Implementation	670
<i>C.H. Kim, S. Kwon, J.J. Kim, C.P. Hong</i>	

Biologically Inspired Designs

A Dynamic Routing Algorithm for a Bio-inspired Reconfigurable Circuit	681
<i>Y. Thoma, E. Sanchez, J.-M.M. Arostegui, G. Tempesti</i>	
An FPL Bioinspired Visual Encoding System to Stimulate Cortical Neurons in Real-Time	691
<i>L. Sousa, P. Tomás, F. Pelayo, A. Martinez, C.A. Morillas, S. Romero</i>	

Low-Power Issues 2

Power Analysis of FPGAs: How Practical is the Attack?	701
<i>F.-X. Standaert, L. van Oldeneel tot Oldenzeel, D. Samyde, J.-J. Quisquater</i>	
A Power-Scalable Motion Estimation Architecture for Energy Constrained Applications	712
<i>M. Martina, A. Molino, F. Quaglio, F. Vacca</i>	

SoC Designs

A Novel Approach for Architectural Models Characterization. An Example through the Systolic Ring	722
<i>P. Benoit, G. Sassatelli, L. Torres, M. Robert, G. Cambon, D. Demigny</i>	
A Generic Architecture for Integrated Smart Transducers	733
<i>M. Delvai, U. Eisenmann, W. Elmenreich</i>	
Customisable Core-Based Architectures for Real-Time Motion Estimation on FPGAs	745
<i>N. Roma, T. Dias, L. Sousa</i>	

Cellular Applications

A High Speed Computation System for 3D FCHC Lattice Gas Model with FPGA	755
<i>T. Kobori, T. Maruyama</i>	

Implementation of ReCSiP: A ReConfigurable Cell SImulation Platform	766
<i>Y. Osana, T. Fukushima, H. Amano</i>	

On the Implementation of a Margolus Neighborhood Cellular Automata on FPGA	776
<i>J. Cerdá, R. Gadea, V. Herrero, A. Sebastià</i>	

Arithmetic 2

Fast Modular Division for Application in ECC on Reconfigurable Logic ..	786
<i>A. Daly, W. Marnane, T. Kerins, E. Popovici</i>	

Non-uniform Segmentation for Hardware Function Evaluation	796
<i>D.-U. Lee, W. Luk, J. Villasenor, P.Y.K. Cheung</i>	

A Dual-Path Logarithmic Number System Addition/Subtraction Scheme for FPGA	808
<i>B. Lee, N. Burgess</i>	

Fault Analysis

A Modular Reconfigurable Architecture for Efficient Fault Simulation in Digital Circuits	818
<i>J.S. Augusto, C.B. Almeida, H.C.C. Neto</i>	

Evaluation of Testability of Path Delay Faults for User-Configured Programmable Devices	828
<i>A. Krasniewski</i>	

Fault Simulation Using Partially Reconfigurable Hardware	839
<i>A. Parreira, J.P. Teixeira, A. Pantelimon, M.B. Santos, J.T. de Sousa</i>	

Switch Level Fault Emulation	849
<i>S.G. Miremadi, A. Ejlali</i>	

Network Applications

An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall	859
<i>J.W. Lockwood, C. Neely, C. Zuver, J. Moscola, S. Dharmapurikar, D. Lim</i>	

IPsec-Protected Transport of HDTV over IP	869
<i>P. Bellows, J. Flidr, L. Gharai, C. Perkins, P. Chodowiec, K. Gaj</i>	
Fast, Large-Scale String Match	
for a 10Gbps FPGA-Based Network Intrusion Detection System	880
<i>I. Sourdis, D. Pnevmatikatos</i>	
Irregular Reconfigurable CAM Structures for Firewall Applications	890
<i>T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, N. Dulay</i>	

High Level Design Tools 2

Compiling for the Molen Programming Paradigm	900
<i>E.M. Panainte, K. Bertels, S. Vassiliadis</i>	
Laura: Leiden Architecture Research and Exploration Tool	911
<i>C. Zissulescu, T. Stefanov, B. Kienhuis, E. Deprettere</i>	
Communication Costs Driven Design Space Exploration	
for Reconfigurable Architectures	921
<i>L. Bossuet, G. Gogniat, J.-L. Philippe</i>	
From Algorithm Graph Specification to Automatic Synthesis	
of FPGA Circuit: A Seamless Flow of Graphs Transformations	934
<i>L. Kaouane, M. Akil, Y. Sorel, T. Grandpierre</i>	

Technologies and Trends (Posters)

Adaptive Real-Time Systems and the FPAAs	944
<i>S. Colsell, R. Edwards</i>	
Challenges and Successes in Space Based Reconfigurable Computing	948
<i>M.E. Dunham, M.P. Caffrey, P.S. Graham</i>	
Adaptive Processor: A Dynamically Reconfiguration Technology	
for Stream Processing	952
<i>S. Takano</i>	

Applications (Posters)

Efficient Reconfigurable Logic Circuits	
for Matching Complex Network Intrusion Detection Patterns	956
<i>C.R. Clark, D.E. Schimmel</i>	
FPGAs for High Accuracy Clock Synchronization	
over Ethernet Networks	960
<i>R. Höller</i>	

XXII Table of Contents

Project of IPv6 Router with FPGA Hardware Accelerator	964
<i>J. Novotný, O. Fučík, D. Antoš</i>	
A TCP/IP Based Multi-device Programming Circuit	968
<i>D.V. Schuehler, H. Ku, J. Lockwood</i>	

Tools (Posters)

Design Flow for Efficient FPGA Reconfiguration	972
<i>R.H. Turner, R.F. Woods</i>	
High-Level Design Tools for FPGA-Based Combinatorial Accelerators	976
<i>V. Sklyarov, I. Skliarova, P. Almeida, M. Almeida</i>	
Using System Generator to Design a Reconfigurable Video Encryption System	980
<i>D. Denning, N. Harold, M. Devlin, J. Irvine</i>	
MATLAB/Simulink Based Methodology for Rapid-FPGA-Prototyping....	984
<i>M. Ličko, J. Schier, M. Tichý, M. Kühl</i>	

FPGA Implementations (Posters)

DIGIMOD: A Tool to Implement FPGA-Based Digital IF and Baseband Modems	988
<i>J. Marín-Roig, V. Torres, M.J. Canet, A. Pérez, T. Sansaloni, F. Cardells, F. Angarita, F. Vicedo, V. Almenar, J. Valls</i>	
FPGA Implementation of a Maze Routing Accelerator	992
<i>J.A. Nestor</i>	
Model Checking Reconfigurable Processor Configurations for Safety Properties	996
<i>J. Cochran, D. Kapur, D. Stefanović</i>	
A Statistical Analysis Tool for FPLD Architectures	1000
<i>R. Huang, T. Cheung, T. Kok</i>	

Video and Image Applications (Posters)

FPGA-Implementation of Signal Processing Algorithms for Video Based Industrial Safety Applications	1004
<i>J. Velten, A. Kummert</i>	
Configurable Hardware Architecture for Real-Time Window-Based Image Processing	1008
<i>C. Torres-Huitzil, M. Arias-Estrada</i>	

- An FPGA-Based Image Connected Component Labeller 1012
K. Benkrid, S. Sukhsawas, D. Crookes, A. Benkrid

- FPGA Implementation of Adaptive Non-linear Predictors
for Video Compression 1016
*R. Gadea-Girones, A. Ramirez-Agundis, J. Cerdá-Boluda,
R. Colom-Palero*

Reconfigurable and Low-Power Systems (Posters)

- Reconfigurable Systems in Education 1020
V. Sklyarov, I. Skliarova

- Data Dependent Circuit Design: A Case Study 1024
S. Yamamoto, S. Ichikawa, H. Yamamoto

- Design of a Power Conscious, Customizable CDMA Receiver 1028
M. Martina, A. Molino, M. Nicola, F. Vacca

- Power-Efficient Implementations of Multimedia Applications
on Reconfigurable Platforms 1032
K. Tatas, K. Siozios, D. Soudris, A. Thanailakis

Design Techniques (Posters)

- A VHDL Library to Analyse Fault Tolerant Techniques 1036
P.M. Ortigosa, O. López, R. Estrada, I. García, E.M. Garzón

- Hardware Design with a Scripting Language 1040
P. Haglund, O. Mencer, W. Luk, B. Tai

- Testable Clock Routing Architecture
for Field Programmable Gate Arrays 1044
L.K. Kumar, A.J. Mupid, A.S. Ramani, V. Kamakoti

Neural and Biological Applications (Posters)

- FPGA Implementation of Multi-layer Perceptrons
for Speech Recognition 1048
E.M. Ortigosa, P.M. Ortigosa, A. Cañas, E. Ros, R. Agís, J. Ortega

- FPGA Based High Density Spiking Neural Network Array 1053
J.M. Xicotencatl, M. Arias-Estrada

- FPGA-Based Computation of Free-Form Deformations 1057
J. Jiang, W. Luk, D. Rueckert

XXIV Table of Contents

FPGA Implementations of Neural Networks – A Survey of a Decade of Progress	1062
<i>J. Zhu, P. Sutton</i>	

Codesign and Embedded Systems (Posters)

FPGA-Based Hardware/Software CoDesign of an Expert System Shell	1067
<i>A. Nețin, D. Roman, O. Creț, K. Pusztai, L. Văcariu</i>	
Cluster-Driven Hardware/Software Partitioning and Scheduling Approach for a Reconfigurable Computer System	1071
<i>T. Wiangtong, P.Y.K. Cheung, W. Luk</i>	
Hardware-Software Codesign in Embedded Asymmetric Cryptography Application – A Case Study	1075
<i>M. Šimka, V. Fischer, M. Drutarovský</i>	
On-chip and Off-chip Real-Time Debugging for Remotely-Accessed Embedded Programmable Systems	1079
<i>J. Harkin, M. Callaghan, C. Peters, T.M. McGinnity, L. Maguire</i>	

Reconfigurable Systems and Architectures (Posters)

Fast Region Labeling on the Reconfigurable Platform ACE-V	1083
<i>C. Schmidt, A. Koch</i>	
Modified Fuzzy C-Means Clustering Algorithm for Real-Time Applications	1087
<i>J. Lázaro, J. Arias, J.L. Martín, C. Cuadrado</i>	
Reconfigurable Hybrid Architecture for Web Applications	1091
<i>D. Rodríguez Lozano, J.M. Sánchez Pérez, J.A. Gómez Pulido</i>	

DSP Applications (Posters)

FPGA Implementation of the Adaptive Lattice Filter	1095
<i>A. Heřmánek, Z. Pohl, J. Kadlec</i>	
Specifying Control Logic for DSP Applications in FPGAs	1099
<i>J. Ballagh, J. Hwang, H. Ma, B. Milne, N. Shirazi, V. Singh, J. Stroomer</i>	
FPGA Processor for Real-Time Optical Flow Computation	1103
<i>S. Maya-Rueda, M. Arias-Estrada</i>	
A Data Acquisition Reconfigurable Coprocessor for Virtual Instrumentation Applications	1107
<i>M.D. Valdés, M.J. Moure, C. Quintáns, E. Mandado</i>	

Dynamic Reconfiguration (Posters)

Evaluation and Run-Time Optimization of On-chip Communication Structures in Reconfigurable Architectures	1111
<i>T. Murgan, M. Petrov, A. García Ortiz, R. Ludewig, P. Zipf, T. Hollstein, M. Glesner, B. Oelkrug, J. Brakensiek</i>	
A Controlled Data-Path Allocation Model for Dynamic Run-Time Reconfiguration of FPGA Devices	1115
<i>D. Carline, P. Coulton</i>	
Architecture Template and Design Flow to Support Application Parallelism on Reconfigurable Platforms	1119
<i>S. Sawitzki, R.G. Spallek</i>	
Efficient Implementation of the Singular Value Decomposition on a Reconfigurable System	1123
<i>C. Bobda, K. Danne, A. Linarth</i>	

Arithmetic (Posters)

A New Reconfigurable-Oriented Method for Canonical Basis Multiplication over a Class of Finite Fields GF(2^m)	1127
<i>J.L. Imaña, J.M. Sánchez</i>	
A Study on the Design of Floating-Point Functions in FPGAs	1131
<i>F.E. Ortiz, J.R. Humphrey, J.P. Durbano, D.W. Prather</i>	
Design and Implementation of RNS-Based Adaptive Filters	1135
<i>J. Ramírez, U. Meyer-Bäse, A. García, A. Lloris</i>	
Domain-Specific Reconfigurable Array for Distributed Arithmetic	1139
<i>S. Khawam, T. Arslan, F. Westall</i>	

Design and Implementations 1 (Posters)

Design and Implementation of Priority Queuing Mechanism on FPGA Using Concurrent Periodic EFSMs and Parametric Model Checking	1145
<i>T. Kitani, Y. Takamoto, I. Naka, K. Yasumoto, A. Nakata, T. Higashino</i>	
Custom Tag Computation Circuit for a 10Gbps SCFQ Scheduler	1149
<i>B. McAllister, S. Sezer, C. Toal</i>	
Exploiting Stateful Inspection of Network Security in Reconfigurable Hardware	1153
<i>S. Li, J. Tørresen, O. Søråsen</i>	

XXVI Table of Contents

Propose of a Hardware Implementation for Fingerprint Systems.....	1158
<i>V. Bonato, R.F. Molz, J.C. Furtado, M.F. Ferrão, F.G. Moraes</i>	

Design and Implementations 2 (Posters)

APPLES: A Full Gate-Timing FPGA-Based Hardware Simulator	1162
<i>D. Dalton, V. Bessler, J. Griffiths, A. McCarthy, A. Vadher, R. O'Kane, R. Quigley, D. O'Connor</i>	
Designing, Scheduling, and Allocating Flexible Arithmetic Components	1166
<i>V. V. Kumar, J. Lach</i>	
UNSHADES-1: An Advanced Tool for In-System Run-Time Hardware Debugging	1170
<i>M.A. Aguirre, J.N. Tombs, A. Torralba, L.G. Franquelo</i>	
Author Index	1175

Reconfigurable Circuits Using Hybrid Hall Effect Devices

Steve Ferrera and Nicholas P. Carter

Department of Electrical and Computer Engineering, University of Illinois,
Urbana-Champaign, Illinois 61801, USA

Abstract. Hybrid Hall effect (HHE) devices are a new class of reconfigurable logic devices that incorporate ferromagnetic elements to deliver non-volatile operation. A single HHE device may be configured on a cycle-by-cycle basis to perform any of four different logical computations (OR, AND, NOR, NAND), and will retain its state indefinitely, even if the power supply is removed from the device. In this paper, we introduce the HHE device and describe a number of reconfigurable circuits based on HHE devices, including reconfigurable logic gates and non-volatile table lookup cells.

1 Introduction

Over the last two decades, CMOS circuitry has become the dominant implementation technology for reconfigurable logic devices and semiconductor systems in general, because advances in fabrication processes have delivered geometric rates of improvement in both device density and speed. However, CMOS circuits suffer from the disadvantage that they require power to maintain their state. When power is removed from the system, all information about the state of a computation and the configuration of a reconfigurable circuit is lost, requiring that the reconfigurable device be configured and any information about the ongoing computation be reloaded from non-volatile storage each time the system containing it is powered on.

Magnetoelectronic circuits [1] overcome this limitation of CMOS systems by incorporating ferromagnetic materials, similar to those used in conventional hard disks. The magnetization state of these materials remains stable when power is removed from the device, allowing them to retain their state without a power supply and to provide “instant-on” operation when power is restored.

Much of the previous work on magnetoelectronic circuits has focused on the use of magnetoelectronic devices to implement non-volatile memory. In this paper, we describe a new class of magnetoelectronic device, the hybrid Hall effect device [2,4], that can be reconfigured on a cycle-by-cycle basis to implement a variety of logic functions, and present two initial applications for these devices: reconfigurable gates and non-volatile lookup table elements.

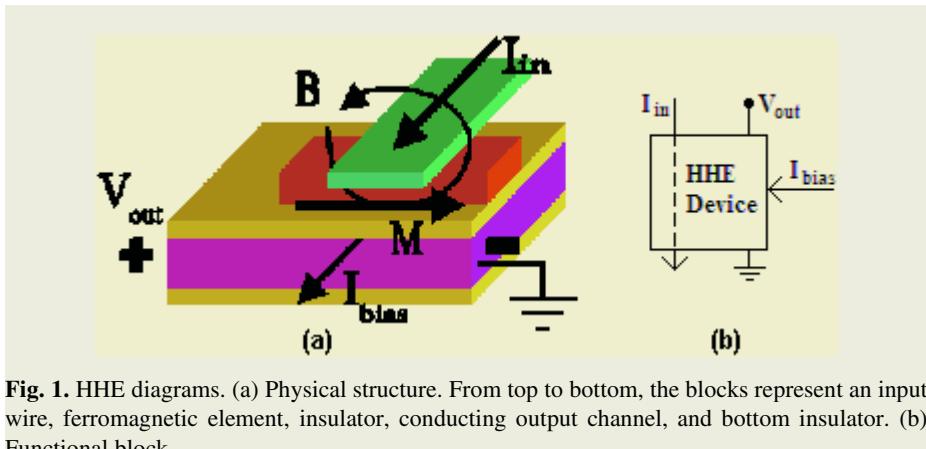
In the next section, we describe the HHE device and its basic operation. Section 3 presents two reconfigurable gate designs based on HHE devices, while Section 4

illustrates how the HHE device could be used to provide non-volatile storage for conventional lookup-table based reconfigurable systems. In Section 5, we present simulation results for our circuits. Related work is mentioned in Section 6, and Section 7 presents conclusions and our plans for future work.

2 HHE Device Description and Operation

The hybrid Hall effect device [2] is a semiconductor structure that contains a ferromagnetic element for non-volatile storage. Fig. 1 shows the physical structure of an HHE device along with a functional block diagram. The input to the device is a current along the input wire, which is at the top of Fig. 1(a). As shown in the figure, the current along the input wire creates a magnetic field in the ferromagnetic element beneath it. If the magnitude of the current is high enough, the induced magnetic field in the ferromagnetic element will magnetize it in the direction of the magnetic field, and the magnetization will remain stable once the input current is removed. An input current of sufficient magnitude in the opposite direction will magnetize the ferromagnetic element in the opposite direction, creating two stable states that can be used to encode binary values. If the magnitude of the input current is below the value required to change the magnetization state of the ferromagnetic element, which is a function of the dimensions of the device and the material used to implement the ferromagnetic element, the ferromagnetic element will retain its old magnetization state indefinitely.

The output voltage of the device is generated by passing a bias current through the insulated conductor at the bottom of Fig. 1(a). According to the Hall effect [3], the interaction of this bias current with the magnetic field generated by the magnetized ferromagnetic element produces a voltage perpendicular to the bias current. The sign of this voltage is determined by the magnetization of the ferromagnetic element and its magnitude is proportional to the magnitude of the bias current. Depending on the intended use of the device, a fabrication offset voltage may be added [4], making the



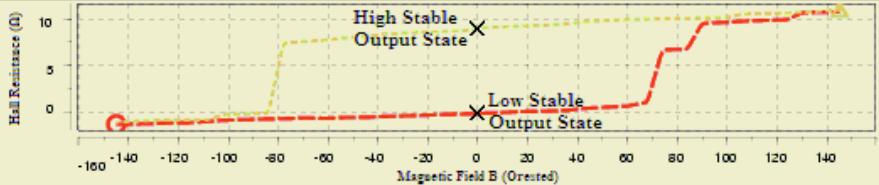


Fig. 2. Hysteresis loop for an HHE device

output voltage approximately 0V for one magnetization state and VDD for the other. Adding this fabrication offset makes it significantly easier to integrate the HHE device with CMOS circuits. Previous experiments [4] have fabricated HHE devices a small number of microns on a side, and the technology is expected to scale to significantly smaller devices in the near future, which will also reduce the amount of current required to set the magnetization state of the device.

The behavior of the HHE device is summarized by the hysteresis graph in Fig. 2. The Hall resistance, which relates the magnitude of the output voltage to that of the bias current, is plotted as a function of the magnetic field generated by the current along the input wire. As shown in the figure, there are two stable states when the input current, and thus the magnetic field, is 0, which correspond to the two magnetization states of the ferromagnetic element. A magnetic field of approximately ± 90 Oersted is required to change the magnetization state of the ferromagnetic element and shift the Hall resistance from one half of the hysteresis curve to the other.

3 HHE as a Reconfigurable Gate

Fig. 3 shows a high-level block diagram of the circuitry required to implement a reconfigurable gate using an HHE device. Signals A and B are the inputs to the gate, while signals G0 and G1 select the logic function to be performed by the gate. In the following subsections, we present two designs for the interface circuitry that converts the CMOS-compatible gate inputs into appropriate input currents for the HHE device.



Fig. 3. HHE reconfigurable gate

3.1 HHE Reconfigurable Gate with Reset

Our first reconfigurable gate design uses a reset-evaluate methodology similar to that used in dynamic CMOS circuits. In the reset phase of each clock cycle, an input current of fixed magnitude and direction is applied to the device to set its magnetization state. In the evaluate phase, a current in the opposite direction whose magnitude is determined by the inputs to the reconfigurable gate is applied, possibly switching the magnetization state to the opposite value. An HHE-based gate using this clocking methodology has been demonstrated in [4].

Fig. 4 illustrates the interface logic for a 2-input HHE reconfigurable gate using this methodology. To simplify the interface logic, we assume the use of an HHE gate with two input wires that are vertically stacked in different metal layers. Our conversations with researchers working at the device level indicate that such an extension to the base HHE device is possible, and we are initiating efforts to fabricate a test device with this design.

As shown in the figure, one of the input wires is only used in the reset phase. During this phase, the RESET signal is high, causing a current to flow upward through transistor MR and setting the magnetization state of the HHE device in the direction that corresponds to a logical 0. The PULSE input is held low during this period to ensure that no current flows through the other input wire.

During gate evaluation, the PULSE input is pulled high while the RESET input remains low. Depending on the inputs to the gate, any or all of transistors Ma, Mb, and MG0 may be turned on, allowing a current I_{in1} to flow downward through the input wire. These three transistors are sized such that at least two of them must be on in order for I_{in1} to be large enough to reverse the magnetization state of the HHE device, creating a majority function. Depending on the value of the G0 configuration input, this causes the device to compute either the AND or OR of its other inputs. Similarly, the value of the G1 input shown in Fig. 3 determines whether or not the inputs to the gate are inverted before they connect to the HHE device, allowing the gate to compute the NAND and NOR of its inputs as well.

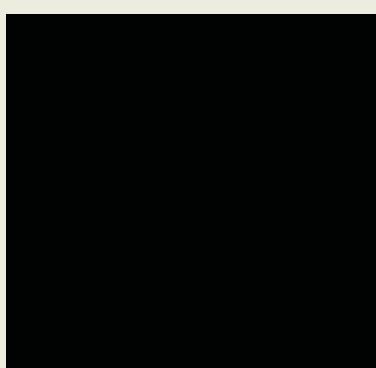


Fig. 4. Interface logic for HHE reconfigurable gate with reset. Current in the left input wire may only flow downwards, while current in the right input wire may only flow upwards.

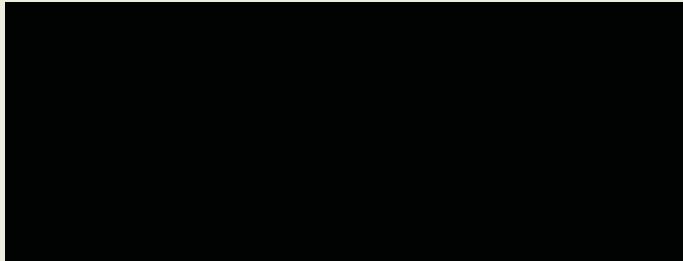


Fig. 5. Waveform of HHE reconfigurable gate operation with reset

Fig. 5 illustrates the operation of the reconfigurable gate when configured to compute the AND of its inputs ($G_0 = 0, G_1 = 0$). During each reset phase, the magnetization state of the HHE device is configured in the direction that represents a logical 0 by the reset path of the gate. During each evaluation phase, the magnetization state of the gate is conditionally set in the direction that represents a logical 1 based on the value of inputs A and B.

The circuit shown in Fig. 4 can be extended to compute functions of additional inputs by adding additional transistors to the pull-down chain shown in the figure and appropriately sizing the transistors in the pull-down chain. In Section 5, we present simulation results for a four-input reconfigurable gate of the type described in the next subsection. In addition, structures that connect additional configuration inputs in parallel with the transistor MG0 are also possible, allowing the gate to compute threshold or symmetric functions [5,6].

3.2 HHE Reconfigurable Gate with Output Feedback

One drawback to the circuit shown in Fig. 4 is that it consumes power each time the RESET signal is asserted, regardless of the value of its inputs and configuration. If the output of the gate remains constant from one cycle to the next, this can result in significant wasted power. To address this limitation, we have designed the static reconfigurable gate shown in Fig. 6, which uses output feedback to eliminate the need for a reset phase. Rather than resetting the magnetization state of the HHE device to a logical 0 on each cycle, this design provides two conditional pull-down chains, one of which allows current to flow in the direction that sets the device to a logical 0, and one of which allows current to flow in the direction that corresponds to a logical 1. The PULSE input to each pull-down chain prevents static power consumption by only allowing current flow during the time required to evaluate the output of the device. (approximately 2ns for current HHE devices) Feedback from the output of the device to the pull-down chains disables the chain that corresponds to the current output value, preventing power from being consumed on input changes that do not change the output of the device.

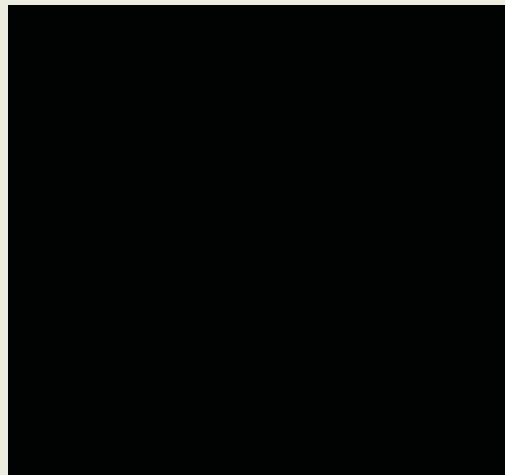


Fig. 6. Interface logic for HHE reconfigurable gate with output feedback

To demonstrate the operation of the gate with output feedback, consider the case where the gate is configured to compute the AND of its inputs ($G0 = 0, G1 = 0$). In this case, the a and b inputs to the circuit receive the uninverted values of the A and B inputs to the gate, while the a' and b' inputs receive the complement of A and B . Assume that the output of the gate starts at a logical 0. In this case, the left-hand pull-down chain in the figure is enabled, while the right-hand chain is disabled. Since $G0$ is set to logical 0, both the A and B inputs to the gate must be high for enough current to flow through the left-hand pull-down chain to flip the magnetization state of the HHE device and change the output of the gate to “1.” If the output of the gate starts at a logical 1, however, the right-hand pull-down chain is enabled while the left-hand one is disabled. Because $G0 = 0, G0' = 1$, and only one of the A or B inputs to the circuit must be 0 for enough current to flow to set the output of the gate to logical 0. Thus, the circuit computes the logical AND of its inputs.

This gate design requires somewhat more configuration logic than the reset-evaluate design, because it is necessary to provide both the true and inverted values of each input signal and the $G0$ configuration bit. The set of logic functions that can be computed by this style of gate can be expanded by including configuration circuitry that allows each input to be inverted or disabled individually, reducing the number of gates required to implement a circuit at the cost of increased gate complexity.

HHE reconfigurable gates with input inversion and input enabling may be incorporated into non-volatile reconfigurable logic devices such as PLAs and CPLDs. Currently, EEPROM transistors are the underlying technology of these systems. EEPROMs are useful for realizing product terms with wide-AND operations such as those commonly used in state machines and control logic. HHE-based logic for these devices will be more efficient than EEPROM-based logic because of its greater flexibility. For example, an HHE-based device would allow either two-level AND-OR or two-level OR-AND implementation of a given logical function, depending on which

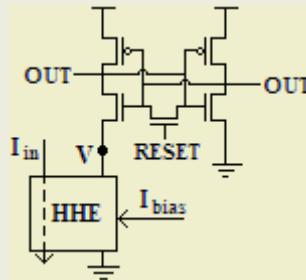


Fig. 7. HHE device incorporated into a logic block LUT cell

resulted in the fewest number of HHE gates used (i.e. fewest number of product/sum terms.) For complete non-volatile operation, the configuration bits for these HHE gates may be stored in non-volatile HHE LUT cells as described in the next section.

4 Non-volatile LUTs Using HHE Devices

HHE devices may also be used to add non-volatile operation to FPGAs based on more-conventional SRAM lookup tables, as shown in Fig. 7. In this circuit, an HHE device fabricated without an offset voltage is used to store the state of each SRAM cell in a lookup table by applying an appropriate current I_{in} to the HHE device during configuration. The HHE device will retain its state without requiring a power supply, acting as non-volatile storage for the configuration of the device.

To copy the state stored in the HHE device into the lookup table, the RESET signal is asserted to equalize the values of OUT and OUT'. When RESET goes low, a bias current is applied to the HHE device, causing it to generate either a positive or a negative voltage on terminal V depending on the magnetization state of its ferromagnetic element (since the HHE device has been fabricated without an offset voltage.) The cross-coupled inverters in the SRAM cell then act as a differential amplifier, bringing the output voltages of the SRAM cell to full CMOS levels. By applying RESET and the bias current to each SRAM cell in an FPGA simultaneously, the entire device can be reconfigured extremely quickly at power-on.

Although only a single HHE device is depicted in Fig. 7, one more may be added to the right leg of the LUT cell. In this manner, one of two configurations may be dynamically loaded into the LUT cell by applying the appropriate read bias current I_{bias} through the desired HHE device.

5 Simulation Results

Using the HSPICETM circuit simulator, we created a circuit model of the HHE device based on the techniques presented in [7], and have simulated HHE designs for recon-

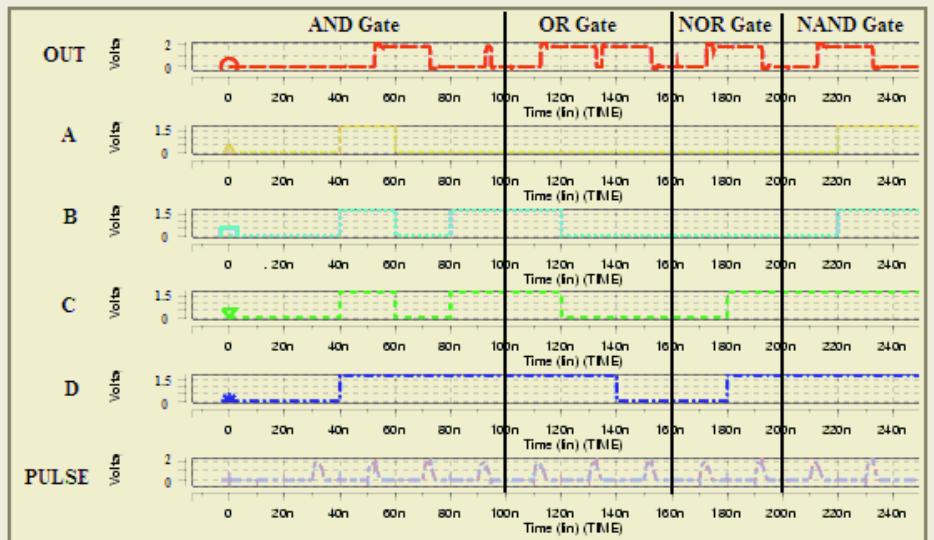


Fig.8. Simulations for HHE reconfigurable gate

figurable gate structures and non-volatile LUTs. Simulations have also been performed comparing power consumption between a reconfigurable gate with output feedback against one that uses reset pulses. The designs incorporate .18 μ CMOS transistors in a 1.8V technology.

In Fig. 8, we illustrate the operation of a 4-input HHE reconfigurable gate with output feedback. The gate is configured to compute different functions of its inputs over the course of the simulation, and inputs are allowed to change on multiples of 20ns. 10ns after each input change, the PULSE input to the gate is asserted to cause the gate to compute its output. The simulated device requires 2ns to compute its outputs, matching current experiments with prototype HHE devices.

One may notice that the HHE gate output attempts to switch at 92ns and 132ns. However, the output does not fully switch because not all of the inputs are logic 1 or logic 0 respectively. This indicates that input currents did not exceed the switching threshold of the ferromagnetic element, so the output reverts back to its previous state when PULSE goes low.

Simulations were also performed to compare the power dissipation of a reconfigurable gate using output feedback against one that uses reset pulses. In Fig. 9, we show the input current pulses associated with the input vectors from Fig. 8 for both types of gate, illustrating that the reset-based design requires more and larger current pulses than the static gate with output feedback. For these input vectors, simulations show an average power consumption of 4.09mW for the reset pulse design. Average power consumption for the design with output feedback is 1.69mW, an improvement of 2.42x, although power consumption for both gates will scale with clock frequency.

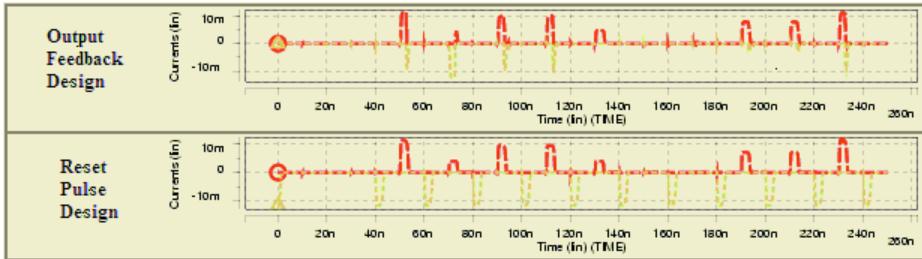


Fig. 9. Input current pulses for output feedback and reset pulse designs for HHE reconfigurable gate. Top curves represent current through left HHE input wire. Bottom curves represent current through right HHE input wire

In Fig. 10 we illustrate the operation of a non-volatile LUT using HHE devices. During the first 40ns, the output state of the HHE device is initialized to logic 1 and the RESET signal is high. At 40ns, the RESET signal is removed, and the LUT output becomes the same as that of the HHE device. At 60ns, the power is turned off ($VDD=0$), and the LUT output decreases exponentially due to discharge effects. At 140ns, power is restored, and the RESET signal is asserted. At 150ns, the RESET signal is disabled, and the LUT output is restored its pre-shutdown value.

6 Related Work

A number of other technologies exist that provide non-volatile storage in reconfigurable devices. Anti-fuses have the benefit of small area, but they are one-time programmable (OTP) and are mainly used for programming interconnections and not logic. EPROMs/EEPROMs are reprogrammable, but consume constant static power since they realize functions using wired-AND logic. Giant-magnetoresistive (GMR) devices are another type of magnetoelectronic device that can easily be integrated into LUT cells [8]. GMR-based designs have the disadvantage that two devices are required to hold the state of each LUT, as opposed to one HHE device, potentially making them less attractive, although this will depend on how well each type of device scales with fabrication technology.

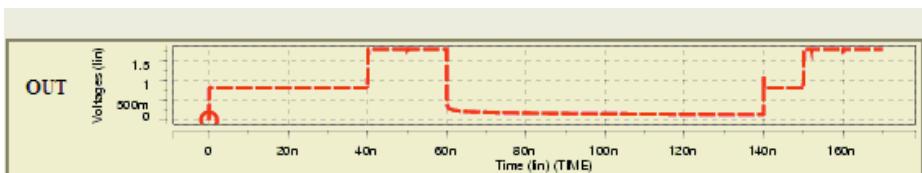


Fig. 10. Simulations for non-volatile LUT using HHE devices

7 Conclusions and Future Work

Hybrid Hall effect devices are a new class of magnetoelectronic circuit element that can be used to implement non-volatile reconfigurable logic and storage. In this paper, we have presented circuit-level designs for reconfigurable gates and non-volatile lookup table cells based on these devices, demonstrating the potential of these devices. We are currently working with researchers at the Naval Research Lab to fabricate prototypes of these circuits.

Future studies of HHE-based reconfigurable logic will focus on the system issues involved in building large-scale reconfigurable logic systems based on magnetoelectronic devices. In particular, the small size and fine-grained reconfigurability of these devices makes them very attractive and is leading us towards the design of systems based on simple logic blocks and regular interconnect patterns, trading reduced logic block utilization for reductions in interconnect area and complexity.

Acknowledgements

The authors would like to thank Mark Johnson of the NRL for information regarding the properties of HHE devices and for the experimental data used in our models. This material is based on work supported by the ONR under award No. N00014-02-1-1038. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the ONR.

References

1. Prinz, G.A.: Magnetoelectronics. *Science* 282 (1998) 1660-1663
2. Johnson, M., Bennett, B.R., Yang, M.J., Miller, M.M., Shanabrook B.V.: Hybrid Hall Effect Device. *App. Phys. Lett.* 71 (1997) 974-976
3. Streetman, B.G., Banerjee S.: *Solid State Electronic Devices*. 5th edn. Prentice Hall, Upper Saddle River, New Jersey (2000)
4. Johnson, M., Bennett, B.R, Hammar, P.R., Miller, M.M.: Magnetoelectronic Latching Boolean Gate. *Solid-State Electronics* 44 (2000) 1099-1104
5. Aoyama, K., Sawada, H., Nagoya, A., Nakajima, K.: A Threshold Logic-Based Reconfigurable Logic Element with a New Programming Technology. *FPL* (2000) 665-674
6. Muroga, S.: *Threshold Logic and Its Applications*. John Wiley & Sons, Inc. (1971)
7. Das, B., Black, Jr., W.C.: A Generalized HSPICE™ Macro-Model for Pinned Spin-Dependent-Tunneling Devices. *IEEE Transactions on Magnetics* 35 (1999) 2889-2891
8. Das, B., Black, Jr., W.C.: Programmable Logic Using Giant-Magnetoresistance and Spin-Dependent Tunneling Devices. *J. Appl. Phys.* 87 (2000) 6674-6679

Gigahertz FPGA by SiGe BiCMOS Technology for Low Power, High Speed Computing with 3-D Memory

Chao You*, Jong-Ru Guo*, Russell P. Kraft, Michael Chu, Robert Heikaus, Okan Erdogan, Peter Curran, Bryan Goda, Kuan Zhou, and John F. McDonald

youc@rpi.edu, guoj@rpi.edu

Abstract. This paper presents an improved Xilinx XC6200 FPGA using IBM SiGe BiCMOS technology. The basic cell performance is greatly enhanced by eliminating redundant signal multiplexing procedures. The simulated combinational logic result has a 30% shorter gate delay than the previous design. By adjusting and properly shutting down the CML current, this design can be used in lower-power consumption circuits. The total saved power is 50% of the first SiGe FPGA developed in the same group. Lastly, the FPGA with a 3-D stacked memory concept is described to further reduce the influence of parasitics generated by the memory banks. The circuit area is also reduced to make dense integrated circuits possible.

1 Introduction

Field Programmable Gate Arrays (FPGAs) have exclusive applications in many areas such as high-speed networking and digital signal processing. The maximum speed of most current FPGAs is around 300 MHz. The first gigahertz FPGA is a Current Mode Logic (CML) version of the Xilinx XC6200 implemented using IBM's SiGe BiCMOS technology [1], [2]. The XC6200 architecture has been selected because of its open source bit stream and available programming tools. Driven by the developing technology, the primary goal of this work is to design high-speed FPGAs while simultaneously focusing on reduced power applications.

The power consumption of this first gigahertz FPGA has a total cell power calculated as following:

$$P_{total} = N_{cell} \times N_{CML-tree} \times V_{supply} \times I_{tree}$$

Where P_{total} is the total cell power, N_{cell} is the total number of cells, $N_{CML-tree}$ is the total number of CML trees inside one basic cell and I_{tree} is the amount of current in each CML tree.

To obtain more power saving, all efforts are focusing on those four factors by reducing the voltage supply, amount of current in CML tree, number of trees in each cell and total number of cells used in an application.

* Both authors have contributed equally to this work.

The layout of the paper is this. Section 2 introduces the SiGe BiCMOS technology. Section 3 elaborates on an improved multiplexer structure to reduce the power supply from 3.4 V to 2 V. Section 4 presents an improved basic cell structure that eliminates redundant multiplexing procedures and thus increases performance. Section 5 illustrates the dynamic routing circuits that shut down unused circuits. Section 6 presents the concept of the 3-D FPGA aiming at reducing the area and the influence of parasitic effects. All the results of different FPGAs utilizing different currents are summarized and compared in Section 7. Finally, a brief future plan is described in Section 8.

2 SiGe BiCMOS Technology from IBM

The FPGA design is implemented by the IBM SiGe BiCMOS 7HP technology. The technology has all features of Si-base transistors, such as polysilicon base contact, polysilicon emitter contact, self-alignment and deep-trench isolation. With the use of a linearly graded Ge profile in the base region, three device factors, current gain, early voltage and base transit time, are improved [3]. Figure 1 shows an I_c and f_T curve in the IBM 7HP technology. The peak f_T point is at 1 mA. Later part of this paper shows a trade off between the power and performance based on the data from this curve.

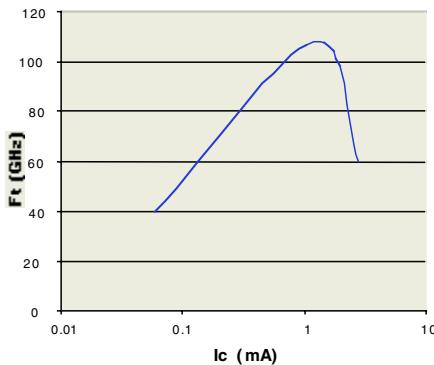
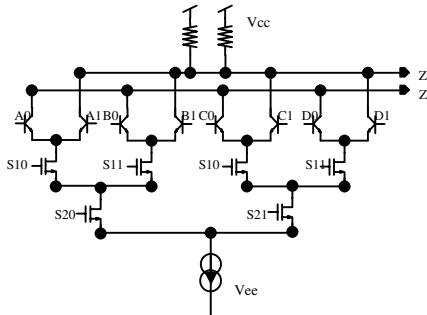
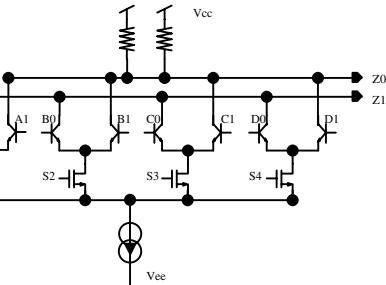


Fig. 1. I_c versus f_T in the IBM 7HP SiGe BiCMOS

3 Improved Multiplexer Structure

The XC6200 is a multiplexer-based FPGA instead of a LUT-based FPGA. As the building block of a basic cell, the single multiplexer design determines the supply voltage, gate delay and total power consumption of the basic cell.

The previous design uses a multiplexer design as shown in Figure 2 (a 4:1 multiplexer is shown as an example). The selection bits come in complementary pairs. Each time, only one branch of the tree is turned on. The corresponding transistor pair will be selected to pass the input signal through. For the NFET gate to work properly, the voltage supply must be large enough. The tallest CML tree in the design, which is an 8:1 multiplexer in the basic cell, determines the chip voltage.

**Fig. 2.** Previous multiplexer**Fig. 3.** New multiplexer

An improved multiplexer design is shown in Figure 3. Instead of using complementary selection bits for decoding, a separate decoder is used in the new multiplexer. During operation, only one NFET obtains a SET signal from the decoder. The corresponding transistor pair will be selected to pass the input signal through. There are two advantages for using this new multiplexer structure.

1. The new multiplexer has a single-level selection tree throughout the basic cell. The power supply is reduced to a minimum. For example, the previous uses a 3.4 V power supply, while the tested new chip uses a power supply of 2.0 V.
2. The previous multiplexer can't be turned off since at least one branch is on no matter what the selection bits are. The new multiplexer can be completely turned off by turning off the decoder. When one multiplexer is not used in an application, it may be turned off to save power. The later part of this paper has a detail description about how dynamic routing works.

By implement the basic cell with the new multiplexer structure, the power consumption can be reduced by 40% without any modification to the other parts of design. The extra decoders added to the circuits can be implemented with CMOS. The programming bits only add negligible leakage power consumption.

4 Improved Basic Cell

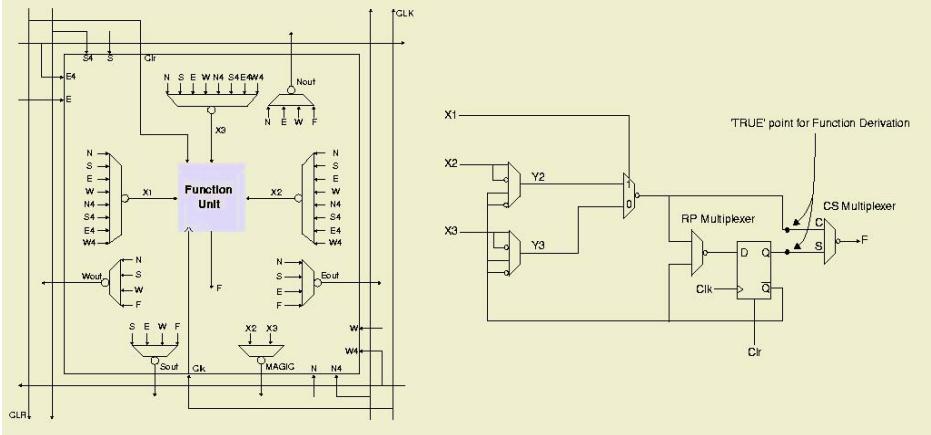
A basic cell is the prime element of the XC6200. The original XC6200 basic cell and the function unit are shown in Figure 4. Each cell obtains two signals in each direction. One signal is from the cell's neighbor in corresponding N, S, E and W directions. The other is the FastLANE® signal, which is a shared signal by four basic cells in the same row or column. The FastLANE provides more routing capability to an XC6200 cell. Each basic cell performs two-variable logic functions. The two-variable inputs are selected from the above eight signals. An example of the logic function table is shown in Table 1.

Logic function result "C" (for combination logic) is sent out to a CS multiplexer (S for sequential logic). The selected signal "F" from the Function Unit is selected again at the output multiplexers before it can reach the next neighbor cell. To provide more routing capability, each basic cell also routes one signal in one direction to another. The function is called a redirection function.

Table 1. Example of Logic Function Table

Function	X1	X2	X3
INV	XX	A1	A1
A1 AND B1	A1	B0	A0
A1 XOR B1	A1	B1	B0

XX: Don't care. "1" is for signal. "0" is for compliment signal

**Fig. 4.** XC6200 and Function Unit

When its neighbor cell finally receives the logic result, the neighbor cell will select this result again among other signals. The neighbor cell will determine which signal to use: combinational logic result, sequential logic result, redirected signal or FastLANE signal. Because of this, all multiplexing procedures at the output stage of a basic cell can be considered as redundant, which must be removed to increase performance.

Figure 5 shows an improved basic cell structure (BCII) and a function unit without the superfluous multiplexing circuits. The combinational logic result and sequential logic result are delivered to neighbor cells directly. The diamond arrow stands for a combinational logic result and the round arrow stands for a sequential logic result in Figure 5. To preserve the routing capability the XC6200 provided, the redirection function is sustained. Instead of sending one output in each direction, BCII sends out three outputs in each direction. They are the combinational logic result, sequential logic result and redirection function result.

With more outputs in each direction, each BCII now receives three inputs from its neighbor cell in one direction. The input multiplexers and redirection multiplexers must be modified to adapt to this change. As shown in Figure 5, the input multiplexers now receive three inputs from each neighbor cell and one signal from FastLANE. The X2 and X3 multiplexers in Figure 4 also receive a feed back signal from the MS-Latch. The implementation of the X2 and X3 multiplexer in BCII are shown in Figure 6. The 16:1 multiplexer and 17:1 multiplexers are implemented by a two-level multiplexing process as shown in the figure. The redirection multiplexers now receive nine signals in three directions. 9:1 multiplexers are used for the

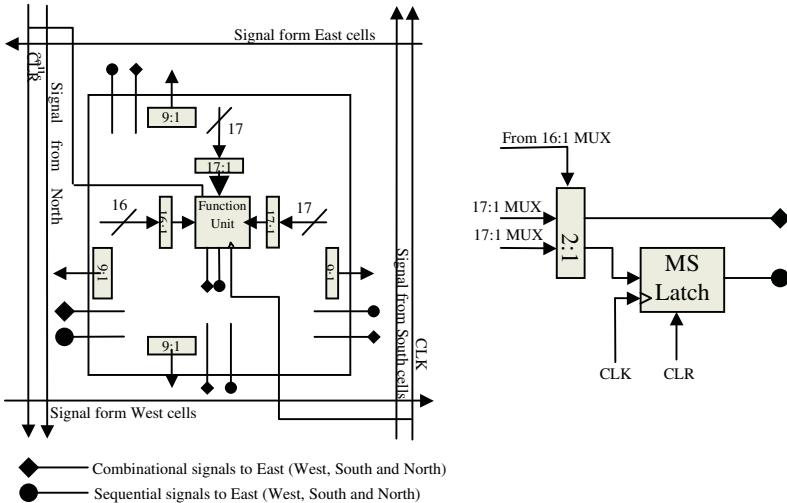


Fig. 5. BCII (Left) and Function Unit (Right)

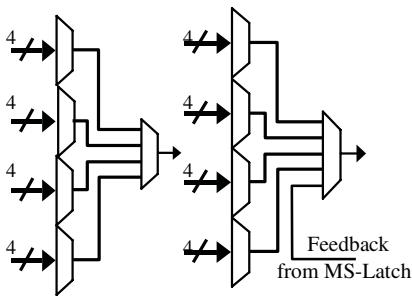


Fig. 6. 16:1 input MUX (Left) and 17:1 input MUX (Right)

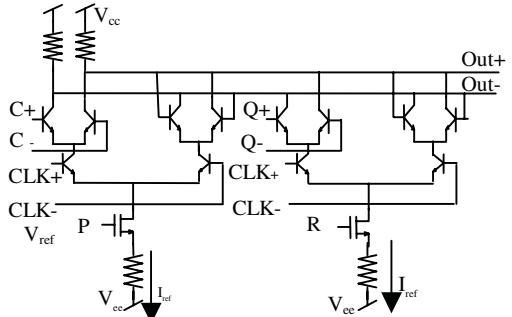


Fig. 7. Improved First Stage MS-Latch

redirection function as shown in Figure 5. 9:1 multiplexers are implemented by the new multiplexer structure described in Section 2.

Besides the modification made on the input multiplexers and redirection multiplexers, the Master-Slave Latch (MS-Latch) can be modified to save power as well. The first stage MS-latch can be combined with the RP multiplexer in Figure 4. The revised circuit is shown in Figure 7. The selection bits for the RP multiplexer are used as enable bits for the two CML trees. In operation, only one CML tree is turned on to pass the signal to the second stage. The benefit of using this revised first stage MS-latch is that it can be turned off by setting both R and P to zero.

The major advantage of BCII is the shorter gate delay on the critical signal path. With the IBM SiGe technology, the current in CML trees can be reduced while still maintaining a shorter or comparable gate delay with the original design.

5 Dynamic Routing

Other factors that can be altered in the power equation are the number of cells and the number of CML trees in each basic cell. One primary notion is to turn off unused cells or unused CML trees when an application is loaded into the FPGA.

In a chip scale, for example, when a loaded application uses only 12 cells in a 4 x 4 gate array, turning off the unused 4 cells will save 25% power. The cells that need to be turned off must be determined by the application. Thus, the turning-off scheme is dynamically controlled by the application instead of hard-wired design. One way to turn off an entire cell is by introducing another enable bit for the cell. The enable bit will enable/disable all CML trees in a single cell.

In a cell scale, there are more complicated schemes to turn off unused CML trees. When one cell is configured to perform a certain function, the incoming signals pass through specific path with several multiplexers involved. All other circuits, which are not involved in the path, have to be turned off to save power. The circuits that have to be turned off are listed as follows.

1. When a cell only performs the combination logic, the sequential logic circuit and all redirection multiplexers may be turned off to save power.
2. Each cell only accepts two signals to generate a logic function. Then at least two output-drivers and the redirection multiplexers in its neighbor cells can be turned off to save power. If the input signal of a cell is selected from FastLANE, all four output-drivers in all neighbor cells can be turned off to save power.
3. When a cell is only used to redirect a signal from one neighbor cell to another neighbor cell, the function unit of the cell may be turned off to save power.

Dynamic Routing Circuits

Dynamic routing circuits are used to turn off CML trees and the whole basic cell. The dynamic routing circuits are the control circuit of the select bit for multiplexers. They are implemented by CMOS to save layout area.

Figure 8 shows the 17:1 multiplexer with dynamic routing circuits. As shown in the figure, the decoder has an enable bit. A reset on the decoder enable bit will turn off all multiplexers in the figure. The Selection-Indicator and the Complement-Selector works as the decoder for the second level multiplexer. Either a signal or its complement signal will be selected as the output. The Sequential Logic Selection Control circuit indicates the feedback from sequential logic will be selected at the second level multiplexer. The Safety Circuit is used to prevent the case that both the input and feedback signal are enabled.

Figure 9 shows a redirection multiplexer. The redirection multiplexer is only needed for redirecting signals. One 4-bit decoder's outputs are used as selection bits. The first design has ten outputs from the decoder. The first nine outputs work as selection bits for the multiplexer. The last output is asserted when the output multiplexer needs to be turned off. The remaining six outputs are unused by any other circuits. Other RAM bits are needed as the Master Key (the master enable for the whole cell), FU Enable and MS latch enable. Further research shows that these RAM bits can be connected to the otherwise unused outputs of the decoder.

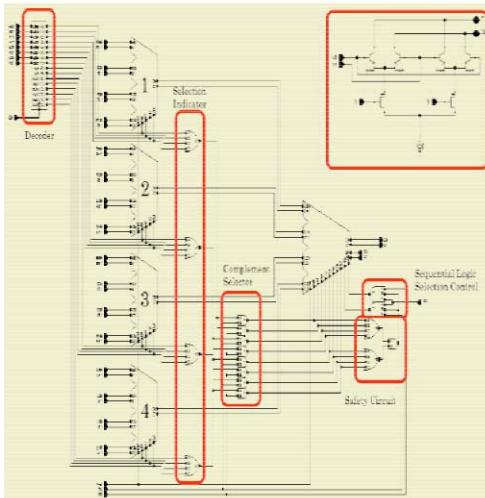


Fig. 8. Dynamic Routing Circuit

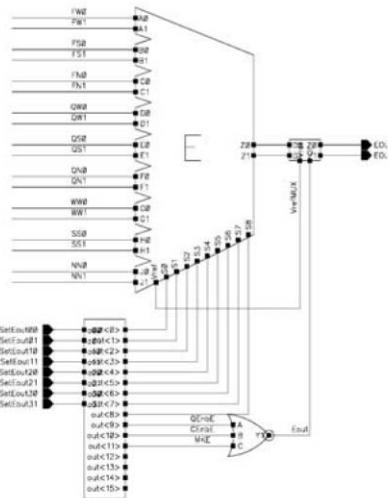


Fig. 9. Output MUX with control circuit

A revised redirecting multiplexer with its control circuits is shown in Figure 9. Output nine is used for the sequential enable. Output ten is used for the combinational enable. Output eleven is used for the master key.

In the same figure, The QEnbE is the enable bit for the driver of sequential logic facing the east side. The CEnbE is the enable bit for the driver of combinational logic facing the east side. The MKE is the master key from the east side. If any of the first nine signals are selected, the output MUX driver is on. Otherwise, the redirection multiplexer and the driver are turned off. The following table shows the actual power reduction in the new design for three cases.

Table 2. Dynamic Routing Power Usage

Design	Tree #	Usage
BCII Maximum Usage	21	100%
Case I (Comb./Sequential. Logic)	10/12	47.6%/57.1%
Case II	Sequential, One Redir.	71.4%
	Sequential, Two Redir.	85.7%
	Sequential, Three Redir	100%
Case III	3 tree/dir	14.2%/dir

Case I: Only combinational logic or sequential logic is used.

Case II: Sequential logic and redirection function are used.

Case III: Only redirection function is used

6 3-D Stack Memory Concept for FPGA

In the FPGA, memory is used to program logic function. However, the considerable amount of memory and wire has occupied a significant area of the chip and reduced the operating frequency due to the longer interconnect. 3-D integration is one of the solutions to reduce the area and alleviate the parasitic effects.

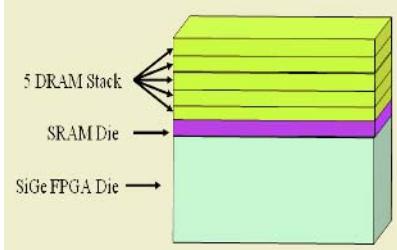


Fig. 10. The 3-D integration of the FPGA/ stack memory.

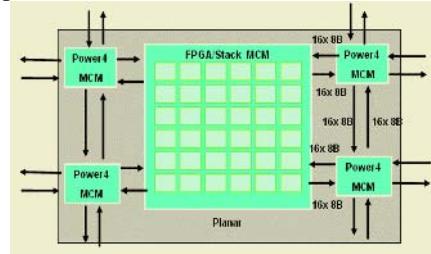


Fig. 11. The block diagram of the 3-D FPGA chip

In the 3-D design concept, similar circuits are grouped together, for example memory, and manufactured on the same wafer. By using “chemical adhesives”, these wafers can be glued and stacked. To connect the different wafer layer circuits, 3-D vias are used. Figure 10 shows the cross section of a 3-D structure.

On the top is the DRAM stacks which provide the FPGA different bit streams to the configuration memory below it (the SRAM die). After selection, the desired bit stream is stored in the SRAM die then passed to the SiGe FPGA die to perform the logic function. It is easy to observe the size of the FPGA is greatly reduced and the parasitic effects can be eased due to the shorter length of the interconnect wires.

Figure 11 shows the block diagram of the 3-D FPGA chip. The 3-D FPGA/Stack Memory block is located in the center and the cells around it are the processors. The FPGA/ stack memory is used to route signals between processors that can broaden the multi-processor applications by programming different personalities in the DRAM. The DSP chip, A/D and D/A converters or communication circuits can replace the processor blocks. Thus, the FPGA chip can serve as the reconfigurable interface between blocks for different applications.

7 Simulation Results

The first gigahertz FPGA chip opens a gate to fast reconfigurable computing. Its continuing work involves performance improvement, lower power consumption design and curbing the basic cell layout area to a scaleable size. High performance is still the primary goal while keeping the power consumption as low as possible. With the IBM SiGe 7HP technology, the performance and power consumption can be balanced by varying the amount of current in CML trees. By moving the current from the peak f_T point, the power consumption is reduced with the lost of performance. Table 3 exhibits the power and delay relationship for a simulated AND gate. Figure 12 shows the power and delay trade-off of BCII in the IBM SiGe 7HP process. The best

trade-off point is at 0.4 mA CML tree. A current larger than 0.8 mA will have a short gate delay at the sacrifice of power savings.

Table 3. Power and Delay Chart for Designs

Design	Power (mW)	Delay (ps)
BC 0.6 mA	53	80
BCII 0.8 mA	16	46
BCII 0.6 mA	12	55
BCII 0.4 mA	8	70
BCII 0.2 mA	4	120

An AND gate is simulated for design comparison
BC has 28 CML trees. BCII has 21 CML trees (10 trees for Combinational Logic)

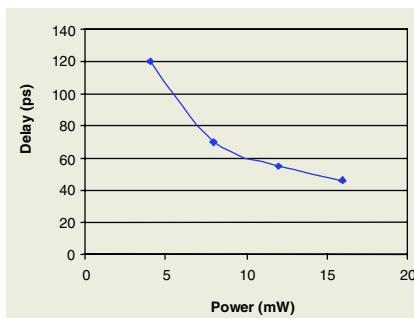


Fig. 12. Power Delay Trade-off in the IBM 7HP Process

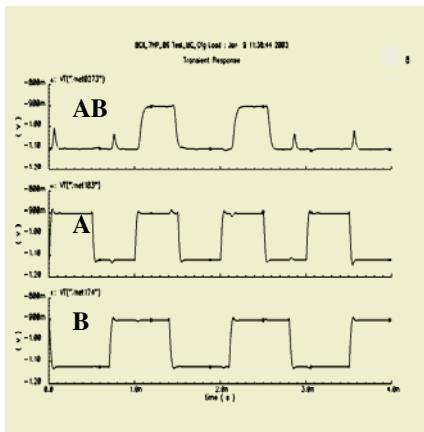


Fig. 13. Simulation result of an AND gate

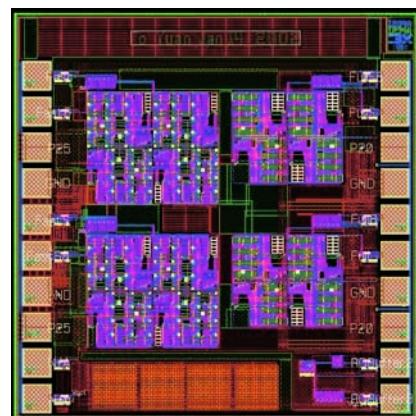


Fig. 14. BCII IBM 7HP layout

Figure 13 shows a simulated AND gate result. The current in the CML tree is 0.6 mA. The gate delay is 55 ps at the temperature of 25°C with a voltage swing of 250 mV.

8 Conclusion and Future Work

With the improved basic cell structure, the performance of a basic cell is improved by 30%. Adjusting the current in CML trees can result in different power savings. The dynamic routing method reduces the power consumption further. One design with the BCII cell has been shipped out for fabrication. The layout of the chip is shown in Figure 14. Future work involves chip test and measurement. The IBM SiGe 8HP technology will be released around August 2003. Further research result will be forth coming after implementing the BCII structure with the faster IBM SiGe 8HP technology.

References

- [1] John F. McDonald and Bryan S. Goda, "Reconfigurable FPGA's in the 1-20GHz Bandwidth with HBT BiCMOS", Proceedings of the first NASA/ DoD Workshop on Evolvable Hardware, pp. 188-192.
- [2] Bryan S. Goda, John F. McDonald, Stephen R. Carlough, Thomas W. Krawczyk Jr. and Russell P. Kraft, "SiGe HBT BiCMOS FPGAs for fast reconfigurable computing," IEE Proc.-Compu. Digi. Tech, vol.147, no. 3 pp. 189-194.
- [3] "IBM SiGe Designer's manual", (IBM Inc. Burlington Vermont. 2001).
- [4] D. Harme, E. Crabbe, J. Cressler, J. Comfort, J. Sun, S. Stiffler, E. Kobeda, M. Burghartz, J. Gilbert, A. Malinowski, S. Dally, M. Rathapanhyarat, W. Saccamango, J. Cotte, C. Chu, and J. Stork, "A High Performance Epitaxial SiGe-Base ECL BiCMOS Technology," IEEE IEDMTech Digest, 1992, pp. 2.1.1-2.1.4.

Implementing an OFDM Receiver on the RaPiD Reconfigurable Architecture*

Carl Ebeling¹, Chris Fisher¹, Guanbin Xing², Manyuan Shen², and Hui Liu²

¹ Department of Computer Science and Engineering
University of Washington

² Department of Electrical Engineering
University of Washington

Abstract. Reconfigurable architectures have been touted as an alternative to ASICs and DSPs for applications that require a combination of high performance and flexibility. However, the use of fine-grained FPGA architectures in embedded platforms is hampered by their very large overhead. This overhead can be reduced substantially by taking advantage of an application domain to specialize the reconfigurable architecture using coarse-grained components and interconnects. This paper describes the design and implementation of an OFDM Receiver using the RaPiD architecture and RaPiD-C programming language. We show a factor of about 6x increase in cost-performance over a DSP implementation and 15x over an FPGA implementation.

1 Introduction

Current SOC platforms provide a mix of programmable processors and ASIC components to achieve a balance between the flexibility and ease-of-use of a processor and the cost/performance advantage of ASICs. Although ASIC designs will continue to provide the best implementation for point problems, typically several orders of magnitude better than processors, platforms require flexibility that ASICs cannot provide. Relying on ASIC components reduces a platform's coverage to only those computations the ASICs can handle. It also reduces the platform's longevity by restricting its ability to adapt to changes caused by new standards or improvements to algorithms.

There has been a recent move to include configurable components on platforms in the form of FPGA blocks. So far this has met with only modest success. The main obstacles have been the large overhead of traditional fine-grained FPGAs and a lack of high-productivity programming tools for would-be users of configurable architectures. It is difficult to find a use for configurable logic when the cost-performance penalty approaches two orders of magnitude over ASIC components.

Our research has focused on showing that coarse-grained configurable architectures specialized to an application domain can substantially reduce the overhead of reconfigurable architectures to the point where they can be a viable alternative to DSP and ASIC components. We have defined a coarse-grained configurable architecture called RaPiD, designed a programming language called RaPiD-C, and implemented a

* This research was funded by the National Science Foundation.

compiler that maps RaPiD-C programs to the RaPiD architecture. This paper begins with a brief description of the RaPiD architecture and programming tools. We then describe the OFDM application and describe in detail the implementation of the compute-intensive parts of this application in RaPiD. Using an emulator of the RaPiD architecture, we have demonstrated the real-time execution of this application. We then compare the performance and cost of this implementation to that of DSP, FPGA and ASIC implementations. Finally we discuss the system issues of constructing a complete implementation of OFDM in a platform that contains RaPiD components.

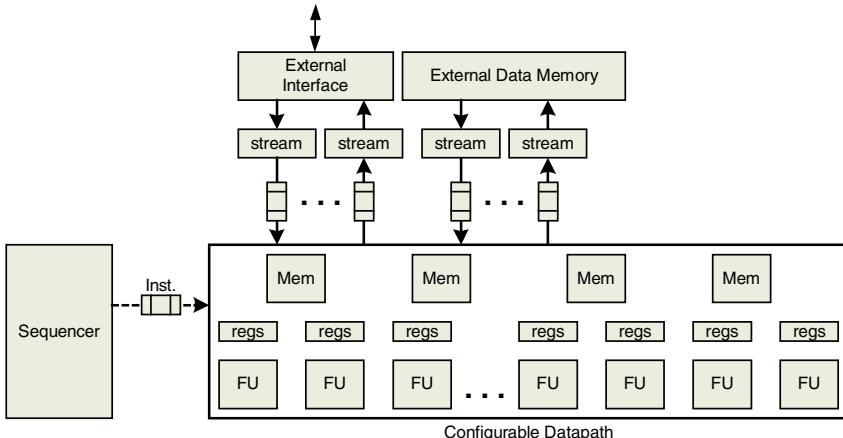


Fig. 1. Architecture of a RaPiD component.

1.1 The RaPiD Architecture

Fig. 1 gives an overview of the RaPiD architecture. RaPiD has been described in detail in previous papers [1,2,3,4] and will only be summarized here. The configurable datapath, which is the main part of a RaPiD component, contains a set of specialized functional units such as ALUs and multipliers that are appropriate to the application domain. The datapath also contains a large number of registers distributed throughout the datapath, which provide the data bandwidth required to keep all the functional units busy on every cycle. There are no register files: Instead, a number of small, embedded memories provide storage for data that is used repeatedly.

The components of the datapath are connected together via a configurable interconnection network comprising segmented busses and multiplexers. The sequencer runs a program that executes a computation on the configured datapath. Part of the interconnection network is statically configured before execution, while the rest is controlled dynamically by the sequencer. The compiler determines which part of the interconnect is statically configured and which is controlled by the program.

The datapath accesses data in external memory and other nodes using a streaming data model. Data streams are created by independent programs that operate on behalf of the datapath to read or write data in memory or communication buffers. The stream programs are decoupled from the datapath using FIFOs and executed ahead/behind the datapath to prefetch/poststore data. Communication with other

components, which can be processors, smart memories or ASIC components, is done using data streams in the style of Unix pipes.

Programs for RaPiD are written in the RaPiD-C programming language, which is an assembly-level language for the parallel RaPiD datapath. The programmer writes programs using datapath instructions, each of which specifies a parallel computation using a data-parallel style of programming. The compiler performs the pipelining and retiming necessary to execute datapath instructions at the rate of one per clock cycle. For more details on the RaPiD-C language and compiler, refer to [5].

In [4] we define a “benchmark cell” which comprises 3 ALUs, 3 embedded memories, one multiplier, and 6 datapath registers, connected using a set of 14 tracks of interconnect buses. The benchmark RaPiD array, comprised of 16 benchmark cells, was shown to perform well across a range of applications.

2 Multiple-Antenna OFDM Application

Orthogonal frequency-division multiplexing (OFDM) is a form of modulation that offers a significant performance improvement over other modulation schemes on broadband frequency selective channels. These inherent advantages make OFDM the default choice of a variety of broadband applications, ranging from digital video broadcasting (DVB-T), to wireless LAN, to fixed broadband access IEEE 802.16a. Currently, OFDM is also regarded as the top candidate for the 4th generation cellular network. OFDM can be combined with an antenna array to perform spatially selective transmission/reception so that information can be delivered to the desired user in an efficient manner. This combination provides an elegant solution to high performance, high data rate multiple-access networks.

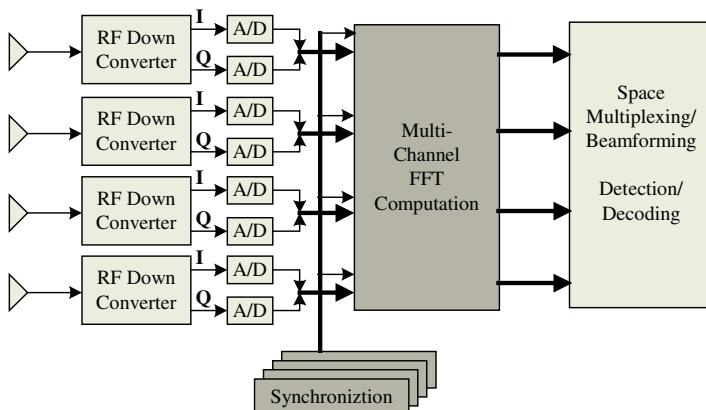


Fig. 2. Antenna Array OFDM Receiver

Fig. 2 shows the block diagram of an antenna array OFDM receiver. The signals from the antennas are first down-converted to base-band and digitized. The OFDM frame timing and carrier frequency are then detected using the synchronization module. Following that, the FFT is computed for each antenna input to convert the incoming signal back to frequency domain, allowing information on each individual

ing signal back to frequency domain, allowing information on each individual sub-carrier to be demodulated.

The shaded portions are the computationally intensive units, namely, synchronization and multi-channel FFTs, which were implemented using the RaPiD architecture. Demodulation and beam-forming typically comprise a small part of the total computation and can be realized using ASIC or programmable DSP components.

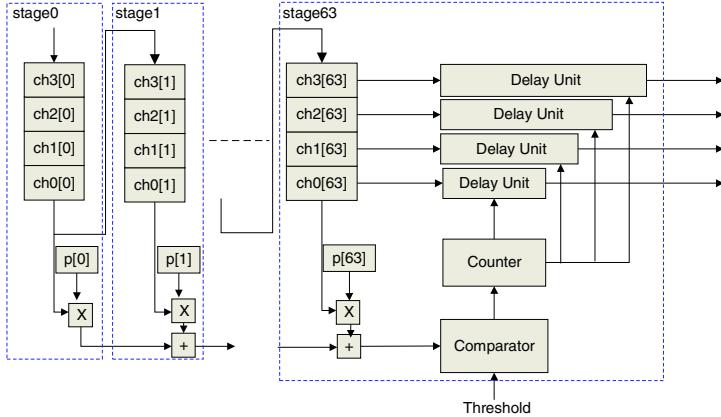


Fig. 3. OFDM Synchronization Module

Synchronization. Synchronization is achieved by correlating the received signal with a known pilot embedded in the transmitted signal. Peaks at the outputs reveal not only the timing of the OFDM frames but also the carrier information. As shown in Fig. 3, the signals from the four antennas are interleaved into a single stream entering the array. Successive windows of 64 samples are correlated with the known pilot signal and the result is compared against a threshold. When a peak is detected, the time offset is used to set the delay of the delay line for the corresponding antenna. This initializes the delay lines so that the signals entering the FFT unit are synchronized to each other and to the start of the frame.

We implemented a maximally parallel correlator that searches for the peak signal over a 64-sample window. This synchronizer can detect a frame-to-frame synchronization offset of ± 32 samples and apply this new offset immediately to the data symbols in the same frame. Since all the coefficients are $+1/-1$, the complex multiplies are implemented by a complex add/subtract. Thus a total of 128 16-bit ALUs and about 520 registers are required, in addition to the 16 embedded memories used to implement the delays. This corresponds to approximately 43 RaPiD benchmark cells.

A less expensive implementation can be used if the offset changes by at most ± 4 samples from one frame to the next. In this case, the requirements are reduced to 22 ALUs and 16 embedded memories, corresponding to 8 cells of the benchmark RaPiD array. We will call these two different implementations “parallel” and “tracking”.

Multi-Channel FFT. We chose the Radix-4-decimated-in-frequency architecture proposed by Gold & Bially [9], which is particularly efficient when the FFT is computed simultaneously on interleaved data streams. Fig. 4 shows one radix-4 stage, which is composed of three main blocks: a) a delay commutator that delays and skews the in-

put; b) a 4-point butterfly computation unit that requires only adders and I/Q swaps; and c) a twiddle factor multiplier. The initial delay unit is implemented as part of the delay lines in the synchronizer. The commutator is also implemented using datapath memories, while the butterfly computation and twiddle factor multiplication are implemented by using the ALU and the multiplier function units, respectively. All calculations are performed using 16-bit fixed-point numbers.

Each radix-4 stage has slightly different memory requirements for the commutator, but the entire 64-way FFT uses a total of 31 datapath memories, 18 ALUs, 12 multipliers and about 120 registers. This fits comfortably in the 16 cells of the benchmark RaPiD array.

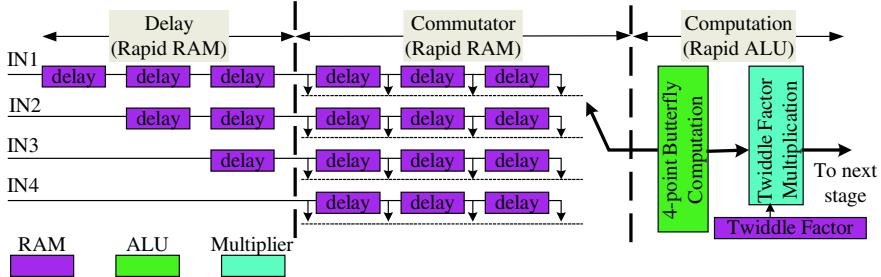


Fig. 4. One stage of the multi-channel FFT implementation

2.1 Implementation Details

This multi-channel OFDM front-end comprising the synchronizer and FFT was programmed using RaPiD-C, compiled using the RaPiD-C compiler [5], and run on the RaPiD architecture emulator [3]. The emulator comprises a set of FPGAs that implement the sequencer, datapath and data streams, and a 64 MByte streaming memory system. This memory system is used to simulate the high-rate data streams that occur in communications systems and can provide 4 I/O streams at an average data rate of close to 1 data value per cycle each. The emulator currently runs at 25 MHz.

To run the OFDM emulation, the external memory is loaded with 200 frames of 4-channel OFDM signals, interleaved, as they would appear in the data stream from the antenna. Each frame consists of one pilot symbol followed by 6 data symbols and one guard interval, as shown below. Both the pilot and data symbol have 64 samples.

Pilot	Data Symbol 1	Data Symbol 2	• • •	Data Symbol 6	Guard Interval
-------	---------------	---------------	-------	---------------	----------------

The RaPiD array then executes the four-channel OFDM demodulation program described above at a clock rate of 25 MHz. The input data stream is read at the rate of one complex value per cycle using two parallel 16-bit streams, and the output values are captured to memory at the same rate. The output values can then be analyzed and displayed using standard analysis modules. Fig. 5 shows a screen shot of data generated during the experiment.

The modulation scheme used in the experiment is QPSK. The current 25MHz RaPiD emulator implementation yields an effective data rate of 4ch x (6 / 8) x

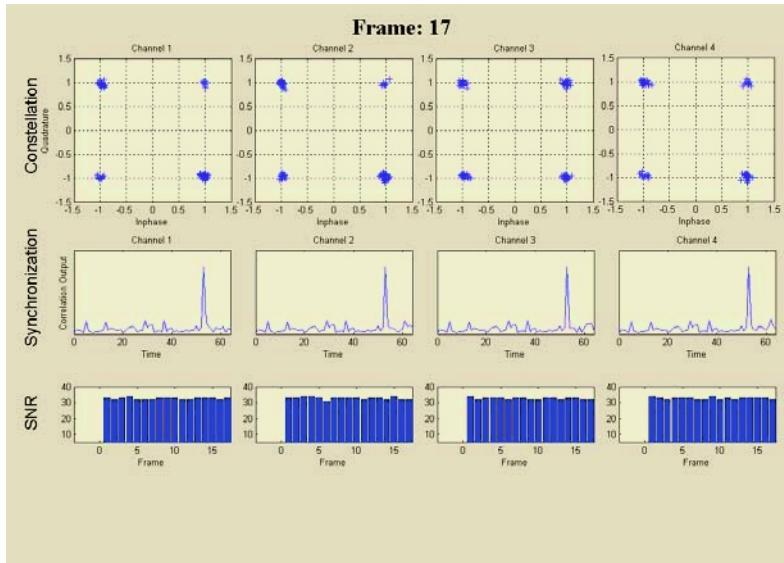


Fig. 5. Screen shot of the experimental results

2bits/symbol \times 25Mhz = 150Mbps when 4 channels transmit different bit streams; or a data rate of 1ch \times (6/8) \times 2bits/symbol \times 25Mhz = 37.5Mbps when 4 channels are combined in a low SNR scenario. In either case, the achieved data rate far exceeds that of any existing standard.

2.2 System Issues and Implications

Our OFDM implementation executes the synchronizer and interleaved FFT in a single RaPiD component. This results in a very complex program and a very large array. The correlator and FFT functions have very different requirements: the correlator has no need of multipliers and memories (unless it is time-multiplexed), while the FFT relies heavily on multipliers and memories. Moreover, the correlator and FFT alternate their computation: The correlator operates only on the pilot frames, while the FFT operates only on the data frames. Finally, there are different performance options for the correlator. Thus, the OFDM receiver can be implemented in a number of different ways in a system comprising multiple reconfigurable components. These alternatives are shown in Fig. 6 and described below. The numbers in the lower right-hand corner of each array indicates the size of the array in terms of the number of RaPiD benchmark cells. We have assumed that arrays are sized in multiples of 8 cells. Thus the 43-cell implementation of the correlator fits into a 48-cell array.

One, high-performance array (a, b): The single, high-performance array implemented as described above (a) is very expensive. If it is implemented as a homogeneous array with evenly distributed resources, then it has a low utilization because of the varying requirements of the correlator and interleaved FFT. Using the more constrained tracking synchronizer reduces the cost of the array dramatically (b).

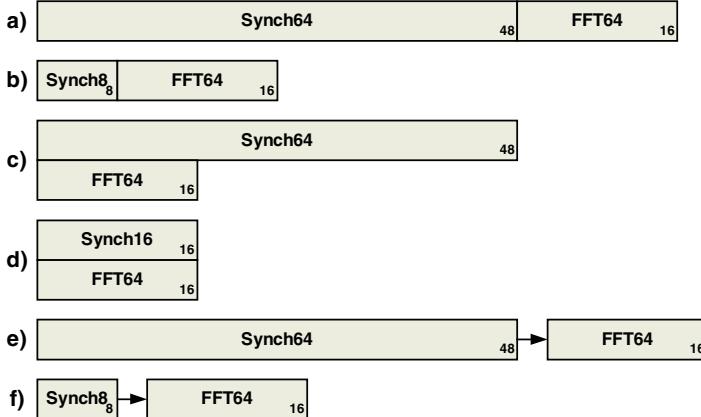


Fig. 6. Alternative OFDM receiver implementations

One array that alternates between the synchronizer and FFT (c, d): This implementation time-multiplexes the array between the two functions. To make this feasible, the datapath must be able to switch between two tasks very quickly, in a few microseconds rather than the many milliseconds required by FPGAs. One advantage of a coarse-grained architecture like RaPiD is that it has many fewer configuration bits. Even so, downloading a new configuration still takes too long for this application. The alternative is to use multiple configuration contexts, allowing reconfiguration in only a few clock cycles. However, the extra context increases the cost by about 10%.

Rapid switching between the correlator and the FFT makes more efficient use of the array since the two computations are not active at the same time. This is especially true if a cheaper tracking synchronizer is used that matches the size of the FFT (d).

Two arrays, one executing the synchronizer and one the FFT (e, f): Instead of providing a single reconfigurable array, platforms typically will provide multiple components with different sizes and capabilities. This permits the functionality of a component to be matched to the requirements of the function. In this case, the synchronizer can be mapped to a component with just ALUs and registers, while the FFT is mapped to a component with multipliers and memories. If a tracking synchronizer is used, then a much smaller array (f) can be used for the correlator.

3 Performance and Area Comparison

In this section we compare the performance and cost of the RaPiD implementation of the OFDM receiver to implementations in a DSP, ASIC and FPGA. We first describe how we obtained the numbers for each of these technologies before presenting the results. Area estimates are given in terms of λ^2 where λ is half the minimum feature size of the given technology. Performance is normalized to a .18 μ process technology. Although we have been very careful in reaching the area and performance estimates, we must stress that they are only estimates. However, even if our estimates are off even by a factor of 2, the overall conclusions do not change.

RaPiD - Our study of the RaPiD benchmark architecture [4] included detailed layouts of a benchmark cell in a .5 μ 3-layer metal CMOS technology and our area estimates for the RaPiD implementation are based on this layout. We also developed a timing model using this layout, and showed that except for some recursive filters, applications can be pipelined to achieve a 100 MHz clock rate. Scaling to a .18 μ technology would push the clock rate above 300 MHz.

Table 1 gives the performance and area results for several alternative RaPiD implementations of the OFDM receiver. The letters in this table refer to the corresponding implementations in Fig. 6. Note that the first implementation (a) is used as the comparison for the ASIC and FPGA implementations, (b) is used to compare against the DSP. Implementations (b) and (f) are the implementations that would most likely be used in practice. Implementation (d) is the most cost-effective option, but it has the disadvantage of requiring rapid run-time reconfiguration, which may reduce the performance slightly, cause increased power dissipation and increase system complexity. Implementations (b) and (f) are almost as good and have better system characteristics in that they use arrays that are more generally useful.

Table 1. Performance and Area Results

Implementation (Synchronization window size)	Performance Msamples/sec	Area (M λ^2)
(a) Single, homogenous array (+/- 32)	75x4 = 300	3485
(b) Single, homogenous array (+/- 4)	75x4 = 300	1413
(d) Shared, homogenous array (+/- 8)	75x4 = 300	1055
(e1) Two, homogenous arrays (+/- 32)	75x4 = 300	3554
(e2) Two, heterogeneous arrays (+/- 32)	75x4 = 300	2184
(f1) Two, homogeneous arrays (+/- 4)	75x4 = 300	1482
(f2) Two, heterogeneous arrays (+/- 4)	75x4 = 300	1264
(f3) Two, heterogeneous arrays (+/- 8)	75x4 = 300	1513
TI C6203 DSP, 300MHz 3 antennas, tracking Correlator (+/- 4)	8x3 = 24	750
Standard-cell ASIC, 600 MHz (+/- 32)	150x4 = 600	1020
Custom ASIC, 600MHz (+/- 32)	150x4 = 600	490
FPGA, 100MHz (+/- 32)	25x4 = 100	19,920 (2938 CLBs)

DSP - We chose a TI C6203 DSP [10] running at 300MHz for comparison. This DSP is based on a VLIW architecture with 6 ALUs and 2 multipliers. This DSP can execute the FFT for 3 antennas with tracking synchronization at a sample rate of 8MHz. For area, we used the published die size of 14.89 mm² in a .18 μ 5-level metal CMOS technology. In the absence of a die photo, we estimated the core of this DSP to be about 40% of the total die area, which corresponds to approximately 750 M λ^2 . This 40% excludes the pads and some of the memory that would not be used with a DSP component implementation.

FPGA - We implemented the OFDM receiver using the same algorithm we used for the RaPiD array and mapped it to the Virtex2 architecture. The FPGA implementa-

tion uses the same number of multipliers and ALUs since there are no constant multiplications besides the +1/-1 multiplications in the correlator. We used CLB-based memories instead of block RAMs because they are a more efficient implementation of the small memories used by this algorithm. We did not use the multipliers in the Virtex2 architecture since we wanted to compare to traditional fine-grained FPGA architectures. Using these multipliers would have reduced the area by less than 10%. We estimated the area of the FPGA implementation by mapping the circuit to Virtex2 CLBs. We have estimated the size of a Virtex2 CLB at $6.78M\lambda^2$ using a published die photo for the Virtex2 1000 [11], which is implemented in a $.15\mu$ 8-level metal CMOS technology, and the architecture description of the Virtex2. We estimate the clock rate of the Virtex2 FPGA implementation for this application, normalized to $.18\mu$ technology, at 100 MHz.

ASIC - We estimated the area of an ASIC implementation using two methods, one for a custom implementation, and one for a standard cell implementation. First, we started with the algorithm implemented used with RaPiD and added up the area of the custom function units and memories used by this implementation. We then multiplied this by a factor of 1.5 to account for interconnect, registers and control to get a conservative area estimate for a custom layout. The second method used the equivalent gate count given by the Xilinx tools for this implementation, and converted this to ASIC area using Toshiba's published density for $.18\mu$ technology of 125,000 gate/mm², and a typical 70% packing factor. We estimated the ASIC clock rate at 600MHz.

4 Conclusions

The results of the experimental OFDM implementation clearly show that there is a role for coarse-grained configurable architectures. The RaPiD implementation has about 6 times the cost-performance of a DSP implementation, while an ASIC has about 7 times the cost-performance of RaPiD. Thus RaPiD falls right between the programmable and ASIC alternatives. Finally, the RaPiD implementation has about 15 times the cost-performance of the FPGA implementation, demonstrating the advantage of a specializing a configurable architecture to a problem domain. It is important to remember, of course, that DSPs and FPGAs have much more flexibility than RaPiD, which trades some flexibility for higher performance and lower cost.

Future system-on-chip platforms will have to provide components that have both high-performance and programmability. Processors and ASICs can provide only one, and FPGAs are ruled out for reasons of cost and power except for a very narrow range of bit-oriented applications. The question is whether coarse-grained configurable architectures will really be able to fill the gap as indicated by our research.

We believe that coarse-grained configurable architectures will become increasingly important as new very-high NRE technologies move the world away from ASICs and towards platforms. High-end FPGAs, which are beginning to take on a distinct platform look, now incorporate specialized coarse-grained components like multipliers, ALUs, and memories in addition to processor cores. It is still not clear what type of coarse-grained architecture will be the most effective. RaPiD achieves very high performance via highly pipelined datapaths and is appropriate for many of the intensive

computations that must be offloaded to a hardware implementation. However, there is still much research to be done to explore this architecture space. The programming and compiler tools required to do this exploration are either non-existent or relatively primitive. Our research is now focused on building a set of architecture-independent programming and compiler tools for coarse-grained configurable architectures.

References

- [1] C. Ebeling, D.C. Cronquist, P. Franklin, J. Secosky and S.G. Berg. "Mapping applications to the RaPiD configurable architecture," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines* pp. 106-115, 1997.
- [2] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg. "Mapping Applications to the RaPiD Configurable Architecture," *Field-Programmable Custom Computing Machines 1997*
- [3] C. Fisher, K. Rennie, G. Xing, S. Berg, K. Bolding, J. Naegle, D. Parshall, D. Portnov, A. Sulejmanpasic, and C. Ebeling. "An Emulator for Exploring RaPiD Configurable Computing Architectures," In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications* (FPL 2001), Belfast, pp. 17-26, August, 2001
- [4] D. Cronquist, C. Fisher, M. Figueroa, P. Franklin and C. Ebeling. "Architecture design of reconfigurable pipelined datapaths," *Proceedings of the Conference on Advanced Research in VLSI* pp. 23-40, 1999.
- [5] D. Cronquist, P. Franklin, S.G. Berg and C. Ebeling. "Specifying and Compiling Applications for RaPiD," *IEEE Symposium on FPGAs for Custom Computing Machines* pp. 116-125, 1998.
- [6] Richard Van Nee and Ramjee Prasad, *OFDM for Wireless Multimedia Communications*, Artech, 1999
- [7] Guanbin Xing, Hui Liu and Shi. R, "A multichannel MC-CDMA demodulator and its implementation", in *Proc. Asilomar '99*
- [8] D. Cronquist. "Reconfigurable Pipelined Datapaths", Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1999.
- [9] B. Gold and T. Bially, "Parallelism in Fast Fourier Transform Hardware", *IEEE Trans. on Audio and Electroacoustics*, 21:5-16, Feb. 1985.
- [10] Texas Instruments Web Page, focus.ti.com/docs/military/catalog/general/general.jhtml?templateId=5603&path=templatedata/cm/milgeneral/data/dsp_die_rev
- [11] Chipworks, www.chipworks.com/Reports/Flyers/Xilinx_XC2V1000.htm

Symbol Timing Synchronization in FPGA-Based Software Radios: Application to DVB-S

Francisco Cardells-Tormo¹, Javier Valls-Coquillat², and Vicenc Almenar-Terre³

¹ Inkjet Commercial Division (ICD) R&D Lab, Hewlett-Packard,
08190 Sant Cugat del Valles, Barcelona, Spain,
francisco.cardells@hp.com

² Department of Electronic Engineering, Polytechnic University of Valencia (UPV),
Camino de Vera s/n, 46022 Valencia, Spain,
jvalls@eln.upv.es

³ Department of Telecommunications, Polytechnic University of Valencia (UPV),
Camino de Vera s/n, 46022 Valencia, Spain,
valmenar@dcom.upv.es

Abstract. The design of all-digital symbol timing synchronizers for FPGAs is a complex task. There are several architectures available for VLSI wireless transceivers but porting them to a software defined radio (SDR) platform is not straightforward. In this paper we report a receiver architecture prepared to support demanding protocols such as satellite digital video broadcast (DVB-S). In addition, we report hardware implementation and area utilization estimation. Finally we present implementation results of a DVB-S digital receiver on a Virtex-II Pro FPGA.

1 Introduction

Software-defined radio (SDR) enables the consumer to roam from a wireless standard to another in a seamless and transparent way. A hardware capable of supporting SDR must have: flexibility and extensibility (to accommodate various physical layer formats and network protocols), high speed of computation (to support broadband protocols) and low power consumption (to be mobile). With the advent of 90 nm and beyond processes, application specific integrated circuits (ASICs) are becoming too expensive to miss a single market. Due to this fact, field-programmable gate arrays (FPGAs) are the hardware platform of choice for SDR: FPGAs can quickly be adapted to new emerging standards or to cope with the last minute changes of the specifications. In particular, this work will focus on a certain subset of FPGA architectures: look-up table (LUT)-based FPGAs. We have targeted commercial devices belonging to this set: Xilinx Virtex-II Pro. Area estimations are to be done in logic cells (LCs), consisting of a 4-input LUT (4-LUT) and a storage element. According to this definition, Xilinx slices consist of two LCs.

As we have previously stated, SDR must support all kinds of wireless physical layers in a flexible way. In transmission, the physical layer must perform

data-to-symbol encoding and carrier modulation, whereas in reception the operations are reversed to recover the data stream. Yet, the recovery process is not straightforward due to the fact that we must cope with signal distortions due to the transmission channel. Therefore the demodulator must be able to estimate the carrier phase shift and the symbol timing delay incurred; and ultimately it must correct those effects. Otherwise, symbols would not correctly be recovered, data detection would fail and in the end, we would obtain an unacceptable high bit-error rate (BER). In this paper we will focus on the latter distortion. Due to the fact that in reception symbols are not aligned with the receiver clock edges, we must know when symbols begin and end to achieve symbol lock (i.e., symbol synchronization). In addition to channel effects, the analog signal is sampled in the digital-to-analog converter (DAC) with a clock independent of the symbol clock, and this must be compensated too. Symbol lock can be achieved in many ways, they all can be found in the literature [1].

The goal of this paper is to report the most suitable architecture of a receiver on FPGAs and to prove the feasibility (in terms of area and clock rates) of performing timing recovery on FPGAs. Although we will base our discussion on a particular standard, the results and ideas found here can be extended to other designs and protocols. Our standard of choice is the satellite digital video broadcasting (DVB-S), an European protocol for satellite communications [2]. The features of this standard are the following: modulation is based on quaternary phase shift keying (QPSK) with absolute mapping (no differential encoding); satellite ground stations could use an intermediate frequency (IF) of 70 MHz; it requires a symbol rate between 20.3 and 42.2 Mbauds (for a satellite transponder bandwidth-symbol rate ratio (BW/R_s) of 1.28).

The structure of the paper is as follows. First we will present the receiver architecture suitable for FPGAs. Secondly we will give implementation details of each block and area estimations. Thirdly, we will report implementation results on Virtex-II Pro FPGAs. Finally, we will present the conclusions of this paper.

2 FPGA-Based Architecture of SDR Receivers

The architecture of an FPGA-based receiver is depicted in figure 1. We have accommodated each functional block of the receiver in its appropriate clock domain according to their throughput requirements. We have chosen to decouple the functionality as much as possible to be able to do this partitioning: the resulting architecture corresponds to a non-data aided synchronizer. Otherwise, we could have used the output of the symbol detector for producing a timing estimate (i.e., a decision-directed architecture).

First of all, $r(t)$ (i.e., the bandlimited signal coming from the RF downconverter) is sampled in the ADC every T'_s seconds. Secondly, the samples $r(mT'_s)$ are down-converted from IF to baseband and then low-pass filtered. The output is downsampled to provide a data rate $1/T_s$ slightly higher than two samples per symbol (e.g., 100 MHz). There are two ways to perform down-conversion. The first technique, covered in detail in [3], multiplies the signal by a carrier

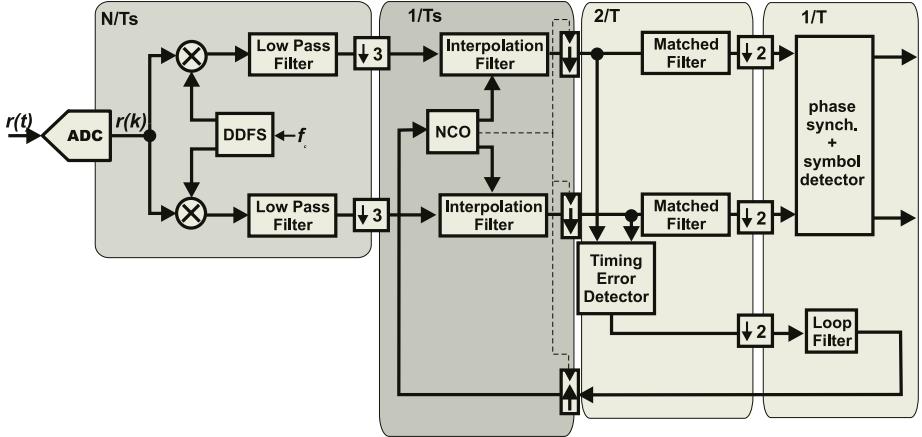


Fig. 1. Receiver architecture: downconverter, timing synchronization loop, matched filter, symbol detector.

centered in IF. Logic must be run at 1.25 times the Nyquist frequency (i.e., at least 232 MHz). In this first stage there are no feed-back loops and therefore pipelining can be used. In addition the output signal must be downsampled to $1/T_s$ using an integer ratio (N), in particular for $N=3$ then $1/T'_s$ would become 300 MHz. The second technique is based on subsampling, it can be applied to low transponder bandwidths but it will not be considered in this discussion.

Thirdly, we have the symbol timing synchronizer. It works at a data rate slightly higher than two samples per symbol (i.e., $1/T_s = 100 \text{ MHz}$). Its working principle can be seen in figure 2. The sampling instants, mT_s , of the incoming data, $x(mT_s)$, are fixed by the ADC sampling clock. New data, $y(kT_i)$, is only generated for the instants kT_i , with T_i the interpolator clock period which is an integer fraction of the symbol period T (e.g. $T_i = T/2$). The ADC sampling rate is asynchronous with the symbol clock (i.e., T/T_s is irrational). Figure 2 shows that the sampling clock and the symbol clock are accommodated using a delayed version of the nearest sample.

$$y(k \cdot T_i) = x((m_k + \mu_k) \cdot T_s) \quad (1)$$

The time delay is defined as $\tau_k = \mu_k \cdot T_s$, where μ_k is the fractional delay. The degree of quantization of μ determines the resolution in the time axis. Besides, we would like to compensate the fact that the receiver and the transmitter clock are not in phase (i.e., they not transition in the same instants). Consequently, the time delay should be slightly shifted from the position shown in figure 2. This correction is achieved using a combinational loop. The error is estimated solely from $y(kT_i)$ and by the timing error detector (TED).

Finally, we have one synchronized sample per symbol. But we must still correct any carrier frequency or phase errors and detect the symbol. This block must have a combinational loop for error detection. Guidelines for the design of

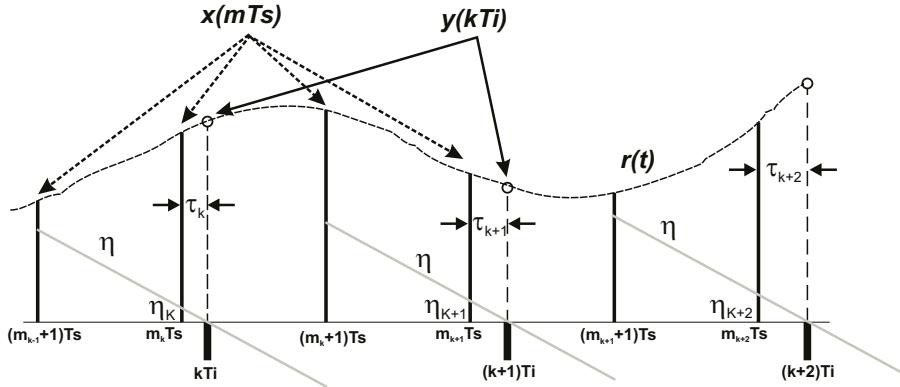


Fig. 2. Sample time relations

this block can be found in [4] and [5]. Current $0.13\mu m$ FPGA technology (e.g., Virtex II Pro) supports bit-rates up to 86 Mbps for DVB-S.

3 Mapping the Symbol Synchronizer in Logic Cells

In this section we will cover the implementation details of each module the symbol synchronizer consists of. Except for the TED, all modules are built in the same way in any all-digital symbol synchronization system, thus making the following discussion fully extensible. In addition to the hardware description, we will estimate area requirements in terms of LCs for our target FPGA. In order to do this in an effective way, we have reduced the design space to a smaller subset by only allowing to set a few parameters. In particular, we can only adjust the wordlength of the signals in table 1. The rest of the signals involved in the design will be affected by those parameters, and they can be found in figures 3, 4, 5, 6 and 7. For our particular case study, we will also use the default values depicted in table 1.

Table 1. Design Parameters.

Parameter	Value	Description
C1	14	Input data wordlength
C2	14	TED output wordlength
C3	28	Loop IIR Filter inner wordlength
C4	4	Fractional delay resolution

3.1 Interpolator

The task of the interpolator is to compute the value of the y signal at the time the interpolator clock transitions (i.e., intermediate values between signal samples $x(m \cdot T_s)$). This can be done using a fractional delay filter [6]. In this paper we will focus on FIR filter implementations [7–9]. The FPGA-implementation of a timing synchronizer using IIR filters can be found in [10]. We must place an interpolation filter in each arm. The number of taps is reduced to four for our normalized bandwidth [9].

$$x[(m_k + \mu_k)T_s] = \sum_{n=-I_1}^{I_2} x[(m_k - n)T_s] \cdot h_n(\mu_k) \quad (2)$$

The fractional delay is variable, and so are the coefficients of the interpolator filter. There are two ways of generating the coefficients, either we pre-compute their values and we store them in a ROM, or we compute them on-the-fly. The two approaches will be discussed next.

ROM-Based Architecture. The direct implementation of equation 2 provides the architecture depicted in figure 3. The filter could not be implemented in its transposed form because we are not using constant coefficients. The filter coefficients can be pre-computed in many ways such as by windowing the impulse response of the ideal interpolator, using optimality criterions and so forth.

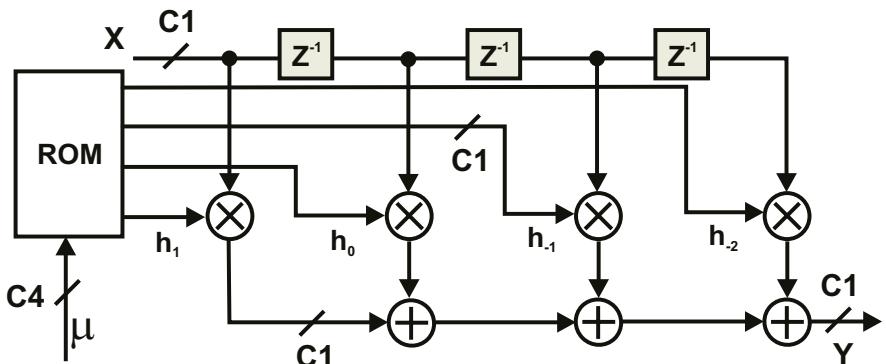


Fig. 3. Fractional-delay FIR filter using for ROM-based coefficients

The area estimation for the interpolator in a Virtex-II Pro (considering the I and Q datapaths) in LCs is

$$\text{Area}(C1, C4) = \left(4 \cdot C1 \cdot \frac{2^{C4}}{15} \right) + 8 \cdot C1^2 + 12 \cdot C1 \quad (3)$$

The ROM-based interpolator requires 114 slices and 8 embedded multipliers.

Farrow Architecture. If we use a polynomial interpolation, then coefficients could be stored in a memory as previously discussed or they could be generated on the fly. The 4 intermediate points can be interpolated by a Lagrange polynomial. By reordering of equation 2 as explained in [9] in chapter 9, we obtain the architecture depicted in figure 4. Each filter bank is a constant coefficient FIR filter and therefore we can use its transposed form and shorten the combinational path. The fact that coefficients are constant also helps with the implementation of multipliers for they can be optimized using canonic signed digit code (CSDC).

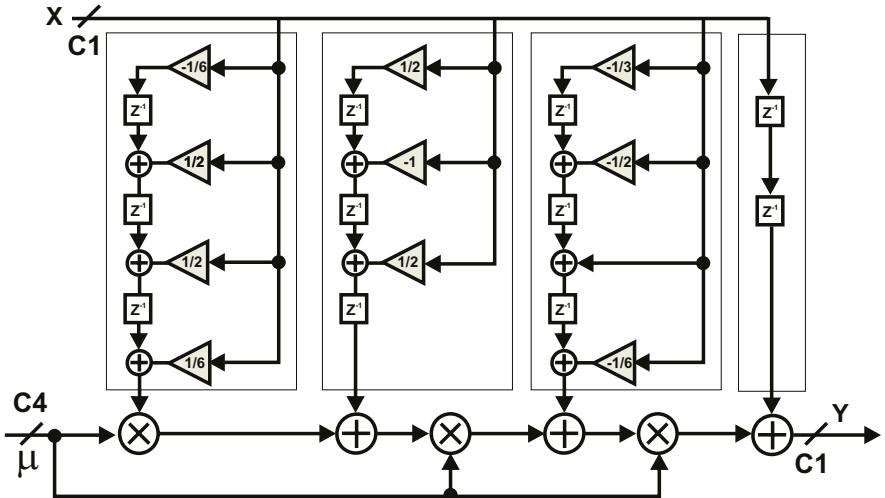


Fig. 4. Fractional-delay FIR filter using the Farrow scheme and Lagrange interpolation

The area estimation for the interpolator in Virtex-II LCs (taking into account that there are two arms) is

$$\text{Area}(C1, C4) = 8 \cdot C1 \cdot C4 + 26 \cdot C1 \quad (4)$$

Although the Farrow interpolator requires fewer logic resources than the ROM-based interpolator, the biggest disadvantage of this technique is the long critical path. The combinational delay is approximately two times longer. Indeed, the ROM-based interpolator has 5 logic levels (an embedded multiplier and four adders), while the Farrow interpolator has 5 logic levels (four adders and three embedded multipliers) plus 3 logic levels in the multiplication with a fixed coefficient.

Polyphase Filters. Although we have only covered interpolation filters, there are other options for performing interpolation such as polyphase filters [11]. A polyphase filter bank consists of M matched filters operating in parallel, the symbol timing synchronizer selects a sample from one filter each cycle. We have

not considered this option in our architecture due to the fact that this architecture is not suitable in FPGAs. The filter bank works at a clock rate as high as M times twice the symbol rate, taking into account that M defines the time precision and should be made as high as possible, it is undeniably true that for high symbol rates the required clock frequency for the banks is not achievable in FPGAs.

3.2 Timing Error Detector

In this paper we have focused on non-data-aided TEDs, thus the estimation will be performed using interpolated samples. The method we will use is the discrete form of the maximum likelihood timing synchronization: the so-called early-late gate detector. This technique consists in finding the point where the slope of the interpolator filter output is zero. If the current timing estimation is too early, then the slope of the interpolator filter output is positive indicating that the timing phase should be advanced. If the current timing estimate is too late, then the slope of the interpolator filter output is negative indicating that the timing phase should be retarded.

The slope is calculated using a derivative. If it is approximated with a finite difference, the TED for QPSK is performed using the following expression [12] (chapter 8):

$$e(k) = \operatorname{Re} \{y^* (k \cdot T + \hat{\varepsilon}_k) \cdot [y((k-1) \cdot T + T/2 + \hat{\varepsilon}_{k-1}) - y(k \cdot T + T/2 + \hat{\varepsilon}_k)]\} \quad (5)$$

Yet this approximation requires sampling at twice the symbol rate. That is the reason why the symbol timing synchronizer uses this data rate. In figure 5 we depict the hardware implementation of the TED. The output of both TEDs is added before being sent to the loop filter. This TED requires 77 slices and 2 multipliers.

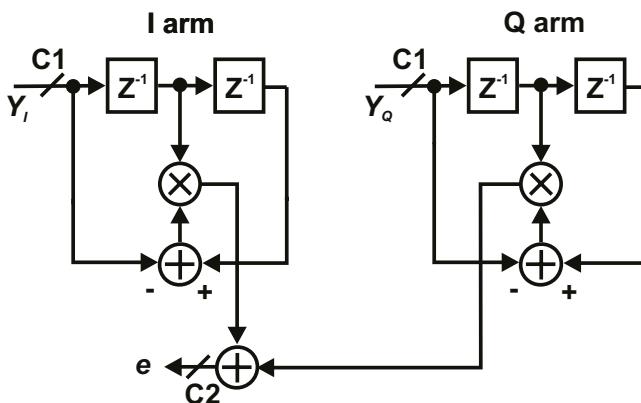


Fig. 5. Timing Error Detector

3.3 Loop Filter

The loop filter implements a standard proportional-integral control as it can be seen in figure 6. Using Virtex II embedded multipliers this module requires 28 slices and 2 multipliers.

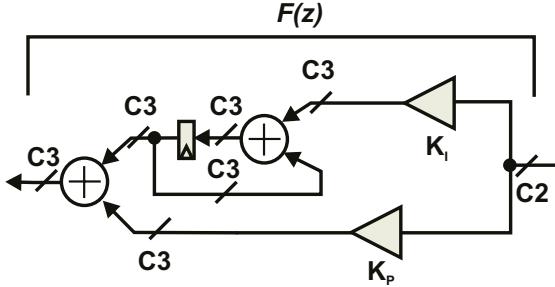


Fig. 6. Loop Filter

3.4 Controller

The controller consists of two timers driven by a clock of frequency of $1/T_s$. In steady-state one timer underflows at the symbol rate while the other underflows at twice the symbol rate. On rollover, the former latches the fractional delay (i.e., the loop filter output), while the latter latches the interpolator output (i.e., the timing error detector input). Therefore a fractional delay value is provided to the interpolator with T -periodicity, and the interpolator outputs two samples per symbol. Timers consists of module-1 counters that are decremented by their nominal period values divided by the clock period as in figure 7. In addition both timers compensate the nominal cycle with a slight shift depending on the timing error-detector output.

In addition to the necessary control signals, the timer with symbol periodicity, generates the fractional interval in the cycle previous to rollover. In figure 2, the relationship between the counter value μ in the cycle previous to rollover and the time delay can be seen. The following equation provides the mathematical formulation as in [1]:

$$\mu_k = \frac{\eta(m_k)}{\eta(m_{k+1}) - 1 + \eta(m_k)} \approx \left(\frac{T_i}{T_s} \right) \cdot \eta(m_k) \quad (6)$$

The two counters are implemented using 56 slices and one extra multiplier to obtain the fractional delay.

4 FPGA-Implementation

The proposed architecture for the symbol timing synchronizer has been fully described in VHDL. We have implemented the design of the synchronizer in a

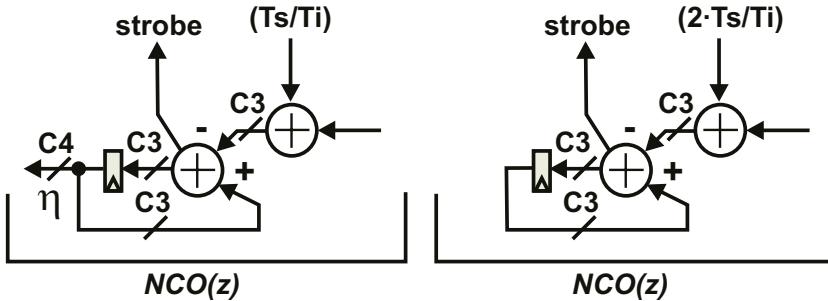


Fig. 7. Receiver architecture showing the timing synchronization loop

Xilinx Virtex-II Pro FPGA. We have used Synplify-Pro 7.1 as synthesis tool and FPGA-vendor design tools, Xilinx ISE 5.1., for place and route. We report the implementation results in table 2.

Table 2. Implementation results of a DVB-S receiver on Virtex-II Pro, speed grade 7.

Receiver module	FPGA Clock rate	Implementation	
		Slices	Multipliers
Down-converter	329 MHz	325	2
Symbol Timing Synchronizer	124 MHz	370	13
Symbol detector	43 MHz	134	6

According to this paper and to [3, 4], the overall area requirements for a DVB-S receiver are: 829 slices and 21 multiplier blocks. We also report the maximum clock rate of each module in the receiver. It can be verified that we are able to run the system fast enough to process DVB-S satellite data. Indeed, the down-converter can be run faster than the Nyquist frequency. The symbol timing synchronizer can be run at twice the maximum symbol rate, i.e. 84 MHz. The symbol detector can be run at the maximum symbol required by DVB-S.

5 Conclusions

In this paper we have presented an SDR receiver architecture adapted for FPGAs. It is based on decoupling carrier phase and symbol timing recovery loops and on performing the functionality in the most suitable clock domain. We have discussed the mapping non-data aided timing synchronizers on FPGAs, and we have found out that FIR ROM-based interpolation filters are the right choice. We have proved that we can meet the timing requirements and that whole DVB-S receiver can be mapped in Virtex-II Pro FPGAs.

Acknowledgements

This work was supported by the Spanish “Ministerio de Ciencia y Tecnología” under grant number “TIC2001-2688-C03-02” and “TIC2001-2688-C03-02”. Francisco Cardells-Tormo acknowledges the support of Hewlett-Packard, Barcelona R&D Lab in the presentation of these results. The opinions expressed by the authors are theirs alone and they do not represent the opinions of HP.

References

1. Gardner, F.M.: Interpolation in Digital Modems – Part I: Fundamentals. *IEEE Transactions on Communications* **41** (1993) 501–507
2. E.T.S.I.: Digital Video Broadcasting (DVB). framing structure, channel coding and modulation for 11/12 Ghz satellite services. European Standard EN 300 421 (1997)
3. Cardells-Tormo, F., Valls-Coquillat, J.: High Performance Quadrature Digital Mixers for FPGAs. In: 12th International Conference on Field Programmable Logic and Applications (FPL2002). (2002) 905–914
4. Cardells-Tormo, F., Valls-Coquillat, J., Almenar-Terre, V., Torres-Carot, V.: Efficient FPGA-based QPSK Demodulation Loops: Application to the DVB Standard. In: 12th International Conference on Field Programmable Logic and Applications (FPL2002). (2002) 102–111
5. Dick, C., Harris, F., Rice, M.: Synchronization in software radios - carrier and timing recovery using FPGAs. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2000)
6. Laakso, T.I., Valimaki, V., Karjalainen, M., Laine, U.K.: Splitting the Unit Delay. *IEEE Signal Processing Magazine* **13** (1996) 30–60
7. Erup, L., Gardner, F.M., Harris, R.A.: Interpolation in Digital Modems – Part II: Implementation and Performance. *IEEE Transactions on Communications* **41** (1993) 998–1008
8. Kootsookos, P.J., Williamson, R.C.: FIR Approximation of Fractional Sample Delay Systems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* **43** (1996) 269–271
9. Meyr, H., Moeneclaey, M., Fechtel, S.A.: Digital Communication Receivers. Synchronization, Channel Estimation and Signal Processing. John Wiley & Sons, (New York)
10. Wu, Y.C., Ng, T.S.: FPGA Implementation of Digital Timing recovery in Software Radio Receiver. In: IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS 2000). (2000) 703–707
11. Harris, F.J., Rice, M.: Multirate Digital Filters for Symbol Timing Synchronization in Software Defined Radios. *IEEE Journal on Selected Areas in Communications* **19** (2001) 2346–2357
12. Mengali, U., D'Andrea, A.N.: Synchronization Techniques for Digital Receivers. Plenum Press, (New York and London)

An Algorithm Designer’s Workbench for Platform FPGAs[∗]

Sumit Mohanty and Viktor K. Prasanna

Electrical Engineering Systems,
University of Southern California, CA, USA,
`{smohanty,prasanna}@usc.edu`

Abstract. Growing gate density, availability of embedded multipliers and memory, and integration of traditional processors are some of the key advantages of Platform FPGAs. Such FPGAs are attractive for implementing compute intensive signal processing kernels used in wired as well as wireless mobile devices. However, algorithm design using Platform FPGAs, with energy dissipation as an additional performance metric for mobile devices, poses significant challenges. In this paper, we propose an algorithm designer’s workbench that addresses the above issues. The workbench supports formal modeling of the signal processing kernels, evaluation of latency, energy, and area of a design, and performance tradeoff analysis to facilitate optimization. The workbench includes a high-level estimator for rapid performance estimation and widely used low-level simulators for detailed simulation. Features include a confidence interval based technique for accurate power estimation and facility to store algorithm designs as library of models for reuse. We demonstrate the use of the workbench through design of matrix multiplication algorithm for Xilinx Virtex-II Pro.

1 Introduction

High processing power, programmability, and availability of a processor for control intensive tasks are some of the unique advantages for Platform FPGAs which integrate traditional processors into the FPGA fabric [13]. Therefore, Platform FPGAs are attractive for implementing complex and compute intensive applications used in wired and wireless devices [13]. Adaptive beam forming, multi-rate filters and wavelets, software defined radio, image processing, etc. are some of the applications that target reconfigurable devices [11]. Efficient design of an application requires efficient implementation of constituent kernels. In this paper, kernel refers to a signal processing kernel such as matrix multiplication, FFT, DFT, etc. Algorithm design for a kernel refers to the design of a datapath and the data and control flow that implements the kernel.

Even though FPGA based systems are not designed for low-power implementations, it has been shown that energy-efficient implementation of signal

[∗] This work is supported by the DARPA Power Aware Computing and Communication Program under contract F33615-C-00-1633.

processing kernels is feasible using algorithmic techniques [3], [5]. However, the major obstacle to the widespread use of Platform FPGAs is the lack of high-level design methodologies and tools [7]. Moreover with energy dissipation as an additional performance metric for the mobile devices, algorithm design using such FPGAs is difficult. In this paper, we discuss the design of a workbench for the algorithm designers based on the domain specific modeling technique [3] that addresses the above issues. A *workbench* in the context of algorithm design refers to a set of tools that aid an algorithm designer in designing energy-efficient algorithms. Although the workbench supports algorithm design for any kernel, we primarily target kernels that can be implemented using regular loop structures enabling maximum exploitation of hardware parallelism.

The workbench enables parameterized modeling of the datapath and the algorithm. Modeling using our workbench follows a hybrid approach, which starts with a top-down analysis and modeling of the datapath and the corresponding algorithm followed by analytical formulation of cost functions for various performance metrics such as energy, area, and latency. The constants in these functions are estimated through a bottom-up approach that involves profiling individual datapath components through low-level simulations. The model parameters can be varied to understand performance tradeoffs. The algorithm designer uses the tools integrated in the workbench to estimate performance, analyze the effect of parameter variation on performance, and identify optimization opportunities. Our focus in designing the workbench is not to develop new techniques for compilation of high-level specifications onto FPGAs, rather to formalize some of the design steps such as modeling, tradeoff analysis, estimation of various performance metrics using the available solutions for simulation and synthesis.

The workbench facilitates both high-level estimation and low-level simulation. High-level refers to the level of abstraction where performance models of the algorithm and the architecture can be defined in terms of parameters and cost functions. In contrast, low-level refers to the level of modeling suitable for cycle-accurate or RT-level simulation and analysis. The workbench is developed using Generic Modeling Environment, GME (GME 3), a graphical tool-suite, that enables development of a modeling language for a domain, provides graphical interface to model specific problem instances for the domain, and facilitates integration of tools that can be driven through the models [4].

The following section discusses some related work. Section 3 describes our algorithm design methodology. Section 4 discusses the design of the workbench. Section 5 demonstrates the use of the workbench. We conclude in Section 6.

2 Related Work

Several tools are available for efficient application design using FPGAs. Simulink and Xilinx System Generator provide a high-level interface to design applications using pre-compiled libraries of signal processing kernels [12]. Other tools such as [6] and [13] provide integrated design environments that accept high-level specification as VHDL and Verilog scripts, schematics, finite state machines, etc. and provide simulation, testing, and debugging support for the designer to

implement a design using FPGAs. However, these tools start with a single conceptual design. Design space exploration is performed as part of implementation or through local optimizations to address performance bottlenecks identified during synthesis. For Platform FPGAs, the ongoing design automation efforts use available support for design synthesis onto the FPGA and processor and focus on communication synthesis through shared memory. However, a high-level programming model suitable for algorithmic analysis still remains to be addressed.

Several C/C++ language based approaches such as SystemC, Handel-C, SpecC are primarily aimed at making the C language usable for hardware and software design and allow efficient and early simulation, synthesis, and/or verification [1], [2]. However, these approaches do not facilitate modeling of kernels at a level of abstraction that enables algorithmic analysis. Additionally, generating source code and going through the complete synthesis process to evaluate each algorithm decision is time consuming.

In contrast, our approach enables high-level parameterized modeling for rapid performance estimation and efficient tradeoff analysis. Using our approach, the algorithm designer does not have to synthesize the design to verify design decisions. Once a model has been defined and parameters have been estimated, design decisions are verified using the high-level performance estimator. Additionally, parameterized modeling enables exploration of a large design space in the initial stages of the design process and hence is more efficient when the final design needs to meet strict latency, energy, or area requirements.

3 Algorithm Design Methodology

Our algorithm design methodology is based on domain specific modeling, a technique for high-level modeling of FPGAs, developed by Choi et al. [3]. This technique has been demonstrated successfully for designing energy efficient signal processing kernels [3], [5]. A domain, in the context of domain specific modeling, refers to a class of architectures such as uniprocessor, linear array, etc. and the corresponding algorithm that implements a specific kernel [3].

In our methodology, the designer initially creates a model using the domain specific modeling technique for the kernel for which an algorithm is being designed. This model consists of RModules, Interconnects, component specific parameters and power functions, component power state matrices, and a system-wide energy function. We have extended the model to include functions for latency and area as well. A *Relocatable Module (RModule)* is a high-level architecture abstraction of a computation or storage module. *Interconnect* represents the resources used for data transfer between the RModules. A component (building block) can be a RModule or an Interconnect. *Component Power State (CPS) matrices* capture the power state for all the components in each cycle.

Parameter estimation refers to the estimation of area and power functions for the components. Power and area functions capture the effect of component specific parameters on the average power dissipation and area of the component respectively. Latency is implicit in the algorithm specification and is also 0spec-

ified as a function. Ultimately, component specific area and power functions and the latency function are used to derive system-wide (for the kernel) energy, area, and latency functions.

Following parameter estimation, the designs are analyzed for performance tradeoffs as follows: a) each component specific parameter is varied to observe the effect on different performance metrics, b) if there exists alternatives for a building block then each alternate is evaluated for performance tradeoffs, and c) fraction of total energy dissipated by each type of component is evaluated to identify candidates for energy optimization. Based on the above analysis, the algorithm designer identifies the optimizations to be performed.

The workbench is used to assist designing using this methodology. Various supports include a graphical modeling environment based on domain specific modeling, integration of widely used simulators, curve-fitting for cost function estimation, and tradeoff analysis. In addition, the workbench integrates a high-level estimator, kernel performance estimator, that rapidly estimates latency, energy, and area for a kernel to enable efficient tradeoff analysis. The workbench also supports storage of various components as a library for reuse and a confidence interval based technique for statistically accurate power estimation.

4 Algorithm Designer's Workbench

We have used GME to create the modeling environment for the workbench [4]. GME provides support to define a metamodel (modeling paradigm). A metamodel contains syntactic, semantic, and visualization specifications of the target domain. GME configures itself using the metamodel to provide a graphical interface to model specific instances of the domain. GME enables integration of various tools to the modeling environment. Integration, in this context refers, to being able to drive the tools from the modeling environment. In the following, we discuss various aspects of the workbench in detail.

4.1 Modeling

We provide a hierarchical modeling support to model the datapath. The hierarchy consists of three types of components; micro, macro, and library blocks. A micro block is target FPGA specific. For example, as shown in Figure 1 (b), the micro blocks specific to Xilinx Virtex II Pro are LUT, embedded memory cell, I/O Pad, embedded multiplier, and interconnects [13]. In contrast, for Actel ProASIC 500 series of devices, there will be no embedded multiplier. Macro blocks are basic architecture components such as adders, counters, multiplexers, etc. designed using the micro blocks. A library block is an architecture component that is used by some instance of the target class of architectures associated with the domain. For example, if linear array of processing elements (PE) is our target architecture, a PE is a library block. Both macro and library blocks are also referred to as composite blocks. We have developed a basic metamodel using GME that provides templates for different kind of building blocks and associated parameters. Once the target device and the kernel are identified, prior

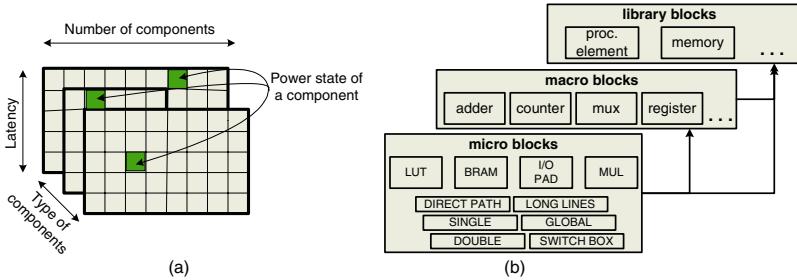


Fig. 1. Component Power State Matrices and Hierarchical Modeling

to using the workbench, the designer needs to modify the basic metamodel to support the specific instances of the building blocks. Being able to modify the modeling paradigm is a considerable advantage over many design environments where the description of a design is limited by the expressive power of the modeling language. Each building block is associated with a set of component specific parameters (Section 3). States is one such parameter which refer to various operating states of each building block. Currently, we model only two states, *ON* and *OFF* for each micro block. In *ON* state the component is active and in *OFF* state it is clock gated. However, for composite blocks it is possible to have more than 2 states due to different combination of states of the constituent micro blocks. Power is always specified as a function with switching activity as the default parameter.

Given an algorithm, Component Power State (CPS) matrices capture the operating state for all the components in each cycle. For example, consider a design that contains k different types of components (C_1, \dots, C_k) with n_i components of type i . If the design has the latency of T cycles, then k two dimensional matrices are constructed where the i -th matrix is of size $T \times n_i$ (Figure 1(a)). An entry in a CPS matrix represents the operating state of a component during a specific cycle and is determined by the algorithm.

Modeling based on the technique described above has the following advantages: a) the model separates the low-level simulation and synthesis tools from high-level algorithmic analysis techniques, b) various parameters are exposed at high-level thus enabling algorithm-level tradeoff analysis, c) performance models for energy, area, and latency are generated in the form of parameterized functions which allows rapid performance estimation, and d) using GME's ability to store models in a database, models for various kernels are stored and reused during the design of other kernels that share the same building blocks.

4.2 Performance Estimation

Our workbench supports performance evaluation through rapid high-level estimation as well as low-level simulation using third-party simulators. We have developed a high-level estimation tool, a kernel performance estimator, to es-

timate different performance metrics associated with the algorithm. The input to the kernel performance estimator is the model of the datapath and the CPS matrices associated with the algorithm. The designer has the option of providing switching activity for each building block. The default assumption is a switching activity of 12.5% (default value used by Xilinx XPower [13]).

Area of the design is evaluated as the sum of the individual areas of the building blocks that constitute the datapath. Latency estimate is implicit in the CPS matrices. Energy dissipation is estimated as a function of the CPS matrices and datapath. Overall energy dissipation is modeled as energy dissipated by each component during each cycle over the complete execution period. As energy can be evaluated as $power \times time$, for a given component in a specific cycle, we use the power function for the operating state specified in the CPS matrices and duration of each cycle to evaluate the energy dissipated. Extending the above analysis for each component over the complete execution period, we evaluate the overall energy dissipation for the design. The kernel performance estimator is based on the above technique and is integrated into the workbench. Once a model is defined, it is possible to automatically invoke the estimator to generate the performance estimates and update the model based on these estimates. The estimator operates at a level of abstraction where the domain specific modeling technique is defined. Therefore, the estimator is not affected by the modifications to the basic metamodel as described in Section 4.1.

Low-level simulation is used in our methodology to estimate various component specific parameters. Low-level simulation is supported in the workbench through widely used simulators available for the FPGAs such as ModelSim, Xilinx XPower, Xilinx Power Estimator, etc. [13]. Different simulators have different input requirements. For example, Power Estimator requires a list of different modules used in a design, expected switching activity, area in CLB slices, and frequency of operation to provide a rapid and coarse estimate of average power. In contrast, ModelSim and XPower accept placed and routed design and input waveforms to perform fairly accurate estimation of power and latency. Therefore, we have added capability in our modeling environment to specify VHDL source code and input vectors as files. Derivation of cost function involves simulation of the design at different instances where each instance refers to a combination of parameter values and curve-fitting to generate functions.

Energy dissipation depends on various factors such as voltage, frequency, capacitance, and switching activity. Therefore, simulating once using a random test vector may not produce statistically accurate results. We use confidence intervals to estimate average power and confidence in the estimate to generate statistically significant results. The approach uses results from multiple simulations performed using a set of randomly generated inputs (Gaussian distribution is assumed) and computes the mean of the performance values and the confidence associated with the mean. Given a synthesized design and set of input test vectors, the workbench automatically performs multiple simulations and evaluates the mean and the confidence interval and updates the model using the results.

4.3 Modeling and Simulating the Processors

Kernels that are control intensive are potential candidates for design using both the FPGA and the processor. We assume shared memory communication between the FPGA and the processor. We model computations on the processor as functions. A function is implemented using high-level languages such as C and is compiled and loaded into the instruction memory. It is assumed that the processor is aware of the memory location to read input and write output. Once a function is invoked the FPGA stalls until the results are available.

Functions are specified in the CPS matrices. A function specified in a cell (i, j) refers to invoking the function in the i th cycle. We model each function as time taken and energy dissipated during execution. Therefore, whenever a function is encountered in the CPS matrices, the kernel performance estimator includes the latency and energy values associated with the function to the overall energy or latency estimates. We use the available simulators for the processors to estimate latency and energy values for each function. For example, ModelSim, which uses the SWIFT model for PowerPC, can be used to simulate the processor and estimate latency [13]. We evaluate energy dissipation by scaling the latency value based on the vendor provided estimates. For example, PowerPC on Virtex-II Pro consumes 0.9 mW per MHz. Using this estimate, a program that executes over 9×10^8 cycles will have a latency of 3 seconds and power dissipation of approximately 810 mW when the processor is operating at 300 MHz.

4.4 Design Flow

We briefly describe the design flow for algorithm design using our workbench. We assume that designer has already identified a suitable domain for the kernel to be designed and the target Platform FPGA.

Workbench Configuration (1): In this step, the designer analyzes the kernel to define a domain specific model. However, the designer does not derive any of the functions associated with the model. The designer only identifies the RModules and Interconnects and classifies them as micro, macro, and library blocks and also identifies the associated component specific parameters. Following this, the designer modifies the basic metamodel to configure GME for modeling using the workbench. The building blocks identified for one kernel can also be used for other kernels implemented using the same target FPGA. Therefore, modifications to the metamodel are automatically saved for future usage.

Modeling (2): The model of the datapath is graphically constructed in this step using GME. GME provides the ability to drag and drop modeling constructs and connect them appropriately to specify the structure of the datapath. If appropriate simulators are integrated, the designer can specify high-level scripts for the building blocks to be used in the next step. In addition, CPS matrices for the algorithm are also specified. The workbench facilitates building a library of components to save models of the building blocks and associated performance estimates (see *Step 3*) for reuse during the design of other algorithms.

Parameter Estimation (3): Estimation of the cost functions for power and area involves synthesis of a building block, low-level simulations, and in case of

power, the use of confidence intervals to generate statistically significant power estimates. The simulations are performed off-line or, if required simulator is integrated, automatically using specified high-level scripts. Latency functions are estimated using the CPS matrices. System-wide energy and area functions are estimated using the latency function and component specific power and area functions.

Tradeoff Analysis and Optimization (4): In this step, the designer uses the workbench to understand various performance tradeoffs as described in Section 3. While the workbench facilitates generation of the comparison graphs, the designer is responsible for specific design decisions based on the graphs. Once the design is modified based on the decisions, kernel performance estimator is used for rapid verification. Tradeoff analysis and optimization is performed iteratively till the designer meets the performance goals.

5 Illustrative Example: Energy-Efficient Design of Matrix Multiplication Algorithm

A matrix multiplication algorithm for linear array architectures is proposed in [10]. We use this algorithm to demonstrate modeling, high-level performance estimation, and performance tradeoff analysis capabilities of the workbench. The focus is to generate an energy efficient design for matrix multiply using Xilinx Virtex-II Pro. The matrix multiplication algorithm and the architecture are shown in Figure 2. In this experiment, only FPGA is considered while designing the algorithm.

In step 1, the architecture and the algorithm were analyzed to define the domain specific model. Various building blocks that were identified are register, multiplexer, multiplier, adder, processing element (PE), and interconnects between the PEs. Among these building blocks only the PE is a library block and the rest of the components are micro blocks. Component specific param-

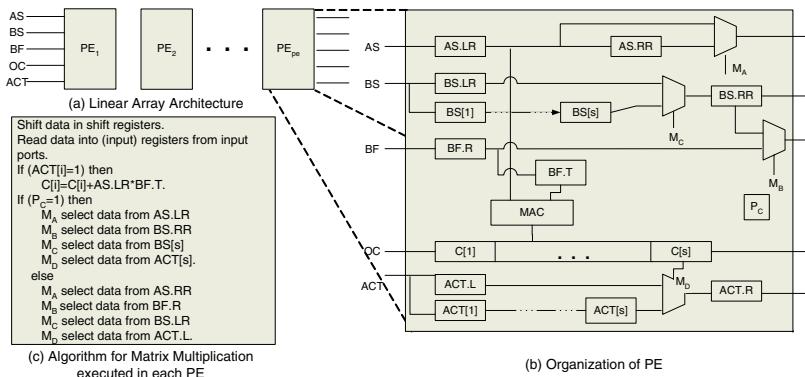


Fig. 2. Architecture and Algorithm for Matrix Multiplication [10]

ters for the PE include number of register (s) and power states *ON* and *OFF*. *ON* refers to the state when the multiplier (within the PE) is in *ON* state and *OFF* refers to the state when the multiplier is in *OFF* state. Additionally, for the complete kernel design number of PEs (pe) is also a parameter. For $N \times N$ matrix multiplication, the range of values for s is $1 \leq s \leq N$ and for pe it is $1 \leq pe \leq N(\lceil N/s \rceil)$. For matrix multiplication with larger size matrices (large values of N) it is not possible to synthesize the required number of PEs due to area constraint. In such cases, block matrix multiplication is used. Therefore, block-size (bs) is also a parameter. Based on the above model, the basic meta-model was enhanced to support modeling of linear array architecture for matrix multiply.

Step 2 involved modeling using the configured GME. Once the datapath was modeled we generated the cost function for power and area for the different components. Switching activity was the only parameter for power functions. To define the CPS matrices, we analyzed the algorithm to identify the operating state of each component in different cycles. As per the algorithm shown in Figure 2 (c), in each PE, the multiplier is in *ON* state for $T/(\lceil n/s \rceil)$ cycles and is in *OFF* state for $T \times (1 - 1/\lceil n/s \rceil)$ cycles [10]. All other components are active for the complete duration.

In *Step 3*, we performed simulations to estimate the power dissipated and area occupied by the building blocks. Currently, we have integrated ModelSim, Xilinx XPower, and Xilinx Power Estimator to the workbench [13]. The latency (T) of this design using $N \lceil N/s \rceil$ PEs and s storage per PE [10] is $T = (N^2 + 2N \lceil N/s \rceil - \lceil N/s \rceil + 1)$. Using the latency function, component specific power functions, and CPS matrices, we derived the system-wide energy function.

Finally, we performed a set of tradeoff analyses to identify suitable optimizations. Figure 3 (a) shows the variation of energy, latency, and area for different block sizes for 16×16 matrix multiplication. It can be observed that energy is minimum at a block size of 4 and area and latency are minimum at block size 2 and 16 respectively. This information is used to identify a suitable design (block size) based on latency, energy, or area requirements. Figure 3 (b) shows energy distribution among multipliers, registers, and I/O pads for three different designs. Design 1 corresponds to the original design described in [10] and Design 2 and 3 are low energy variants discussed in [5]. Using Figure 3, we identify

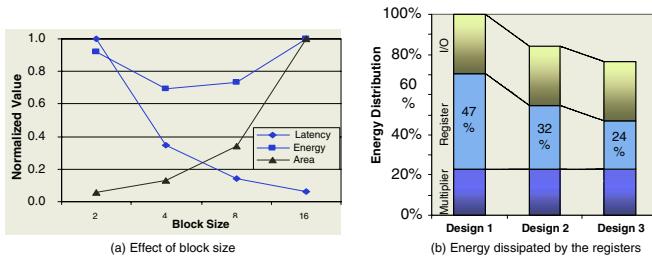


Fig. 3. Analysis of Matrix Multiplication Algorithm

that the registers dissipate the maximum energy and select them as candidates for optimization. Optimizations considered include reduction of number of registers through analysis of data movements (Design 2) and use of CLB based SRAMs instead of registers to reduce energy dissipation (Design 3). Details of the optimized algorithm are available in [5], [9].

Using the workbench, an optimized matrix multiplication algorithm was designed and compared against the Xilinx IP core for matrix multiplication. The best design obtained using our approach achieved 68% reduction energy dissipation, 35% reduction in latency while occupying $2.3 \times$ area when compared with the design provided by Xilinx [9].

6 Conclusion

We discussed an algorithm designer's work bench suitable for a general class of FPGA and Platform FPGA devices. The work discussed in this paper is part of the MILAN project. MILAN addresses the issues related to the design of end-to-end applications [8]. In contrast, the workbench addresses the issues pertaining to the design of application kernels only. The workbench is developed as a stand-alone tool with plans for integration into the MILAN environment.

References

1. Cai, L., Olivarez, M., Kritzinger, P., and Gajski, D: C/C++ Based System Design Flow Using SpecC, VCC, and SystemC. Tech. Report 02-30, UC, Irvine, June 2002.
2. The Handel-C language. <http://www.celoxica.com/>
3. Choi, S., Jang, J., Mohanty, S., and Prasanna, V.K.: Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures. Engineering of Reconfigurable Systems and Algorithms, 2002.
4. Generic Modeling Environment. <http://www.isis.vanderbilt.edu/> Projects/gme/
5. Jang, J., Choi, S., and Prasanna, V.K.: Energy-Efficient Matrix Multiplication on FPGAs. Field Programmable Logic and Applications, 2002.
6. Mentor Graphics FPGA Advantage. <http://www.mentor.com/fpga-advantage/>
7. McGregor, G., Robinson, D., and Lysaght, P.: A Hardware/Software Co-design Environment for Reconfigurable Logic Systems. Field-Programmable Logic and Applications, 1998.
8. Model-based Integrated Simulation. <http://milan.usc.edu/>
9. Mohanty, S., Choi, S., Jang, J., and Prasanna, V.K.: A Model-based Methodology for Application Specific Energy Efficient Data Path Design using FPGAs. Conference on Application-Specific Systems, Architectures and Processors, 2002.
10. Prasanna, V.K. and Tsai, Y.: On Synthesizing Optimal Family of Linear Systolic Arrays for Matrix Multiplication. IEEE Tran. on Computers, Vol. 40, No. 6, 1991.
11. Srivastava, N., Trahan, J., Vaidyanathan, R., and Rai, S.: Adaptive Image Filtering using Run-Time Reconfiguration. Reconfigurable Architectures Workshop, 2003.
12. System Generator for Simulink. http://www.xilinx.com/products/software/sysgen/product_details.htm.
13. Xilinx Virtex-II Pro and Xilinx Embedded Development Kit (EDK). <http://www.xilinx.com/>

Prototyping for the Concurrent Development of an IEEE 802.11 Wireless LAN Chipset

Ludovico de Souza, Philip Ryan, Jason Crawford, Kevin Wong,
Greg Zyner, and Tom McDermott

Cisco Systems, Wireless Networking Business Unit, Sydney, Australia,
ludi@cisco.com

Abstract. This paper describes how an FPGA based prototype environment aided the development of two multi-million gate ASICs: an IEEE 802.11 medium access controller and an IEEE 802.11a/b/g physical layer processor. Prototyping the ASICs on a reconfigurable platform enabled concurrent development by the hardware and software teams, and provided a high degree of confidence in the designs. The capabilities of modern FPGAs and their development tools allowed us to easily and quickly retarget the complex ASICs into FPGAs, enabling us to integrate the prototyping effort into our design flow from the start of the project. The effect was to accelerate the development cycle and generate an ASIC which had been through one pass of beta testing before tape-out.

1 Introduction

The IEEE 802.11 standards [1] describe high-speed wireless local area networks. We have implemented an 802.11 solution as two SoCs, corresponding to the low-level physical communication layer (the PHY), and the high-level medium access control layer (the MAC), as shown in Figure 1.

The complexity and evolving nature of the 802.11 standards make a degree of programmability essential for the MAC. Most MACs achieve this by embedding a processor coupled to hardware acceleration blocks, such as a packet engine or a cryptography module. However, it is difficult a-priori to make good decisions as to what areas require hardware acceleration. The ability to run firmware on real hardware in a real network allows design decisions to be based on real data.

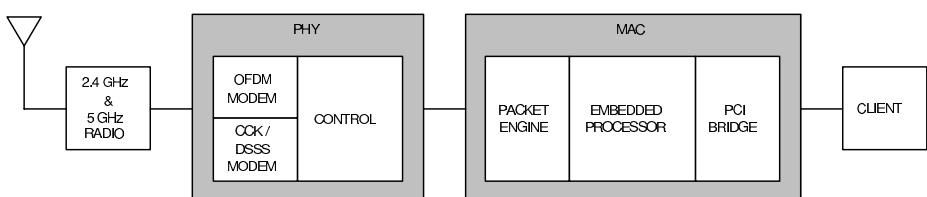


Fig. 1. An 802.11 Wireless LAN, showing the PHY and MAC SoCs.

Conversely, the PHY standard is stable and well understood, and the computational requirements of the PHY require dedicated hardware [2]; but this makes it vulnerable to its own problems. There is little, if any, scope to fix design faults after tape out. Mistakes are expensive, fabrication NRE costs are in the order of a million dollars. Simulation is too slow to exhaustively test the complex design. It is very difficult to envisage and plan for all the ways in which the system might be excited in practice. The vagaries of the medium mean that it is very hard to simulate - the only way to verify is to do it “virtual” real time.

Our FPGA prototype addressed both of these problems. Interfacing the two developing systems with each other and with external entities provided each system with the necessary real-world stimulus to develop and verify its implementation. The ability to execute firmware in a real system environment, months in advance of first silicon, accelerated the development of the product by allowing both hardware and software to be developed simultaneously rather than sequentially, improving confidence and reducing risk in the complete design.

An FPGA based prototype was the best solution to meet our needs and fit in with our existing development process. The capacity and speed of modern FPGAs offered us a near full-speed implementation; a realistic prototype running at one quarter of the final system speed. The maximum benefit of prototyping was achieved by implementing the core functionality of the prototype directly from the ASIC Verilog source, without modification. The pushbutton process from synthesis to bitstream of our multi-million transistor designs was achievable on a time-scale of hours, without user intervention, allowing many design/test/debug iterations per day.

In this paper we share our experience of how modern FPGA tools and hardware allowed us to implement a successful solution, and the positive impact it has had on our ASIC development flow.

2 Requirements

The prototype served three different purposes; as a platform for firmware development, as a testbed, and to interface with real hardware to expose the design to real world conditions. The following sections address these issues.

2.1 MAC Hardware/Firmware Co-development

The attraction of a prototype implementation of the MAC was the ability to execute firmware, running code on the actual platform, months before real silicon was available. When developing software for an embedded processor on a yet to be built SoC, one could make use of simulators, such as the Armulator in the ARM developer suite [3], or one could attempt to integrate a physical processor into the prototype environment, such as using the ARM Integrator Core Module, or using the PowerPC embedded in the Xilinx Virtex-II Pro FPGAs. However our experience is that this is somewhat cumbersome; there is a lack of observability in the implementation, and we are subject to constraints specific to

that implementation, such as differences in the cache configuration. The concern is that we are not using the actual hardware that will be implemented in the final ASIC.

A much more accurate and insightful analysis of the firmware execution is achieved by synthesising the processor soft-core into the prototype, with the rest of the system. Our FPGA implementation did this, implementing the complete MAC SoC within an FPGA, and so conserved the timing relationship between the processor and the rest of the system. An in-house developed RISC processor, a V8 SPARC written in synthesizable behavioural Verilog without any FPGA specific timing optimisation, was used on this project. The FPGA implementation was quite acceptable, synthesizing to over 50MHz while using about 10 percent of the XC2V6000 FPGA.

2.2 PHY Hardware Testability

The motivation for a prototype implementation of the PHY was increased testability; the ability to throw huge amounts of data at a complex design, to run the design.

The PHY ASIC supports the OFDM, CCK, and DSSS modulations used in the 802.11a, 802.11b, and 802.11g physical layers. This entails considerable computational effort, covering operations such as synchronisation and frequency correction, transforms such as the FFT and IFFT, equalisation, Viterbi decoding, interleaving, deinterleaving, scrambling, filtering, and so on. The magnitude of this processing, coupled with a limited power budget, demands a highly optimised data-flow implementation, providing little scope for post-silicon design fixes. Simulation of the complex design is time consuming, having a large number of variables that influence the performance of the system. This means that only a limited test coverage is possible in simulation, which is usually sufficient to highlight critical design faults, but may leave many lesser problems undetected. The real-time nature of the prototype allows for instantaneous feedback and analysis. Experimentation and verification are performed with the system running in a much more realistic environment than could be feasibly achieved in any reasonable time in simulation. A packet, lasting 200 microseconds in real time, corresponds to 2 minutes of simulation time, an incredible speed difference - on the prototype, 5000 packets can be demodulated every second, a task that takes almost 3 hours in simulation.

2.3 Why Emulation Wasn't an Option

A requirement common to the PHY and the MAC was to interface with real hardware. We wanted the prototype to be part of a real system, to expose the design to real world conditions. This meant interfacing the PHY with a radio, in our case an existing 5GHz radio, and interfacing the MAC with a host device, in our case an existing Cisco Access Point.

This doesn't imply that a full speed real time prototype is necessary. Because a full speed prototype was unachievable given the speed of current FPGAs, the

prototype had to be a time scaled implementation. A quarter speed implementation was achieved, meaning that the prototype PHY operates on signals of one quarter the bandwidth used in real systems. This is not a problem for communicating with other prototype PHY implementations, or with other appropriately modified equipment. It does however prevent communication with regular 802.11 systems. We allowed for this by having a mechanism to bypass the FPGA PHY implementation, to interface the MAC with a silicon PHY implementation, to allow the prototype to communicate with any 802.11 system, permitting more extensive MAC level testing.

These requirements demanded a real implementation, not just emulation.

3 Implementation

3.1 Integrating with the ASIC Design Flow

Our group has a standard process for the development of its ASICs. Designs are coded in synthesizable behavioural Verilog, simulated with tools such as Cadence's NC-Verilog [4], and synthesized and implemented as an ASIC using Cadence's physical design tools, for $0.13\mu\text{m}$, $0.18\mu\text{m}$ and $0.25\mu\text{m}$ standard CMOS processes.

We decided to start the prototype implementation flow from the behavioural Verilog code base, the same code used for ASIC synthesis. This allows the FPGA synthesis tool to perform FPGA-centric optimisations and infer instantiations of embedded blocks such as memories and multipliers in the FPGAs. All FPGA specific details, such as the instantiation of the clock buffers and the partitioning of the design between FPGAs, were contained within high level wrapper modules. These wrapper modules together instantiated the complete, unmodified, ASIC core. By confining the FPGA implementation details entirely within these wrapper files we avoided inconsistencies associated with maintaining parallel source trees; the FPGA and ASIC implementations used an identical source base, giving us a very high confidence in the match between ASIC and FPGA implementations. As a side effect, synthesis warnings from the FPGA tools often provided meaningful feedback to the designers for coding changes in advance of the ASIC synthesis.

3.2 Choosing a Platform

An enormous amount of logic was required for the implementation, the ASICs contain more than 10 million transistors. A similar approach to wireless baseband modelling has been employed by Berkeley Wireless Research Center with their custom BEE FPGA processor [5]. BWRC chose to implement their own system whereas while we originally did this about four years ago, we found it more cost effective to buy off-the-shelf units; the capacity and speed of modern FPGAs met our needs. We chose the Dini Group's DN3000K10 [6] populated with 3 Xilinx Virtex-II 6 million gate parts in the FF1152 package [7].

The Virtex-II parts provide clock gating and embedded hardware multipliers, both of which our PHY uses. Gated clocking allows us to reduce power consumption by removing the clock to inactive circuits. Because the Virtex-II parts have support for gated clocking our ASIC Verilog was directly implementable. The computationally intensive nature of the PHY meant that we made significant use of the embedded hardware multipliers in the Virtex-IIs.

There were several clocks used in the project, principally the 33MHz PCI clock and a 20MHz PHY clock. The clocks were distributed to all FPGAs simultaneously, to avoid difficulties with skew across the partition.

3.3 Partitioning

We partitioned the two ASICs into three FPGAs, as shown in Figure 2. The MAC was sufficiently small to fit within a single FPGA. The PHY, however, had to be split across two FPGAs. One problem faced was the large number of configuration and status registers; the PHY contains roughly a thousand configuration and status registers, each up to 16 bits in length, which are accessed through a narrow configuration interface. It was infeasible to carry this large number of registers across the partition, so the interface and configuration blocks were duplicated, appearing in both PHY FPGAs. Having done so, there was then a natural split within the ASIC that supported a partition given the available IOs. Because we could partition satisfactorily by hand, no effort was made to use an automated partitioning tool.

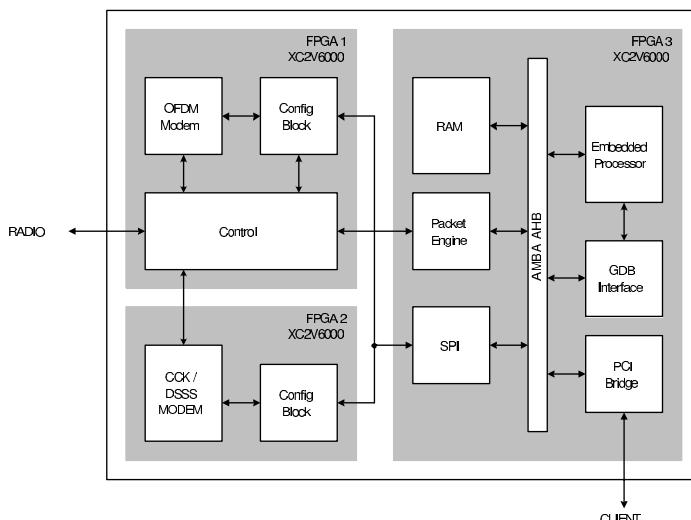


Fig. 2. The partitioned implementation, with the PHY in FPGAs 1 and 2 on the left, and the MAC in FPGA 3 on the right.

3.4 Interfaces

The full benefit of the prototype could only be achieved if the system was able to interface with existing hardware, forming a complete system. For one interface the PCI connector was used, for the other interface a custom interposer board was constructed. The abundance of FPGA IOs were necessary to meet this need.

The Cisco Access Point that was used for product development interfaces with our MAC across a MiniPCI interface. Using a passive PCI to MiniPCI adapter we were able to connect the prototyping board directly to the Access Point. Having the PCI bus directly connected to an FPGA was very important, a number of FPGA development boards have inbuilt PCI implementations, but our MAC SoC already contains a PCI implementation that we wanted to exercise.

A PCI solution is also capable of being inserted into any standard PC, opening up interesting debugging opportunities. We used the PC as a platform for running development support software, such as GUIs to control and observe the configuration of the PHY, tools to monitor the execution of software on the embedded processor, and even to act as an 802.11 client, sending and receiving network traffic through our development system.

Two interposer cards were developed to provide an interface to an existing radio. For development involving the prototype PHY the interposer card included the missing analog features of the PHY, the DAC and ADC. For development requiring a full speed PHY the interposer card included a previous generation silicon PHY. There were also test headers for devices such as logic analysers and oscilloscopes.

A photo of the completed system, showing the prototype interfacing between a Cisco Access Point and a 5GHz radio is shown in Figure 3.

3.5 Tool Flow

The process of building a new implementation following design changes was completely automated. Makefiles running on Linux servers executed first the Synplify synthesiser, which was the native Linux version, and then the Xilinx ISE implementation software, which was the PC version executed in Linux under the WINE Windows emulation environment [8]. The resulting bitstreams were loaded into the FPGAs using the Smart Media flash memory based configuration system present on the Dini cards.

The Synplify synthesizer was capable of automatically inferring instances of Xilinx features from the behavioural Verilog. Block RAMs were automatically inferred from behavioural descriptions of our ASIC memories, although some initial non-functional coding-style changes had to be made to the ASIC Verilog. The hardware multiplier units were also automatically inferred. There were no modifications required to the ASIC design for FPGA synthesis, which was critical to automating the process.

The Xilinx ISE flow was also completely automated. No hand placement was necessary; even as the designs stretched the capacity of the chips, the automatic placement was always satisfactory.

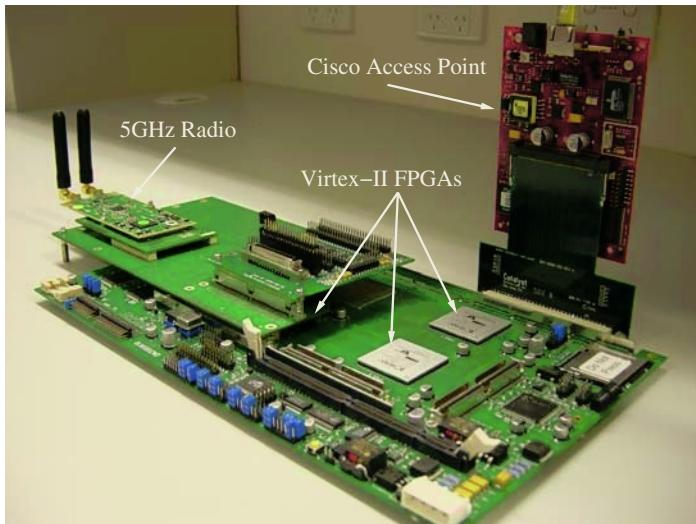


Fig. 3. The prototyping system, showing the 5GHz radio and Access Point.

The utilisation and build times for the three Xilinx Virtex-II XC2V6000 parts are shown below. We found that executing the tools on Linux Pentium 4 servers was many times quicker than executing the same sequence using the Solaris versions of the tools on our high-end Sun servers; this meant saving hours per re-implementation.

Table 1. FPGA Usage.

FPGA	LUTs	Registers	Block RAMs	Multipliers	Build Time
1 (PHY)	57,000 (83%)	32,000 (46%)	48 (33%)	56 (38%)	6 Hours
2 (PHY)	40,000 (59%)	11,000 (15%)	9 (6%)	48 (33%)	3 Hours
3 (MAC)	10,000 (14%)	7,000 (9%)	93 (64%)	0 (0%)	1 Hour

3.6 Custom Development Tools

Being able to plug the development board into a PCI slot and have a simple, high bandwidth interface from the PC into the system offered great development opportunities. Having software to take advantage of this interface as the opportunities became apparent was well worth the effort. The PHY chip contains around a thousand configuration and status registers. Providing a simple interface to manage this configuration was critical to the success of the system. We ported the existing powerful graphical user interface [10] used to configure our group's existing ASICs. This program was written in Python [11], an interpreted programming language. This allowed the hardware developers to write

their own scripts to communicate with the prototype and debug their hardware. An example view is shown in Figure 4. Because this is the same interface that we will use to configure and debug the silicon ASICs, it will be a transparent switch for the developers to move from working with the prototype to the ASIC, they are already experienced with the test environment.

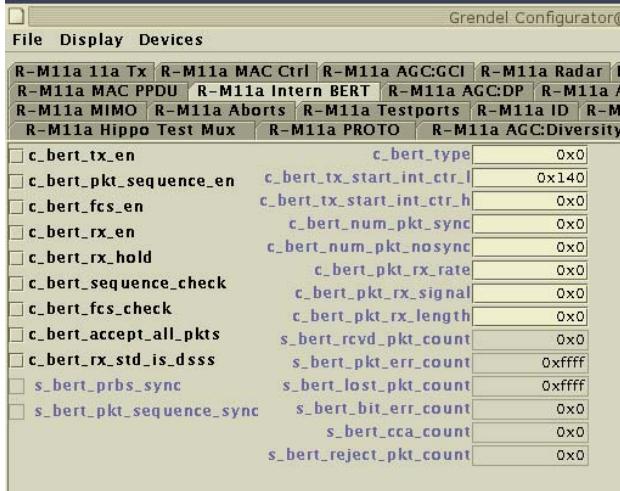


Fig. 4. A screen capture showing the software used to configure and debug the implementation.

Firmware development for the embedded processor also made use of the PCI interface. Firmware is loaded into the processor memory across the PCI bus. Hardware support was added to the embedded processor to provide an interface with the GNU GDB debugger [12] running on the host PC. This allowed for the observation of code execution in the embedded processor, and provided features such as stepping, breakpoints, and watchpoints.

4 Experiences and Lessons

4.1 Revision Control

A difficulty that became apparent from having a rapid re-implementation was keeping the software and the hardware in synchronisation. Modifications to the register map in the hardware would cause existing software to break. A method was implemented to automatically generate a hardware and software implementation of the memory map from a common abstract view, meaning that any changes to the mapping were simultaneously updated in both the hardware Verilog and software map files.

Given the 6 hour re-implementation time, and the fact that the source database was actively being updated, there was sometimes confusion about the versions of files that had been taken to construct the implementation. The build scripts were modified to automatically embed checks such as time stamps and revision numbers into the implementation. These were accessible from our development software, which could detect if the hardware and software were not using the same version.

4.2 Simulating vs Prototyping

When making enhancements to the hardware it was often quicker to test changes through the prototype implementation than to extensively simulate them. Still, the 6 hour implementation time was long enough to warrant a careful analysis before re-implementation, guessing the cause of unexpected behaviour could waste this time.

Simulation and prototyping complemented each other; bugs found in simulation were examined on the prototype, and bugs found in the prototype were examined in simulation. Using both methods increased our understanding and confidence in the design.

4.3 Tool Problems

We found that the FPGA tools were lacking in their ability to handle partial resynthesis. There are large blocks in our design, such as the Viterbi decoder and the processor core, that are not actively developed, remaining constant between each resynthesis. Having blocks such as these reused from previous implementations could save on re-implementation time. But it was non-trivial to make the tools do this, and having done so the time saved was not great.

The FSM optimisations performed by Synplify were, after much head scratching, found to be broken. Several FSMs were observed to lock up in unexpected states, a behaviour inconsistent with the source code. The problem was that the FSMs were being recoded as one-hot state machines, but the synthesizer failed to add logic to check when the FSM was in an illegal state. Disabling the FSM optimisations resolved this problem. This highlighted the need for equivalence checking of the synthesis output.

4.4 Concurrent Development

Having a firmware development proceed concurrently with the development of the MAC and PHY hardware allowed for an evolution and fine tuning of the hardware / software interface. The ability to tightly integrate observability, such as trace-logs, instruction counters, and timers woven deep into the processor, provided us with useful profiling data.

Having the prototype PHY interface directly with a real radio, allowing exposure to real and simulated radio channels with issues such as multipath, allowed

us to verify and fine-tune our algorithms. The system continues to be used to implement experimental ideas.

Having the prototype MAC interact with other 802.11 MACs allowed for a better understanding of how these implementations behave and affect our performance.

When using the system we often discovered ways to improve observability. There were test ports that we had not anticipated, that only became apparent once we used the system. Some functionality, such as a trace buffer, was found to be incredibly useful in practice, and so was added to the ASIC code base.

5 Conclusion

The FPGA implementation of our MAC and PHY has become an extensively used, powerful tool. It is vital to the success of our chips. Its ability to transparently fit into the existing ASIC development process, and to automatically produce a re-implementation, without user intervention, have been key to its success.

It has provided great confidence to the hardware and software developers to extensively exercise their implementation, beta-testing in a very realistic environment before tape-out, enhancing the success of the first spin silicon. It has allowed development and experience with our test environment, saving time when the ASICs return from fab.

The system continues to be used for firmware and hardware development.

References

1. The IEEE 802.11 Standards, <http://grouper.ieee.org/groups/802/11/>
2. Philip Ryan et al., "A Single Chip COFDM Modem for IEEE 802.11a with Integrated ADCs and DACs", ISSCC 2001
3. ARM Developer Suite, ARM Ltd, <http://www.arm.com/devtools/ads/>
4. Cadence Design Systems, <http://www.cadence.com/>
5. Berkeley Wireless Research Center, Berkeley Emulation Engine, <http://bwrc.eecs.berkeley.edu/Research/BEE/index.htm>
6. The Dini Group, <http://www.dinigroup.com/>
7. Xilinx, <http://www.xilinx.com/>
8. WINE, <http://www.winehq.org/>
9. Synplicity, Inc, <http://www.synplicity.com/>
10. Tom McDermott et al., "Grendel: A Python Interface to an 802.11a Wireless LAN Chipset", The Tenth International Python Conference, 2002
11. The Python Programming Language, <http://www.python.org/>
12. GDB, The GNU Project Debugger, <http://www.gnu.org/software/gdb/gdb.html>

ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix

Bingfeng Mei^{1,2}, Serge Vernalde¹, Diederik Verkest^{1,2,3},
Hugo De Man^{1,2}, and Rudy Lauwereins^{1,2}

¹ IMEC vzw, Kapeldreef 75, Leuven, B-3001, Belgium

² Department of Electrical Engineering, Katholieke Universiteit Leuven, Belgium

³ Department of Electrical Engineering, Vrije Universiteit Brussel, Belgium

Abstract. The coarse-grained reconfigurable architectures have advantages over the traditional FPGAs in terms of delay, area and configuration time. To execute entire applications, most of them combine an *instruction set processor*(ISP) and a reconfigurable matrix. However, not much attention is paid to the integration of these two parts, which results in high communication overhead and programming difficulty. To address this problem, we propose a novel architecture with tightly coupled *very long instruction word* (VLIW) processor and coarse-grained reconfigurable matrix. The advantages include simplified programming model, shared resource costs, and reduced communication overhead. To exploit this architecture, our previously developed compiler framework is adapted to the new architecture. The results show that the new architecture has good performance and is very compiler-friendly.

1 Introduction

Coarse-grained reconfigurable architectures have become increasingly important in recent years. Various architectures were proposed [1][2][3][4]. These architectures often comprise a matrix of functional units (FUs), which are capable of executing word- or subword-level operations instead of bit-level ones found in common FPGAs. This *coarse* granularity greatly reduces the delay, area, power and configuration time compared with FPGAs, however, at the expense of flexibility. Other features include predictable timing, a small configuration storage space, flexible topology, etc. However, the reconfigurable matrix alone is not capable of executing entire applications. Most coarse-grained architectures are coupled with processors, typically RISCs. The execution model of such hybrid architectures is based on the well-known 90/10 locality rule[5], i.e., *a program spends 90% of its execution time in only 10% of the code*. Some computational-intensive kernels are mapped to the matrix, whereas the rest code is executed by the processor. So far not much attention is paid to the integration of the two parts of the system. The coupling between the processor and the reconfigurable matrix is often loose, which is essentially two separated parts connected by a

communication channel. This results in programming difficulty and communication overhead. In addition, the coarse-grained reconfigurable architecture consists of components which are similar to those used in processors. This represents a major resource-sharing and cost-saving opportunity, which is not extensively exploited in traditional coarse-grained architectures.

To address the above problems, in this paper we presents a novel architecture called ADRES (*Architecture for Dynamically Reconfigurable Embedded System*), which tightly couples a VLIW processor and a coarse-grained reconfigurable matrix. The VLIW processor and the coarse-grained reconfigurable matrix are integrated into one single architecture but with two virtual functional views. This level of integration has many advantages compared with other coarse-grained architectures, including improved performance, a simplified programming model, reduced communication costs and substantial resource sharing. Nowadays, new programmable architecture can not succeed without good support for mapping applications. In our previous work, we built a compiler framework for a family of coarse-grained architectures [6]. A novel modulo scheduling algorithm was developed to exploit the loop-level parallelism efficiently[7]. In this paper, we present how this compiler framework can be adapted to the ADRES architecture. In addition, some new techniques are proposed to solve the integration problem of the VLIW processor and the reconfigurable matrix.

The paper is organized as follow. Section 2 describes the proposed ADRES architecture and analyzes its main advantages. Section 3 discusses how the compiler framework is ported to the ADRES architecture and some considerations of the compilation techniques. Section 4 reports experimental results. Section 5 covers related work. Section 6 concludes the paper and presents future work.

2 ADRES Architecture

2.1 Architecture Description

Fig. 1 describes the system view of the ADRES architecture. It is similar to a processor with an execution core connected to a memory hierarchy. The ADRES core(fig 3) consists of many basic components, including mainly FUs and register files(RF), which are connected in a certain topology. The FUs are capable of executing word-level operations selected by a control signal. The RFs can store intermediate data. The whole ADRES matrix has two functional views, the VLIW processor and the reconfigurable matrix. These two functional views share some physical resources because their executions will never overlap with each other thanks to the processor/co-processor model. For the VLIW processor, several FUs are allocated and connected together through one multi-port register file, which is typical for VLIW architecture. Compared with the counterparts of the reconfigurable matrix, these FUs are more powerful in terms of functionality and speed. They can execute more operations such as branch operations. Some of these FUs are connected to the memory hierarchy, depending on available ports. Thus the data access to the memory is done through the load/store operation available on those FUs.

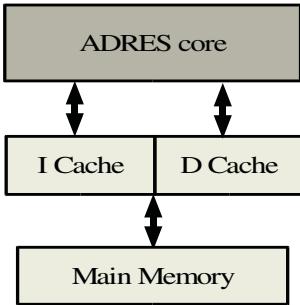


Fig. 1. ADRES system

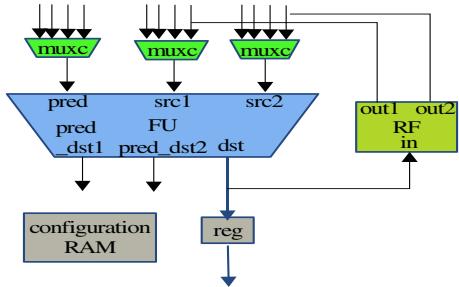


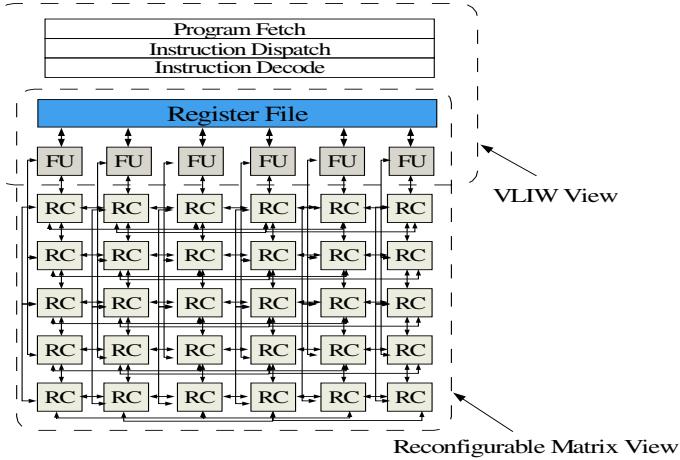
Fig. 2. Example of a Reconfigurable Cell

For the reconfigurable matrix part, apart from the FUs and RF shared with the VLIW processor, there are a number of *reconfigurable cells*(RC) which basically comprise FUs and RFs too(fig. 2). The FUs can be heterogeneous supporting different operation sets. To remove the control flow inside loops, the FUs support predicated operations. The distributed RFs are small with less ports. The multiplexors are used to direct data from different sources. The configuration RAM stores a few configurations locally, which can be loaded on cycle-by-cycle basis. The configurations can also be loaded from the memory hierarchy at the cost of extra delay if the local configuration RAM is not big enough. Like instructions in ISPs, the configurations control the behaviour of the basic components by selecting operations and multiplexors. The reconfigurable matrix is used to accelerate the dataflow-like kernels in a highly parallel way. The matrix also includes the FUs and RF of the VLIW processor. The access to the memory of the matrix is also performed through the VLIW processor FUs.

In fact, the ADRES is a template of architectures instead of a fixed architecture. An XML-based architecture description language is used to define the communication topology, supported operation set, resource allocation and timing of the target architecture [6]. Even the actual organization of the RC is not fixed, FUs and RFs can be put together in several ways, for example, two FUs can share one RF. The architecture shown in fig. 3 and fig. 2 is just one possible instance of the template. The specified architecture will be translated to an internal architecture representation to facilitate compilation techniques.

2.2 Improved Performance with the VLIW Processor

Many coarse-grained architectures consist of a reconfigurable matrix and a relatively slow RISC processor, e.g., TinyRisc in MorphoSys [1] and ARC in Chameleon [3]. These RISC processors execute the unaccelerated part of the application, which only represents a small portion of execution time. However, such a system architecture has some problems due to the huge performance gap between the RISC and the reconfigurable matrix. According to Amdahl's law [5], the performance gain that can be obtained by improving some portion of an application can be calculated as equation 1. Suppose the kernels, representing 90%

**Fig. 3.** ADRES core

of execution time, are mapped to the reconfigurable matrix to obtain 30 times of acceleration over the RISC processor, the overall speedup is merely 7.69. Obviously a high kernel speedup is not translated to a high overall speedup. The unaccelerated part, which is often irregular and control-intensive, becomes a bottleneck. Speeding up this part is essential for the overall performance. Although it is hard to exploit higher parallelism for the unaccelerated part on the reconfigurable matrix, it is still possible to discover *instruction-level parallelism* (ILP) using a VLIW processor, where 2-4 times speedup over the RISC is reasonable. If we recalculate the speedup with the assumption of 3 times acceleration for the unaccelerated code, the overall acceleration is now 15.8, much better than the previous scenario. This simple calculation proves the importance of a balanced system. The VLIW can help to improve the overall speedup dramatically in certain circumstances.

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \quad (1)$$

2.3 Simplified Programming Model and Reduced Communication Cost

A simplified programming model and reduced communication cost are two important advantages of the ADRES architecture. These are achieved by making the VLIW processor and the reconfigurable matrix share access to the memory.

In traditional reconfigurable architectures, the processor and the reconfigurable matrix are essentially separated. The communication is often through explicit data copying. The normal execution steps are: (1) copy the data from the VLIW memory to that of the reconfigurable matrix; (2) the reconfigurable matrix part computes the kernel; (3) the results are copied back from the memory of the reconfigurable matrix to that of the VLIW processor. Though some

techniques are adopted to reduce the data copying, e.g., wider data bus and DMA controller, the overhead is still considerable in terms of performance and energy. From the programming point of view, the separated processor and reconfigurable matrix require significant code rewriting. Starting from a software implementation to map kernels to the matrix, we have to identify the data structures used for communication and replace them with communication primitives. Data analysis should be done to make sure as few as possible data are actually copied. In addition, the kernels and the rest of the code have to be cleanly separated in such a way that no shared access to any data structure remains. These transformations are often complex and error-prone.

In the ADRES architecture, the data communication is performed through the shared RF and memory. This feature is very helpful to map high-level language code such as C to the ADRES architecture without major changes. When a high-level language is compiled to a processor, the local variables are normally allocated in the RF, whereas the static variables and arrays are allocated in the memory space. When the control of the program is transferred between the VLIW processor and the reconfigurable matrix, those variables used for communication can stay in the RF or the memory as they were. The copying is unnecessary because both the VLIW processor and the reconfigurable matrix share access to the RF and memory hierarchy. From programming point of view, this *shared-memory* architecture is more compiler-friendly than the *message-passing* one. Moreover, the RF and memory are alternately shared instead of being simultaneously shared. This eliminates data synchronizing and integrity problems. Code doesn't require any rewriting and can be handled by compiler automatically.

2.4 Substantial Resource Sharing

Since the basic components such as the FUs and RFs of the reconfigurable matrix and those of the VLIW processor are basically the same, one natural thinking is that resources might be shared to have substantial cost-saving. In other coarse-grained reconfigurable architectures, the resources cannot be effectively shared because the processor and the reconfigurable matrix are two separated parts. For example, the FU in the TinyRisc of MorphoSys cannot work cooperatively with the reconfigurable cells in the matrix. In the ADRES architecture, since the VLIW processor and the reconfigurable matrix are indeed two virtual functional views of the same physical entity, many resources are shared among these two parts. Due to its processor/co-processor model, only one of the VLIW processor and the reconfigurable matrix is active at any time. This fact makes the resource sharing possible. Especially, most components of the VLIW processor are reused in the reconfigurable matrix as shown in fig. 3. Although the amount of VLIW resources is only a fraction of those of the reconfigurable matrix, they are generally more powerful. For example, the FUs of the VLIW processor can execute more operations. The register file has much more ports than the counterparts in the reconfigurable matrix. In other words, the resources of the VLIW processor are substantial in terms of functionality. Reusing these resources can help to improve the performance and increase the schedulability of kernels.

3 Adaptations of Compilation Techniques

Given the ever-increasing pressure of time-to-market and complexity of applications, the success of any new programmable architecture is more and more dependent on good design tools. For example, VLIW processors have gained huge popularity among DSP/multimedia applications although they are neither the most power- or performance-efficient ones. One important reason is that they have mature compiler support. An application written in a high-level programming language can be automatically mapped to a VLIW with reasonable quality. Compared with other coarse-grained reconfigurable architectures, the ADRES architecture is more compiler-friendly due to the simplified programming model discussed in section 2.3. However, some new compilation techniques need to be adopted to fully exploit the potential of the architecture.

3.1 Compilation Flow Overview

Previously, we have developed a compiler framework for a family of coarse-grained reconfigurable architectures [6]. A novel modulo scheduling algorithm and an abstract architecture representation were also proposed to exploit loop-level parallelism [7]. They have been adapted to the ADRES architecture. The overall compilation flow is shown in fig. 4. We use the IMPACT compiler framework [8] as a frontend to parse C source code, do some optimization and analysis, and emit the intermediate representation (IR), which is called *lcode*. Taking *lcode* as input, the compiler first tries to identify the pipelineable loops, which can be accelerated by the reconfigurable matrix. Then, the compilation process is divided into two paths that are for the VLIW processor and the reconfigurable matrix respectively. The identified loops are scheduled on the reconfigurable matrix using the modulo scheduling algorithm we developed [7]. The scheduler takes advantage of the shared resources, e.g., the multi-port VLIW register file,

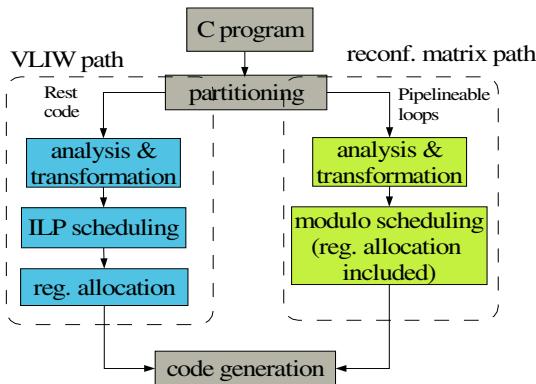


Fig. 4. Compilation Flow for the ADRES architecture

to maximize performance. The remaining code is mapped to the VLIW processor using regular VLIW compilation techniques, including ILP scheduling and register allocation. Afterwards, the two parts of scheduled code are put together, ready for being executed by the ADRES architecture.

3.2 Interface Generation

The compilation techniques for the VLIW architecture are already mature and the main compilation techniques for the coarse-grained architecture were developed in our previous work. Adapted to the ADRES architecture, the most important problem is how to make the VLIW processor and the reconfigurable matrix work cooperatively and communicate with each other. Thanks to ADRES's compiler-friendly features, interface generation is indeed quite simple (fig. 5).

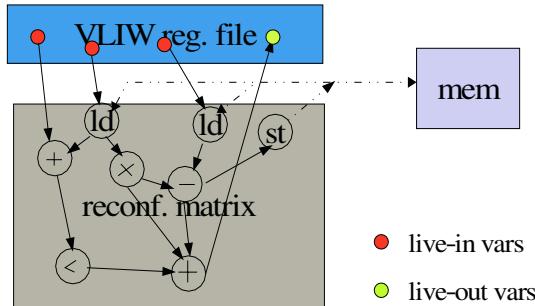


Fig. 5. Interfacing between the VLIW processor and the Reconfigurable matrix

Each loop mapped to the matrix has to communicate with the rest of application, e.g., taking input data and parameters, and writing back results. As mentioned in section 2.3, the communication of the ADRES architecture is performed through shared register file and shared memory. Using dataflow analysis, the *live-in* and *live-out* variables are identified, which represents the input and output data communicated through the shared register file. These variables will be allocated in the VLIW register file. Since these variables will occupy some register space throughout the lifetime of the loops, they are subtracted from the capacity of the VLIW register file. Therefore the scheduler won't overuse the VLIW register file for other tasks. As to the variables mapped to the memory, we don't need to do anything. The mapped loop can access the correct address through the load/store operations available on some FUs of the VLIW processor.

4 Experimental Results

For the purpose of experiment, an architecture resembling the topology of MorphiSys [1] is instantiated from the ADRES template. In this configuration, a

total of 64 FUs are divided into four tiles, each of which consists of 4x4 FUs. Each FU is not only connected to the 4 nearest neighbor FUs, but also to all FUs within the same row or column in this tile. In addition, there are row buses and column buses across the matrix. The first row of FUs is also used by the VLIW processor, and are connected to a multi-port register file. Only the FUs in the first row are capable of executing memory operations, i.e., load/store operations.

The testbench consists of 4 programs, which are derived from C reference code of TI's DSP benchmarks and MediaBench [9]. The *idct* is a 8x8 inverse discrete cosine transformation, which consists two loops. The *adpcm-d* refers to an ADPCM decoder. The *mat_mul* computes matrix multiplication. The *fir_cpl* is a complex FIR filter. They are typical multimedia and digital signal processing applications with abundant inherent parallelism.

Table 1. Schedule results

loop	no. of ops	live-in vars	live-out vars	II	IPC	sched. density
idct1	93	4	0	3	31	48.4%
idct2	168	4	0	4	42	65.6%
adpcm-d	55	9	2	4	13.8	21.5%
mat_mul	20	12	0	1	20	31.3%
fir_cpl	23	9	0	1	23	35.9%

The schedule results are shown in table 1. The second column refers to the total number of operations within the loop body. The II is *initiation interval*, meaning the loop starts a new iteration every II cycles [10]. The live-in and live-out variables are allocated in the VLIW register file. The instructions-per-cycle (IPC) reflects how many operations are executed in one cycle on average. Scheduling density is equal to *IPC/No.ofFUs*. It reflects the actual utilization of all FUs for computation. The results show the IPC is pretty high, ranging from 13.8 to 42. The FU utilization is ranged from 21.5% to 65.6%. For kernels such as *adpcm-d*, the results are constrained by achievable minimal II(MII).

The table 2 shows comparisons with the VLIW processor. The tested VLIW processor has the same configuration as the first row of the tested ADRES architecture. The compilation and simulation results for VLIW architecture are obtained from IMPACT, where aggressive optimizations are enabled. The re-

Table 2. Comparisons with VLIW architecture

app.	total ops (ADRES)	total cycles (ADRES)	total ops (VLIW)	total cycles (VLIW)	speed-up
idct	211676	6097	181853	38794	6.4
adpcm_d	8150329	594676	5760116	1895055	3.2
mat_mul	20010518	1001308	13876972	2811011	2.8
fir_cpl	69126	3010	91774	18111	6.0

sults for the ADRES architecture are obtained from a developed co-simulator, which is capable of simulating the mixed VLIW and reconfigurable matrix code. Although these testbenches are small applications, the results already reflect the integration impact of the VLIW processor and the reconfigurable matrix. The speed-up over the VLIW is from 2.8 to 6.4, showing pretty good performance.

5 Related Work

Many coarse-grained reconfigurable architectures have been proposed in recent years. MorphoSys [1] and REMARC [4] are typical ones consisting of a RISC processor and a fabric of reconfigurable units. For MorphoSys the communication is performed through a DMA controller and a so-called frame buffer. In REMARC, the coupling is tighter. The matrix is used as a co-processor next to the MIPS processor. Neither of these architectures has compiler support for the matrix part. Chameleon [3] is a commercial architecture that comprises an ARC processor and a reconfigurable processing fabric as well. The communication is through a 128-bit bus and a DMA controller. The data has to be copied between the two memory spaces. Compiler support is limited to the processor side.

Another category of reconfigurable architectures presents much tighter integration. Examples are ConCise [11], PRISC [12] and Chimaera [13]. In these architectures, the reconfigurable units are deeply embedded into the pipeline of the processor. Customized instructions are built with these reconfigurable units. The programming model is simplified compared with the previous category because resources such as memory ports and register file are exposed to both the processor and the reconfigurable units. This leads to good compiler support. However, these architectures do not have much potential for performance, constrained by limited exploitable parallelism.

6 Conclusions and Future Work

Coarse-grained reconfigurable architectures have been gaining importance recently. Many new architectures are proposed, which normally comprise a processor and a reconfigurable matrix. In this paper, we address the integration problem between the processor and the reconfigurable matrix, which has not received enough attention in the past. A new architecture called ADRES is proposed, where a VLIW processor and a reconfigurable matrix are tightly coupled in a single architecture and many resources are shared. This level of integration brings a lot of benefits, including increased performance, simplified programming model, reduced communication cost and substantial resource sharing.

get fpl03.bbl Our compiler framework was adapted to the new without much difficulty. It proves that the ADRES architecture is very compiler-friendly. The VLIW compilation techniques and the compilation techniques for the reconfigurable matrix can be applied to the two parts of the ADRES architecture respectively. The partitioning and interfacing of the accelerated loops and the rest of code can be handled by the compiler without requiring code rewriting.

However, we have not implemented the ADRES architecture at the circuit level yet. Therefore, many detailed design problems have not been taken into account and concrete figures such as area and power are not available. Hence, to implement the ADRES design is in the scope of our future work. On the other hand, we believe the compiler is even more important than the architecture. We will keep developing the compiler to refine the ADRES architecture from the compiler point of view.

References

1. Singh, H., Lee, M.H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Filho, E.M.C.: Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers* **49** (2000) 465–481
2. C. Ebeling, D. Cronquist, P.F.: RaPiD - reconfigurable pipelined datapath. In: Proc. of Field Programmable Logic and Applications. (1996)
3. : Chameleon Systems Inc. (2002) <http://www.chameleonsystems.com>.
4. Miyamori, T., Olukotun, K.: REMARC: Reconfigurable multimedia array coprocessor. In: FPGA. (1998) 261
5. Patterson, D.A., Hennessy, J.L.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, Inc. (1996)
6. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In: International Conference on Field Programmable Technology. (2002)
7. Mei, B., Vernalde, S., Verkest, D., Man, H.D., Lauwereins, R.: Exploiting loop-level parallelism for coarse-grained reconfigurable architecture using modulo scheduling. In: Proc. Design, Automation and Test in Europe (DATE). (2003)
8. Chang, P.P., Mahlke, S.A., Chen, W.Y., Warter, N.J., Hwu, W.W.: IMPACT: An architectural framework for multiple-instruction-issue processors. In: Proceedings of the 18th International Symposium on Computer Architecture (ISCA). (1991)
9. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In: International Symposium on Microarchitecture. (1997) 330–335
10. Rau, B.R.: Iterative modulo scheduling. Technical report, Hewlett-Packard Lab: HPL-94-115 (1995)
11. Kastrup, B.: Automatic Synthesis of Reconfigurable Instruction Set Accelerations. PhD thesis, Eindhoven University of Technology (2001)
12. Razdan, R., Brace, K., Smith, M.D.: Prisc software acceleration techniques. In: Proc. 1994 IEEE Intl. Conf. on Computer Design. (1994)
13. Hauck, S., Fry, T.W., Hosler, M.M., Kao, J.P.: The Chimaera reconfigurable functional unit. In: Proc. the IEEE Symposium on FPGAs for Custom Computing Machines. (1997)

Inter-processor Connection Reconfiguration Based on Dynamic Look-Ahead Control of Multiple Crossbar Switches

Eryk Laskowski¹ and Marek Tudruj^{1,2}

¹ Institute of Computer Science Polish Academy of Sciences
ul. Ordona 21, 01-237 Warsaw, Poland

² Polish-Japanese Institute of Information Technology
ul. Koszykowa 86, 02-008 Warsaw, Poland
{laskowsk, tudruj}@ipipan.waw.pl

Abstract. A parallel system architecture for program execution based on the look-ahead dynamic reconfiguration of inter-processor connections is discussed in the paper. The architecture is based on inter-processor connection reconfiguration in multiple crossbar switches that are used for parallel program execution. Programs are structured into sections that use fixed inter-processor connections for communication. The look-ahead dynamic reconfiguration assumes that while some inter-processor connections in crossbar switches are used for current section execution, other connections are in advance configured for execution of further sections. Programs have to be decomposed into sections for given time parameters of reconfiguration control, so, as to avoid program execution delays due to connection reconfiguration. Automatic program structuring is proposed based on the analysis of parallel program graphs. The structuring algorithm finds the partition into sections that minimizes the execution time of a program executed with the look-ahead created connections. The program execution time is evaluated by simulated program graph execution with reconfiguration control modeled as an extension of the basic program graph.

1 Introduction

A new kind of parallel program execution environment based on dynamically reconfigurable connections between processors [1-4] is the main interest of this paper. The proposed environment assumes new paradigm of point-to-point inter-processor connections. Link connection reconfiguration involves time overheads in program execution. These overheads cannot be completely eliminated by an increase of the speed of communication/reconfiguration hardware. However, they can be neutralized by a special method applied at the level of system architecture and at the level of program execution control. This special method is called the look-ahead dynamic link connection reconfigurability [5,6]. Special architectural solutions for the look-ahead reconfigurability consist in increasing the number of hardware resources used for link connection setting (multiple crossbar switches) and using them interchangeably in parallel for program execution and run-time look-ahead reconfiguration. Application programs are partitioned into sections executed by clusters of processors whose mu-

tual connections are prepared in advance. The connections are set in crossbar switches and remain fixed during section execution. At sections boundaries, processor's communication links are switched to look-ahead configured crossbar switches.

If a program can be partitioned into sections whose connection reconfiguration does not delay communication, we obtain the quasi-time transparency of reconfiguration control since connection reconfiguration overlaps with program execution. Then, the multi-processor system behaves as a fully connected processor structure.

An algorithm for program partitioning into sections for execution with the look-ahead prepared connections was designed. Sections are defined by program graph analysis performed at compile time. The algorithm finds the graph partition and the number of crossbar switches that provide time transparent connection control. It is based on list scheduling and iterative task clustering heuristics. The optimization criterion is the total execution time of a program. This time is determined by program graph symbolic execution, which takes into account time parameters of the system.

The paper consists of three parts. In the first part, the look-ahead inter-processor connection reconfiguration principles are discussed. In the second part, main features of the applied graph partitioning algorithm are discussed. In the third part, experimental results of efficiency measures of program execution based on the look-ahead dynamic reconfiguration are shown and discussed.

2 The Principle of the Look-Ahead Connection Reconfiguration

The look-ahead dynamic connection reconfiguration assumes anticipated inter-processor connection setting in communication resources provided in the system. An application program is partitioned into sections, which assume fixed direct inter-processor connections, Fig. 1. Connections for next sections are prepared while current sections are executed. Before execution of a section, the prepared connections are enabled for use in parallel and in a very short time. Thus, this method can provide inter-processor connections with almost no delay in the linear program execution time. In other words it can provide a time transparent control for dynamic link connection

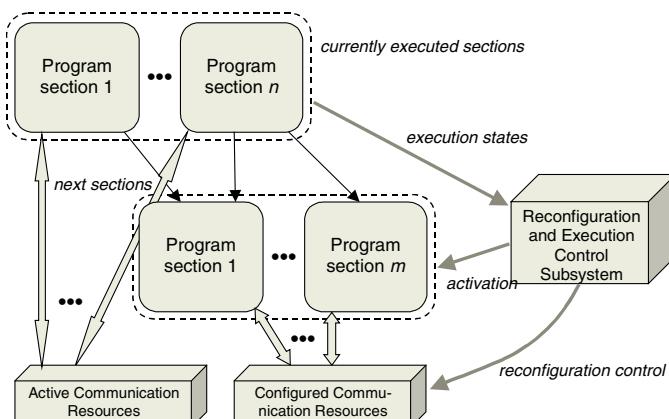


Fig. 1. Program execution based on the look-ahead connection reconfiguration

reconfiguration. The redundant resources used for anticipated connection setting can be link connection switches (crossbar switches, multistage connection networks), processor sets and processor link sets [5].

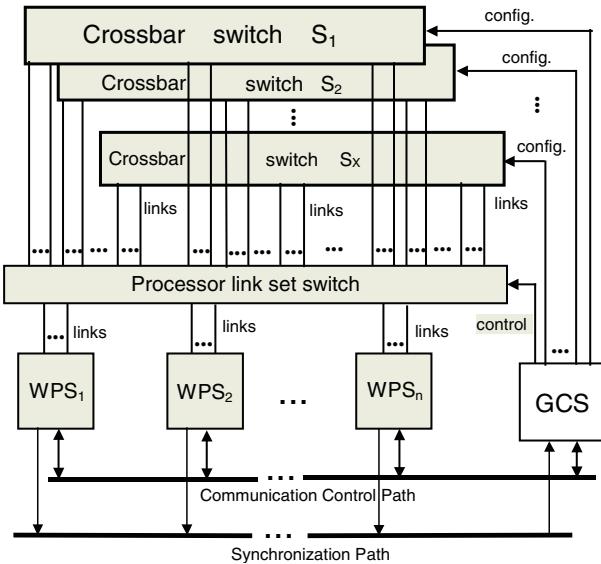


Fig. 2. Look-ahead reconfigurable system with multiple connection switches

The general scheme of a look-ahead dynamically reconfigurable system based on redundancy of link connection switches is shown in Fig. 2. Worker processor subsystems (WPS_i) can consist of a single processor or can include a data processor and a communication processor sharing a common memory. WPS_is have sets of communication links connected to crossbar switches S_1, \dots, S_x , which are interchangeably used as active and configured communication resources. The subsystem links can be switched between the switches by the Processor Link Set Switch. This switch is controlled by the Global Control Subsystem (GCS). GCS collects messages on the section execution states in worker subsystems (link use termination) sent via the Control Communication Path. The simplest implementation of such path is a bus. Depending on the availability of links in the switches S_1, \dots, S_x GCS prepares connections for execution of next program sections. In parallel with reconfiguration, synchronization of states of all processors in clusters for next sections is performed using a hardware Synchronization Path [6]. When all connections for a section are ready and the synchronization has been reached, GCS switches all links of processors, which will execute the section, to the look-ahead configured connections in a proper switch. Then, it enables execution of the section in involved worker processors.

Program sections for execution with the look-ahead connection reconfiguration are defined at compile time by an analysis of the program graph. Three basic program execution strategies can be identified which differ in granularity of control:

- 1) synchronous, with inter-section connection switching controlled at the level of all worker processors in the system,

- 2) asynchronous processor-restrained, where inter-section connection switching is controlled at the level of dynamically defined worker processor clusters,
- 3) asynchronous link-restrained, with granularity of control at the level of single processor links.

The strategies require synchronization of process states in processors in different ways. In the synchronous strategy, processes in all processors in the system have to be synchronized when they reach the end of section points. In the asynchronous processor-restraint strategy, process states in selected subsets of processors are synchronized when they reach the end of use of all links in a section. In the asynchronous link-restraint strategy, the end of use of links in pairs of processors has to be synchronized. In this paper we will be interested in the asynchronous processor-restrained strategy.

3 Program Graph Partitioning

The initial program representation is a weighted Directed Acyclic Graph (DAG) with computation task nodes and communication i.e. node data dependency edges. Programs are executed according to the macro-dataflow model governed by arrivals of all data from all node predecessors. A task executes without pre-emption and, after completion, it sends data to all successor tasks. The graph is static and deterministic.

A two-phase approach is used to tackle the problem of scheduling and graph partitioning in the look-ahead reconfigurable environment [9]. In the first phase, a scheduling algorithm reduces the number of communications and minimizes program execution time, based on program DAG. The program schedule is defined as task-to-processor and communication-to-link assignment with specification of starting time of each task and communication. Assignment preserves the precedence constraints coming from the DAG. The scheduling algorithm is an improved version of ETF /Earliest Task First/ scheduling [8]. In the second phase, scheduled program graph is partitioning into sections for the look-ahead execution in the assumed environment.

A program with specified schedule is expressed in terms of the Assigned Program Graph (APG), Fig. 3. This graph is composed of the non-communicating code nodes (rectangles) and communication instruction nodes (circles). There are activation edges and communication edges in the graph. Weights of nodes represent execution times. Communication is done according to the synchronous model. All nodes are assigned to processors and all communication edges to processor link connections. The graph is assumed static and acyclic. All control in the graph is deterministic.

Program sections correspond to such sub-graphs in the APG for which the following conditions are fulfilled:

- i. section sub graphs are mutually disjoint in respect to communication edges connecting processes allocated to different processors,
- ii. sub-graphs are connected in respect to activation and communication edges,
- iii. inter-processor connections do not change inside a section sub-graph,
- iv. section sub-graphs are complete in respect to activation paths and include all communication edges incident to all communication nodes on all activation paths between any two communication nodes which belong to a section,
- v. all connections for a section are prepared before this section activation, which simultaneously concerns all processors involved in section execution.

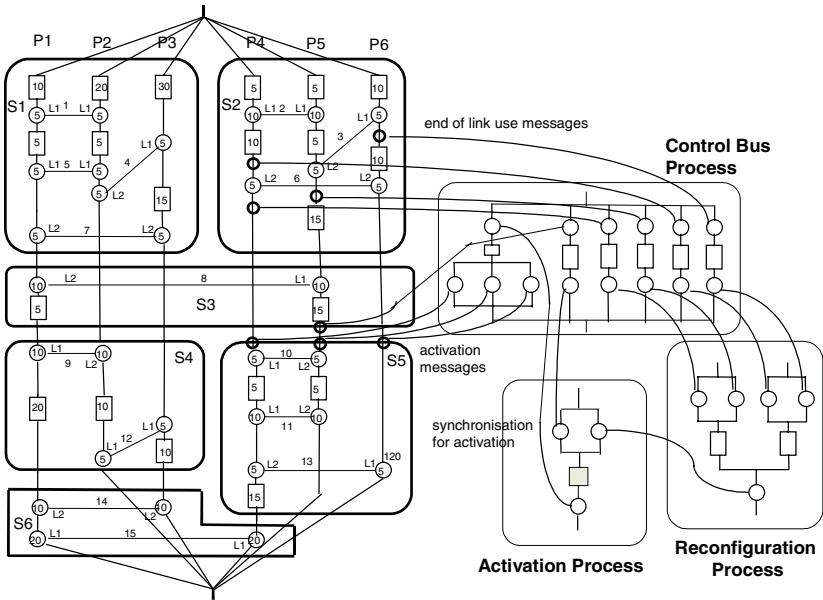


Fig. 3. Modeling the program and the reconfiguration control as a common graph

Program execution time is the optimization criterion. It is estimated by simulated execution of the program graph in a modeled look-ahead reconfigurable system. A partitioned APG graph is automatically extended at all section boundaries by sub-graphs, which model the look-ahead reconfiguration control, Fig. 3. The Communication Control Path, Synchronization Path and Global Control Subsystem are modeled as PAR structures executed on virtual additional processors. Connection setting is modeled by pairs of parallel nodes with input communications (“end of links use” messages) from worker subsystems. After their synchronization, a reconfiguration process is executed by a “Reconfiguration Processor”. Weights of nodes correspond to latencies of control actions, such as crossbar switch reconfiguration, bus latency, etc. Synchronization of processors for link switching between crossbars is modeled in the “Activation Processor”. Activation of next section, is done by multicast transmissions over control bus to all involved worker subsystems.

The algorithm finds program graph partition into sections assigned to crossbar switches. It also finds the minimal number of switches that allow reconfiguration time transparency. Its outline is given in Fig. 4. To simplify graph analysis, a Communication Activation Graph (CAG) is introduced, which contains nodes and edges that correspond to communication edges and activation paths between communication edges of the initial APG graph, respectively. The algorithm starts with all communications assigned to the same crossbar switch. In each step, a vertex of CAG is selected and then, the algorithm tries to include this vertex to a union of existing sections determined by edges of the current vertex. The heuristics tries to find such a union of sections, which doesn't break rules of graph partitioning. The union, which gives the shortest program execution time, is selected. The choice of the vertex for visiting depends on the following APG and CAG graph parameters used in heuristic rules: the critical path of APG, the delay of vertex of CAG, reconfiguration criticality function

for the investigated vertex, and the dependency on link use between communications, see [10] for details. When section clustering doesn't give any execution time improvement, the section of the current vertex is left untouched and crossbar switch is assigned to it. As with section clustering, the choice of the switch depends on program execution time. When algorithm cannot find any crossbar switch for section that allows creating connections with no reconfiguration time overhead, then current number of used switches ($curr_x$ in Fig. 4) is increased by 1 and algorithms is restarted. Vertices can be visited many times. The algorithm stops when with all vertices visited, no further time improvement is obtained. A list of visited vertices (*tabu list*) is used to prevent algorithm from frequent visiting small subset of vertices.

```

Begin
B := initial set of section, each section is composed of
      single communication and assigned to crossbar 1
curr_x := 1      {current number of switches used}
finished := false
While not finished
  Repeat until each vertex of CAG is visited and there is no
    execution time improvement during last  $\beta$  steps {1}
    v := vertex of CAG which maximizes the selection function
        and which is not placed in tabu list
    S := set of sections that contain communications of all
        predecessors of v
    M := Find_sections_for_clustering( v, S )
    If M  $\neq \emptyset$  Then
      B := B - M
      Include to B a new section built of v and
      communications that are contained in sections in M
    Else
      s := section that consists of communication v
      Assign crossbar switch (from 1..curr_x) to section s
      If reconfiguration introduces time overheads Then
        curr_x := curr_x + 1
        Break Repeat
      EndIf
    EndIf
  EndRepeat
  finished := true
EndWhile
End

```

Fig. 4. The general scheme of the graph partitioning algorithm

4 Experimental Results

The described algorithm finds optimal program partitions, which correspond to given reconfiguration control parameters. It can be also used for evaluation and comparative studies of execution efficiency of programs for different program execution control strategies and system parameters. The following parameters were used to characterize an application program in the experiments that were performed:

t_r - reconfiguration time of a single connection in a crossbar switch,

- t_v - section activation time overhead,
 a - average time interval between connection reconfigurations in a program,
 $R = a / (t_r + t_v)$ – reconfiguration control efficiency for a program.

Reconfiguration control efficiency of a given system for the program is a ratio of the average time interval between reconfigurations of processor links in the program to the connection reconfiguration service time in the system. It represents suitability of the reconfiguration control for execution of the program.

Experiments with execution of exemplary program graphs have shown that the look-ahead connection reconfiguration behaves better (positive and large program execution speedup S_q with the look-ahead reconfiguration versus execution with the on-request reconfiguration) for systems in which the reconfiguration efficiency is poor (low values of the parameter R), Fig. 5. For systems with sufficiently fast connection reconfiguration control, the on-request reconfiguration can be sufficient (introduces lower reconfiguration control overhead). This confirms the suitability of the look-ahead reconfiguration for fine-grain parallel systems.

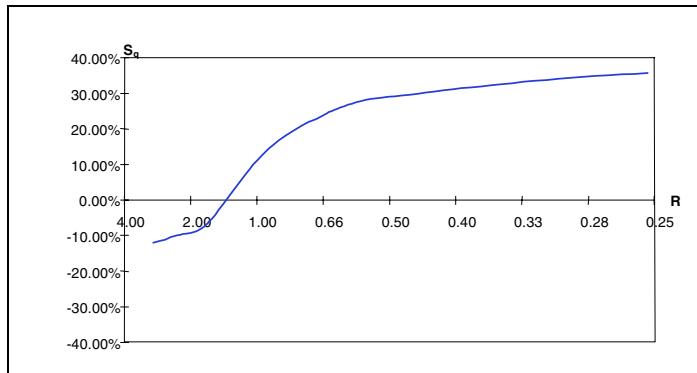
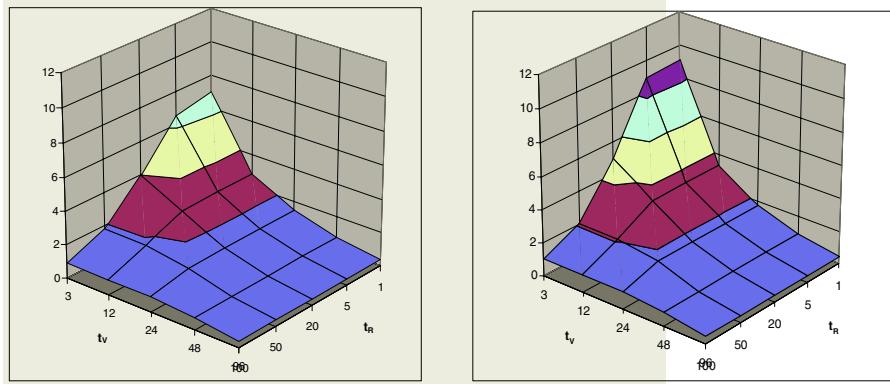


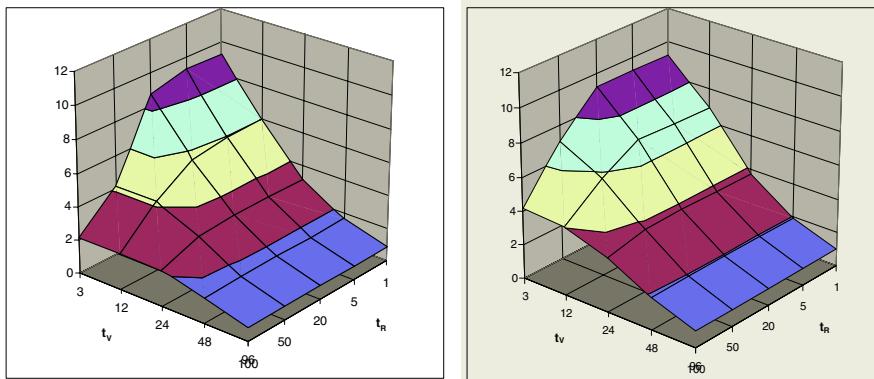
Fig. 5. On-request connection reconfiguration versus look-ahead reconfiguration.

We have examined program execution speedup versus parameters of reconfiguration control (t_r and t_v), the number of crossbar switches, the number of processor links, for the parallel program of Strassen matrix multiplication [11] (two recursion levels). Fig. 6 presents speedup on 12 processors with 2 and 4 links vs. execution on 1 processor with the on-request reconfiguration. The speedup is 7 and 9 in a narrow area of the lowest values of t_v , t_r . With the look-ahead reconfiguration, Fig. 7, the speedup increases to 9.5 in a larger area of lowest t_v , t_r . With an increase in the number of crossbars, the high speedup area increases and is much larger than for the on-request reconfiguration. Fig. 8 shows the reduction of the reconfiguration control time overhead when look-ahead control is used instead of on-request for 2 and 4 crossbars. When reduction is close to 100%, the system behaves as a system with fully-connected inter-processor network. It happens in a narrow area of t_v , t_r that increases when there are more crossbars used. When the number of switches is 6, Fig. 9, the good speedup and reconfiguration overhead area is much larger. So, multiple crossbar switches used with the look-ahead control strongly reduce reconfiguration time overheads. The larger is the number of processor links, the look-ahead method is prevailing vs. on-request for a wider range of reconfiguration and activation time parameters.



a) 12 processors, 2 links

b) 12 processors, 4 links

Fig. 6. Speed-up for Strassen algorithm in on-request environment, single crossbar switch.

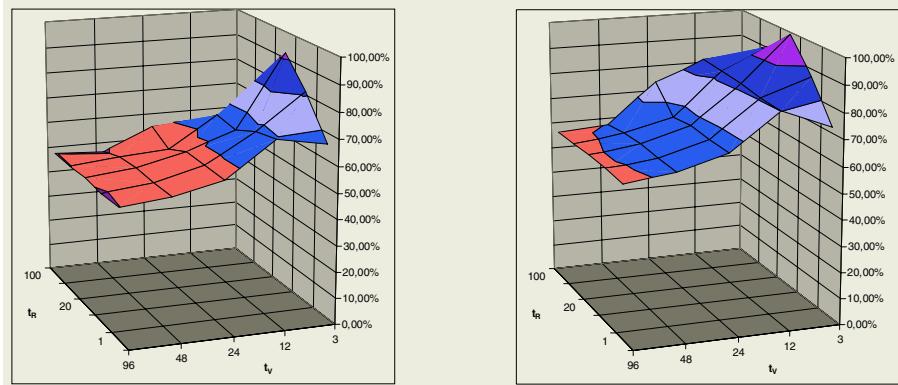
a) 12 processors, 4 links, 2 x-bar switches

b) 12 processors, 4 links, 4 x-bar switches

Fig. 7. Speedup for Strassen algorithm executed in the look-ahead environment.

5 Conclusions

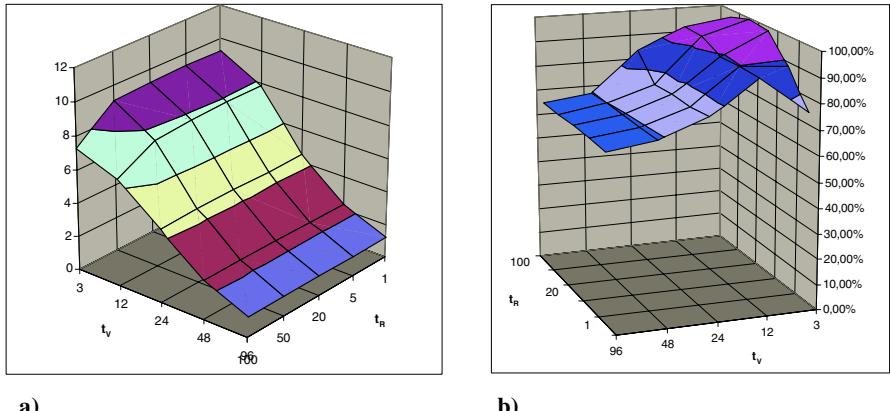
The paper has presented and discussed the concept of the connection look-ahead dynamic reconfigurability in multi-processor systems based on multiple crossbar switches. Program partitioning into sections executed with fixed inter-processor connections is the essential feature of this environment. Programs can be automatically structured for the look-ahead reconfiguration paradigm using the graph partitioning algorithm presented in the paper. For a sufficient number of crossbar switches, the system can behave - from a program point of view - as fully connected multi-processor cluster. This number depends on relation between program time parameters – the granularity of parallelism and time parameters of the dynamic reconfiguration control.



a) 12 processors, 4 links, 2 x-bar switches

b) 12 processors, 4 links, 4 x-bar switches

Fig. 8. Reduction of reconfiguration control overhead for Strassen algorithm with the look-ahead reconfiguration vs. on-request method.



a)

b)

Fig. 9. Speedup (a) and reduction of reconfiguration time overhead (b) for Strassen algorithm executed in the look-ahead environment with 12 processors, 4 links, 6 x-bar switches

This relation can be evaluated using the notion of the reconfiguration control efficiency of a system for a program, introduced in the paper. Experiments show that the lower is the reconfiguration efficiency of the system, the better results are obtained from the application of the look-ahead dynamic connection reconfiguration. It confirms the value of the look-ahead reconfiguration for fine-grained parallel programs, which have time-critical reconfiguration requirements.

Experiments with Strassen algorithm show that quasi time transparency of inter-processor connection setting can be obtained for reconfiguration control time parameters whose area depends on the number of crossbar switches used for dynamic look-ahead connection reconfiguration. This area is narrow for 2 crossbar switches and becomes larger with the bigger number of switches used. Comparing the on-request re-

configuration, the look-ahead reconfiguration gives better program execution speedup and larger applicability area in respect to various system time parameters.

This work has been partially supported by the KBN research grant T11C 007 22.

References

- [1] T. Muntean, SUPERNODE, Architecture Parallele Dynamiquement Reconfigurable de Transputers, *11-emes Journees sur l'Informatique*, Nancy, Janvier 1989.
- [2] A. Bauch, R. Braam, E. Maehle, DAMP - A Dynamic Reconfigurable Multiprocessor System With a Distributed Switching Network, *2-nd European Conf. on Distributed Memory Computing*, Munich, 22-24 April, 1991, pp. 495-504.
- [3] M. Tudruj, "Connection by Communication" Paradigm for Dynamically Reconfigurable Multi-Processor Systems, *Proceedings of the PARELEC 2000*, Trois Rivieres, Canada, August 2000, IEEE CS Press, pp. 74-78.
- [4] M. Tudruj, Embedded Cluster Computing Through Dynamic Reconfigurability of Inter-Processor Connections, *Advanced Environments, Tools and Applications for Cluster Computing*, NATO Advanced Workshop, Mangalia, 1-6 Sept. 2001, Springer Verlag, LNCS 2326.
- [5] M. Tudruj, Multi-transputer architectures with the look-ahead dynamic link connection reconfiguration, *World Transputer Congress '95*, Harrogate, Sept. 1995.
- [6] M. Tudruj, Look-Ahead Dynamic Reconfiguration of Link Connections in Multi-Processor Architectures, *Parallel Computing '95*, Gent, Sept. 1995, pp. 539-546.
- [7] M. Tudruj, O. Pasquier, J.P. Calvez, Fine-grained process synchronization in multiprocessors with the look-ahead inter-processor connection setting, *Proc. of 22nd Euromicro Conference: Short Contributions*, Prague, IEEE CS, Sept. 1996.
- [8] Jing-Jang Hwang, Yuan-Chien Chow, Frank D. Angers, Chung-Yee Lee; Scheduling Precedence Graphs in Systems with Inter-processor Communication Times, *Siam J. Comput.*, Vol. 18, No. 2, April 1989, pp. 244-257.
- [9] E. Laskowski, Program Graph Scheduling in the Look-Ahead Reconfigurable Multiprocessor System, *Proceedings of the PARELEC 2000*, Trois Rivieres, Canada, August 2000, IEEE CS Press, pp. 106-110.
- [10] E. Laskowski, New Program Structuring Heuristics for Multi-Processor Systems with Redundant Communication Resources, *Proc. of the PARELEC 2002*, Warsaw, Poland, September 2002, IEEE CS Press, pp. 183-188.
- [11] B. Wilkinson, M. Allen, *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall 1999, p. 327.

Arbitrating Instructions in an $\rho\mu$ -Coded CCM

Georgi Kuzmanov and Stamatios Vassiliadis

Computer Engineering Lab, Electrical Engineering Dept.,

EEMCS, TU Delft, The Netherlands,

{G.Kuzmanov,S.Vassiliadis}@ET.TUDelft.NL, <http://ce.et.tudelft.nl/>

Abstract. In this paper, the design aspects of instruction arbitration in an $\rho\mu$ -coded CCM are discussed. Software considerations, architectural solutions, implementation issues and functional testing of an $\rho\mu$ -code arbiter are presented. A complete design of such an arbiter is proposed and its VHDL code is synthesized for the VirtexII Pro platform FPGA of Xilinx. The functionality of the unit is verified by simulations. A very low utilization of available reconfigurable resources is achieved after the design is synthesized. Simulations of an MPEG-4 case study suggest considerable performance speed-up in the range of 2,4-8,8 versus a pure software PowerPC implementation.

1 Introduction

Numerous design concepts and organizations have been proposed to support the Custom Computing Machine (CCM) paradigm from different prospectives [6, 7]. An example of a detailed classification of CCMs can be found in [8]. In this paper we propose a design of a potentially performance limiting unit of the MOLEN $\rho\mu$ -coded CCM: the arbiter [11]. We discuss all design aspects of the arbiter, including software considerations, architectural solutions, implementation issues and functional testing. A synthesizable VHDL code of the arbiter has been developed and simulated. Performance has been evaluated theoretically and by experimentation using Virtex II Pro technology of Xilinx. Synthesis results and performance evaluation for an MPEG-4 case study suggest:

- Less than 1% of the reconfigurable hardware resources available on the selected FPGA (xc2vp20) chip are spent for the implementation of the arbiter.
- Considerable speed-ups in the range of 2,4-8,8 of the MPEG-4 encoder are feasible when the SAD function is implemented in the proposed framework.

The remainder of this paper is organized as follows. The section to follow contains brief background on the MOLEN $\rho\mu$ -coded processor and describes the requirements to the design of a general arbiter. In Section 3, software-hardware considerations for a particular arbiter design for PowerPC and Virtex II Pro FPGA are presented. Section 4 discusses functional testing, performance analysis and an MPEG-4 case study with experimental results obtained from a real chip implementation of the arbiter. Finally, we conclude the paper with Section 5.

2 Background

This section presents the MOLEN $\rho\mu$ -coded Custom Computing Machine organization, introduced in [11] and described in Fig. 1. The ARBITER performs a partial decoding on the instructions in order to determine where they should be issued. Instructions implemented in fixed hardware are issued to the core processor. Instructions for custom execution are redirected to the *reconfigurable unit*. The reconfigurable unit consists of a custom computing unit (CCU) and the $\rho\mu$ -code unit. An operation, executed by the reconfigurable unit, is divided into two distinct phases: **set** and **execute**. The **set** phase is responsible for reconfiguring the CCU hardware enabling the execution of the operation. This phase may be divided into two subphases - partial set (**pset**) and complete set (**cset**). In the **pset** phase the CCU is partially configured to perform common functions of an application (or group of applications). Later, the **cset** sub-phase only reconfigures that blocks in the CCU, which are not covered in the **pset** sub-phase in order to *complete* the functionality of the CCU.

General Requirements to the Arbiter. The arbiter controls the proper co-processing of the GPP and the reconfigurable units. It is closely connected to three major units of the CCM, namely the GPP, the memory and the $\rho\mu$ -unit. Each of these parts of the organization has its own requirements, which should be considered when an arbiter is designed. Regarding the core processor, the arbiter should: 1) Preserve the original behavior of the core processor when no reconfigurable instruction is executed. Create the shortest possible critical path penalties. 2) Emulate reconfigurable instruction execution behavior on the core processor using its original instruction set and/or other architectural features.

Regarding the $\rho\mu$ -unit the arbiter should: 1) Distribute control signals and the starting microcode address to the $\rho\mu$ -unit. 2) Consume minimal hardware resources if implemented in the same FPGA with the $\rho\mu$ -unit. Thus more resources will be available for the CCU.

For proper memory management the arbiter should be designed to: 1) Arbitrate the data access between the $\rho\mu$ -unit and the core processor. 2) Allow

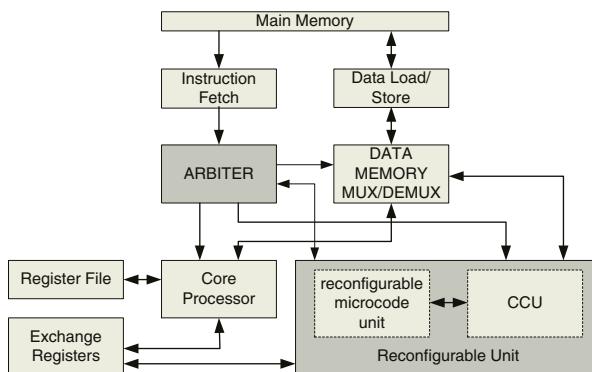


Fig. 1. The MOLEN machine organization

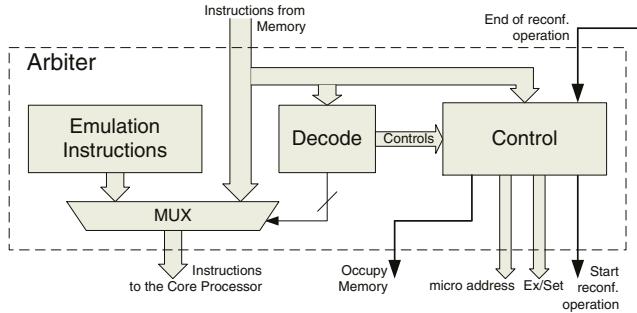


Fig. 2. General Arbiter organization

speeds within the capabilities of the utilized memory technology, i.e., not creating performance bottlenecks in memory transfers.

Reconfigurable instructions should be encoded in consistence with the instruction encoding of the targeted general purpose architecture. The arbiter should also provide proper timing for reconfigurable instruction execution to all units referred above. In Fig.2, a general view of an Arbiter organization is presented. The operation of such an arbiter is entirely based on decoding the input instruction flow. The unit either redirects instructions, or generates an instruction sequence to control the state of the core processor during reconfigurable operations. In such an organization, the critical path penalty to the original instruction flow can be reduced to just one 2-1 multiplexer. Once either of the three reconfigurable instructions has been decoded, the following actions are initiated:

1. Emulation instructions are multiplexed to the processor instruction bus. These instructions drive the processor into wait or halt state.
2. Control signals from the decoder are generated to the control block in Fig.2. Based on these controls, the control block performs the following: a) Redirects the microcode location address of the corresponding reconfigurable instruction to the $\rho\mu$ -unit. b) Generates an internal code signal (Ex/Set) for the decoded reconfigurable instruction and delivers it to the $\rho\mu$ -unit. c) Initiates reconfigurable operation by generating '*start reconf. operation*' signal to the $\rho\mu$ -unit. d) Reserves the data memory control for the $\rho\mu$ -unit by generating *memory occupy* signal to the (data) memory controller. e) Enters a wait state until signal '*end of reconf. operation*' arrives.

An active '*end of reconf. operation*' signal initiates the following actions: 1) Data memory control is released back to the core processor. 2) An instruction sequence is generated to ensure proper exiting of the core processor from the wait state. 3) After exiting the wait state, the program flow control is transferred to the instruction immediately after the reconfigurable instruction, executed last.

3 CCM Implementation

The general organization presented in the previous section has been implemented on Virtex II Pro, a platform FPGA chip of Xilinx.

Software Considerations: Because of performance reasons, we decided not to use PowerPC special operating modes instructions (exiting special operating modes is usually performed by interrupt). We employed the '*branch to link register*' (**blr**) to emulate a wait state and '*branch to link register and link*' (**blrl**) to get the processor out of this state. The difference between these instructions is that **blrl** modifies the link register (LR), while **blr** does not. The next instruction address is the effective address of the branch target, stored in LR. When **blrl** is executed, the new value loaded into LR is the address of the instruction following the branch instruction. Thus the emulation instructions, stored into the corresponding block in Fig.2 are reduced to only one instruction for wait and one for 'wake-up' emulation.

Let us assume the following mnemonics for the three reconfigurable instructions: '*pset rm_addr*', '*cset rm_addr*' and '*exec rm_addr*'. To implement the proposed mechanism, we only need to initialize LR with a proper value, i.e. the address of the reconfigurable instruction. This should be done by the compiler with the '*branch and link*' (**bl**) instruction of PowerPC. In the assembly code of the application program the '*complete set*' instruction should look like this:

bl *label1* \rightarrow **bl** — branch to *label1* ; LR = *label1*

label1: **cset** *rm_addr* → *blr* — branch to *label1* ; LR = *label1*

Obviously, the processor will execute branch instruction to the same address, because LR remains unchanged and points to an address containing **blr** instruction. Thus we drive the processor into an eternal loop. It is the responsibility of the arbiter to get the processor out of this state. When the reconfigurable instruction is complete, an '*end_op*' signal is generated by the μ -unit to the arbiter, which initiates the execution of **blrl** exactly twice. Thus, the effective address of the next instruction is loaded into the LR, which points to the address of the instruction immediately following the reconfigurable one and the processor exits the eternal loop. Below, the instructions generated by the arbiter to finalize a reconfigurable operation are displayed (instruction alignment is at 4 bytes):

label1: cset rm_addr → blrl — branch to *label1* ; LR = *label1+4*
→ blrl — branch to *label1+4* ; LR = *label1+4*

label1+4: next instr → — next instruction ; LR = *label1+4*

This approach allows executions of *blocks of reconfigurable instructions* (BRI): We define **BRI** as any sequence of reconfigurable instructions starting with the instruction ‘bl’ and containing arbitrary number of consecutive reconfigurable instructions. No other instructions can be utilized within a BRI. Utilizing BRI saves the necessity to initialize LR every time a reconfigurable instruction is invoked, thus saving a couple of **bl** instructions. In this scheme only one **bl** instruction is used to initialize LR in the beginning of the BRI. The time spent for executing a single reconfigurable operation (T_ρ) is estimated to be the time for the *reconfigurable execution* ($T_{\rho E}$), consumed by the μ -unit, plus the time for

three *unconditional taken branch instructions* (T_{UTB}) : $T_\rho = 3 \times T_{UTB} + T_{\rho E}$. Assuming the number of reconfigurable instructions in the BRI to be N_{BRI} , the execution time of a reconfigurable instruction within a BRI costs: $T_\rho = 2 \times T_{UTB} + T_{\rho E} + \frac{T_{UTB}}{N_{BRI}}$. In other words, the *time penalty for single reconfigurable instruction execution* is $3 \times T_{UTB}$ and within a BRI execution - between $2 \times T_{UTB}$ and $3 \times T_{UTB}$. Optionally, the ‘*instruction synchronization*’ instruction (**isync**) can be added before a BRI to avoid out-of-order executions of previous instructions during reconfigurable operation.

Instruction Encoding. To perform the MOLEN processor reconfigurations, the PowerPC Instruction Set Architecture (ISA) is extended with three instructions. To encode these three instructions, we have considered the following: 1.) The encoding scheme should be consistent with the PowerPC instruction format with opcodes (OPC) encoded in the six most-significant bits of the instruction word (see Fig.3). 2.) All three instructions have the same OPC field and same instruction form, which is similar to the I-form. Let us call the new form of the reconfigurable instructions *R-form*. 3.) The OPCodes of the instructions are as close as possible to the OPC of the emulation instructions (shortest Hamming distance), i.e. **blr** and **blrl**. From the free opcodes of the PowerPC architecture, such is opcode ‘6’ (‘000110_b’). 4.) Instruction modifiers are implemented in the two least-significant fields of the instruction word, to distinguish the three reconfigurable instructions. 5.) A 24-bit address, embedded into the instruction word, specifies the location of the microcode in memory. A modifier bit R/P (Resident/Pageable), assumed to be a part of the address field, specifies where the microcode is located and how to interpret the address field. If R/P=1 a memory address is specified, otherwise an address of the on-chip storage in the $\rho\mu$ -code unit is referred. The address always points to the location of the first microcode instruction. This first address should contain the length or the final address of the microcode. A microprogram is terminated by an *end_op* microinstruction.



Fig. 3. Reconfigurable instruction encoding: R-form

Hardware Requirements. To implement the instruction bus arbitration, we have considered the following: 1) Information, related to instruction decoding, arbitration and timing is obtained only through the instruction bus. 2) PowerPC instruction bus is 64-bit wide and instructions are fetched in couples. 3) Speculative dummy prefetches should not disturb the correct timing of a reconfigurable instruction execution. 4) Both the arbiter and the $\rho\mu$ -unit strobe input signals on rising clock edges and generate output controls on falling clock edges.

The $\rho\mu$ -code arbiter for PowerPC has been described in synthesizable VHDL and mapped on the Virtex II Pro FPGA. Fig. 4 depicts the timing of this im-

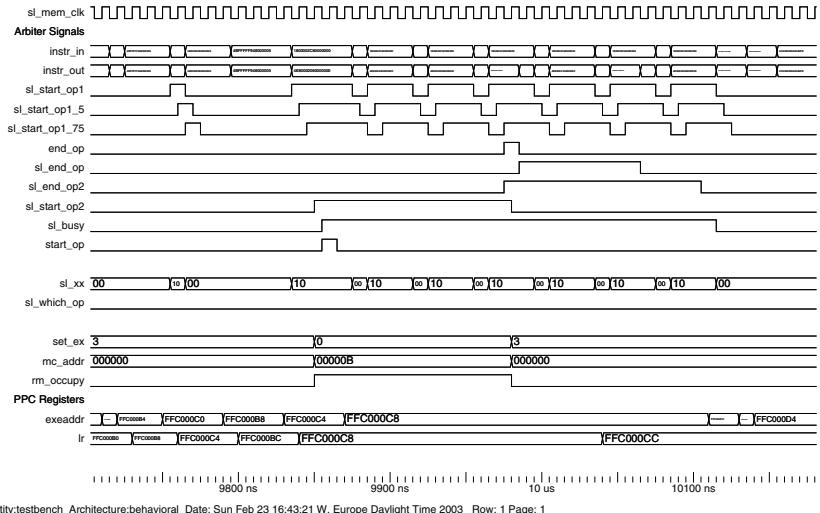


Fig. 4. Reconfigurable instruction execution timing

plementation. The unit uses the same clock ('`sl_mem_clk`') as the instruction memory, in this case a fast ZBT RAM. The only inputs of the arbiter are the *input instruction bus* ('`instr_in`') and *end of (reconfigurable) operation* ('`end_op`').

The *decode unit* of the arbiter (Fig. 2) decodes both OPCodes of the fetched instructions. Non-reconfigurable instructions are redirected (via MUX) to output '`instr_out`', directly driving the instruction bus of PowerPC. Alternatively, when either of the decoded two instructions is reconfigurable, the instruction code of `blr` is multiplexed via '`instr_out`' from the '*emulation instructions*' block. Obviously, the critical path penalty to the original instruction flow is just one 2-1 multiplexer and the decoding logic for a 6-bit value. The decode block generates two internal signals to the control block - `sl_start_op1` (explained later) and `sl_xx`. The latter signal indicates the alignment of the fetched instructions with respect to the reconfigurable ones. A one represents a reconfigurable instruction, a zero - any other instruction. For example, assuming big endian alignment: "`sl_xx=10`" means a reconfigurable instruction at the least-significant and a non-reconfigurable instruction at the most-significant address.

The *control block* generates signal *start (reconfigurable) operation* ('`start_op`') for one clock cycle delayed with two cycles after the moment a reconfigurable operation is prefetched and decoded, thus filtering short (dummy) prefetches. In Fig. 4 the rising edge of the internal signal `sl_start_op1` indicates the moment a reconfigurable operation is decoded. One can see that signal ('`start_op`') is generated only when the reconfigurable instruction is really fetched, i.e. when `sl_start_op1` takes longer than one clock cycle. Dummy prefetch filtration has been implemented by two flip-flops, connected in series and clocked by complementary clock edges. The outputs of these flip-flops are denoted by signals `sl_start_op1_5` and `sl_start_op1_75`. The output control to the $\rho\mu$ -unit, `sl_start_op` is generated between two falling clock edges.

Synchronously with the decoding of a reconfigurable instruction, the two instruction modifier fields (output signal *set_ex*) and *microcode address* (24-bit output *mc_addr*) are registered on rising clock edge (recall Fig.3). The internal flip-flop *sl_which_op* is used only when both of the fetched instructions are reconfigurable (*sl_xx*=“11”) to ensure the proper timely distribution of *set_ex*, *mc_addr* and controls. In addition, two internal signals (flip-flops) are set when reconfigurable instruction is decoded. These two signals denote that the $\rho\mu$ -unit is performing an operation (*sl_start_op2*) and that the arbiter is busy (*sl_busy*) with such an operation, therefore another reconfigurable execution can not be executed. To multiplex the data memory ports to the $\rho\mu$ -unit during reconfigurable operations, signal *rm_occupy* is driven to the data memory controller.

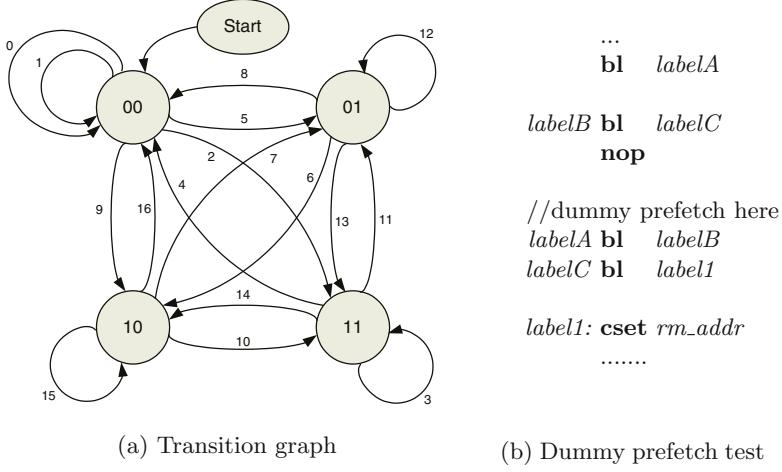
When a reconfigurable instruction is over, ‘*end_op*’ is generated by the $\rho\mu$ -unit and the *sl_start_op2* flip-flop is reset, thus releasing the data memory (via *rm_occupy*) for access by other units. Now, the control logic should guarantee that the **blrl** instruction is decoded exactly twice. This is done by a counter issuing active *sl_end_op* for precisely two **blrl** cycles, i.e., eight clocks. Instruction codes of **blr** and **blrl** differ only in one bit position. Therefore, redirecting *sl_end_op* via the MUX to this exact position of ‘*instr_out*’ while **blr** is issued, drives **blrl** to the PowerPC. When ‘*end_op*’ is strobed by the arbiter, another counter generates the *sl_end_op2* signal to prevent other reconfigurable operations from starting executions before the current reconfigurable operation has finished properly. The falling edge of signal *sl_end_op2* synchronously resets signal *busy*, thus enabling the execution of reconfigurable operations coming next.

4 Arbiter Testing, Analysis, and Case Study

Testing. To test the operation of the arbiter, we need a program, strictly aligned into memory, which tests all possible sequences of instruction couple alignments. Fig. 5(a) depicts the transition graph of such a test sequence, where a bubble represents an instruction couple alignment (1=reconfigurable instruction, 0 = other instruction). Arrows (transitions) fetch the next aligned instruction couple. Minimum 16 transitions cover all possible situations. The number next to each arrow indicates its position in the program sequence. An extra transition (arrow 0) tests the dummy prefetch filtration. Fig. 5(b) depicts its assembly code.

Performance Analysis. Let us assume T to be the execution time of the original program (say measured in cycles) and T_{SEi} - time to execute kernel i in software, which we would like to speed-up in reconfigurable hardware. With respect to T_ρ from Section 3, $T_{\rho i}$ is the execution time for the reconfigurable implementation of kernel i . Assuming $a_i = \frac{T_{SEi}}{T}$ and $s_i = \frac{T_{SEi}}{T_{\rho i}}$, the speed-up of the program with respect to the reconfigurable implementation of kernel i is:

$$S_i = \frac{T}{T - T_{SEi} + T_{\rho i}} = \frac{1}{1 - (a_i - \frac{a_i}{s_i})} \quad (1)$$



Identically, assuming $a = \sum_i a_i$, all kernels potential candidates for reconfigurable implementation would speed-up the program with:

$$S = \frac{T}{T - \sum_i T_{SEi} + \sum_i T_{\rho i}} = \frac{1}{1 - (a - \sum_i \frac{a_i}{s_i})}, \quad S_{max} = \lim_{\forall s_i \rightarrow \infty} S = \frac{1}{1-a} \quad (2)$$

Where S_{max} is the theoretical maximum speed-up. Parameters a_i may be obtained by profiling the targeted program, or along with s_i , by running an application on the real MOLEN CCM. Further in this Section, we will use (1) and (2) to compare theoretical to actual experimental speed-up.

Experimental Testbench. To prove the benefits of the proposed design we followed an experimental scenario. First, we use *profiling data* for the application to *extract computationally demanding kernels*. Second, we *design hardware engines*, which implement these kernels in performance efficient hardware. Further, we go through the following steps, to get experimental data for analysis:

1. Describe the MOLEN organization and the hardware kernel designs in VHDL and synthesize them for the selected target FPGA technology.
 2. Simulate the pure software implementations of the kernels on a VHDL model of the core processor to obtain performance figures.
 3. Simulate the hardware implementations of the same kernels, embedded in the MOLEN organization, and mapped on the target FPGA (i.e., VirtexIIPro).
 4. Estimate the speed-ups of each kernel (s_i) and for the entire application (S_i, S), based on data from the previous steps and the initial profiling.
 5. Download the FPGA programming stream into a real chip and run the application, to validate the figures from simulations.

Performance Speed-Up: An MPEG-4 Case Study. The application domain of interest in our experimentations is the visual data compression and in particular the MPEG-4 standard. For parameter a_i , we use profiling data

Table 1. Speed-up for pageable $\rho\mu$ -code, fixed $\rho\mu$ -code, and theoretical maximum.

MPEG-4 SAD	$T_{SEi} = 3404[\text{cyc}]$		
	Pag.	Fixed	Theor.
$T_{\rho i}, [\text{cyc}]$	87	51	-
s_i	39	67	∞
$S_i, (a_i = 0, 6)$	2,41	2,45	2,50
$S_i, (a_i = 0, 66)[3], [4]$	2,80	2,86	2,94
$S_i, (a_i = 0, 88)[2], [10]$	7,01	7,51	8,33
$S_i, (a_i = 0, 90)[1]$	8,13	8,82	10

Table 2. Synthesis Results for xc2vp20, Speed Grade -5

Slices	84 of 10304	< 1%
Flip Flops	69 of 20608	< 1%
4 input LUTs	150 of 20608	< 1%
Clock period	7.004ns	
Frequency	142.776MHz	

reported in literature [1–4, 10]. Values of some “global” parameters (a_i) regarding overall MPEG-4 performance may be within a standard deviation of 20% [3], with respect to the particular data. On the other hand, “local” parameters regarding implemented kernels ($T_{SEi}, T_{\rho i}, s_i$) are less data dependent, thus more predictable (accuracy within 5%). Table 1 contains experimental results for the implementation of the most demanding function in MPEG-4 encoder, the Sum-of-Absolute Differences (SAD), utilizing a design, described in [9] and assuming memory addressing schemes, discussed in [5]. The SAD kernel takes 3404 PowerPC cycles to execute in pure software. For its reconfigurable execution in MOLEN, we run two scenarios: a)worst case, when SAD execution microcode address is pageable and not residing in the $\rho\mu$ -unit; and best case, when the microcode is fixed into the $\rho\mu$ -unit. Experimental results in Table 1 strongly suggest that considerable speed-up of MPEG-4 encoders in the range of 2,41-8,82 is achievable only by implementing the SAD function as CCU in the MOLEN CCM organization. Both experimental and theoretical results indicate that for great kernel speed-ups ($s_i \gg 1$), the difference in overall performance (S_i) between worst and best case (pageable and fixed μ -code) is diminishing.

FPGA Synthesis Results. The VHDL code of the arbiter has been simulated with Modeltech’s ModelSim and synthesized with Project Navigator ISE 5.1 S3 of Xilinx. The target FPGA chip was XC2VP20. Hardware costs obtained by the synthesis tools are reported in Table 2. Post-place-and-route results indicate the total number of slices to be 80 and memory clock frequency of 100 MHz to be feasible. These results strongly suggest that at trivial hardware costs the $\rho\mu$ -arbiter design can arbitrate the PowerPC instruction bus without causing severe critical path penalties and frequency decreases. Moreover, virtually all reconfigurable resources of the FPGA remain available for CCUs. Regarding the total number of flip-flops in the arbiter design (69), most of them (52) are used for registering mc_addr and set_ex outputs. Thus only 17 flip-flops are spent for the control block, including the two embedded counters (2×4 flip-flops).

5 Conclusions

In this paper, we proposed an efficient design of a potentially performance limiting unit of an $\rho\mu$ -coded CCM: the arbiter. The general $\rho\mu$ -coded machine orga-

nization MOLEN was implemented on the platform FPGA Virtex II Pro and the arbitration between reconfigurable and fixed PowerPC instructions investigated. All design aspects of the arbiter have been described, including software considerations, architectural solutions, implementation issues and functional testing. Performance has been evaluated analytically and by experimentation. Synthesis results indicate trivial hardware costs for an FPGA implementation. Simulations suggest that considerable speed-ups (in the range of 2,4-8,8) of an MPEG-4 case study are feasible when the SAD function is implemented in the proposed framework. The presented design will be implemented on an FPGA prototyping board.

Acknowledgements

This research is supported by PROGRESS, the embedded systems research program of the Dutch scientific organization NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW.

References

1. H.-C. Chang, L.-G. Chen, M.-Y. Hsu, and Y.-C. Chang. Performance analysis and architecture evaluation of MPEG-4 video codec system. In *IEEE International Symposium on Circuits and Systems*, vol. II, pp. 449–452, 28-31 May 2000.
2. H.-C. Chang, Y.-C. Wang, M.-Y. Hsu, and L.-G. Chen. Efficient algorithms and architectures for MPEG-4 object-based video coding. In *IEEE Workshop on Signal Processing Systems*, pp. 13–22, 11-13 Oct 2000.
3. J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann. The MPEG-4 video coding standard - a VLSI point of view. In *IEEE Workshop on Signal Processing Systems,(SIPS98)*, pp. 43–52, 8-10 Oct. 1998.
4. P. Kuhn and W. Stechel. Complexity analysis of the emerging MPEG-4 standard as a basis for VLSI implementation. In *SPIE Visual Communications and Image Processing (VCIP)*, vol. 3309, pp. 498–509, Jan. 1998.
5. G. Kuzmanov, S. Vassiliadis, and J. van Eijndhoven. A 2D Addressing Mode for Multimedia Applications. In *SAMOS 2001*, vol. 2268 of *Lecture Notes in Computer Science*, pp. 291–306, July 2001. Springer-Verlag.
6. M.Wazlowski, L.Agarwal, T.Lee, A.Smith, E.Lam, H.Silverman, and S.Ghosh. PRISM-II Compiler and Architecture. In *Proc.IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 9–16, 5-7 April, 1993.
7. R.W.Hartenstein, R.Kress, and H.Reining. A new FPGA Architecture for Word-Oriented Datapaths. In *FPL 1994*, pp. 144–155, 1994.
8. M. Sima, S. Vassiliadis, S. Cotofana, J. T. van Eijndhoven, and K. Vissers. Field-Programmable Custom Computing Machines. A Taxonomy. In *FPL 2002.*, vol. 2438 of *Lecture Notes in Computer Science*, pp. 79–88, Sept. 2002. Springer-Verlag.
9. S. Vassiliadis, E. Hakkenes, J. Wong, and G. Pechaneck. The Sum Absolute Difference Motion Estimation Accelerator. In *EUROMICRO 98*, vol. 2, Aug. 1998.
10. S. Vassiliadis, G. Kuzmanov, and S. Wong. MPEG-4 and the New Multimedia Architectural Challenges. In *15th SAER'2001*, 21-23 Sept. 2001.
11. S. Vassiliadis, S. Wong, and S. Cotofana. The MOLEN $\rho\mu$ -coded processor. In *FPL 2001*, pp. 275–285, Aug. 2001.

How Secure Are FPGAs in Cryptographic Applications?*

Thomas Wollinger and Christof Paar

Chair for Communication Security (COSY), Horst Görtz Institute for IT Security,
Ruhr-Universität Bochum, Germany,
`{wollinger,cpaar}@crypto.rub.de`

Abstract. The use of FPGAs for cryptographic applications is highly attractive for a variety of reasons but at the same time there are many open issues related to the general security of FPGAs. This contribution attempts to provide a state-of-the-art description of this topic. First, the advantages of reconfigurable hardware for cryptographic applications are listed. Second, potential security problems of FPGAs are described in detail, followed by a proposal of a some countermeasure. Third, a list of open research problems is provided. Even though there have been many contributions dealing with the algorithmic aspects of cryptographic schemes implemented on FPGAs, this contribution appears to be the first comprehensive treatment of system and security aspects.

Keywords: cryptography, FPGA, security, attacks, reconfigurable hardware

1 Introduction

The choice of the implementation platform of a digital system is driven by many criteria and heavily dependent on the application area. In addition to the aspects of algorithm and system speed and costs — which are present in most other application domains too — there are crypto-specific ones: physical security (e.g., against key recovery and algorithm manipulation), flexibility (regarding algorithm parameter, keys, and the algorithm itself), power consumption (absolute usage and prevention of power analysis attacks), and other side channel leakages.

Reconfigurable hardware devices, such as Field Programmable Gate Arrays (FPGAs), seem to combine the advantages of SW and HW implementations. At the same time, there are still many open questions regarding FPGAs as a module for security functions. There has been a fair amount of work been done by the research community dealing with the algorithmic and computer architecture aspects of crypto schemes implemented on FPGAs since the mid-1990s (see, e.g., relevant articles in [KP99, KP00, KNP01, KKP02]), often focusing on high-performance implementations. At the same time, however, very little work has

* This research was partially sponsored by the German Federal Office for Information Security (BSI).

been done dealing with the system and physical aspects of FPGAs as they pertain to cryptographic applications. It should be noted that the main threat to a cryptographic scheme in the real world is *not* the cryptanalysis of the actual algorithm, but rather the exploration of weaknesses of the implementation. Given this fact, we hope that the contribution at hand is of interest to readers in academia, industry and government sectors.

In this paper we'll start in Section 2 with a list of the advantages of FPGAs in cryptographic applications from a systems perspective. Then, we highlight important questions pertaining to the *security* of FPGAs when used for crypto algorithms in Section 3. A major part of this contribution is a state-of-the-art perspective of security issues with respect to FPGAs, by illuminating this problem from different viewpoints and by trying to transfer problems and solutions from other hardware platforms to FPGAs (Section 4). In Section 5, we provide a list of open problems. Finally, we end this contribution with some conclusions. We would like to stress that this contribution is not based on any practical experiments, but on a careful analysis of available publications in the literature and on our experience with implementing crypto algorithms.

2 System Advantages of FPGAs for Cryptographic Applications

In this section we list the potential advantages of reconfigurable hardware (RCHW) in cryptographic applications. More details and a description to each item can be found in [EYCP01, WP03]. Note that the listed potential advantages of FPGAs for cryptographic applications can only be exploited if the security shortcomings of FPGAs discussed in the following have been addressed.

- Algorithm Agility
- Algorithm Upload
- Architecture Efficiency
- Resource Efficiency
- Algorithm Modification
- Throughput
- Cost Efficiency

3 Security Shortcomings of FPGAs

This section summarizes security problems produced by attacks against given FPGA implementations. First we would like to state what the possible goals of such attacks are.

3.1 Objectives of an Attacker

The most common threat against an implementation of cryptographic algorithm is to learn a confidential cryptographic key. Given that the algorithms applied are

publicly known in most commercial applications, knowledge of the key enables the attacker to decrypt future and past communication. Another threat is the one-to-one copy, or “cloning”, of a cryptographic algorithm *together* with its key. In some cases it can be enough to run the cloned application in decryption mode to decipher past and future communication. In other cases, execution of a certain cryptographic operation with a presumably secret key is in most applications the sole criteria which authenticates a communication party. An attacker who can perform the same function can masquerade as the attacked communication party. Yet another threat is given in applications where the cryptographic algorithms are proprietary e.g. pay-TV and government communication. In such scenarios it is already interesting for an attacker to reverse-engineer the encryption algorithm itself.

The discussion above assumes mostly that an attacker has physical access to the encryption device. We believe that in many scenarios such access can be assumed, either through outsiders or through dishonest insiders. In the following we discuss vulnerabilities of modern FPGAs against such attacks.

3.2 Black Box Attack

The classical method to reverse engineer a chip is the so called Black Box attack. The attacker inputs all possible combinations, while saving the corresponding outputs. The intruder is then able to extract the inner logic of the FPGA, with the help of the Karnaugh map or algorithms that simplify the resulting tables. This attack is only feasible if a small FPGA with explicit inputs and outputs is attacked and a lot of processor power is available.

3.3 Readback Attack

Readback is a feature that is provided for most FPGA families. This feature allows to read a configuration out of the FPGA for easy debugging. The idea of the attack is to read the configuration of the FPGA through the JTAG or programming interface in order to obtain secret information (e.g. keys) [Dip]. The readback functionality can be prevented with security bits provided by the manufacturers.

However, it is conceivable, that an attacker can overcome these countermeasures in FPGA with fault injection. This kind of attack was first introduced in [BDL97] and it was shown how to break public-key algorithms by exploiting hardware faults. It seems very likely that these attacks can be easily applied to FPGAs, since they are not especially targeted to ASICs. If this is in fact feasible, an attacker is able to deactivate security bits and/or the countermeasures, resulting in the ability to read out the configuration of the FPGA [Kes, Dip].

3.4 Cloning of SRAM FPGAs

In a standard scenario, the configuration data is stored (unprotected) externally in nonvolatile memory (e.g., PROM) and is transmitted to the FPGA at

power-up in order to configure the FPGA. An attacker could easily eavesdrop on the transmission and get the configuration file. This attack is therefore feasible for large organizations as well as for those with low budgets and modest sophistication.

3.5 Reverse-Engineering of the Bitstreams

The attacks described so far output the bitstream of the FPGA design. In order to get the design of proprietary algorithms or the secret keys, one has to reverse-engineer the bitstream. The condition to launch the attack is that the attacker has to be in possession of the (unencrypted) bitstream.

FPGA manufacturers claim that the security of the bitstream relies on the disclosure of the layout of the configuration data. This information will only be made available if a non-disclosure agreement is signed, which is, from a cryptographic point of view, an extremely insecure situation. This security-by-obscenity approach was broken at least ten years ago when the CAD software company NEOCad reverse-engineered a Xilinx FPGA [Sea]. Even though a big effort has to be made to reverse engineer the bitstream, for large organizations it is quite feasible. In terms of government organizations as attackers, it is also possible that they will get the information of the design methodology directly from the vendors or companies that signed NDAs.

3.6 Physical Attack

The aim of a physical attack is to investigate the chip design in order to get information about proprietary algorithms or to determine the secret keys by probing points inside the chip. Hence, this attack targets parts of the FPGA, which are not available through the normal I/O pins. This can potentially be achieved through visual inspections and by using tools such as optical microscopes and mechanical probes. However, FPGAs are becoming so complex that only with advanced methods, such as Focused Ion Beam (FIB) systems, one can launch such an attack. To our knowledge, there are no countermeasures to protect FPGAs against this form of physical threat. In the following, we will try to analyze the effort needed to physically attack FPGAs manufactured with different underlying technologies.

SRAM FPGAs: Unfortunately, there are no publications available that accomplished a physical attack against SRAM FPGAs. This kind of attack is only treated very superficially in a few articles, e.g. [Ric98]. In the related area of SRAM memory, however there has been a lot of effort by academia and industry to exploit this kind of attack [Gut96, Gut01, AK97, WKM⁺96, Sch98, SA93, KK99]. Due to the similarities in structure of the SRAM memory cell and the internal structure of the SRAM FPGA, it is most likely that the attacks can be employed in this setting.

Contrary to common wisdom, the SRAM memory cells do not entirely loose the contents when power is cut. The reason for these effects are rooted in the

physical properties of semiconductors (see [Gut01] for more details). The physical changes are caused mainly by three effects: electromigration, hot carriers, and ionic contamination. Most publications agree that device can be altered, if 1) threshold voltage has changed by 100mV or 2) there is a 10% change in transconductance, voltage or current.

One can attack SRAM memory cells using the access points provided by the manufactures. An extreme case of data recovery, was described in [AK97], where a cryptographic key was recovered without special equipment. “ I_{DDQ} testing” is one of the widely used methods to analyze SRAM cells and it is based on the analysis of the current usage [Gut01, WKM⁺96, Sch98]. Another possibilities for the attack are also to use the scan path that the IC manufacturers insert for test purposes or techniques like bond pad probing [Gut01].

When it becomes necessary to use access points that are not provided by the manufacturer, the layers of the chip have to be removed. Mechanical probing with tungsten wire with a radius of $0,1 - 0,2\mu m$ is the traditional way to discover the needed information. Focused Ion Beam (FIB) workstations can expose buried conductors and deposit new probe points [KK99]. Electron-beam tester (EBT) is another measurement method. EBT measures the energy and amount of secondary electrons that are reflected.

Resulting from the above discussion of attacks against SRAM memory cells, it seems likely that a physical attack against SRAM FPGAs can be launched successfully, assuming that the described techniques can be transferred. However, the physical attacks are quite costly and having the structure and the size of state-of-the-art FPGA in mind, the attack will probably only be possible for large organizations, for example intelligence services.

Antifuse FPGAs: In order to be able to detect the existence or non-existence of the connection one has to remove layer after layer, or/and use cross-sectioning. Unfortunately, no details have been published regarding this type of attack. In [Dip], the author states that a lot of trial-and-error is necessary to find the configuration of one cell and that it is likely that the rest of the chip will be destroyed, while analyzing one cell. The main problem with this analysis is that the isolation layer is much smaller than the whole AF cell. One study estimates that about 800,000 chips with the same configuration are necessary to explore one configuration file of an Actel A54SX16 chip with 24,000 system gates [Dip]. Another aggravation of the attack is that only about 2–5 % of all possible connections in an average design are actually used. In [Ric98] a practical attack against AF FPGAs was performed and it was possible to alter one cell in two months at a cost of \$1000.

Flash FPGAs: Flash FPGAs can be analyzed by placing the chip in a vacuum chamber and powering it up. Other possible attacks against flash FPGAs can be found in the related area of flash memory. The number of write/erase cycles are limited to $10,000 - 100,000$, because of the accumulation of electrons in the floating gate causing a gradual rise of the transistors threshold voltage. This fact increases the programming time and eventually disables the erasing of the cell [Gut01]. Another less common failure is the programming disturbance

in which unselected erased cells gain charge when adjacent selected cells are written [ASH⁺93, Gut01]. Furthermore, electron emission causes a net charge loss [PGP⁺91]. In addition, hot carrier effects build a tunnel between the bands [HCSL89]. Another phenomenon is overerasing, where an erase cycle is applied to an already-erased cell leaving the floating gate positively charged [Gut01].

All the described effects change in a more or less extensive way the cell threshold voltage, gate voltage, or the characteristic of the cell. We remark that the stated phenomena apply as well for EEPROM memory and that due to the structure of the FPGA cell these attacks can be simply adapted to attack flash/EEPROM FPGAs.

3.7 Side Channel Attacks

Any physical implementation of a cryptographic system might provide a *side channel* that leaks unwanted information. Examples for side channels include in particular: power consumption, timing behavior, and electromagnet radiation. Obviously, FPGA implementations are also vulnerable to these attacks. In [KJJ99] two practical attacks, Simple Power Analysis (SPA) and Differential Power Analysis (DPA) were introduced. Since their introduction, there has been a lot of work improving the original power attacks (see, e.g., relevant articles in [KP99, KP00, KNP01, KKP02]). There seems to be very little work at the time of writing addressing the feasibility of actual side channel attacks against FPGAs. However, it seems almost certain that the different side channels can be exploited in the case of FPGAs as well.

4 How to Prevent Possible Attacks?

This section shortly summarizes possible countermeasures that can be provided to minimize the effects of the attacks mentioned in the previous section. Most of them have to be realized by design changes through the FPGA manufacturers, but some could be applied during the programming phase of the FPGA.

Preventing the Black Box Attack: The Black Box Attack is not a real threat nowadays, due to the complexity of the designs and the size of state-of-the-art FPGAs. Furthermore, the nature of cryptographic algorithms prevents the attack as well. Today's stream ciphers output a bit stream, with a period length of 128 bits (e.g. w7). Block ciphers, like AES, are designed with a minimum key length of 128 bits. Minimum length in the case of public-key algorithms is 160 bits for elliptic curve cryptosystems and 1024 bits for discrete logarithm and RSA-based systems. It is widely believed that it is infeasible to perform a brute force attack and search a space with 2^{80} possibilities. Hence, implementations of this algorithm can not be attacked with the black box approach.

Preventing the Cloning of SRAM FPGAs: There are many suggestions to prevent the cloning of SRAM FPGAs, mainly motivated by the desire to prevent reverse engineering of general, i.e., non-cryptographic, FPGA designs. One solution would be to check the serial number before executing the design and

delete the circuit if it is not correct. Another solution would be to use dongles to protect the design (a survey on dongles can be found in [Kea01]). Both solutions do not provide the necessary security, see [WP03] for more details. A more realistic solution would be to have the nonvolatile memory and the FPGA in one chip or to combine both parts by covering them with epoxy. However, it has to be guaranteed that an attacker is not able to separate the parts.

Encryption of the configuration file is the most effective and practical countermeasure against the cloning of SRAM FPGAs. There are several patents that propose different encryption scenarios [Jef02, Aus95, Eri99, SW99, Alg] and a good number of publications, e.g. [YN00, KB00]. The 60RS family from Actel was the first attempt to have a key stored in the FPGA in order to be able to encrypt the configuration file. The problem was that every FPGA had the same key on board.

An approach in a completely different direction would be to power the whole SRAM FPGA with a battery, which would make transmission of the configuration file after a power loss unnecessary. This solution does not appear practical, however, because of the power consumption of FPGAs. Hence, a combination of encryption and battery power provides a possible solution. Xilinx addresses this with an on-chip 3DES decryption engine in its Virtex II [Xil] (see also [PWF⁺00]), where the two keys are stored in the battery powered memory.

Preventing the Physical Attack: To prevent physical attacks, one has to make sure that the retention effects of the cells are as small as possible, so that an attacker can not detect the status of the cells. Already after storing a value in a SRAM memory cell for 100–500 seconds, the access time and operation voltage will change [vdPK90]. The solution would be to invert the data stored periodically or to move the data around in memory. Neutralization of the retention effect can be achieved by applying an opposite current [TCH93] or by inserting dummy cycles into the circuit [Gut01]. In terms of FPGA application, it is very costly or even impractical to provide solutions like inverting the bits or changing the location for the whole configuration file. A possibility could be that this is done only for the crucial part of the design, like the secret keys. Counter techniques such as dummy cycles and opposite current approach can be carried forward to FPGA applications.

Antifuse FPGAs can only be protected against physical attack, by building a secure environment around them. If an attack was detected every cell should be programmed in order not to leak any information or the antifuse FPGA has to be destroyed.

In terms of flash/EEPROM memory cell, one has to consider that the first write/erase cycles causes a larger shift in the cell threshold [SKM95] and that this effect will become less noticeable after ten write/erase cycles [HCSL89]. Thus, one should program the FPGA about 100 times with random data, to avoid these effect (suggested for flash/EEPROM memory cells in [Gut01]). The phenomenon of overerasing flash/EEPROM cells can be minimized by first programming all cells before deleting them.

Preventing the Readback Attack: The readback attack can be prevented with the security bits set, as provided by the manufacturers, see Section 3.3. If one wants to make sure that an attacker is not able to apply fault injection, the FPGA has to be embedded into a secure environment, where after detection of an interference the whole configuration is deleted or the FPGA is destroyed.

Preventing the Side Channel Attack: In recent years, there has been a lot of work done to prevent side-channel attacks (see, e.g., relevant articles in [KP99, KP00, KNP01, KKP02]). There are “Software” countermeasures that refer primarily to algorithmic changes which are also applicable to implementations in FPGA. Furthermore, there are Hardware countermeasures that often deal either with some form of power trace smoothing or with transistor-level changes of the logic. Neither seem to be easily applicable to FPGAs without support from the manufacturers. However, some proposals such as duplicated architectures might work on today’s FPGAs.

5 Open Problems

At this point we would like to provide a list of open questions and problems regarding the security of FPGAs. If answered, such solutions would allow stand-alone FPGAs with much higher security assurance than currently available. A more detailed description to all points can be found in [WP03].

- Side channel attacks
- Fault injection
- Key management for configuration encryption
- Secure deletion
- Physical attacks

6 Conclusions

This contribution analyzed possible attack against the use of FPGA in security applications. For black box attacks, we stated that they are not feasible for state-of-the-art FPGAs. However, it seems very likely for an attacker to get the secret information stored in a FPGA, when combining readback attacks with fault injection. Cloning of SRAM FPGA and reverse engineering depend on the specifics of the system under attacked, and they will probably involve a lot of effort, but this does not seem entirely impossible. Physical attacks against FPGAs are very complex due to the physical properties of the semiconductors in the case of flash/SRAM/EEPROM FPGAs and the small size of AF cells. It appears that such attacks are even harder than analogous attacks against ASICs. Even though FPGA have different internal structures than ASICs with the same functionality, we believe that side-channel attacks against FPGAs, in particular power-analysis attacks, will be feasible too.

From the discussion above it may appear that FPGAs are currently out of question for security applications. We don’t think that this the right conclusion,

however. It should be noted that many commercial ASICs with cryptographic functionality are also vulnerable to attacks similar to the ones discussed here. A commonly taken approach to prevent these attacks is to put the ASIC in a secure environment.

References

- AK97. R.J. Anderson and M.G. Kuhn. Low Cost Attacks on Tamper Resistant Devices. In *5th International Workshop on Security Protocols*, pages 125–136. Springer-Verlag, 1997. LNCS 1361.
- Alg. Algotronix Ltd. Method and Apparatus for Secure Configuration of a Field Programmable Gate Array. PCT Patent Application PCT/GB00/04988.
- ASH⁺93. Seiichi Aritome, Riichiro Shirota, Gertjan Hemink, Tetsup Endoh, and Fujio Masuoka. Reliability Issues of Flash Memory Cells. *Proceedings of the IEEE*, 81(5):776–788, May 1993.
- Aus95. K. Austin. Data Security Arrangements for Semiconductor Programmable Devices. United States Patent, No. 5388157, 1995.
- BDL97. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *EUROCRYPT '97*, pages 37–51. Springer-Verlag, 1997. LNCS 1233.
- Dip. B. Dipert. Cunning circuits confound crooks. <http://www.e-insite.net/ednmag/contents/images/21df2.pdf>.
- Eri99. C. R. Erickson. Configuration Stream Encryption. United States Patent, No. 5970142, 1999.
- EYCP01. A. Elbirt, W. Yip, B. Chetwynd, and C. Paar. An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. *IEEE Transactions on VLSI Design*, 9(4):545–557, August 2001.
- Gut96. P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Sixth USENIX Security Symposium*, pages 77–90, July 22–25, 1996.
- Gut01. P. Gutmann. Data Remanence in Semiconductor Devices. In *10th USENIX Security Symposium*, pages 39–54, August 13–17, 2001.
- HCSL89. Sameer Haddad, Chi Chang, Balaji Swaminathan, and Jih Lien. Degradations due to hole trapping in flash memory cells. *IEEE Electron Device Letters*, 10(3):117–119, March 1989.
- Jef02. G. P. Jeffrey. Field programmable gate arrays. United States Patent, No. 6356637, 2002.
- KB00. S. H. Kelem and J. L. Burnham. System and Method for PLD Bitstream Encryption. United States Patent, No. 6118868, 2000.
- Kea01. T. Kean. Secure Configuration of Field Programmable Gate Arrays. In *FPL 2001*, pages 142–151. Springer-Verlag, 2001. LNCS 2147.
- Kes. D. Kessner. Copy Protection for SRAM based FPGA Designs. <http://www.free-ip.com/copyprotection.html>.
- KJJ99. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO '99*, pages 388–397. Springer-Verlag, 1999. LNCS 1666.
- KK99. O. Kommerling and M.G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *Smartcard '99*, pages 9–20, May 1999.

- KKP02. B. S. Kaliski, Jr., Ç. K. Koç, and C. Paar, editors. *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2002*, Berlin, Germany, August 13-15, 2002. Springer-Verlag. LNCS 2523.
- KNP01. Ç. K. Koç, D. Naccache, and C. Paar, editors. *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2001*, Berlin, Germany, May 13-16, 2001. Springer-Verlag. LNCS 2162.
- KP99. Ç. K. Koç and C. Paar, editors. *Workshop on Cryptographic Hardware and Embedded Systems — CHES'99*, Berlin, Germany, August 12-13, 1999. Springer-Verlag. LNCS 1717.
- KP00. Ç. K. Koç and C. Paar, editors. *Workshop on Cryptographic Hardware and Embedded Systems — CHES 2000*, Berlin, Germany, August 17-18, 2000. Springer-Verlag. LNCS 1965.
- PGP⁺91. C. Papadas, G. Ghibaudo, G. Pananakakis, C. Riva, P. Ghezzi, C. Gounelle, and P. Mortini. Retention characteristics of single-poly EEPROM cells. In *European Symposium on Reliability of Electron Devices, Failure Physics and Analysis*, page 517, October 1991.
- PWF⁺00. R. C. Pang, J. Wong, S. O. Frake, J. W. Sowards, V. M. Kondapalli, F. E. Goetting, S. M. Trimberger, and K. K. Rao. Nonvolatile/battery-backed key in PLD. United States Patent, No. 6366117, Nov. 28 2000.
- Ric98. G. Richard. Digital Signature Technology Aids IP Protection. In EETimes - News, 1998. <http://www.eetimes.com/news/98/1000news/digital.html>.
- SA93. J. Soden and R.E. Anderson. IC failure analysis: techniques and tools for quality and reliability improvement. *Proceedings of the IEEE*, 81(5):703–715, May 1993.
- Sch98. D.K. Schroder. *Semiconducor Material and Device Characterization*. John Wiley and Sons, 1998.
- Sea. G. Seamann. FPGA Bitstreams and Open Designs. <http://www.opencollector.org/>.
- SKM95. K.T. San, C. Kaya, and T.P. Ma. Effects of erase source bias on Flash EPROM device reliability. *IEEE Transactions on Electron Devices*, 42(1):150–159, January 1995.
- SW99. C. Sung and B. I. Wang. Method and Apparatus for Securing Programming Data of Programmable Logic Device. United States Patent, Patent Number 5970142, June 22 1999.
- TCH93. Jiang Tao, Nathan Cheung, and Chenming Ho. Metal Electromigration Damage Healing Under Bidirectional Current Stress. *IEEE Transactions on Elecron Devices*, 14(12):554–556, December 1993.
- vdPK90. J. van der Pol and J. Koomen. Relation between the hot carrier lifetime of transistors and CMOS SRAM products. In *IRPS 1990*, page 178, 1990.
- WKM⁺96. T.W. Williams, R. Kapur, M.R. Mercer, R.H. Dennard, and W. Maly. IDDQ Testing for High Performance CMOS - The Next Ten Years. In *ED&TC'96*, pages 578–583, 1996.
- WP03. T. Wollinger and C. Paar. How Secure Are FPGAs in Cryptographic Applications? (Long Version). Report 2003/119, IACR, 2003. <http://eprint.iacr.org/>.
- Xil. Xilinx Inc. Using Bitstream Encryption. Handbook of the Virtex II Platform. <http://www.xilinx.com>.
- YN00. Kun-Wah Yip and Tung-Sang Ng. Partial-Encryption Technique for Intellectual Property Protection of FPGA-based Products. *IEEE Transactions on Consumer Electronics*, 46(1):183–190, 2000.

FPGA Implementations of the RC6 Block Cipher

Jean-Luc Beuchat

Laboratoire de l'Informatique du Parallelisme, Ecole Normale Supérieure de Lyon,
46, Allée d'Italie, F-69364 Lyon Cedex 07,
Jean-Luc.Beuchat@ens-lyon.fr

Abstract. RC6 is a symmetric-key algorithm which encrypts 128-bit plaintext blocks to 128-bit ciphertext blocks. The encryption process involves four operations: integer addition modulo 2^w , bitwise exclusive or of two w -bit words, rotation to the left, and computation of $f(X) = (X(2X + 1)) \bmod 2^w$, which is the critical arithmetic operation of this block cipher. In this paper, we investigate and compare four implementations of the $f(X)$ operator on Virtex-E and Virtex-II devices. Our experiments show that the choice of an algorithm is strongly related to the target FPGA family. We also describe several architectures of a RC6 processor designed for feedback or non-feedback chaining modes. Our fastest implementation achieves a throughput of 15.2 Gb/s on a Xilinx XC2V3000-6 device.

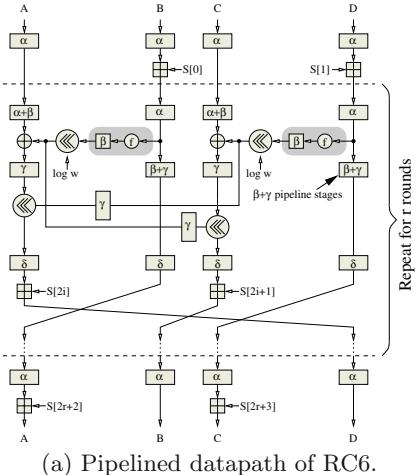
1 Introduction

In 1997, the National Institute of Standards and Technology (NIST) initiated a process to specify a new symmetric-key encryption algorithm capable of protecting sensitive data. RSA Laboratories submitted RC6 [9] as a candidate for this Advanced Encryption Standard (AES). NIST announced fifteen AES candidates at the First AES Candidate Conference (August 1998) and solicited public comments to select five finalist algorithms (August 1999): MARS, RC6, Rijndael, Serpent, and Twofish. Though the algorithm Rijndael was eventually selected, RC6 remains a good choice for security applications and is also a candidate for the NP 18033 project (via the Swedish ISO/IEC JTC 1/SC 27 member body¹) and the Cryptrec project initiated by the Information-technology Promotion Agency in Japan².

A version of RC6 is more exactly specified as RC6- $w/r/b$, where the parameters w , r , and b respectively express the word size (in bits), the number of rounds, and the size of the encryption key (in bytes). Since all actual implementations are targeted at $w = 32$ and $r = 20$, we use RC6 as shorthand to refer to RC6-32/20/b. A key schedule generates $2r + 4$ words (w bits each) from the b -bytes key provided by the user (see [9] for details). These values (called round

¹ <http://www.din.de/ni/sc27>

² <http://www.ipa.go.jp/security/enc/CRYPTREC/index-e.html>



```

entity rc6_f is
port (
  D : in std_logic_vector (31 downto 0);
  Q : out std_logic_vector (31 downto 0));
end rc6_f;
architecture behavioral of rc6_f is
  signal d0 : std_logic_vector (31 downto 0);
  signal d1 : std_logic_vector (31 downto 0);
  signal p : std_logic_vector (63 downto 0);
begin
  begin behavioral
    d0 <= D;
    d1 <= D (30 downto 0) & '1';
    p <= d0 * d1;
    Q <= p (31 downto 0);
  end behavioral;
end;

```

(b) Straightforward implementation of $f(X)$.**Fig. 1.** Encryption with RC6.

keys) are stored in an array $S[0, \dots, 2r+3]$ and are used in both encryption and decryption. The encryption algorithm involves four operations (Figure 1a):

- Integer addition modulo 2^w (denoted by $X \boxplus Y$).
- Bitwise exclusive or of two w -bit words (denoted by $X \oplus Y$).
- Computation of $f(X) = (X(2X + 1)) \bmod 2^w$, where X is a w -bit integer.
- Rotation of the w -bit word X to the left by an amount given by the $\log_2 w$ least significant bits of Y (denoted by $X \lll Y$).

Note that the decryption process requires moreover integer subtraction modulo 2^w and rotation to the right. As the algorithm is similar to encryption, we will not consider it here.

In this paper, we study several hardware architectures of RC6 using Virtex-E and Virtex-II field programmable gate arrays (FPGA). Virtex-E and Virtex-II Configurable Logic Blocks (CLB) provide functional elements for synchronous and combinational logic. Each CLB includes respectively two (Virtex-E) and four (Virtex-II) slices containing basically two 4-input look-up tables (LUT), two storage elements, fast carry logic dedicated to addition and subtraction, and two dedicated AND gates (referred to as MULT_AND) which improve the efficiency of multiplier implementation. Furthermore, Virtex-II devices embed many 18-bit \times 18-bit multiplier blocks (also referred to as MULT18x18 blocks) supporting two independent dynamic data input ports: 18-bit signed or 17-bit unsigned. Arithmetic operators dedicated to FPGAs should therefore involve such building blocks.

This paper is organized as follows: Section 2 describes several architectures of a RC6 processor. We then investigate various implementations of $f(X)$ and show that the choice of an algorithm depends on the target FPGA family (Section 3). Finally, Section 4 digests our main results and compare them with recent works on RC6.

2 Architecture of a RC6 Processor

RC6 encrypts plaintext in fixed-size 128-bit blocks. However, messages will often exceed 128 bits and a simple solution, known as Electronic Codebook (ECB) mode, consists in partitioning the plaintext into 128-bit blocks and encrypting each independently. This ECB mode has a major drawback in that identical ciphertext blocks imply identical plaintext blocks and is therefore inadvisable if the secret key is reused for more than one message. More sophisticated chaining modes bring a solution to this problem. For instance, in the Cipher Block Chaining (CBC) mode, a feedback mechanism causes the j th ciphertext block to depend on the first j plaintext blocks and an n -bit initialization vector. Since the entire dependency on preceding blocks is contained in the previous ciphertext block [6], all blocks must be processed sequentially (CBC decryption can however be performed in parallel). This property forbids to pipeline the computation path and implies a slightly different hardware architecture of the block cipher with a lower throughput. The counter (CTR) mode, a non-feedback mode described for example in [3], could remedy the situation if it becomes a standard as recommended in [5]. It is also possible to pipeline the processor in feedback modes if we accept the decomposition of the data stream into d separately encrypted messages, where d is the pipeline depth [11]. Also note that RC6 involves forty 32-bit \times 32-bit unsigned multipliers. The implementation of the 20 rounds is therefore only possible on rather large and expensive FPGAs.

Consequently, the hardware architecture of a RC6 processor depends as well on the required chaining mode and the target FPGA. We adopt here a design methodology initially proposed for the hardware implementation of the IDEA block cipher [7, 11]: the simplest RC6 processor contains a single round, the input round, and the output round (Figure 2a). This architecture is tailored to feedback chaining modes: a single plaintext block is encrypted at a time and we can provide a new input block after 21 clock cycles.

Assume now that a non-feedback chaining mode is required or that the data stream is decomposed into several separately encrypted messages. In order to shorten the critical path, each round has a parametric number of internal pipeline stages (parameters α , β , γ , and δ on Figure 1a). Figure 2b depicts an iterative architecture with partial loop unrolling and pipelining. The circuit implements k rounds (k is an integer divisor of the total number of rounds r), the input round, and the output round. Finally, Figure 2c illustrates an architecture with full loop unrolling dedicated to high throughput implementations of the RC6 block cipher.

In addition to the RC6 computation path, each processor contains a subkey memory implemented on CLBs and a control unit. The latter simply consists in a token associated with each plaintext block. This token indicates the validity of the data and selects the correct subkeys in iterative architectures. We have written a C program which generates a structural VHDL description of such a RC6 processor according to several parameters (partial or full loop unrolling, inner-round pipeline stages, and outer-round pipeline stages). Some examples are freely available at <http://www.ens-lyon.fr/~jlbeucha>.

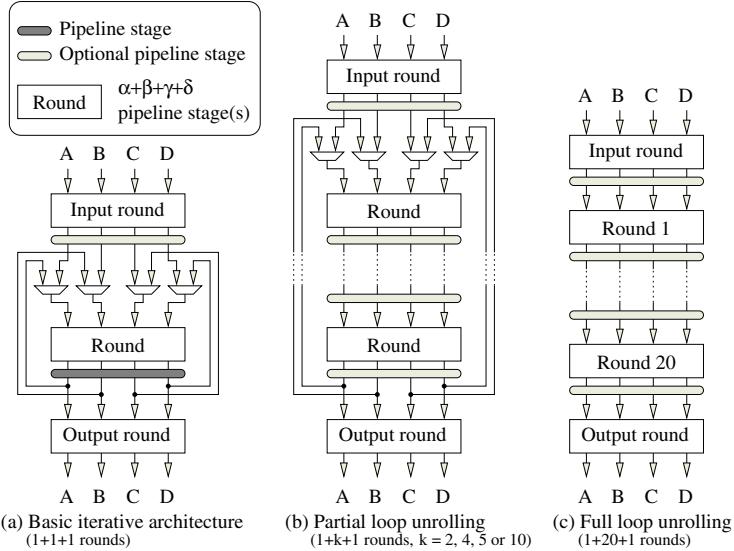


Fig. 2. Some architectures of a RC6 processor.

3 Computation of $f(X)$

The computation of $f(X)$ is the critical arithmetic operation of the block cipher. Therefore, both area and delay of a RC6 processor are closely related to the hardware operator carrying out $f(X) = (X(2X + 1)) \bmod 2^w$. In this section, we investigate and compare a method involving an array of AND gates and carry-propagate adders (CPA), and three algorithms dedicated to FPGA embedding small multiplier blocks. In the following, $X_{q:p}$ denotes $\sum_{i=p}^q x_i 2^i$.

3.1 Adder-Based Algorithm

This first algorithm is based on a standard method for squaring described for instance in [8]. Let us consider the problem of computing $f(X)$ when X is a 8-bit unsigned integer. As shown in Figure 3, the partial products can be significantly simplified before performing their addition according to the identities $x_i x_i = x_i$ and $x_i x_j + x_j x_i = 2x_i x_j$. Finally, based on the well-known relation $x_i x_j + x_i = 2x_i x_j + x_i \bar{x}_j$, we remove $x_3 x_2$ and x_2 from the leftmost column and replace them by $x_3 \bar{x}_2$. As $f(X)$ is computed modulo 2^8 , we ignore the term $2x_3 x_2$.

Let us formalize the algorithm sketched out in this example. If w is even, the computation of $f(X)$ involves the addition of $\frac{w}{2}$ partial products PP_i defined as follows³:

³ A proof of correctness is provided in [1].

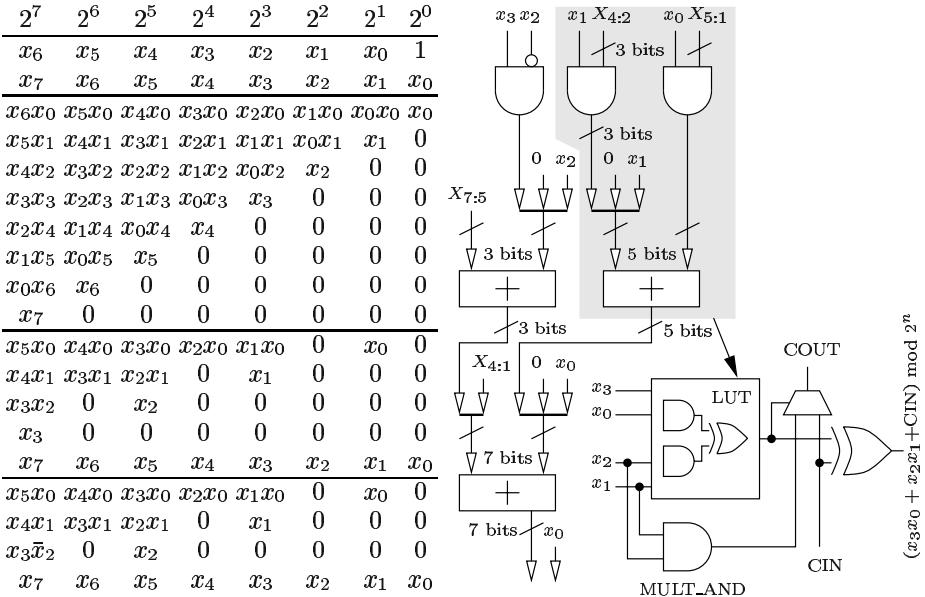


Fig. 3. Algorithm 1: computation of $f(X) = (X(2X + 1)) \bmod 2^8$ with AND gates and carry-propagate adders.

$$\text{PP}_i = \begin{cases} \sum_{j=0}^{w-1} x_j 2^j & \text{if } i = \frac{w}{2} - 1, \\ x_{i+1} \bar{x}_i 2^{w-1} + x_i 2^{w-3} & \text{if } i = \frac{w}{2} - 2, \\ x_i 2^{2i+1} + \sum_{j=2i+3}^{w-1} x_{j-i-2} x_i 2^j & \text{if } 0 \leq i \leq \frac{w}{2} - 3. \end{cases}$$

The above equation allows to automatically generate the VHDL code of the partial product generator for any even w . Synthesis tools are then able to put to good use the MULT_AND gates in order to generate and add partial products using the same logic resources as a simple multioperand tree adder (Figure 3).

3.2 Multiplier-Based Algorithms

A straightforward algorithm reported in [10] consists in writing the VHDL code depicted by Figure 1b. Since $w = 32$ and Virtex-II devices embed 17-bit \times 17-bit unsigned multipliers, commercial tools like Synplify Pro or XST (Xilinx Synthesis Technology) resort to the well-known divide-and-conquer scheme [8] in order to build the operator (Figure 4).

Each $f(X)$ operator based on this scheme involves three multiplier blocks and two carry-propagate adders. Consequently, a RC6 processor with full loop unrolling requires $2r \cdot 3 = 120$ MULT18x18 blocks and fits in a XC2V4000 device. Let us define the lower and higher words of the operand X as follows:

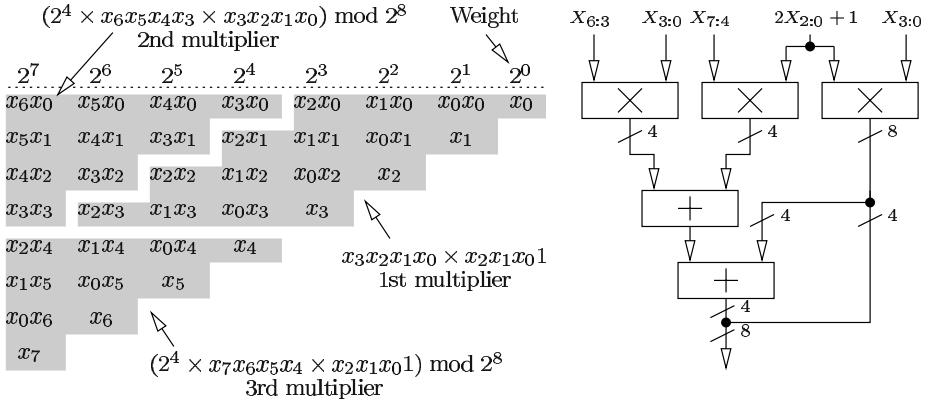


Fig. 4. Algorithm 2: computation of $f(X) = (X \cdot (2X + 1)) \bmod 2^w$ with a divide-and-conquer strategy.

$X_L = \sum_{i=0}^{w/2-1} x_i 2^i$ and $X_H = \sum_{i=0}^{w/2-1} x_{n+i} 2^i$. A solution to reduce the amount of 17-bit \times 17-bit unsigned multipliers, and therefore the price of the processor, is to evaluate $f(X)$ according to:

$$\begin{aligned} f(X) &= (2X^2 + X) \bmod 2^w = (2 \cdot (2^{w/2} X_H + H_L)^2 + X) \bmod 2^w \\ &= (2 \cdot (2^w X_H^2 + 2^{1+w/2} X_H X_L + X_L^2) + X) \bmod 2^w \\ &= (2^{2+w/2} X_H X_L + 2X_L^2 + X) \bmod 2^w. \end{aligned}$$

Figure 5 illustrates how this algorithm works. In this example, we assume that $w = 8$ and that 4-bit \times 4-bit multiplier blocks are available. Since $(2 \cdot 2^7 x_0 x_6 + 2 \cdot 2^7 x_2 x_4) \bmod 2^8 = 0$, we can discard these terms. For $w = 32$, this third algorithm involves a 16-bit squarer, a 14-bit \times 14-bit multiplier, a 14-bit CPA, and a 31-bit CPA.

Remember that Virtex-II devices embed 17-bit \times 17-bit unsigned multipliers. In the following, we describe how to put to good use the most significant bit of their input ports in order to further reduce the area of the $f(X)$ operator. Consider again the computation of $f(X)$ with $w = 8$ (Figure 6). The trick consists in performing the *rectangular* multiplication $(2 \cdot X_{3:0} + 1) X_{3:0}$ and allows to replace the $(w - 1)$ -bit CPA by a $w/2$ -bit CPA.

3.3 Comparison of the Four Algorithms

Table 1 summarizes the main characteristics of the four $f(X)$ operators previously described⁴. From these results, we can conclude that:

⁴ The VHDL code was synthesized and implemented on Virtex-E and Virtex-II devices with Synplify Pro 7.0.3 and Xilinx Alliance Series 4.1.03i. All input and output signals were routed through the D-type flip-flops available in the Input/Output blocks of Virtex-E or Virtex-II devices. No specific constraints were given to the synthesis tools and it should be possible to improve the results.

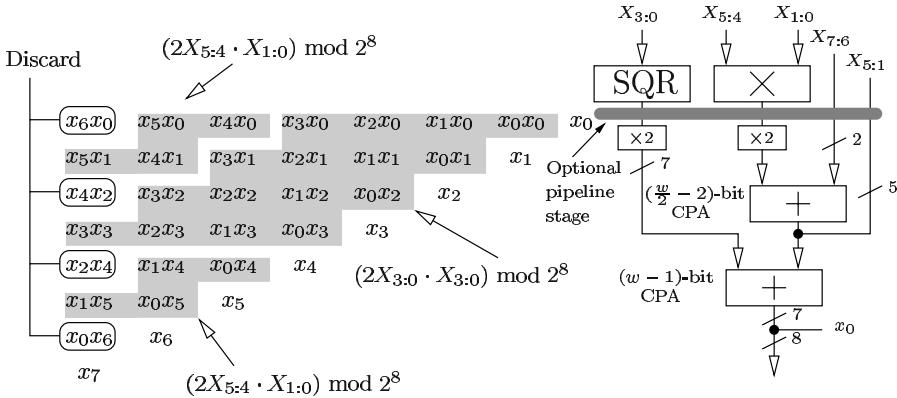


Fig. 5. Algorithm 3: computation of $f(X)$ with a squarer, a multiplier, and two carry-propagate adders.

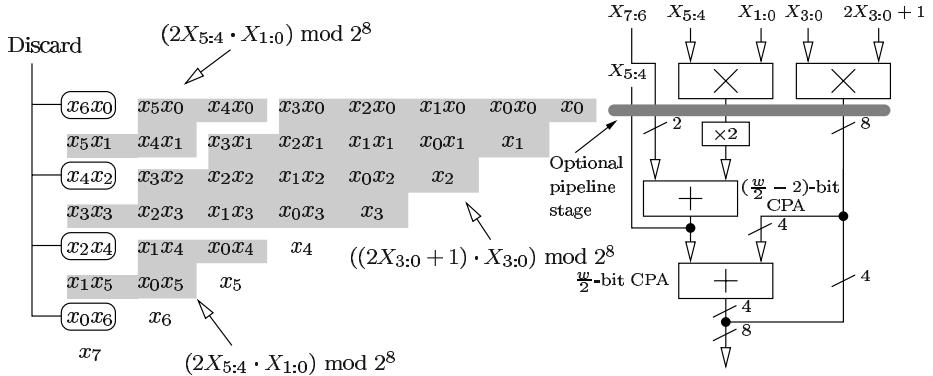


Fig. 6. Algorithm 4: computation of $f(X) = (X(2X + 1)) \bmod 2^8$ with two multipliers and two carry-propagate adders.

- On a Virtex-II device, Algorithm 4 leads to the smallest circuit.
- The adder-based operator (Algorithm 1) is the best alternative for the Virtex-E family. This result is not surprising since current synthesis tools are unable to build efficient multipliers from a high-level VHDL description such that shown in Figure 7. We also notice that Algorithm 3 and Algorithm 4 lead to the same circuit area: the fact that both multipliers and adders are now implemented on CLBs explains this result.

In order to improve the frequency of the RC6 processor, our VHDL code generator is able to insert optional pipeline stages in the $f(X)$ operator (parameter β on Figure 1a). For Virtex-II devices, we take advantage of the internal pipeline stage of each MULT18x18 block. Unfortunately, the VHDL coding style depends on both the chosen algorithm and the synthesis tools:

```

entity rc6_f is
  port (
    D : in std_logic_vector (31 downto 0);
    Q : out std_logic_vector (31 downto 0));
end rc6_f;
architecture behavioral of rc6_f is
  signal sqr_q : std_logic_vector (31 downto 0);
  signal mult_q : std_logic_vector (27 downto 0);
  signal add_q : std_logic_vector (31 downto 0);
begin
  begin -- behavioral
    sqr_q <= D (15 downto 0) * D (15 downto 0);
    mult_q <= D (29 downto 16) * D (13 downto 0);
    add_q (17 downto 0) <= D (17 downto 0);
    add_q (31 downto 18) <= D (31 downto 18) +
      mult_q (13 downto 0);
    Q (0)           <= add_q (0);
    Q (31 downto 1) <= add_q (31 downto 1) + sqr_q (30 downto 0);
  end behavioral;

```

(a) VHDL code for Algorithm 3.

```

entity rc6_f is
  port (
    D : in std_logic_vector (31 downto 0);
    Q : out std_logic_vector (31 downto 0));
end rc6_f;
architecture behavioral of rc6_f is
  signal multi1_q : std_logic_vector (32 downto 0);
  signal multi2_q : std_logic_vector (27 downto 0);
  signal add_q : std_logic_vector (15 downto 0);
begin
  begin -- behavioral
    multi1_q <= (D (15 downto 0) & '1') * D (15 downto 0);
    multi2_q <= D (29 downto 16) * D (13 downto 0);
    add_q (15 downto 2) <= D (31 downto 18) +
      multi2_q (13 downto 0);
    add_q (1 downto 0) <= D (17 downto 16);
    Q (15 downto 0) <= multi1_q (15 downto 0);
    Q (31 downto 16) <= multi1_q (31 downto 16) + add_q;
  end behavioral;

```

(b) VHDL code for Algorithm 4.

Fig. 7. Two VHDL descriptions of the $f(X)$ operator.**Table 1.** Comparison of several $f(X)$ operators.

	XCV50E-6		XC2V40-6			
	Slices	Delay [ns]	Pipeline	Slices	Mult. blocks	Delay [ns]
Algorithm 1	134	~ 18	—	148	—	~ 12
Algorithm 2	274	~ 20	—	18	3	~ 12
Algorithm 3	229	~ 20	—	25	2	~ 12
			1 stage	41	2	~ 8
Algorithm 4	230	~ 20	—	17	2	~ 12
			1 stage	25	2	~ 8

- For Algorithms 3 and 4, it sometimes suffices to insert a register after each multiplication. For instance, Synplify Pro is able to infer registered multipliers (option `-pipe 1` in the synthesis script). Synthesis tools also provide the designer with libraries containing the basic building blocks of a given FPGA family. It is therefore possible to instantiate a pipelined MULT18x18 block in the VHDL code, instead of expressing the multiplication operator.
- However, if the multiplier does not fit in a single MULT18x18 block, current synthesis tools are unable to simultaneously apply the divide-and-conquer scheme and the retiming algorithm. It is therefore impossible to automatically pipeline the VHDL description of Algorithm 2 depicted on Figure 1. The solution consists here in performing the divide-and-conquer scheme by hand: the VHDL description will then contain three 16-bit \times 16-bit multipliers.

The VHDL code illustrated on Figure 7 leads however to poor results on Virtex-E devices. A structural description of the operator (partial product generation, carry-propagate adders, and registers) gives better results.

4 Implementation Results

Table 2 summarizes the main characteristics of several RC6 processors for Virtex-E and Virtex-II devices. For non-feedback chaining modes, processors with full-

Table 2. Some RC6 processors for Virtex-II and Virtex-E devices. CAD tools: Synplify Pro 7.0.3 and Xilinx Alliance Series 4.1.03i (*) or ISE 5.1.03i (†).

	Device	Algo	Rounds	Pipeline				Slices	Mult. blocks	Through-put [Gb/s]
				α	β	γ	δ			
Non-feedback chaining modes	XCV2000E-6†	1	1 + 20 + 1	1	2	1	1	19198 (99%)	—	~ 10.6
	XCV300E-6*	1	1 + 1 + 1	1	2	1	1	2068 (67%)	—	~ 0.5
	XC2V3000-6*	4	1 + 20 + 1	1	1	1	0	8554 (59%)	80 (83%)	~ 15.2
	XC2V3000-6†	4	1 + 20 + 1	1	1	1	0	10288 (71%)	80 (83%)	~ 14.2
	XC2V3000-6*	1	1 + 10 + 1	1	1	1	0	7456 (52%)	0 (0%)	~ 4.8
	XC2V1000-6*	4	1 + 10 + 1	1	1	1	0	4391 (85%)	40 (100%)	~ 7.4
	XC2V500-6*	4	1 + 5 + 1	1	1	1	0	2365 (76%)	20 (62%)	~ 3.9
Feedback chaining modes	XC2V250-6*	4	1 + 4 + 1	1	1	1	0	1534 (99%)	16 (66%)	~ 2.8
	XCV300E-6*	1	1 + 1 + 1	0	0	0	0	1709 (55%)	—	~ 0.16
	XCV300E-6*	4	1 + 1 + 1	0	0	0	0	1902 (61%)	—	~ 0.15
	XCV400E-6*	1	1 + 4 + 1	1	0	0	0	3932 (81%)	—	~ 0.16
	XC2V1000†	4	1 + 1 + 1	0	0	0	0	1560 (30%)	4 (10%)	~ 0.29
	XC2V1000†	4	1 + 4 + 1	1	0	0	0	2902 (56%)	16 (40%)	~ 0.34

loop unrolling achieve high throughputs. The area and the critical path however depend on the synthesis tools: we have obtained better results with Symplify Pro 7.0.3 and Xilinx Alliance 4.1.03i than with ISE 5.1.03i. Also note that XC2V500 and XC2V250 devices have not enough I/Os to deal with 128-bit words. Our solution consists in defining 64-bit input and output ports and spending two clock cycles for data transmission.

The basic iterative architecture (Figure 2a) seems to be the best one for feedback chaining modes: it requires less slices than systems with partial loop unrolling and achieves the same throughput. As the rounds are combinational, the critical path increases and we obtain very low encryption rates.

A NSA team has implemented RC6 with semi-custom ASICs based on a $0.5 \mu\text{m}$ CMOS library [10]. Using the architecture depicted by Figure 2c with a pipeline stage between two consecutive rounds and Algorithm 2 to compute $f(X)$, the NSA team reports a throughput of 2.2 Gbits/s. Gaj et al. have proposed an architecture similar to Figure 2 [2, 4]. The main differences lie in the $f(X)$ operator and in the number of pipeline stages per cipher round (3 in our case versus 28 in their system). However, four XCV1000-6 devices are required to implement the algorithm with full loop unrolling and to achieve a throughput of 13.1 Gbits/s. While the throughput is close to ours, this solution is more expensive and requires a more complex PCB.

5 Conclusions

In this paper, several architectures of the RC6 block cipher for Virtex-E and Virtex-II FPGAs have been described. We have also investigated four algorithms computing $f(X)$, which is the critical arithmetic operation of the block cipher. Our experiments indicate that both the choice of an algorithm and the VHDL

coding style are strongly related to the target FPGA family. Our VHDL generator allows to quickly explore a wide parameter space and to determine the best architecture for a given set of constraints (feedback or non-feedback chaining mode, FPGA device, ...).

Acknowledgments

The author would like to thank the “Ministère Français de la Recherche”, the Swiss National Science Foundation, and the Xilinx University Program for their support.

References

1. J.-L. Beuchat. *Etude et conception d'opérateurs arithmétiques optimisés pour circuits programmables*. PhD thesis, Swiss Federal Institute of Technology Lausanne, 2001.
2. P. Chodowiec, P. Khuon, and K. Gaj. Fast Implementations of Secret-Key Block Ciphers Using Mixed Inner- and Outer-Round Pipelining. In *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 94–102, 2001.
3. M. Dworkin. Recommandation for Block Cipher Modes of Operation, 2001. NIST Special Publication 800-38A.
4. K. Gaj and P. Chodowiec. Fast implementation and fair comparison of the final candidates for Advanced Encryption Standard using Field Programmable Gate Arrays. In *Proc. RSA Security Conf. - Cryptographer's Track*, pages 84–99. Springer-Verlag, 2001. Available at <http://ece.gmu.edu/crypto/publications.htm>.
5. H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption.
6. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
7. E. Mosanya, C. Teuscher, H. F. Restrepo, P. Galley, and E. Sanchez. Crypto-Booster: A Reconfigurable and Modular Cryptographic Coprocessor. In C. K. Koc and C. Paar, editors, *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems, CHES'99, Worcester, MA*, volume 1717 of *Lecture Notes in Computer Science*, pages 246–256. Springer-Verlag, 1999.
8. B. Parhami. *Computer Arithmetic*. Oxford University Press, 2000.
9. R.L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 Block Cipher, 1998.
10. B. Weeks, M. Bean, T. Rozylowicz, and C. Ficke. Hardware Performance Simulations of Round 2 Advanced Encryption Standard Algorithms. Technical report, National Security Agency, 2000.
11. R. Zimmermann, A. Curiger, H. Bonnenberg, H. Kaeslin, N. Felber, and W. Fichtner. A 177 Mbit/s VLSI Implementation of the International Data Encryption Algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, 1994.

Very High Speed 17 Gbps SHACAL Encryption Architecture

Máire McLoone and J.V. McCanny

DSiP™ Laboratories, School of Electrical and Electronic Engineering,
The Queen's University of Belfast, Belfast BT9 5AH, Northern Ireland
maire.mcloone@ee.qub.ac.uk, j.mccanny@ee.qub.ac.uk

Abstract. Very high speed and low area hardware architectures of the SHACAL-1 encryption algorithm are presented in this paper. The SHACAL algorithm was a submission to the New European Schemes for Signatures, Integrity and Encryption (NESSIE) project and it is based on the SHA-1 hash algorithm. To date, there have been no performance metrics published on hardware implementations of this algorithm. A fully pipelined SHACAL-1 encryption architecture is described in this paper and when implemented on a Virtex-II X2V4000 FPGA device, it runs at a throughput of 17 Gbps. A fully pipelined decryption architecture achieves a speed of 13 Gbps when implemented on the same device. In addition, iterative architectures of the algorithm are presented. The SHACAL-1 decryption algorithm is derived and also presented in this paper, since it was not provided in the submission to NESSIE.

Keywords: NESSIE, SHACAL

1 Introduction

New European Schemes for Signatures, Integrity and Encryption (NESSIE) was a three-year research project, which formed part of the Information Societies Technology (IST) programme run by the European Commission. The main aim of NESSIE is to present strong cryptographic primitives covering confidentiality, data integrity and authentication, which have been open to a rigorous evaluation and cryptoanalytical process. Therefore, submissions not only included private key block ciphers, as with the Advanced Encryption Standard (AES) project [1], but also hash algorithms, digital signatures, stream ciphers and public key algorithms. The first call for submissions was in March 2000 and resulted in 42 submissions. The first phase of the project involved conducting a security and performance evaluation of all the submitted algorithms. The performance evaluation hoped to achieve performance estimates for software, hardware and smartcard implementations of each algorithm. 24 algorithms were selected for further scrutiny in the second phase, which began in July 2001. The National Institute for Standards and Technology's (NIST) Triple-DES and Rijndael private key algorithms and SHA-1 and SHA-2 hash algorithms were also considered for evaluation. The following 7 block cipher algorithms were chosen for further investigation in phase two of the NESSIE project: IDEA, SHACAL,

SAFER++, MISTY1, Khazad, RC6 and Camellia. The final selection of the NESSIE cryptographic primitives took place on 27 February 2003. MISTY1, Camellia, SHACAL-2 and Rijndael are the block ciphers chosen to be included in the NESSIE portfolio of cryptographic primitives. Overall, 12 of the original submissions are included along with 5 existing standard algorithms.

Currently, the fastest known FPGA implementations of the NESSIE finalist algorithms are as follows: Leong *et al.*[2] estimate that their IDEA architecture can achieve a throughput of 2 Gbps on a Virtex XCV1000 FPGA device. Standaert *et al.*'s [3] Khazad architecture runs at 9.4 Gbps on the XCV1000 FPGA. Their MISTY1 architecture runs at 10 Gbps on the same device. A pipelined Camellia implementation by Ichikawa *et al.* [4] on a Virtex-E XCV1000E device performs at 6.7 Gbps. The fastest FPGA implementation of the RC6 algorithm is the 15.2 Gbps implementation by Beuchat [5] on the Virtex-II XC2V3000 device. Finally, an iterative architecture of the SAFER++ algorithm by Ichikawa *et al.* [6] achieves a data-rate of 403 Mbps on a Virtex-E XCV1000E device. For the SHACAL algorithm, only published work on software implementations is currently available. Hence, in this paper, the authors hope to provide a performance evaluation of SHACAL-1 hardware implementations.

Section 2 of the paper provides a description of the SHACAL-1 algorithm. Section 3 outlines the SHACAL-1 architectures for hardware implementation. Performance results are given in section 4 and conclusions are provided in section 5.

2 A Description of the SHACAL-1 Algorithm

The SHACAL algorithm [7], developed by Helena Handschuh and David Naccache, is based on the NIST's SHA hash algorithm. SHACAL is defined as a variable block and key length family of ciphers. There are two versions specified in the submission: SHACAL-1, which is derived from SHA-1 and SHACAL-2, which is derived from SHA-256. However, other versions can easily be derived from the SHA-384 and SHA-512 hash algorithms. Only the SHACAL-1 algorithm is considered for implementation in this paper. However, the SHACAL-1 architecture described can be easily adapted for the other variants.

SHACAL-1 operates on a 160-bit data block utilising a 512-bit key. The key, k , can vary in length within the range, $128 \leq k \leq 512$. If $k < 512$, it is appended with zeros to a length of 512-bits. SHACAL-1 encryption, outlined in Fig. 1, is performed by splitting the 160-bit plaintext into five 32-bit words, A, B, C, D and E. Next, 80 iterations of the compression function are performed. The resulting A, B, C, D and E values are concatenated to form the ciphertext. The compression function is defined as follows:

$$\begin{aligned} A_{i+1} &= ROT_{LEFT\text{-}5}(A_i) + F_i(B_i, C_i, D_i) + E_i + Cnst_i + W_i \\ B_{i+1} &= A_i \\ C_{i+1} &= ROT_{LEFT\text{-}30}(B_i) \\ D_{i+1} &= C_i \\ E_{i+1} &= D_i \end{aligned} \quad (1)$$

where, W_i are the subkeys generated from the key schedule and the $ROT_{LEFT\text{-}n}$ function is defined as a 32-bit word rotated to the left by n positions.

The constants, $Cnst_i$, in hexadecimal, are,

$$\begin{aligned} Cnst_i &= 5a827999 & 0 \leq i \leq 19 \\ Cnst_i &= 6ed9eba1 & 20 \leq i \leq 39 \\ Cnst_i &= 8f1bbcd \bar{c} & 40 \leq i \leq 59 \\ Cnst_i &= ca62c1d6 & 60 \leq i \leq 79 \end{aligned} \quad (2)$$

and the function $F_i(x, y, z)$ is defined as,

$$F_i(x, y, z) = \begin{cases} (x \text{ AND } y) \text{ OR } (\bar{x} \text{ AND } z) & 0 \leq i \leq 19 \\ x \oplus y \oplus z & 20 \leq i \leq 39 \\ (x \text{ AND } y) \text{ OR } (x \text{ AND } z) \text{ OR } (y \text{ AND } z) & 40 \leq i \leq 59 \\ x \oplus y \oplus z & 60 \leq i \leq 79 \end{cases} \quad (3)$$

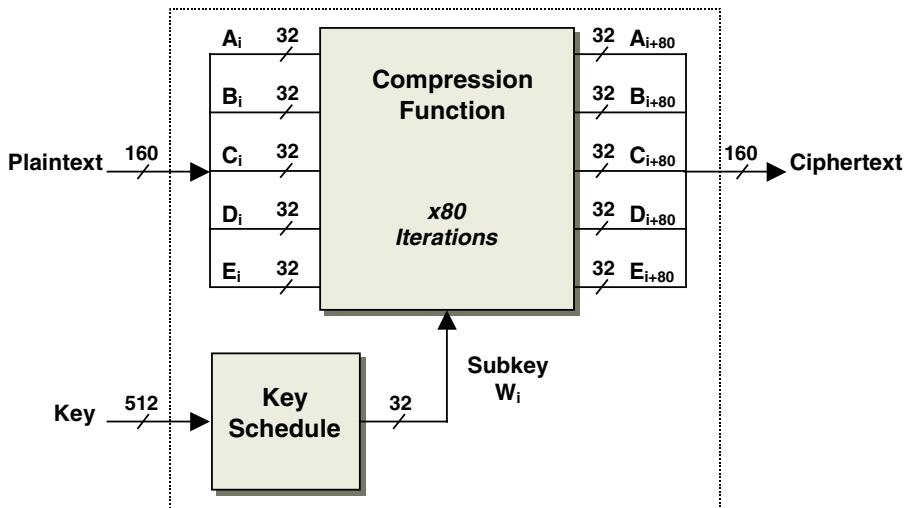


Fig. 1. Outline of SHACAL-1 Encryption Algorithm

In the SHACAL-1 key schedule, the 512-bit input key is expanded to form eighty 32-bit subkeys, W_i , such that,

$$W_i = \begin{cases} Key_i & 0 \leq i \leq 15 \\ ROT_{LEFT-1}(W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16}) & 16 \leq i \leq 79 \end{cases} \quad (4)$$

The first 16 subkeys are formed by splitting the input key into sixteen 32-bit values.

2.1 SHACAL-1 Decryption

SHACAL-1 decryption is not defined in the submission. Therefore, it has been derived and is presented here. It requires an inverse compression function and an inverse key schedule.

The inverse compression function is as follows:

$$\begin{aligned} A_{i+1} &= B_i \\ B_{i+1} &= \text{ROT}_{\text{RIGHT-30}}(C_i) \\ C_{i+1} &= D_i \\ D_{i+1} &= E_i \\ E_{i+1} &= A_i - [\text{ROT}_{\text{LEFT-5}}(B_i) + \text{InvF}_i(\text{ROT}_{\text{RIGHT-30}}(C_i), D_i, E_i) + \text{InvCnst}_i + \text{InvW}_i] \end{aligned} \quad (5)$$

where InvW_i are the inverse subkeys generated from the inverse key schedule and the $\text{ROT}_{\text{RIGHT-}n}$ function is defined as a 32-bit word rotated to the right by n positions. The constants, InvCnst_i , in hexadecimal, are,

$$\begin{aligned} \text{InvCnst}_i &= \text{ca62c1d6} & 0 \leq i \leq 19 \\ \text{InvCnst}_i &= \text{8f1bbcde} & 20 \leq i \leq 39 \\ \text{InvCnst}_i &= \text{6ed9eba1} & 40 \leq i \leq 59 \\ \text{InvCnst}_i &= \text{5a827999} & 60 \leq i \leq 79 \end{aligned} \quad (6)$$

and the function $\text{InvF}_i(x, y, z)$ is defined as,

$$\text{InvF}_i(x, y, z) = \begin{cases} x \oplus y \oplus z & 0 \leq i \leq 19 \\ (x \text{ AND } y) \text{ OR } (x \text{ AND } z) \text{ OR } (y \text{ AND } z) & 20 \leq i \leq 39 \\ x \oplus y \oplus z & 40 \leq i \leq 59 \\ (x \text{ AND } y) \text{ OR } (\bar{x} \text{ AND } z) & 60 \leq i \leq 79 \end{cases} \quad (7)$$

The subkeys generated from the key schedule during encryption are used in reverse order when decrypting data. Therefore, it is necessary to wait for all of the subkeys to be generated before beginning decryption. However, an inverse key schedule can be utilised to generate the subkeys in the order that they are required for decryption. This inverse key schedule is defined as,

$$\text{InvW}_i = \begin{cases} \text{InvKey}_i & 0 \leq i \leq 15 \\ \text{ROT}_{\text{RIGHT-1}}(\text{InvW}_{i-16}) \oplus \text{InvW}_{i-13} \oplus \text{InvW}_{i-8} \oplus \text{InvW}_{i-2} & 16 \leq i \leq 79 \end{cases} \quad (8)$$

The InvKey is created by concatenating the final 16 subkeys generated during encryption, such that,

$$\text{InvKey} = \{ W_{69}, W_{69}, W_{69}, W_{69}, W_{69}, W_{69}, W_{69}, W_{70}, W_{71}, \\ W_{72}, W_{73}, W_{74}, W_{75}, W_{76}, W_{77}, W_{78}, W_{79} \} \quad (9)$$

3 SHACAL-1 Hardware Architectures

The SHACAL-1 algorithm is derived from the SHA hash algorithm. Therefore, the design of a SHACAL-1 hardware architecture can be derived from the design of a SHA hash algorithm architecture. Previous efficient implementations of the SHA-1 and SHA-2 hash algorithms [8], [9], [10] have utilised a shift register design approach. Thus, this methodology has also been used in the iterative and fully pipelined SHACAL-1 architectures described here.

3.1 Iterative SHACAL-1 Architecture

In the iterative encryption and decryption architectures data is passed through the compression or inverse compression function component eighty times. The initial five data blocks are generated from the input block split into five 32-bit blocks. The outputs of the function, A to E form the inputs on consecutive clock cycles. After 80 iterations, the A to E outputs are concatenated to form the 160-bit plaintext/ciphertext.

The main components in both the iterative and fully pipelined architectures are the compression function and the key schedule. The key schedule is implemented using a 16-stage shift register design, as illustrated in Fig. 2.

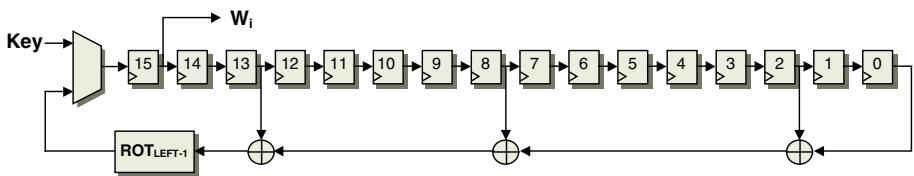


Fig. 2. SHACAL-1 Key Schedule Design

The input 512-bit key is loaded into the registers in 32-bit blocks over 16 clock cycles. On the next clock cycle, the value of register 15 is updated with the result of equation 4. An initial delay of 16 clock cycles is avoided by taking the values, W_i , from the output of register 15 and not from the output of register 0. The subkeys, W_i , are generated as they are required by the compression function.

The inverse key schedule is designed in a similar manner and is shown in Fig. 3.

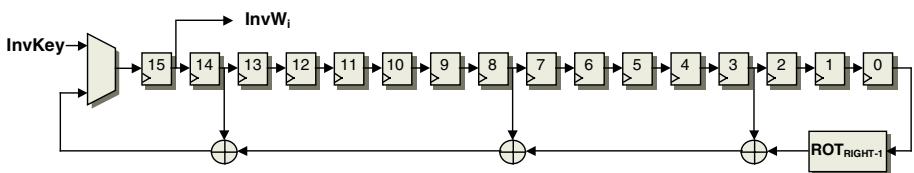


Fig. 3. SHACAL-1 Inverse Key Schedule Design

Typically, for decryption, the sender of the ciphertext will send the receiver the original key used to encrypt the message. Hence, the receiver will have to generate all eighty 32-bit subkeys before commencing decryption. However, this can be avoided if

the sender of the ciphertext sends the receiver the final sixteen 32-bit subkeys that were created during encryption of the message as a 512-bit inverse key. Now, the receiver can immediately begin to decrypt the ciphertext, since the subkeys required for decryption can be generated as they are required using this inverse key.

The design of the SHACAL-1 compression function is depicted in Fig. 4. The design requires 5 registers to store the continually updating values of A, B, C, D & E. The values in registers B, C and D are operated on by one of four different functions every 20 iterations, as given in equation 3.

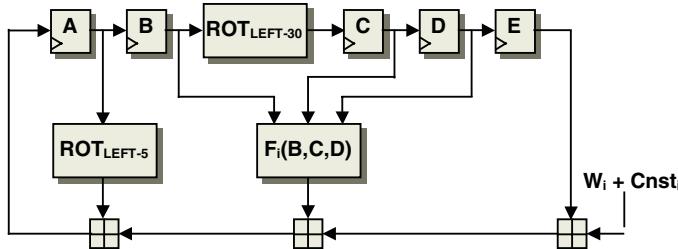


Fig. 4. SHACAL-1 Compression Function Design

The critical path of the overall SHACAL design occurs in the compression function in the calculation of,

$$A_{i+1} = ROT_{LEFT-5}(A_i) + F_i(B_i, C_i, D_i) + E_i + Cnst_i + W_i \quad (10)$$

In the architectures described here, this critical path is reduced in 2 ways. Firstly, the addition, $T = Cnst_i + W_i$, is performed on the previous clock cycle and thus, is removed from the critical path. Also, Carry-Save-Adders (CSAs) are utilised. With CSAs, the carry propagation is avoided until the final addition. Therefore, equation 10 is implemented using two CSAs and one Full Adder (FA) rather than three FAs, as shown in Fig. 5. Since the CSAs involve 32-bit additions, this implementation is faster than an implementation using only FAs [11].

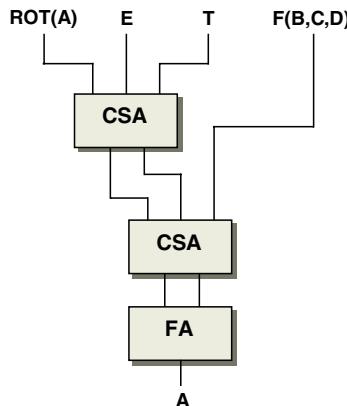


Fig. 5. CSA Implementation in Compression Function

The inverse compression function, outlined in Fig. 6, contains a subtraction. Hence, the throughput of the SHACAL-1 decryption architecture will be slower than that of the encryption architecture. During decryption the critical path of the overall SHACAL design occurs in the inverse compression function, where,

$$E_{i+1} = A_i - [ROT_{LEFT-5}(B_i) + InvF_i(ROT_{RIGHT-30}(C_i), D_i, E_i) + InvCnst_i + InvW_i] \quad (11)$$

Once again, it can be reduced by performing the addition, $InvCnst_i + InvW_i$, on the previous clock cycle. No significant performance benefit is achieved by using CSAs in the decryption architecture. Since equation 11 includes a subtraction, only one CSA could be used and two full adders would still be required in an implementation.

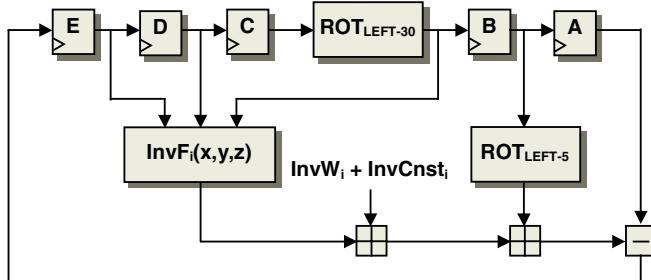


Fig. 6. SHACAL-1 Inverse Compression Function Design

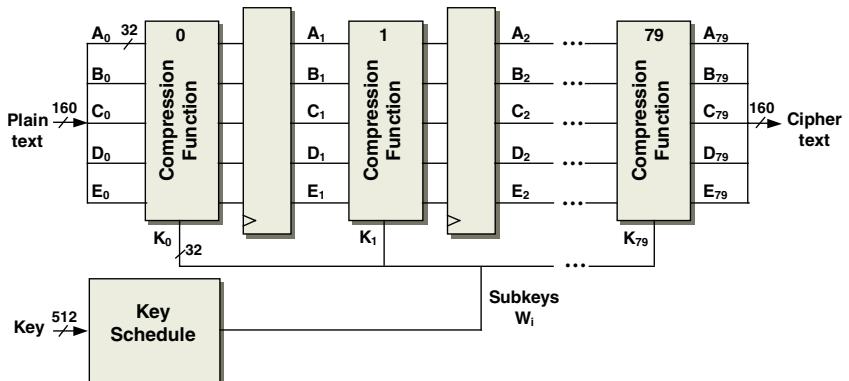


Fig. 7. SHACAL-1 Pipelined Architecture

3.2 Pipelined SHACAL-1 Architecture

In the pipelined SHACAL-1 encryption architecture, the compression function and key schedule are designed as for the iterative architecture. However, the eighty compression function iterations are fully unrolled and registers placed between each component, as depicted in Fig. 7. It is assumed that the same key is used throughout a data transfer session. Every twenty compression function components contain a different function according to equation 3. New plaintext blocks can be accepted on every clock cycle and after an initial delay, the corresponding ciphertext blocks will

appear on consecutive clock cycles. This leads to a very high speed design. The pipelined decryption architecture is designed in a similar manner using the inverse compression function and inverse key schedule.

4 Performance Evaluation

To provide a hardware performance evaluation for SHACAL-1, the hardware architectures described in this paper are implemented on Xilinx FPGA devices for demonstration purposes. The designs were simulated using Modelsim and synthesised using Synplify Pro v7.2 and Xilinx Foundation Series 5.1i software. They were verified using the SHACAL-1 test vectors provided in the submission to NESSIE.

The iterative encryption architecture implemented on the Virtex-II XC2V500 device runs at a clock speed of 110 MHz and hence, achieves a throughput of 215 Mbps. The design utilises just 994 CLB slices. The iterative decryption architecture implemented on the same device has a data-rate of 177 Mbps and requires only 877 slices. The decryption design is slower since its critical path contains a subtraction. However, it is smaller in area since the CSAs used in the design of the compression function in the encryption design incur a slight area penalty. The iterative architectures result in highly-compact yet efficient implementations. In both designs the 160-bit plaintext/ciphertext blocks and 512-bit key are input and output in 32-bit blocks. This implies a lower IOB count and less routing, thus, the designs can be implemented efficiently on smaller FPGA devices.

The pipelined architectures result in very high speed designs. The encryption design when implemented on the Virtex-II XC2V4000 device, operates at 106 MHz with a throughput of 17 Gbps utilising 13,729 CLB slices. The equivalent decryption design metrics are 83 MHz, 13 Gbps and 10,844 slices. Even higher data-rates are attainable if the architectures are implemented on ASIC technology. Table 1 provides a summary of published work on fast, pipelined hardware implementations of the NESSIE block cipher finalists on FPGA devices. The SAFER++ algorithm is not included in the table since an iterative design [6] with a throughput of 403 Mbps is the fastest implementation published to date. For comparison purposes, the table includes performance metrics for a fully pipelined single-chip implementation of the Rijndael algorithm [12]. Overall, the SHACAL-1 algorithm provides the fastest pipelined implementation. However, it is difficult to compare the performance of the algorithms as they are implemented on different devices. The closest in speed to the SHACAL-1 implementation is the 15.2 Gbps RC6 design by Beuchat. Beuchat [5] also carried out an implementation of the pipelined RC6 design on the XCV1600E device, achieving a data-rate of 9.7 Gbps. The SHACAL-1 pipelined design implemented on the same device performs at 10 Gbps. Therefore, the SHACAL-1 algorithm still provides the fastest implementation. Although it is the fastest, the pipelined design is also the largest in area. High-speed, yet lower area implementations are possible by unrolling the algorithm by a lower number of compression function components, while still adopting pipelining. The SHACAL specification provided performance metrics for a software implementation of SHACAL-1 on an 800 MHz Pentium III processor. Encryption was achieved at a data-rate of 52 Mbps, decryption at 55 Mbps and key

setup at 180 Mbps [7]. Therefore, even the iterative hardware design outlined in this paper is 4 times faster than their software implementation.

The efficiency of the algorithm implementations is also given in Table 1. Efficiency calculations are not provided for implementations that have been designed specifically to a device, and thus, include features such as multipliers and BRAM components. The MISTY1 implementation is the most efficient pipelined design while the SHACAL-1 implementation on the Virtex-II device is the next most efficient.

Table 1. Summary of NESSIE Algorithm Hardware Implementations

Authors	Algorithm	Device Used	Area	Through-put (Mbps)	Efficiency (Mbps/slices)
Authors	SHACAL-1	XC2V4000	13,729 slices	17021	1.24
Authors	SHACAL-1	XCV1600E	14,768 slices	10123	0.68
Leong <i>et al.</i> [2] (<i>estimated</i>)	IDEA	XCV1000	11,204 slices	2003	0.18
Standaert <i>et al.</i> [3]	Khazad	XCV1000	8,800 slices	9472	1.08
Standaert <i>et al.</i> [3]	MISTY1	XCV1000	6,322 slices	10176	1.6
Icikawa <i>et al.</i> [4]	Camellia	XCV1000E	9,692 slices	6750	0.7
Beuchat[5]	RC6	XC2V3000	8,554 slices 80 multipliers	15200	-
Beuchat[5]	RC6	XCV1600E	14110 slices	9700	0.7
McLoone, <i>et al.</i> [12]	Rijndael	XCV812E	2679 slices 82 BRAMs	6956	-

5 Conclusions

Throughout the NESSIE project, there has been no performance evaluation provided for hardware implementations of the SHACAL algorithm. In this paper, the SHACAL-1 algorithm is studied and both low area iterative and high speed pipelined architectures are presented. These are implemented on Xilinx FPGA devices for demonstration purposes but can readily be implemented on other FPGA or ASIC technologies. The iterative architectures are highly compact, yet efficient, when implemented on Virtex-II devices. A very high speed 17 Gbps design is achieved when the fully pipelined SHACAL-1 architecture is implemented on the Virtex-II XC2V4000 device. Therefore, SHACAL-1 is the fastest algorithm in hardware when

compared to pipelined hardware implementations of the other NESSIE block cipher finalists. The other SHACAL algorithm version specified in the submission, SHACAL-2, operates on a 256-bit data block utilising a 512-bit key. The SHACAL-1 architectures described in this paper can be easily adapted for SHACAL-2. SHACAL-2 iterative and pipelined architectures are anticipated to achieve similar speeds to that achieved by SHACAL-1.

The SHACAL-2 algorithm was selected as one of four block ciphers to be included in the NESSIE portfolio of cryptographic primitives and future work will be carried out on SHACAL-2 hardware implementations. The SHACAL algorithm has proven to be very secure with a large security margin. Since it is derived from the SHA hash algorithm, it has already undergone much cryptanalysis. It performs well when implemented in software [13] and as is evident from this paper, very high speed hardware implementations are also possible. Since security and performance were the two main criteria in the NESSIE selection process, the SHACAL algorithm is an ideal candidate for selection.

References

1. US NIST Advanced Encryption Standard, URL: <http://csrc.nist.gov/encryption/aes/>
2. M.P. Leong, O.Y.H. Cheung, K.H. Tsoi, P.H.W. Leong, "A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA", IEEE Symposium on FCCMs 2000, California, April 2000.
3. F.X. Standaert, G. Rovroy, "Efficient FPGA Implementation of Block Ciphers Khazad and MISTY1", 3rd NESSIE Workshop, URL: <http://www.di.ens.fr/~wwwgrecc/NESSIE3/>, Germany, November 2002.
4. T. Ichikawa, T. Sorimachi, T. Kasuya, M. Matsui, "On the criteria of hardware evaluation of block ciphers(1)", Techn report of IEICE, ISEC2001-53, Sept 2001.
5. J.L. Beuchat, "High Throughput Implementations of the RC6 Block Cipher Using Virtex-E and Virtex-II Devices", INRIA Research Report, URL: <http://www.ens-lyon.fr/~jlbeucha/publications.html>, July 2002.
6. T. Ichikawa, T. Sorimachi, T. Kasuya, "On Hardware Implementation of Block Ciphers Selected at the NESSIE Project Phase 1", 3rd NESSIE Workshop, URL: <http://www.di.ens.fr/~wwwgrecc/NESSIE3/>, Germany, November 2002.
7. H. Handschuh, D. Naccache, "SHACAL", 1st NESSIE Workshop, URL: <https://www.cosic.esat.kuleuven.ac.be/nessie/workshop/>, Belgium, Nov 2000.
8. K.K. Ting, S.C.L. Yuen, K.H. Lee, P.H.W. Leong, "An FPGA based SHA-256 Processor", 12th International FPL Conference, France, September 2002.
9. T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, B. Schott, "Comparative Analysis of the Hardware Implementations of Hash Functions SHA-1 and SHA-512", Information Security Conference", Oct 2002.
10. M. McLoone, J.V. McCanny, "Efficient Single-Chip Implementation of SHA-384 & SHA-512", IEEE International FPT Conference, Hong Kong, Dec 2002.
11. T. Kim, W. Jao, S. Tjiang, "Circuit Optimization Using Carry-Save-Adder Cells", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 17, No.10, October 1998.
12. M. McLoone, J.V. McCanny, "High Performance Single-Chip FPGA Rijndael Algorithm Implementations", 3rd International CHES Workshop, pp 65-77, France, May 2001.
13. NESSIE, "Performance of Optimized Implementations of the NESSIE Primitives", <http://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/D21-v2.pdf> February 2003.

Track Placement: Orchestrating Routing Structures to Maximize Routability

Katherine Compton¹ and Scott Hauck²

¹ Northwestern University, Evanston, IL USA
kati@ece.northwestern.edu

² University of Washington, Seattle, WA USA
hauck@ee.washington.edu

Abstract. The design of a routing channel for an FPGA is a complex process requiring a careful balance of flexibility with silicon efficiency. With a growing move towards embedding FPGAs into SoC designs, and the new opportunity to automatically generate FPGA architectures, this problem is even more critical. The design of a routing channel requires determining the number of routing tracks, the length of the wires in those tracks, and the positioning of the breaks between wires on the tracks. This paper focuses on the last problem, the placement of breaks in tracks to maximize overall flexibility. Our optimal algorithm for track placement finds a best solution provided the problem meets a number of restrictions. Our relaxed algorithm is without restrictions, and finds solutions on average within 1.13% of optimal.

1 Introduction

The design of an FPGA interconnect structure has usually been a hand-tuning process. A human designer, with the aid of benchmark suites and trial-and-error, develops an interconnect structure that attempts to balance flexibility with silicon efficiency. Often, the concentration is on picking the number and length of tracks – long tracks give global communication but with high silicon and delay costs, while short wires can be very efficient only if signals go a relatively short distance.

An area that can sometimes be ignored is the placement of the breaks in these interconnect wires. If we have a symmetric interconnect, with N length- N wires, we simply break one wire at each cell. However, for more irregular interconnects, it can be difficult to determine the best positioning of these breaks.

While a manual solution may be feasible in many cases when only a single architecture is being examined, it is not always practical. For example, track placement becomes extremely critical when we consider automatic generation of custom FPGA architectures for systems-on-a-chip [1]. Here, a track placement may be performed a very large number of times within the inner loop of an architecture generator, and a fast but effective algorithm for automatic track placement becomes a necessity.

In this paper, we address the issue of routing architecture design for reconfigurable architectures with segmented channel routing, such as RaPiD [2] and Garp [3]. We formalize the track placement problem, define a cost metric, and introduce track placement algorithms, including one proven optimal for a subset of these problems.

Achieving the best track placement requires a careful positioning of the breaks on multiple, different-length, routing tracks. For example, in Fig. 1 right, the breaks between wires in the routing tracks are staggered. This helps to provide similar routing options regardless of location in the array. If instead all breaks were lined up under a single unit, as in Fig. 1 left, a signal might become very difficult to route if its source was on one side of the breaks and at least one sink on the other. When large numbers of tracks or tracks of different wire lengths are involved, it can become difficult to find a solution where the breaks are evenly distributed through the array.

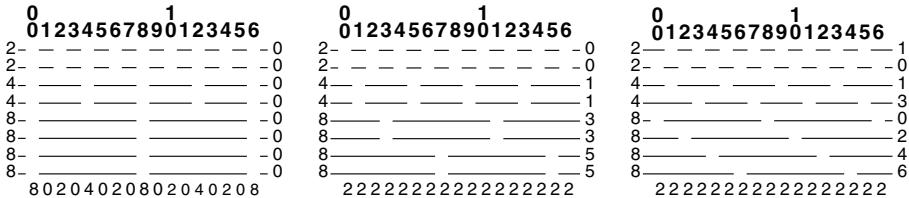


Fig. 1. Three different track placements for the same problem. A very poor one (left), an improved one (middle) and an even better placement (right). In each, the numbers on top indicate position in the architecture, and the numbers on the bottom indicate the number of breaks at the given position. Each track has its associated wire length at left, and offset at right.

The issue of determining the positioning (or offset) of a track within an architecture is referred to as track placement. The goal is to pick the offset that should be used for each track in order to maximize the routability, given a predetermined set of tracks with fixed length wires. For simplicity, each track is restricted to contain wires of only one length, which is referred to as the track’s S value, or track length. The actual determination of the quantity and S values of tracks is discussed elsewhere [1], as are issues specific to 2D routing architecture design [4].

2 Problem Description

Finding a good track placement is a complex task. Intuitively, we wish to space tracks with the same S value evenly, such as by placing the length-8 tracks from the problem featured in Fig. 1 at offsets 0, 2, 4, and 6. However, a placement of tracks of one S value can affect the best placement for tracks of another S value. For example, Fig. 1 right shows that the length-4 tracks are placed at offsets 1 and 3 in order to avoid having the breaks of those tracks fall at the same locations as the breaks from the length 8 tracks. This effect is called “correlation” between the affected tracks. Correlations occur between tracks when their S values contain at least one common prime factor. It is these correlations that make track placement difficult.

From the previous example we might conclude that a possible metric for measuring the quality of a track placement would be to compute the “evenness” or “smoothness” of the break distribution throughout the architecture. However, the smoothness metric fails to capture the idea of maintaining similar routing options for every location in the array. The placement in Fig. 1 center has the same smoothness of breaks as the solution at right, but is not an equally good solution. For example, although

there are two length-4 tracks, each logic unit position is at the same location along the length-4 wires in both tracks. On the other hand, the architecture at right provides for two different locations along the length-4 wires at every logic unit position. For this reason, we consider the placement at right to be superior in terms of routing options at each position despite the two architectures having the same break smoothness.

Instead, our chosen goal in track placement is to ensure that signals of all lengths have the most possible routes. To quantitatively measure this routability, we examine each possible signal length, and find the region of the interconnect that gives the fewest possible routes for this length signal. Summing across all possible signal lengths yields the “diversity score” of a track placement, as shown in Equation 1. The fewer and smaller the routing bottlenecks, the more routing choices are available throughout the architecture, and the higher the diversity score.

Equation 1. The diversity score for a particular track placement routing problem T and solution set of offsets O for the tracks in T can be calculated using the equation:

$$\text{diversity_score}(T, O) = \sum_L \left(\min_{\text{all positions}} \left(\sum_{Ti \in T} \text{uncut}(T_i, O_i, L, \text{position}) \right) \right)$$

for all tracks T_i with their given wire lengths and offsets O_i , possible signal lengths L, and all possible positions in the interconnect. The $\text{uncut}()$ function returns a binary result that is 0 if there is a break within the range [position, position + L) on track T_i that has been placed at offset O_i , and 1 otherwise.

Fig. 2 shows the diversity score calculation for two different placements of the same track problem. Here we consider a four position window that encapsulates the full repeating pattern of breaks of the placement. The length of the window that needs to be considered can be found by taking the LCM of all S values in the problem. Examining a larger window would simply yield the same results. Within this window we count the number of tracks at each position that can be used to route a signal of the given length towards the right. The different possible signal lengths (L) are listed, and at each position, the number of tracks useable to route that signal length is given.

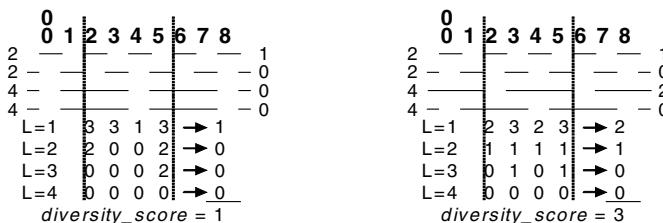


Fig. 2. Diversity score for two different track placements. Positions are given by the top row of numbers. For each track, segment length S is at left, and offset O is at right. The number of routing possibilities is given for each potential signal length L. Diversity score is at bottom.

In Fig. 2 left, two different tracks can be used to route length-2 signals to the right at position 2, but at position 3, no tracks are useable. At right, exactly one track can be used to route length-2 signals from any position in our window. We then take the

minimum value at each length (representing a routing bottleneck for that signal length), and sum across L values to arrive at the diversity score.

We have also determined a bound on this value, as described in Theorem 1 (proven elsewhere [5]). Note that comparing diversity scores is only valid across different solutions to the same track placement problem. We discuss the issue of determining the actual quantity and types of tracks that should be used elsewhere [1].

Theorem 1. For all possible offset assignments O to track set T,

$$\text{diversity_score}(T, O) \leq \sum_L \text{floor} \left(|T| - \sum_{T_i \in T} \min(1, L/S_i) \right)$$

We focus on the worst-case (the regions with the fewest possible routes) instead of the average case. The average number of possible routes for a signal is independent of track placement, and only depends on the number of tracks and their wire lengths. Thus, an average case metric cannot tell the difference between the placements in Fig. 2, while the worst-case will show that the rightmost version is superior.

3 Proposed Algorithms

We have developed a number of algorithms to solve the track placement problem based on the diversity score cost function. These track placement algorithms, both optimal and heuristic, are discussed in depth in the next few sections.

3.1 Brute Force Algorithm

Using a brute force approach, we can guarantee finding a solution with the highest possible diversity score. However, examining the entire solution space is a very slow process. The number of cases that the brute force approach must examine for a given problem is the product, over all distinct track lengths in the problem, of a multichoose of the track length and quantity of that length. For example, a modest architecture with 8 length-12 tracks, 4 length-6 tracks and 2 length-4 tracks will require the examination of *over 95 million* distinct cases. Since one of our targeted applications is within an inner loop of an automatic architecture generator [1], this approach is far too slow. Instead, it provides a bound on the diversity score, which is used to verify the results of our optimal algorithm and measure the quality of our other algorithms.

3.2 Simple Spread Algorithm

The Simple Spread algorithm is a simple heuristic for track placement. Tracks are grouped by segment length. For each group, the tracks are spaced evenly within the offsets 0 through $S_i - 1$ (where S_i is the segment length of that group), regardless of the placement of tracks from other groups. This algorithm is simple in design and fast in execution, but disregards correlations between S values. It is included to demonstrate in the results section the necessity of considering these correlations.

3.3 Optimal Factor Algorithm

One of our goals was to develop a fast algorithm that, with some restrictions, would provably find the optimal solution. The Optimal Factor algorithm is the culmination of many theorems and proofs. While many of the theorems will be presented here, due to reasons of space, their proofs are presented elsewhere [5].

Any optimal algorithm must consider the correlations between breaks both within an S group and across S groups. Correlations between two different S values occur when those S values share a common factor. For example, a track with S=6 and one with S=2 (a common factor of 2) will either have breaks at the same location once every 6 positions, or not at all, depending on the offsets chosen for the two tracks. On the other hand, a track with S=2 and one with S=5 will have breaks at the same location once every 10 positions regardless of the offsets chosen, as they are completely uncorrelated (no common factors). The following theorem is therefore used to divide a problem into smaller independent (uncorrelated) problems when possible.

Theorem 2. If T can be split into two sets $G1 \subset T$ and $G2 = T - G1$, where the S values of all tracks in $G1$ are relatively prime with the S values of all tracks in $G2$, then $\text{diversity_score}(T, O) = \text{diversity_score}(G1, O1) + \text{diversity_score}(G2, O2)$, where O is the combination of sets $O1$ and $O2$. Thus, $G1$ and $G2$ may be solved independently.

We can also use the fact that correlation is based on common factors to further simplify the track placement problem. Theorem 3 states that whenever one track's S value has a prime factor that is not shared with any other track in the problem (or a track has a higher quantity of a prime factor than any other track in the problem), the prime factor can be removed. For example, with 2 length-6 tracks and one length-18 track, we can remove a 3 from 18, and essentially have 3 length-6 tracks, which can then be placed evenly using a later theorem. We do not actually change the S value of the length-18 track, just the effective S value used during track placement. After the offsets of tracks are determined, the original S values are restored if necessary.

Theorem 3. If the S_i of unplaced track T_i contains more of any prime factor than the S_j of each and every other track T_j ($i \neq j$), then for all solutions O , $\text{diversity_score}(T, O) = \text{diversity_score}(T', O)$, where T' is identical to T , except T'_i has $S'_i = S_i$ with the unshared prime factor removed. This rule can be applied recursively to eliminate all unshared prime factors.

Next the tracks are grouped by S value (which may have been factored using Theorem 3). These groups are then sorted such that the largest S value is considered first. Any full sets (a set of N tracks all with $S_i = N$) are removed from the problem by Theorem 4, placing one track from the set at each possible offset 0 to $N-1$. Note we are only considering as possible offsets the range $0 \dots |S_i|-1$ for a track with $S=S_i$. While it is perfectly valid to place a track at an offset greater than $|S_i|-1$, the fact that all wires within the track are of length $|S_i|$ causes a break to be located every $|S_i|$ locations. Therefore, there will always be a break on track T_i within the range 0 to $|S_i|-1$.

The basic idea of the remainder of the algorithm is to space the tracks of the largest S value (S_{\max}) evenly using Theorems 4 and 5, then remove these tracks from further consideration. However, in order to respect the effect that their breaks have on the placement of tracks from successive S groups, we add pre-placed placeholder tracks

(with $S =$ the next largest S value S_{next}) such that they have the same number of breaks at the same positions as the original tracks, but in terms of S_{next} instead of S_{max} . This enables us to determine the best offsets for the real tracks with $S=S_{\text{next}}$. This process is repeated within a loop, where S_{next} becomes S_{max} , and we find the new S_{next} .

Theorem 4. Given a set $G \subset T$ of tracks, each with $S=|G|$. If there exists a solution O' for tracks $T'=T-G$ that meets the bound, then there is also solution O for tracks T that meets the bound (and thus is optimal), where each track in G is placed at a different offset $0\dots|G|-1$.

We do set a number of restrictions, however, in order to ensure that at each loop iteration, (1) we can perfectly evenly space the tracks with $S=S_{\text{max}}$, and (2) the breaks from the set of tracks with $S=S_{\text{max}}$ can be exactly represented by some integer number of tracks with $S=S_{\text{next}}$. These restrictions are outlined in the next two theorems.

Theorem 5. Let S_{max} be the maximum S value currently in the track placement problem, M be the set of all tracks with $S = S_{\text{max}}$, $N = |M|$, and S_{next} be the largest $S \neq S_{\text{max}}$. If $N > 1$, S_{max} is a proper multiple of N , and $S_{\text{next}} \leq S_{\text{max}} * (N-1)/N$, then any solution to T that meets the bound must evenly space out the N tracks with $S = S_{\text{max}}$. That is, for each track M_i , with a position O_i , there must be another track M_j with a break at $O_i + S_{\text{max}}/N$.

Theorem 6. Given a set of tracks G , all of segment length X , where X is evenly divisible by $|G|$, and a solution O with these tracks evenly distributed. There is another set of tracks G' , all of length $Y = |G'| * X / |G|$, with a solution O' where the number of breaks at each position is identical for solution O of G and solution O' of G' . If solution in which G has been replaced with G' meets its bound, the solution with the original G also meets its bound.

```

Optimal_Factor(T) {
    Run independently on relatively prime track sets (Th. 2)
    Factor out unshared prime factors from S values (Th. 3)
    While tracks exist without an assigned Oi {
        Place and remove any full sets (Th. 4)
        If all tracks have their Oi assigned, end.
        Let Smax = the largest Si amongst unplaced tracks
        Let Snext = 2nd largest Si amongst unplaced tracks
        Let M be the set of tracks with Si = Smax
        Require: Smax % |M| = 0, Snext ≤ Smax*(|M|-1)/|M| (Th. 5)
        Assign all unassigned tracks in M to Oi, s.t. all tracks in M are
            at a k*Smax/|M|, for all k 0≤k<|M|.
        If all tracks have their Oi assigned, end.
        Require: Snext = c*Smax/|M| for some integer c≥1 (Th. 6), and Snext
            % c = 0 (to make Th. 5 work)
        Use c placeholder tracks w/S=Snext to model existing breaks
    }
}

```

Fig. 3. The pseudocode for the Optimal Factor algorithm. This algorithm has been proven optimal [5] provided all restrictions listed in the pseudocode are met.

Using the theorems we have presented, we can construct an algorithm (Fig. 3) which is optimal [5] provided our restrictions are met. There is one additional restriction that is implied. Because the tracks at a given S value must be evenly spread, the offsets assigned to the placeholder tracks added using Theorem 6 in the previous iteration must fall at offsets calculated using Theorem 5 on the next iteration. This is

accomplished by requiring that S_{next} also be evenly divisible by the number of pseudotracks of that length added during the track conversion phase.

3.4 Relaxed Factor Algorithm

While the Optimal Factor Algorithm generates placements that are optimal in diversity score, there are significant restrictions on segment length and track quantity. However, not all architectures may meet these requirements. Therefore we developed a relaxed version, which may not always be optimal, but does demonstrate good results.

The general framework remains basically the same as the optimal version, although the placement and track conversion phases differ in operation. Before we used restrictions on track length and quantity to ensure that our methods are optimal. In the Relaxed Algorithm, when the optimal restrictions are not met, we instead use a number of heuristics where the goal is the even spreading of the breaks in tracks.

The routing architecture can be considered in terms of a topography, where elevation at a given location is equivalent to the number of breaks at that position. Because “mountains” represent areas with many breaks, and therefore fewer potential routing paths, we attempt to avoid their creation. Instead, we focus on placing our tracks to evenly fill the “plains” in our topography, where the breaks from the additional tracks will have a low effect on the overall routability of the resulting architecture. This topography is represented by the `breaks[]` array, which holds a count of the number of breaks occurring at every position $0 \dots K-1$. In this case, K is the number of positions required to observe the full behavior of the break pattern. As mentioned earlier, this value can be determined by finding the LCM of all S values in the problem.

When each S group is placed, we need to look at the `breaks[]` array in terms of its effect on that S group. The `tracks[]` array summarizes the information from the `breaks[]` array within the potential offsets for a track. Fig. 4 indicates how this is accomplished. Using the `tracks[]` array, we can choose offsets which place new breaks in slots with the fewest amount of existing breaks. Each time a track is placed, both the `breaks[]` array and the `tracks[]` array are updated to reflect the additional breaks. As long as we have at least as many tracks in the current S group as minimum slots in `tracks[]`, this is a simple procedure. However, when there are more minimum locations than tracks, we need to decide which of those minimum offsets should be used.

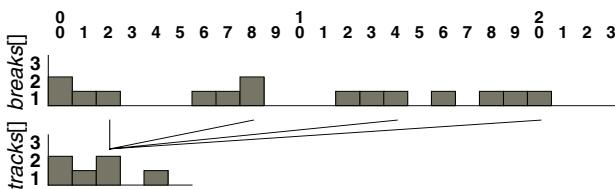


Fig. 4. An example of a `breaks[]` array for $K = 24$, and the corresponding `tracks[]` array for $S_{\text{next}} = 6$. The lines between the two arrays indicate the locations in the `breaks[]` array used to compute the value of `tracks[2]`.

In that case, we use density-based placement. We first compute the “ideal” density of breaks (ie, if they were placed perfectly smooth) if we were able to achieve a perfect placement of the tracks thus far plus our current S group. This represents our placement goal for the S group. In order to achieve this goal, we consider an increasing region of the array, and attempt to bring its density close to the goal density. This region first begins as a single plain and mountain in the `tracks[]` array, and the number of tracks needed to bring the density of that region close to the goal density is added. These tracks are spaced evenly in the plain. The region is now increased to include the next plain and mountain in the array (where we treat `tracks[]` as a circular array). We then add tracks to the new plain in a similar manner to bring the density of the new region close to the goal density. This continues until the entire `tracks[]` array is included in the region, and we have placed all our tracks in the current S group.

Once we have placed all tracks with $S = S_{\max}$, we need to prepare for the next iteration when we place all tracks with $S=S_{\text{next}}$. This means that we need to update the `tracks[]` array to be in terms of S_{next} instead of S_{\max} . The `tracks[]` array is resized to be S_{next} slots in length, and is recomputed using the technique from Fig. 4.

4 Results

A number of terms are used to describe the problems that we have covered in our testing of our algorithms. The value `numT` refers to the total number of tracks in a particular track placement problem, `numS` refers to the number of discrete S values in the problem, `maxS` is the largest S value in the problem, and `maxTS` is the maximum number of tracks at any one S value in the problem. We have also reduced our very large search space by only considering problems where the number of tracks at each S value is less than the S value itself, since Theorem 4 strongly implies that cases with S or more tracks of a particular S value will yield similar results by placing tracks from any full sets one per potential offset. Cases with only one track, or all track lengths less than 3, are trivial and thus ignored. The three terms, `numT`, `numS`, and `maxS`, along with the restrictions above, define the track placement problems we consider.

Our first test was to verify that the Optimal Factor Algorithm yields a best possible diversity score in practice as well as theory. The results of this algorithm were compared to those of the Brute Force Algorithm for all cases with $2 \leq \text{numTracks} \leq 8$, $\leq \text{numS} \leq 4$, and $3 \leq \text{maxS} \leq 9$, which represents 5236 different routing problems. Note that even with these significant restrictions the runtime of brute-force is over a week of solid computation. In all cases where a solution could be found using the Optimal Factor Algorithm, the resulting diversity score was identical to the Brute Force method. Furthermore, we compared the Relaxed Factor Algorithm to the Optimal Factor Algorithm for this same range and found that Relaxed Factor produces optimal results for all cases that meet the Optimal Factor restrictions within that range.

Next, we compared the performance of the Relaxed and Simple Spread to the results of the Brute Force method for the same search space as above to determine the quality of our algorithms. The results for the heuristics, normalized to Brute Force,

are shown categorized by numT, numS, and maxTS in Fig. 5. In these graphs Min is the worst performance of the algorithm at that data point, Max is the best, and Avg is the average performance. In this figure, the Brute Force result is a constant value of 1. Fig. 5 left indicates that both heuristics achieve optimal results in some cases, with the average of the Relaxed algorithm nearly optimal across the entire range. Simple Spread improves with the increasing number of tracks, and both algorithms degrade with an increase in the number of different S values, though only slightly for Relaxed.

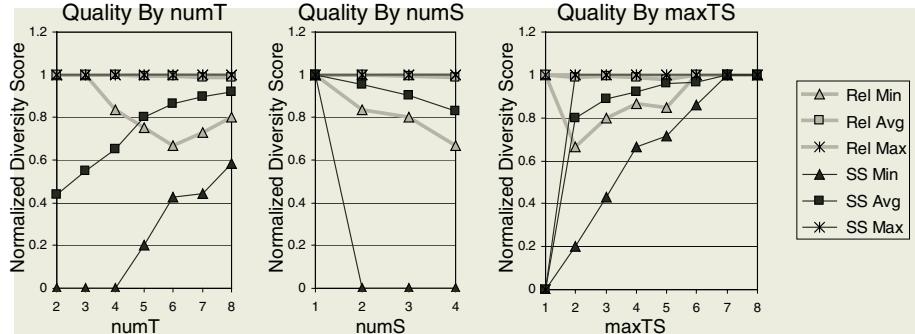


Fig. 5. A comparison of Relaxed and Simple Spread to the Brute Force method, with respect to numT (left), numS (center), and maxTS (right). The diversity scores for each case were first normalized to the Brute Force result, which is represented by a horizontal line at the value 1.

The upswing of both algorithms' minimums towards higher values of numT may be an artifact of our benchmark suite – since we only allow at most 4 unique S values, when there are more than 4 tracks we have at least two tracks with the same S value. Fig. 5 right shows that as the number of tracks per S value increases, the quality of all algorithms improves. The only exception is for the relaxed algorithm when maxTS=1; when there is only one track per S value the Relaxed Algorithm is always optimal. All throughout these tests the Relaxed algorithm is superior to the Simple Spread algorithm. This indicates the critical importance of correlation between S values in track placement. Fig. 5 center also demonstrates how as numS increases, the more difficult it is to solve the problem well, which we expected to be the case. Note that the results for both heuristics are optimal for the case when there is only one S value, as in this case there are no correlations to contend with, and only an even spreading is required.

Next, we used our place and route tool [1] to test the correspondence between diversity score and routability of architectures created using the Relaxed and Simple Spread algorithms. These architectures are based on a tileable coarse-grained architecture similar in structure to RaPiD. This architecture has two length-2 local tracks, four length-4 local tracks, eight length-4 distance tracks, and eight length-8 distance tracks. Note that local tracks do not allow connections between wire segments to form longer wires. We used seven different multi-netlist applications, and created a test case for each application/track placement algorithm. By keeping the proportion of track types constant but varying the total quantity of tracks, the minimum number of tracks required to successfully place and route every netlist in the application onto

this target architecture was determined. Fig. 6 lists the results of this experiment. Performing track placement using the Relaxed Algorithm allowed netlists to be routed with on average 27% (and up to 45%) fewer tracks than the Simple Spread Algorithm.

	OFDM	Camera	Radar	Image Processing	Sort	Matrix Multiply	FIR Filters
Simple Spread	27	23	20	23	23	11	20
Relaxed	24	19	13	15	13	10	11

Fig. 6. The number of tracks in our target architecture required to successfully place and route all netlists in an application using the given track placement algorithm.

5 Conclusions

As we have shown, the track placement problem involved fairly subtle choices, including balancing requirements between tracks of the same length, and between tracks of different, but not relatively prime, lengths. We introduced a quality metric, the diversity score, which captures the impact of track placement. Multiple algorithms were presented to solve the track placement problem. One of these algorithms is provably optimal for some situations, though it is complex and works for only a relatively restricted set of cases. We also developed a relaxed version of the optimal algorithm, which appears to be optimal in all cases meeting the restrictions of the optimal algorithm, and whose average appears near optimal overall.

We envision two situations where this presented research can be applied. First, we believe that there is a growing need for automatic generation of FPGA architectures for systems-on-a-chip. Domain-specific FPGAs can achieve much better area, performance, and power than standard FPGAs. Furthermore, because the FPGA subsystem will be within a custom fabricated SoC, the advantage of pre-made silicon of commodity FPGAs is irrelevant. Second, these techniques can be used as a guideline for manual FPGA designers to potentially improve routing architecture quality.

References

1. K. Compton, S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems", *International Conference on Field-Programmable Logic and Applications*, pp. 59-68, 2002.
2. D. C. Cronquist, P. Franklin, C. Fisher, M. Figueroa, C. Ebeling, "Architecture Design of Reconfigurable Pipelined Datapaths", *Twentieth Anniversary Conference on Advanced Research in VLSI*, 1999.
3. J. R. Hauser, "The Garp Architecture", *University of California at Berkeley Technical Report*, 1997.
4. V. Betz, J. Rose, "Automatic Generation of FPGA Routing Architectures from High-Level Descriptions," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 175 – 184, 2000.
5. K. Compton, S. Hauck, "Track Placement: Orchestrating Routing Structures to Maximize Routability", *University of Washington Technical Report UWEETR-2002-0013*, 2002.

Quark Routing

Sean T. McCulloch¹ and James P. Cohoon²

¹ Ohio Wesleyan University, Department of Computer Science, Delaware,
OH, 43015 USA

² University of Virginia, Department of Computer Science
Charlottesville, VA 22903, USA

Abstract. With inherent problem complexity, ever increasing instance size and ever decreasing layout area, there is need in physical design for improved heuristics and algorithms. In this investigation, we present a novel routing methodology based on the mechanics of auctions. We demonstrate its efficacy by exhibiting the superior results of our auctionbased FPGA router QUARK on the standard benchmark suite.

1 Introduction

Most typical routers are based on a *sequential* strategy. With this methodology, one of the nets to be routed is chosen first, and is given a path anywhere among the initially unclaimed set of routing resources. Then a second net is routed in the space unused by the first net, and so on, until the last nets must find a path that has not been used by the preceding nets. We feel that the sequential strategy has inherent limitations that impact both solution quality and runtime.

The main contributions of this investigation are a routing methodology based on the mechanics of an auction that supports a simultaneous routing philosophy; and an FPGA router QUARK that implements a version of our methodology. The idea of using an auction for decision-making is not new, and appears with some frequency in the operating systems literature. One particular area is in the allocation of resources in distributed environments. Tasks are given virtual money with which to bid on the various resources in the system. When a resource becomes available, all tasks can bid on that resource. The highest bidder pays for the resource and utilizes it [GAG95].

There are also two similar negotiation-based routers that have some slight resemblance to our methodology—the PathFinder and NC routers [McMU95, CHAN00]. The routers represent an FPGA as a graph. Initially, each net is assigned an optimal path, even if it conflicts with the paths of other nets. The algorithm iterates until there are no conflicting assignments. The current cost of a shared resource is set to the number of nets that want it. Each net then finds a new shortest path. Because the shared resources become more expensive, nets heuristically end up negotiating an alternate path. Although this process is not dependent on an initial net ordering, it is not a truly simultaneous strategy. In practice, nets are placed down in order of congestion and are typically ripped up in that order if another net overlaps any part of that net's path.

In our methodology, the entire auction of all the routing resources takes place concurrently. This property has led to a different computational model that represents

a novel approach to performing simultaneous routing. In addition, our negotiation processes are both different and more personalized.

2 Basic FPGA Auction Methodology

The concept of our auction-based routing methodology is straightforward—the pins of an FPGA are resources that can be bid upon by the nets. Each net seeks to control a set of pins that realize a complete detailed route for that net. For discussion ease, we define two terms.

- A *pin-auction* is the local auction of a single specific pin. The auction generally consists of several nets bidding for the right to route on that pin.
- A *chip-auction* is the complete auction process over the entire circuit. Thus, this auction comprises all of the various pin-auctions.

Thus, a run of an auction-based router corresponds to a single chip-auction, where a chip auction consists of collection of pin auctions, one for each pin.

Only one net can win a given pin-auction, and thus have the right to be routed upon that pin. The goal of each net, while working independently of the other nets, is to win sufficient pin auctions to realize its detailed routing. The chip auction completes successfully after all nets have achieved a detailed routing. It is important to stress that all of the pin-auctions are taking place simultaneously. The individual pin-auctions start when the chip-auction starts and finish when the chip-auction completes. This requirement gives quick proof to our claim that QUARK is a simultaneous router.

2.1 Income

The algorithm begins with each net being given an initial allocation of funds. These funds are normally the only source of assets available to a net for bidding on various pin-auctions. An alternative method would have been to have periodic “pay periods” for nets. For example, nets that are losing several pin-auctions could be given extra money. However, it is our experience that routinely adding income to the system merely cause the prices in the pin-auctions to increase and hampers the chip-auction’s ability to finish. Thus, we recommend limiting the application of additional funds to extreme cases.

After a net has been given its money, the net then places its initial bids. We view this initial bidding as being a distinct process from subsequent bidding actions. The initial bidding process must select the specific path to realize the net. Subsequent bidding processes generally only require checking of pin-auctions and updating of bids if necessary.

Once each net has placed its initial set of bids, the chip-auction enters its iterative main phase. In each iteration of the main phase, all nets are given an opportunity to place or modify bids in the various pin-auctions as they see fit. The nets make bids in such a way as to eventually claim ownership of pins that realize a complete detailed route. If two or more nets enter a pin-auction, the first net processed with a maximal bid has current control of the pin.

A net is considered to have a *complete detailed route* if the set of pin-auctions that it is currently winning comprise a path that would be a legal detailed route for the net. There are no “freeze points” where if a net is winning a pin, it can hold it. Doing so would violate the simultaneous nature of the router. As stated before, there is only one chip-auction for the entire run of the router, and that all nets must constantly have bids on the pins that they need to realize their route.

The chip-auction completes *successfully* on the first iteration in which all nets have a complete detailed route. The chip-auction completes *unsuccessfully* if at some iteration it is determined that it is not possible for a given net to realize a complete detailed route. It is possible for the chip-auction to continue indefinitely, with neither of these criteria being met. However, this situation does not seem to arise in practice.

Figure 1 is an example of the bidding process at work on a sample block. Observe that several nets have active bids on different pins of that block at the current time. It is responsibility of each net to ensure that it is bidding sufficiently high in its various pinauctions to have control of pins that realize a complete detailed route for itself.

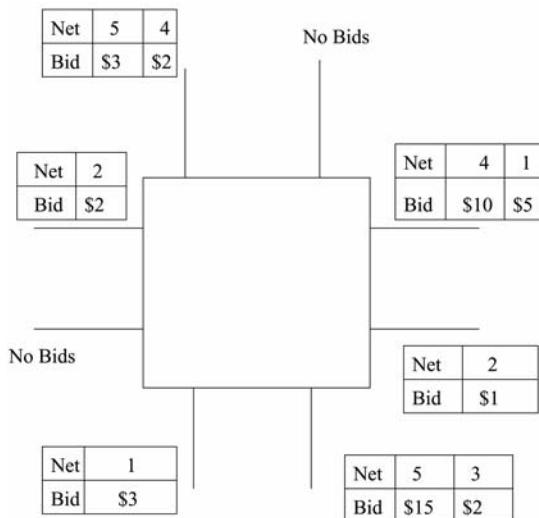


Fig. 1. Hypothetical bidding scenario.

3 Design Details

We now turn our attention to design details. Firstly, there is the issue of the bidding agents. In our model, each net is an active bidding agent, making decisions on which pin-auctions to participate, and once participating, on how much to bid. As a consequence, routing decisions can be made on a low level, which in turn means *a priori* that there is no general routing policy for deciding which nets gain which pins. Thus, each net individually determines how to best realize its route. Because the nets are simultaneously performing bidding operations, it makes sense to give each net its own view of the state of the various pin-auctions.

An important benefit of delegating these low-level decisions is that a routing tool is then free to allow different nets to use different routing algorithms for determining on which pins to bid. We call these local decision processes the net's *personality*.

While it is important to enable the nets to act individually, and simultaneously, it is also helpful to have a central authority, which we call the *overseer*, for policy decisions that require more information than can be found with regard to an individual net or pin.

3.1 Income and Priorities

When assigning a net an initial amount of money at the beginning of the chip-auction, the amount should be sufficiently large, to give the net some flexibility with regard to the bids of other nets. For example, suppose a net was given only \$10 and was limited to bidding integral amounts. If the net needed ten pins to realize its route, it could only bid \$1 per pin. It would have no excess money in a competition with other nets for its pins.

The initial income given to each net is a very natural way to implement a priority structure. In some designs, there is a need for a *critical net*—a net of high priority that must be given preference in finding its preferred path. In our design, a critical net can be given more initial funds. Heuristically, it is more likely in practice to win the pinauctions along its preferred path. If a net is absolutely critical and must be given its preferred path at the expense of the other nets, then it can be effectively given infinite funds to ensure that it gets that path.

3.2 Overseer

To ensure that the flow of the auction is simultaneous, it is important that there be a rapid flow of information regarding bids amongst the nets. The overseer must process bids quickly, and neither go into a failure state nor allow the router to end prematurely because nets have had an inadequate opportunity to bid against each other.

The overseer can determine whether a chip-auction has successfully completed by simply polling the nets whether they have won a complete detailed route. However, determining whether a chip-auction will be ultimately unsuccessful cannot be answered algorithmically (The problem is a variant of the Halting problem). Thus, the overseer must implement a heuristic to recognize this eventuality.

To decrease the likelihood of an unsuccessful routing, an overseer can be helpful in the case of a net lacking sufficient funds to complete any possible path to realize its detailed route. The routing analogy of eminent domain can be applied in such situations —the overseer steps in and in some manner assists the net in acquiring its path. Care needs to be taken so that the implementation of eminent domain does not effectively turn the router into a sequential one by simply assigning a path outright to the indigent net.

It is the overseer that also extracts the technology files; creates an internal representation of the FPGA that meets the specifications of the technology files specify; and extracts the global routing information for each net.

3.3 Pins

Our methodology gives individual pins the responsibility of processing the pin-auctions associated with them. As a result, the pin reports which net is currently winning its pin-auction. Furthermore, the pin ensures that only the current winner of the pinauction has the right to be routed on it.

3.4 Nets

Our net representation is also markedly different from similar representations found in traditional routers. For example, nets must have the ability to receive money and to bid this money in the pin-auctions. Most importantly, the nets must have the ability to bid on pin-auctions in a effective manner that realized a complete detailed route. This decision making is quite different from that found in previous routers. Routers traditionally have enforced routing decisions at a high-level and have required all nets to all perform homogeneously.

The placing of routing decisions at the net level has several benefits. By allowing different nets to have different personalities, we add great flexibility to the router's functionality. By changing the personality of an individual net, it is possible to optimize that net's routing path goal without impacting the routing decisions of the other nets. In addition, it is also possible to individually change the routing qualities that the nets are trying to optimize. For example, it is possible to route certain nets while trying to minimize delay, and to route other nets to minimize net length.

4 Personalities

The concept of a net personality is a novel one, and leads to many benefits in the routing process. As mentioned earlier, a net personality encapsulates a collection of routing decision processes. Thus allowing the router to give different nets different personalities based on their structure and importance. Our implementation provides three personalities.

The three personalities do have some commonality due to the nature of the routing task. Firstly in our environment, a net must have the ability to receive money from the overseer. One allocation activity always happens at the beginning of the chip-auction. An allocation can also occur as the result of an eminent domain request.

Secondly, a net must have the ability to make a constructive bidding decision in reaction to a request for bid(s) from the overseer. The algorithms that determine these bidding decisions are the central component of a net's personality. Although personalities will make different decisions as to where to bid, they will all share the goal of finding a complete detailed route. A net typically has three main bidding actions: increasing its bids when losing, decreasing its bids when winning, or deciding to bid on an alternate path if it deems its current path is too expensive. Besides being able to perform such actions, a net must be able to determine when they apply. For example, when a net is being outbid in a pin-auction and has sufficient unallocated funds to gain the pin, the net needs a process for determining what bid is sufficient. It will be an aspect of the net's personality to determine that new bid size.

Thirdly, each personality must have a mechanism for finding an alternate path if it deems the current path that it is attempting to acquire as too expensive. In implementing this mechanism, a net must be able to remove bids, modify existing bids, and place new bids.

Fourthly, a net needs a mechanism for determining when there are more affordable possible paths that it can construct. If an eminent domain request is viable in such a case, the request is signaled to the controller.

The personalities have been designed to optimize for the traditional FPGA routing problem of minimizing channel width and total net length. The first personality that we present is a very basic one. As such, we have named it the *baseline personality*. Because the baseline personality makes very simple decisions, it served as a framework for the succeeding personalities. The other two personalities—the *multiple personality* and the *focused personality*—extended the baseline personality in very different directions with regard to the management of resources. Both directions are heuristically promising in practice. The extensions were

- Allowing a net to bid on more than one possible path.
- Allowing a net to concentrate its resources in a small number of important locations.

We next discuss the three individual personalities in more detail.

4.1 Baseline Personality

The primary goal of a baseline personality net is to realize a detailed routing by trying to win the pin-auctions along its current global routing. (The global routing is maintained as the list of blocks that connect the various terminals of the net together.)

At the start of the chip-auction, the net is given the global route as determined by a global router in a previous step of the design process. Using the global route, an arbitrarily-determined legal detailed route is chosen. A minimal bid is placed upon each of these pins in the detailed route.

The personality favors pins that have no interest among the other nets. This bias helps the net to avoid congestion and to minimize overall resource expenditure. Once every pin in the detailed route has a bid of \$1 on it, the net is ready to enter the main phase of the auction.

In the main phase, when a baseline personality net is signaled to react to the current state of the chip-action, the net first checks the status of each pin-auction along its current detailed route. If a net is winning a pin-auction by a substantial amount, the net heuristically concludes that some other net has lost interest in this pin-auction. The reason being that the net has minimally raised its bid in the past. As a result, the net reduces its bid so that it is only winning by only \$1. The bidding decrease frees up funds to bid in other pin-auctions.

If instead a net is losing a pin-auction and has sufficient funds to gain the pin, the net bids the minimal amount to lead the pin-auction.

If the needed funds are not there, the net enters a re-route state. In this state, the net first tries to make changes that minimally alter the current detailed route. For example, the net might try to switch to a cheaper pin on the same block. However, switch blocks on most conventional FPGAs do not have great flexibility. Therefore, these attempts may not be successful. In such cases, the baseline personality follows a

maze-routing strategy similar in nature to the UPSTART detailed router [McC02]. The UPSTART strategy considers a bounding region around the block that contains the expensive pin for the net. The width and length of the bounding region are specified by parameters. All of the net's bids on pins in the interior of this bounding region are removed. The locations where the detailed routing intersects the bounding box are marked as virtual terminals. An alternate detailed route is calculated that connects these virtual terminals together. If such a path can be found and if the net has sufficient funds to gain the associated pin auctions, then minimal winning bids are placed on these pins.

4.2 Multiple Personality

The baseline personality is clearly a very simple one and definitely has much room for improvement. One interesting way to improve the quality of the routing is to exploit some of the flexibility provided by the auction methodology. For example, we can have a net simultaneously show interest in multiple detailed routes. Sometime later in the course of the chip-auction, the net can commit to some particular routing. We have implemented such a personality and call it our *multiple personality*.

As a way of also demonstrating the fact that different personalities can be designed for different types of nets, we have designed the multiple personality to work on two-terminal nets whose terminals do not lie within the same channel. (If the terminals did lie so, there is a unique shortest path with regard to blocks.)

The key to understanding our multiple personality is to recognize that a two-terminal net with terminal blocks in differing channels has multiple minimum-length paths that are of the dogleg form. A dogleg is a connection in the form of a single stair step, where the step can be oriented either horizontally or vertically. The flat segment can be preceded and/or followed by riser segments of orthogonal orientation. To instantiate a dogleg, we only need to specify its orientation and the length of the risers. Our multiple personality starts by bidding all of its initial allocation as possible in equal amounts on each of these paths.

In subsequent bids, the personality first determines whether there are routes that it is currently bidding on which cannot be won with the available funds. If so, one such route is removed from consideration and the funds are reallocated to the other routes. In particular, the funds are first spent on routes that are most in danger of being lost and then on the other routes.

If the path that was out-bid was the last one being considered by the personality, then the net enters a re-route phase. In rerouting, the personality examines all possible dogleg paths. If there is a path that the net can afford, it is taken. The net then bids all of its funds along this path, in the expectation that the bids will be sufficient enough to retain control. If this action proves unsuccessful, later iterations will try a different dogleg path that the net can afford. The process continues until a net retains the path that was found, or the net realizes that all possible paths cost more money than the net has. In this case, an eminent domain request is considered.

4.3 Focussed Personality

Just as the multiple personality was designed to be used for two-terminal nets, so have we designed the focused personality to be used on particular types of nets. We expect

that a “focus attention on a set of pins that are important to win” strategy would work well if we focus the bidding on certain pins of a multi-terminal net. We define a *branching point* of a multi-terminal net to be a point that has more than two edges incident on it. In QUARK, we treat all blocks where the global route has greater than two edges incident on the block as a branching point. The personality assumes that the global router has intelligently placed these branching points. Therefore, it is important to win the bid on these pins.

The focused personality divides the initial money resources into two main parts. The amount to be used in bidding on branching points, *branching money*; and the amount to be used in bidding on the remainder of the routing, *connection money*. Clearly, the balance between these two amounts is important. If one component does not receive sufficient funds, then the net will be out-bid by some other net and thus have trouble completing a routing.

QUARK allocates enough connection money such that a bid of \$2 can be made on each pin in the given global route. This allows the router to find connections that are longer than the length given by the global route. In doing so, it will be necessary to give some pins \$1 bids. Another benefit of this behavior is that a net can bid more than \$2 on certain pins by reducing the bid of other pins to \$1. The different gives the router some flexibility in finding a connection path that it can actually keep.

At the beginning of the chip-auction, the division between branching money and connection money is made. The branching money is initially divided evenly among all pins on all branching points.

Once the branching points are bid, the net bids on the connections between branching points. It is important to observe that all effective routings segments leaving branching points either eventually lead to a single terminal, or a single pin on another branching point. Our implementation uses a quick maze-routing strategy to connect the two pins. The resulting routing can be of arbitrary length and go in arbitrary directions. The limitations are based only on the amount of connection money.

When the router signals a focused personality net during the main bidding phase, the net makes two separate checks: one for the connection paths, and one for the branching points. Each of the current connection paths are examined to make sure that the net is winning the pin-auctions along the entire path. If a net is winning, then an additional check is made to see if connection money can be freed. If a net is not winning and there is enough connection money remaining to make the bid higher, then that money is spent to increase the bid. If there still is not enough excess money, the net enters the reroute state.

Note that we do not use branching money to help these connection paths become routed. Doing so would slowly siphon funds away from the branching points into the connection paths. This action would violate the principles of this personality, because it would result in the branching points being more likely to be out-bid.

If a branching pin is being out-bid, then the net examines the bids on all of the other branching pins. The net searches for a branching pin that is winning by a amount that allows for some reduction. The reduction will free up branching money to help the out-bid branching pin to regain control of the pin-auction. The process continues until either the out-bid pin has enough excess money to control the pin-auction again, or it is the case that none of the other branching pins can lower their bid without losing their pin-auction. If this condition results, the net makes an eminent domain request.

The re-routing phase for the connection paths uses a maze-routing algorithm to find a different inexpensive connection path.

5 Experiments

We now analyze a series of experiments that we performed on the Quark router. The experiments were conducted using the standard set of benchmarks maintained by the University of Toronto. The benchmarks have been the basis for the testing of a significant number of tools. Different benchmarks have different numbers of nets, blocks, and block layouts. In our experiments, we use the QUARK tool as a detailed router. The placement and global routing were performed by the SPIFFY tool [SELF]. Because SPIFFY is nondeterministic with regard to its output, five separate runs of SPIFFY were generated for each benchmark. Besides producing a placement and global routing, each run also heuristically specified an expected channel width given complete switchboxes.

Our first set of experiments tested the functionality of the various personalities relative to each other. For these tests, we created four versions of QUARK. The versions differed in their use of various personalities. By using these tests, we are able to make a preliminary decision on which versions of QUARK are best in practice.

- QUARK-BASELINE: every net uses the baseline personality.
- QUARK-MULTIPLE: all nets that qualify for the multiple personality use that personality, and all other nets use the baseline personality.
- QUARK-FOCUSSED: all multi-terminal nets use the focussed personality, and all two-terminal nets use the baseline personality.
- QUARK-ALL: all multi-terminal nets use the focussed personality, all two-terminal nets that qualify for the multiple personality use that personality, and the remaining two-terminal nets use the baseline personality.

A summary of these results are presented in Table 1. This table shows the minimum channel width successfully routed to by the various versions of Quark, along with the average smallest channel width of several runs.

We also note best results tend to be generated by the multiple personality and the baseline personality. Although the baseline personality is slightly better on average, it is significantly slower to run. The results suggest an interesting composite algorithm—run QUARK-MULTIPLE until it discovers a channel width that is too small for it to successfully route. The router then switches to using the slower but more effective QUARK-BASELINE. We call this version QUARK-PRIME. QUARK-PRIME required several minutes to produce its solution for the smaller instances. For the largest instances, it required slightly more than 200 minutes on a Solaris workstation.

Table 2 compares QUARK-PRIME to several state-of-the-art routers: Graph-based router [ALEX98], SEGA tool [LEMI93], CGE tool [BROW92], and VPR tool [BETZ97]. Because the SEGA and CGE tools are designed for different architectures, their results are combined. We note that only VPR betters QUARK-PRIME performance. However, VPR is not running the same instances as QUARK-PRIME. The VPR layout system uses a technology mapper to reduce the number of logic blocks that is unavailable to QUARK-PRIME. This table shows the minimal channel width routed to by each of the routers on these benchmarks.

Table 1. Channel widths successfully routed

Circuit	QUARK BASELINE		QUARK MULTIPLE		QUARK FOCUSSED		QUARK ALL	
	Avg	Best	Avg	Best	Avg	Best	Avg	Best
d fsm	8.8	8	10.4	9	11	10	10.6	10
9sym	8.0	8	7.4	7	7.6	7	8	8
term1	5.8	5	5.8	5	7	5	7	6
apex7	6.2	6	6.6	6	6.2	6	7.4	7
alu2	8.4	7	8.6	8	9.4	9	9.2	9
alu4	8.4	7	9.8	9	12	10	11.4	11
Total	45.6	41	48.6	44	53.2	47	53.6	51

Table 2. Inter-router comparison.

Circuit	QUARK PRIME	Graph- Based	SEGA/ CGE	VPR
d fsm	8	9	10	—
9symml	6	8	10	5
z03	9	11	13	—
term1	5	8	10	5
apex7	5	14	10	4
alu2	7	9	11	6
alu4	8	11	15	7
Total	48 (31)	70	79	(27)

References

- [ALEX98] M. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins, Placement and Routing for High-Performance FPGA layout, *VLSI Design: International Journal of Custom-Chip Design, Simulation, and Testing*, 97-110, January 1998.
- [BETZ97] V. Betz and J. Rose, VPR: a new packing placement and routing tool for FPGA research. International Workshop on Field Programmable Logic and Application, 1997.
- [BROW92] S. D. Brown, J. S. Rose, and Z. G. Vranesic, A detailed router for field programmable gate arrays, *International Conference on Computer-Aided Design*, 382-385, 1990.
- [CHAN00] P. K. Chan and M. D.F. Schlag, New parallelization and convergence results for NC: a negotiation-based FPGA route, *International Symposium on Field Programmable Gate Arrays*, 165-174, 2000.
- [GAG95] R. A. Gagliano, M. D. Fraser, and M. E. Schaefer, Auction allocation of computing resources, *Communications of the ACM*, J88-102, June 1995.
- [LEM93] G. G. Lemieux and S. D. Brown, A detailed routing algorithm for allocating wire segments in field programmable gate arrays, *ACM-SIGDA Physical Design Workshop*, April 1993.
- [MCU03] Sean T. McCulloch, Auction-based routing for FPGAs, University of Virginia, Doctoral Dissertation, 2002.
- [MCMU95] L. McMurchie and C. Ebeling, Pathfinder: A negotiation-based performance-driven router for FPGAs, *International Symposium on Field Programmable Gate Arrays*, 111-117, 1995.
- [SHRA87] E. Shragowitz and S. Keel, A global router based on a multicommodity flow model, *INTEGRATION: the VLSI Journal*, 3-16, 1987.

Global Routing for Lookup-Table Based FPGAs Using Genetic Algorithms¹

Jorge Barreiros^{1,2} and Ernesto Costa²

¹ Departamento de Engenharia Informática e Sistemas,
Instituto Superior de Engenharia de Coimbra.

² Centro de Informática e Sistemas da Universidade de Coimbra.
jmsousa@isec.pt , ernesto@dei.uc.pt

Abstract. In this paper we present experiments concerning the feasibility of using genetic algorithms to efficiently build the global routing in lookup-table based FPGAs. The algorithm is divided in two steps: first, a set of viable routing alternatives is pre-computed for each net, and then the genetic algorithm selects the best routing for each one of the nets that offers the best overall global routing. Our results are comparable to other available global routers, so we conclude that genetic algorithms can be used to build competitive global routing tools.

Introduction

With the increasing growth of the complexity of electronic devices, the good performance of synthesis tools is critical for the success of design and manufacturing of non-trivial projects. Concerning the development of FPGA (*Field Programmable Gate Arrays*) systems, one of the fundamental tasks of these tools is to perform the routing of the circuit constrained to the limited resources available in the device. Although a lot of research has been made in this area [4,5,6,7,8,9,10,11,12,13], little has been done concerning the study of the feasibility of using genetic algorithms [2,3] to generate the required routing (see on [17] for a survey of genetic algorithms used for VLSI design). With this purpose, in this work we developed a global routing tool based on the application of a genetic algorithm for selection for viable routing paths on a LUT-based FPGA. Results are very close to those of other available tools, so we believe that, with further refinement, the genetic approach can became competitive with current techniques.

¹ This work was partially supported by the Portuguese Ministry of Science and High Education, under program POSI

FPGA Architecture

The FPGA architecture we used to test our algorithm is similar to the one described in [1] (see Figure 1). In this model, there are four distinct types of blocks, interconnected by several routing channels:

- **L-Blocks** – These blocks implement any logic function with N inputs. These inputs are equally distributed over all sides of the block, and a single output is provided on the right side of the block. We chose to select N=4 for all our experiments. These blocks can also be referred to as *logic cells* or *logic blocks*.
- **IO Blocks** – These blocks represent the input and output pins of the FPGA. Since the number of pins in real FPGAs is far greater than those that fit along the square formed by the L-Blocks, we consider that multiple I/O pins are contained in each I/O block. In this work, we consider that two I/O pins are available for each I/O block.
- **C Block** – These are the blocks through which the L and IO blocks are connected to routing channels. The number of alternative tracks to which a connection from a L-block or IO-block can be made, characterizes these blocks. In this work, this value was predefined to be equal to the width of the routing channel.
- **S-Block** – These blocks allow switching the tracks between different routing channels. Their *Flexibility* is defined as the number of different outputs to which a given input can be connected. We use S-Blocks with a *Flexibility* of 3 in this work (see Figure 2).

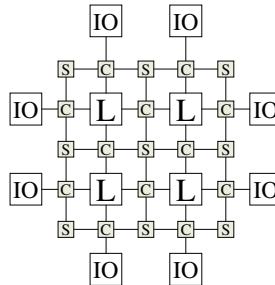


Fig. 1.- Architecture of the LUT-based FPGA

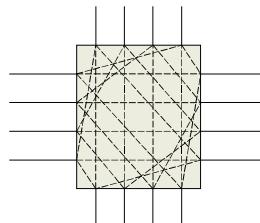


Fig. 2. S-Block configuration used in this work

Genetic Algorithms

Genetic algorithms (GA) are a group of stochastic optimization techniques [2,3]. These algorithms work with a set of candidate solutions to the problem (a population of individuals, using GA terminology) and seek to evolve them using concepts derived from genetics and natural selection. Each individual holds enough information (the *genes*) to describe a possible solution to the problem. They are evaluated regarding the quality of that solution (i.e. their *fitness* is computed), and a probabilistic selection method (based on the fitness of each individual) is used to find group of individuals (the *parents*) that will be used to create the next *generation* of the population. They individuals of the new generation are created by applying genetic-inspired transformations (*operators*) to the parents. Among these transformations we can find *mutations* and *crossover*. The mutation operator does random changes to the genes, while the crossover combines parts of the genes of two parents to create a single individual. After multiple iterations, the quality of the population will increase, and when a predetermined stopping condition is met, the solution for the problem will be found on the genes of the best individual of the last generation.

1. Randomly initialize population
2. **While** stopping condition is not met
 - a) Evaluate population
 - b) Select parents
 - c) Crossover
 - d) Mutation
 - e) Substitute old population

Fig. 3. Simple Genetic Algorithm

There are, of course, multiple variants of this simple framework.

Global Routing for FPGAs

The synthesis of circuits for FPGAs can be decomposed in several steps:

1. **Logic optimization** – The circuits are optimized, shrunk and redundant logic is eliminated.
2. **Technology Mapping** – The elements of the circuit description are assigned to specific classes of resources available in the actual FPGA. For example, logic expressions are broken into sub-expressions implemented in a single logic block.
3. **Placement** – When there are multiple components in the FPGA capable of implementing a specific aspect of the circuit description, a selection needs to be made about which one of them will be used (for example, what logic blocks will actually be used to implement the sub-expressions generated by the technology mapping?)
4. **Routing** – The communication resources available in the FPGA are used to connect adequately all the components of the circuit. Ideally, the routing should

use the smallest routing channel width and have minimum source-to-sink distance, to maximize circuit speed.

Some approaches combine some of these steps or further refine them into additional iterations. The last step is sometimes separated in global and detailed routing. The objective of the global routing step is to ensure a balanced occupation of available channels. The global router decides which routing channels will be used, without deciding about specific track usage inside those channels. Detailed routing will complete the process, by making the necessary track assignments within those channels. Frequently, it may be impossible to perform detailed routing with the same channel width found by the global routing because of restrictions on the flexibility of S-Blocks (the *routing anomaly*, see [1,4]), so the global routing channel width is actually a low bound for the width of the routing channels after detailed routing.

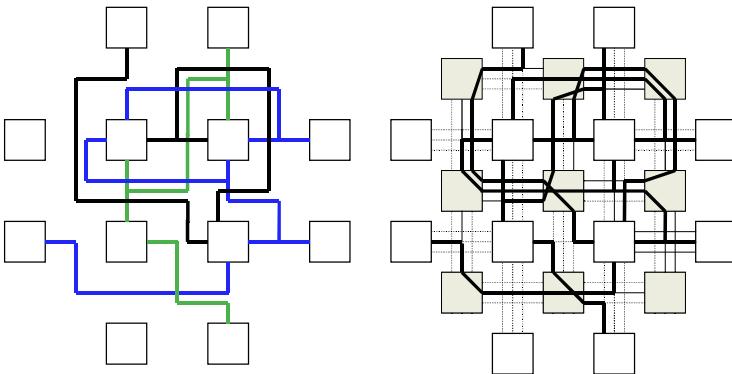


Fig. 4. Global vs. detailed routing

For further details about previous work on global and detailed routing, see [4,5,6,7,8,9,10,11,12,13]. For information regarding other steps of the synthesis process, please consult [14, 15, 16].

The Routing Algorithm

We used a two-step approach for building the global routing:

1. For each net of the circuit, generate a set of alternative routing paths.
2. Find, for all nets, the combination of alternative routing paths that offers the best overall global routing

This approach allows using different optimization techniques for each step. We developed three different heuristics for creating the alternate routing paths, and used a genetic algorithm for making the optimization described in step 2, according to the objectives of this work.

Generation of Alternate Routing Paths

The generation of the alternate paths is made with the following algorithm:

For each net,

1. Generate one routing path that connects all nodes in net
2. While the desired number of alternate paths isn't met, build new routing path by randomly selecting and mutating one of the previously built paths

This algorithm will first generate one model net. All other alternative paths are deviations from this model net. The rationale behind this option is that it should be more efficient to build alternate solutions by making minor changes to pre-existing ones than re-computing a new net from scratch.

Computing the Model Routing Path for Each Net

We developed three different algorithms for computing the first model routing. Two are graph-based search techniques and the other is based on a heuristic algorithm for computing rectilinear Steiner trees. The first graph-based technique is simply a greedy search from the source node to each sink node. This algorithm is fast but offers, as could be expected, poor quality solutions. The other graph algorithm is based on Dijkstra's algorithm [18] for finding the shortest path between two nodes. Every source/sink is connected sequentially using Dijkstra's algorithm, and the cost of previously taken segments is reduced. This algorithm offers reasonable performance, but is not as good as our heuristic for computing near optimal rectilinear Steiner trees (RST). A RST is the shortest rectilinear tree connecting a given set of points. Computing a RST instead of using graph-transversal algorithms eliminates the problem of having to route multipoint nets as groups of source/sink paths. The performance of those algorithms is very dependant on the order by which those paths are processed, and it isn't clear how to determine what the optimal order is, although some heuristic can be used (i.e. longest paths first). Our approach to computing the RST for a given set of points is based on a hill-climbing search algorithm that finds an optimal partition of those points into smaller RST, as explained below and illustrated in Figure 5, where a tree representation of the decomposition is presented.

The points are partitioned across horizontal and vertical axis that transverse de median point of each set, sharing one point to ensure a connected net is built. This partitioning is applied recursively until no more than a predetermined number of points (for illustration purposes, this number is 3 in Figure 5) are contained in every partition. The rectilinear tree at the nodes is built by constructing an axis along the median (vertical or horizontal) point, and connecting all other nodes to that is by transversal segments.

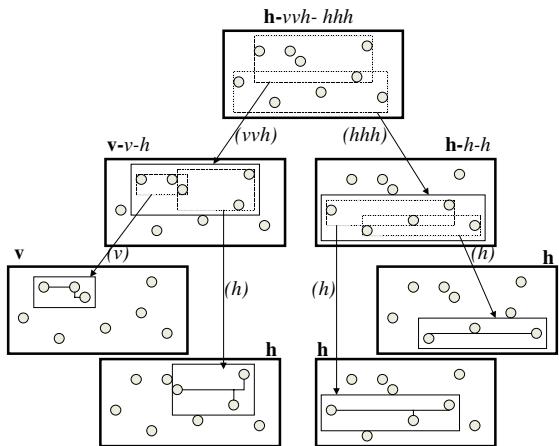


Fig. 5. Example of decomposition of point set into multiple RSTs

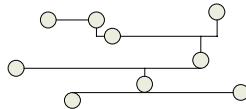


Fig. 6. (non-optimal) Rectilinear tree resulting from the decomposition illustrated in Figure 5

The actual decomposition is dependant on whether a vertical or horizontal axis is chosen on each node of the decomposition tree. This information can be represented linearly by a vector² where the first element represents the orientation of the axis on the root node, and the remaining information is split in half and interpreted accordingly by the left and right descendants of the root. For example, for the specific decomposition shown in Figure 5, this string is “hvvhvv” for the root node.

The simple linear representation of the decomposition is important, because it enables us to use a simple hill-climbing optimization algorithm for finding the partition that offers the smallest length rectilinear tree. Figure 7 shows the rectilinear trees computed for a random point set with different values for maximum points per leaf. The dashed rectangles represent the points contained in a single leaf. (These trees are not optimal; they just represent a possible result of the algorithm.)

Generating Mutations from Model Path

Deviations from a model routing path are built by applying a mutation to previously built nets. The algorithm is based on the idea of generating an intersecting rectangle over the model network and re-routing the internal connections on the borders of that rectangle. For space considerations, some details are omitted (e.g. handling nodes internal to the rectangle, or multiple branches), but the general idea is illustrated in Figure 8.

² Although padding may be required for unbalanced decomposition trees.

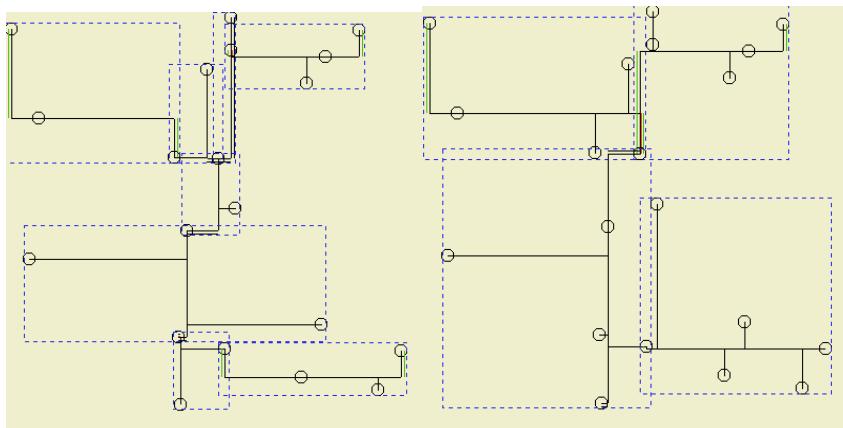


Fig. 7. Rectilinear trees for random set of points, with maximum of 4 points per leaf (left) and 8 points per leaf (right).

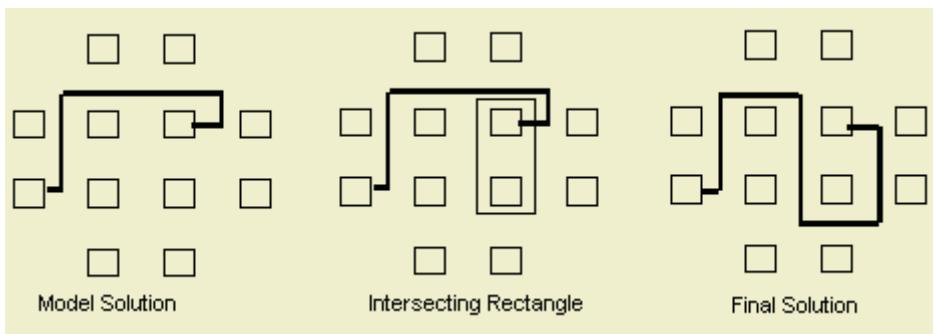


Fig. 8. Generating alternative routing paths.

Genetic Algorithm

The genetic algorithm decides among the possible alternate routes which ones will be used in the global routing. Each individual (solution) is represented only by an integer vector³. Each position of this vector is associated to a net of the circuit, and in that position it is held an index that identifies what is the chosen routing alternative for that net.

After some tuning tests, the following configuration was found to offer the best results for the GA:

- Population size: 50
- Maximum number of generations: 2500
- Mutation rate: 8%
- Crossover rate: 80%

³ Actually, some auxiliary data is also kept in each individual to help to speed up some computations. That detail isn't relevant to understand the main operation of the algorithm, so it is left out of the description.

- Elitism: 4 individuals
- Lamarckian learning⁴

Results

We used circuits from LGSynth93 test suite [19] to benchmark our algorithm. A set of 30 circuits of diverse complexity was chosen. The circuits were placed using VPR [10] and our tool was used to generate the global routing. For comparison purposes, VPR was also used to globally route the same circuits. A single run of both tools was used for all circuits. The test machine was a Pentium 3 processor at 866MHz with 384 MB RAMS. The results are presented in Table 1.

As we can see, although VPR offers slightly better results, both in terms of CPU time and track number. However, results are reasonably close, and in some cases the AG actually computed a better solution. We believe the small difference in performance can be further reduced if other heuristics for net generation are tried and more exhaustive tuning is made.

Additionally, the AG seems to have a very large performance advantage when compared with reported results for older global routers [11], which suggests that our approach seems valid and worthy of further development.

Conclusions and Future Work

Our main goal was to investigate the feasibility of constructing a global router for FPGAs using genetic algorithms. We believe that our results show that this class of algorithms is worthy of consideration when designing this kind of tools. Results seem to suggest that the AG offers performance that is comparable with that of other known algorithms, although in the current implementation it is on average slightly inferior. However, this difference in performance isn't very significant, and further improvement of the algorithm may offer better results. Although some tuning tests were made, a more exhaustive set of tests might reveal some better parameterization. Also, one improvement that might significantly improve efficiency would be to generate alternative routings dynamically, only when judged necessary, instead of pre-computing a fixed-size set of alternate paths. Further work could also be done by adapting the algorithm to use architectural features of current FPGAs, like routing segments of heterogeneous size.

⁴ A quick, non-exhaustive local search is conducted for each individual, and if any improvement is found it the individual is changed accordingly.

Table 1. Results for LGSynth93 test suite benchmark

Circuit	Channel	Channel	CPU (s)	CPU (s)
	Width (VPR)	Width (AG)	(AG)	(VPR)
pdc	20	20	4299	8154
ex1010	11	13	2609	952
spla	16	17	1881	2654
s298	9	10	1247	814
seq	13	14	1158	278
alu4	11	11	807	184
misex3	12	12	661	210
ex5p	15	14	518	824
apex3	13	12	558	134
pair	8	9	476	59
C6288	6	6	89	11
i8	9	9	200	95
table5	11	12	109	33
cordic	10	10	99	68
C3540	8	9	40	27
i9	6	7	116	27
x3	6	6	273	12
vda	9	9	77	35
s1238	7	7	43	13
e64	9	8	158	45
planet	6	7	29	14
i7	4	5	12	93
mm9b	7	7	42	8
i6	5	5	13	61
alu2	7	7	33	8
too-large	7	8	8	8
C880	8	7	31	5
example2	6	6	48	3
term1	5	5	4	4
misex2	5	5	1	1
TOTAL	269	277	15639	14834

References

- [1] "On two-step routing for FPGAs", Lemieux, G. ; Brown, S. ; Vranesic, D. , International Symposium on Physical Design, Abril 1997
- [2] "Genetic Algorithms in Search, optimization and machine learning", Goldber, D. Addison Wesley, 1989
- [3] "An introduction to Genetic Algorithms", Mitchel, M., MIT Press, 1996
- [4] "New performance-driven FPGA routing algorithms", Alexander, M. ; Robins, G; Design Automation Conference, June 1995
- [5] "A detailed router for Field-Programmable gate arrays", Brown, G. ; Rose, Z. ; Vranesic, G., IEEE Transactions on Computer-Aided Design, Vol. 11, No. 5, May 1992
- [6] "Plane parallel A* maze router and it's application to FPGA's"; Palczewski, M. ; Proceedings of the Design Automation Conference, 1992.
- [7] "New performance-driven FPGA routing algorithms", Alexander, M. ; Robins, G; Design Automation Conference, June 1995
- [8] "Performance-oriented placement and routing for Field-Programmable gate arrays", Alexander, M; Cohoon , J. ; Ganley, J. ; Robins, G., European Design Automation Conference, Sept. 1995
- [9] "New performance-driven FPGA routing algorithms"; Alexander, M ; Robins , G. ; IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 15, N. 12, Dec. 1996.
- [10] "Directional bias and Non-uniformity in FPGA global routing architectures", Betz, V.; Rose, J. ; IEEE/ACM International Conference on Computer Aided Design, 1996
- [11] "LocusRoute: A parallel global router for standard cells"; Rose, J. ; Proceedings of the Design Automation Conference, 1988.
- [12] "A detailed routing algorithm for allocating wire segments in field-programmable gate arrays"; Lemieux, G; Brown, S; Proceedings of the ACM Physical Design Workshop, 1993.
- [13] "A performance and routability driven router for FPGAs considering path delays", Lee, Y. ; Wu, A.; IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol 16, n°2, Feb. 1997
- [14] "Optimal FPGA mapping and retiming with efficient initial state computation", Cong ; J. ; Wu, C. Proceedings of the 35th Design Automation Conference, 1998.
- [15] "Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams", Chang, Shih-Chieh ; Marek-Sadowska, M. ; Hwang, T., IEEE Transactions on computer aided design of integrated circuits and systems, Vol. 15, N.10, 1996
- [16] "Combining technology mapping and placement for delay-minimization in FPGA designs", Chen, C.-S. ; Tsay, Y.-W. ; Hwang, T.; Wu, A. ; Lin, Y.-L.; IEEE
- [17] "Genetic Algorithms for VLSI design, layout & test automation", Mazumder, P., Rudnik, E., 1999, Prentice Hall, ISBN 0-13-011566-5
- [18] "A note on two problems in connection with graphs", Dijkstra, E. W.; Numerische Mathematik, vol. 1, 1959
- [19] CAD Benchmarking Laboratory, North Carolina State University, LGSynth93 suite, <http://www.cbl.ncsu.edu/www/>

Virtualizing Hardware with Multi-context Reconfigurable Arrays

Rolf Enzler, Christian Plessl, and Marco Platzner

Swiss Federal Institute of Technology (ETH) Zurich*, Switzerland,
`enzler@ife.ee.ethz.ch`

Abstract. In contrast to processors, current reconfigurable devices totally lack programming models that would allow for device independent compilation and forward compatibility. The key to overcome this limitations is hardware virtualization. In this paper, we resort to a macro-pipelined execution model to achieve hardware virtualization for data streaming applications. As a hardware implementation we present a hybrid multi-context architecture that attaches a coarse-grained reconfigurable array to a host CPU. A co-simulation framework enables cycle-accurate simulation of the complete architecture. As a case study we map an FIR filter to our virtualized hardware model and evaluate different designs. We discuss the impact of the number of contexts and the feature of context state on the speedup and the CPU load.

1 Introduction

Reconfigurable computing fabrics have shown great potential in many high-performance applications that benefit from hardware customization while still relying on some amount of programmability. A major drawback of current reconfigurable devices, in particular field-programmable gate arrays (FPGAs), is the lack of programming models. Applications are compiled (synthesized) to given fixed-size hardware. The resulting configuration bitstream cannot be reused to program a device of different type or size. Thus, to leverage advances in VLSI technology, i. e. increased transistor count and higher clock rates, a complete recompilation is required.

The key to overcome this limitation is *hardware virtualization* [1–3]. In order to achieve hardware virtualization, we have to define a set of basic operators a hardware can execute. Together with a description of the data flow (communication paths between operators) and the control flow (sequencing of operators) a hardware programming model is defined that compilers can target. Processors use a well-established form of hardware virtualization and define an instruction set architecture that decouples the compiler from the actual hardware organization. Achieving virtualization of reconfigurable hardware is more complex. Reconfigurable hardware excels when computations are organized spatially. The

* This work is supported by ETH Zurich under the ZIPPY project and the Wearable Computing Polyproject.

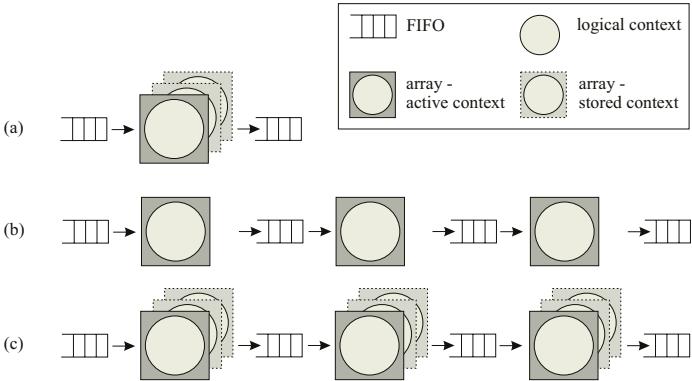


Fig. 1. Models for virtualized macro-pipelining

basic operators will thus have greater complexities than processor instructions and the number of possible operators is very large. Further, the reconfigurability allows to implement many basic operators with just one type of hardware block.

In this paper, we consider data streaming applications that map well to (macro-)pipelines, where one pipeline stage is implemented by one basic hardware block. Our basic hardware block is a 4×4 coarse-grained reconfigurable array. The inputs and outputs of the array connect to FIFO buffers to facilitate data streaming. One set of configuration data for the array is denoted as a *context*. Applications are organized by pipelining several *logical context* executions. Although this model is rather restrictive, it is amenable to true hardware virtualization and targets an important application domain.

Figure 1(a) shows our model with one physical array that is reconfigured to implement logical contexts as needed. To minimize or even hide the reconfiguration time the array stores multiple *physical contexts*. Figure 1(b) displays an alternative implementation with several single-context physical arrays arranged in a pipelined fashion. The arrays are still reconfigured to execute different logical contexts. However, as several contexts run in parallel the throughput increases. Both multiple contexts and physical pipelining can be combined which is shown in Fig. 1(c). All these architectures achieve virtualization as they provide the logical pipeline of array executions as programming model, but differ in their performance and hardware cost.

While there exists already a substantial body of work on coarse-grained arrays, macro-pipelining of stream computations and multi-context devices, a system-level evaluation of the performance and the various features of multi-context devices is missing. To this end, we form a reconfigurable hybrid system by coupling our multi-context array to a CPU. The CPU takes care of data I/O, context loading, and control of the multi-context array. We develop a system-wide, cycle-accurate architecture model and investigate the following issues by means of a co-simulation environment: First, we determine the performance gains for the hybrid over the CPU only, depending on the number of physical contexts.

Second, we try to identify whether and when the capability to resume the state of a previous context is advantageous. Third, we measure the CPU load for the different designs.

Section 2 summarizes related work. The hybrid architecture model and our co-simulation environment are discussed in Section 3. Section 4 presents an FIR filter case study, while Section 5 discusses the results. Finally, Section 6 summarizes our findings and points to further work.

2 Related Work

PipeRench [1, 4] is a reconfigurable architecture that supports hardware virtualization. The device is organized into a physical pipeline of stripes, which represent the minimal reconfigurable hardware blocks. A stripe’s output is strictly pipelined and connects to the next stripe via an interconnection network. Thus, PipeRench is similar to the model in Fig. 1(b). Fast reconfiguration of stripes is supported by 256 contexts held on-chip. Each stripe comprises 16 processing elements, which implement addition/subtraction or a programmable logic function. Application kernels are mapped to virtual pipeline stages. During runtime, the virtual stages are configured to the physical stripes that are available on the device. The implementation described in [4] features 16 physical stripes.

Multi-context techniques for both fine-grained and coarse-grained reconfigurable devices have been investigated by several researchers. DeHon [5] demonstrated that adding multi-context support to FPGAs can increase computational density. Due to the moderate contribution of the configuration memory to the total chip area, a small number of contexts can be added with reasonable impact on cost. Trimberger et al. [2] introduce a multi-context extension of the Xilinx XC4000 architecture. The proposed device holds eight contexts on-chip. The flip-flops of the device are eight times replicated and each logic cell can write to any of these flip-flops. The authors propose to use the multi-context feature for emulation of arrays of arbitrary size. The fine-grained, memory-poor architecture prefers logic emulation rather than macro-pipelining.

PACT’s XPP device [6] uses cells with a functionality similar to our model, but targets a different execution model. Data is transferred between the cells using a handshake protocol. This ensures that dataflow dependencies are met and makes the computation self-timed. Configurations are loaded on demand using a hierarchical configuration management. A configuration context is not necessarily activated for the whole device at the same point in time. For each cell, the new configuration is activated as soon as the current configuration is not used anymore. MorphoSys [7] integrates a CPU with a coarse-grained, multi-context ALU array. The device holds 32 contexts on-chip.

Our work targets a coarse-grained, multi-context reconfigurable hybrid. The main differences to related approaches are that we focus on macro-pipelining of contexts that execute for a longer time period, use FIFOs to transfer data between contexts, and couple the reconfigurable array with a CPU to form a hybrid device.

3 Architecture Model and Co-simulation

3.1 System Model

We investigate a hybrid reconfigurable device, which couples a coarse-grained reconfigurable unit closely to a CPU core. Figure 2 outlines the basic system model, which comprises the CPU core, instruction and data caches, and the reconfigurable unit (RU). The reconfigurable unit is attached to the CPU via a dedicated coprocessor interface and provides a number of coprocessor registers.

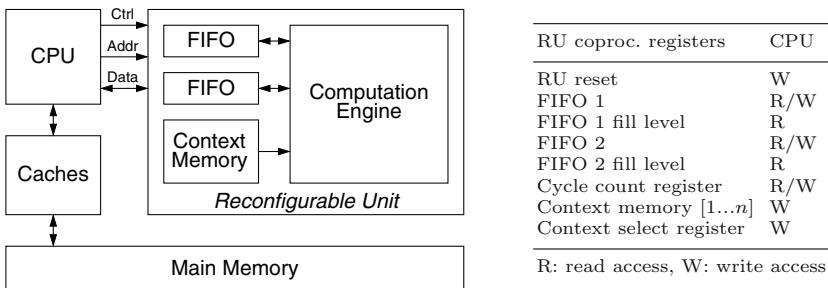


Fig. 2. System model outline

We have developed a co-simulation framework that combines a cycle-accurate CPU model with an RU model specified in VHDL and allows for cycle-accurate simulation of the whole system. Details on the design and implementation of the co-simulation framework have been published in [8].

Currently the RU does not have its own memory access port, but all data communicated to and from the RU is passed via the CPU's register file. On the RU side, data transfers are performed via the FIFO buffers. Both FIFOs are readable and writable by the CPU as well as the RU.

The synchronization mechanism between CPU and RU is similar to the one proposed in the Garp processor [9]. The execution of the RU is started by writing the number of clock cycles the RU shall perform to the *cycle count register*. In every clock cycle, the cycle count register is decremented by one and stops the execution of the RU when reaching zero. By reading the cycle count register the remaining execution cycles can be determined.

3.2 CPU Model

For CPU simulation, we leverage on the SimpleScalar processor simulator [10]. SimpleScalar's CPU model is based on a 32-bit RISC processor architecture and has a MIPS-like instruction set. The CPU core's data and control path as well as the memory hierarchy are widely parameterizable. Thus, the CPU model can be configured to resemble a broad range of CPU architectures, from small embedded CPUs to powerful high-end CPUs.

In order to couple the RU to the CPU, we have extended SimpleScalar with a coprocessor interface. To this end, coprocessor read and write instructions have been added to the instruction set, which allow the CPU to access the coprocessor registers of the RU.

3.3 Model of the Reconfigurable Unit

The RU model comprises two FIFO buffers, the context memory, and the computation engine. Some RU characteristics are parameterizable: the data path width, the depth of the FIFO buffers, and the number of configurations the context memory holds. Another RU parameter determines whether the contexts contain state or not. An RU with context states replicates the registers in the data path in a way that each context is assigned a separate set of registers. An RU without context states provides only one register set that all contexts must share.

The context memory holds a set of configurations for the computation engine. The configuration data is written from the CPU to the RU via the configuration interface. The RU supports the download of full and partial configurations for any of the contexts. The CPU selects a context on the RU for execution by writing the number of the context to the *context selection register*. The context is immediately switched and the CPU can trigger the RU to run by writing the desired number of cycles to the *cycle count register*.

The computation engine is a 4×4 array of homogeneous, coarse-grained cells, which are connected by a 2-level network: direct interconnects between certain adjacent cells, Fig. 3(a), and horizontal buses between cell rows, Fig. 3(b). The computation engine has two input and two output ports, which are connected to the two FIFOs of the RU. Inside the computation engine, they are routed via the horizontal buses.

Figure 4 outlines the data path of a cell consisting of a fixed-point arithmetic logic unit (ALU), several multiplexers and registers. Figure 4(a) shows a cell without context state; all contexts have to share the same registers which are reset on context switches. Figure 4(b) displays a cell supporting context state. All the registers are replicated according to the number of physical contexts. This allows to preserve register values over several context switches. Alternatively, the register can also be reset on a context switch. The ALU implements the common arithmetic and logic operations (addition, subtraction, shift, OR, NOR, NOT, etc.) as well as multiplication. The control signals for the ALU and the multiplexers are part of the RU's configuration. The configuration contains also a constant operator, which can be routed to both ALU inputs.

The configuration of the computation engine is responsible for the functionality of the cells and the routing of the data path between the cells, from the input ports to the cells, and from the cells to the output ports. Since the configuration incorporates constant cell operators, the amount of required configuration bits depends on the datapath width. Given a datapath width of 16 bit, the configuration data results in 918 bits.

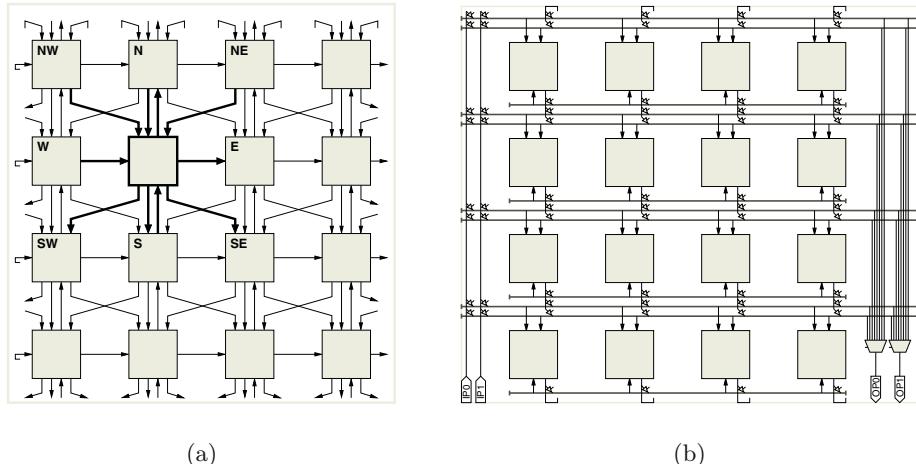


Fig. 3. 2-level interconnect scheme of the computation engine: (a) direct interconnects (*highlighted connections of one cell*), and (b) horizontal buses and I/O ports (*IP_x, OP_x*)

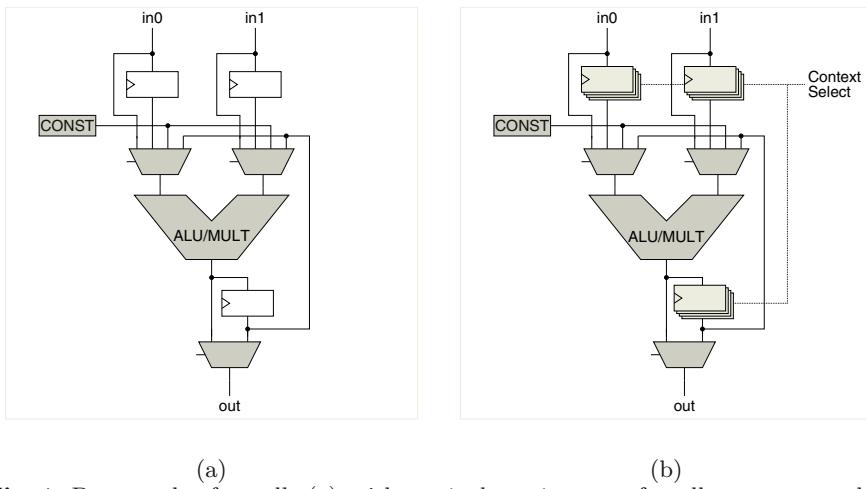


Fig. 4. Data path of a cell: (a) with *a single register set for all contexts*, and (b) with *a dedicated register set for each context*. The shaded parts are controlled by the configuration

4 Case Study and Experimental Setup

4.1 FIR Filter Partitioning and Mapping

As a case study, we have implemented a 56th-order FIR filter on our virtualized hardware. The filter is implemented as a cascade of eight subfilters of 7th-order. The input samples are processed in data blocks. An FIR filter implementation requires delay registers. These registers form the state of the context, which

Table 1. CPU model resembling an embedded CPU

Parameter Class	Setup
Computation units	1 int. ALU, 1 int. multiplier 1 FP ALU, 1 FP multiplier
Caches	32-way 16K L1 I-cache, 32-way 16K L1 D-cache, no L2 cache
Memory interface	32-bit memory bus, 1 memory port
Queue sizes ¹	instruction fetch: 1, register update unit: 4, load/store: 4
Bandwidths ¹	decode width: 1, issue width: 2, commit width: 2
Execution order	in-order
Branch prediction	always not-taken

¹ in number of instructions

must be saved between two executions of the same context. Depending on the capabilities of the reconfigurable array, there are two ways to achieve this:

- If all contexts of the RU share the same register set, the state must be explicitly saved and later on restored. For the filter implementation this is achieved by overlapping subsequent data blocks, which forms an execution overhead.
- If the RU provides dedicated register sets for each context the state is kept automatically. For the filter implementation, no extra cycles are needed for state handling if we can hold all logical contexts on the array.

4.2 System Model Setups

We have set up our system model to study the following cases: CPU only (no RU present), CPU with attached single-context RU, CPU with attached multi-context RU having 2, 4 and 8 contexts, and finally a CPU with attached 8-context RU incorporating a dedicated register set for each context.

The SimpleScalar CPU model is configured such that it resembles an embedded CPU. Table 1 lists the most important CPU parameters. In each experiment, 64K samples organized in data blocks are processed. The size of the data blocks depends on the FIFO depth available on the RU (cf. Fig. 2). We vary the depth of the FIFO buffers between 128 and 1k words.

For the coprocessor cases, a data block is written to the RU, processed sequentially by the eight FIR filter stages (the eight logical contexts), and finally read back. At the beginning, a controller task running on the CPU downloads as many contexts as fit onto the RU. If not all logical contexts fit, the contexts are loaded on demand. Each time a filter context is required that is not present, the controller performs the download by overriding always the same physical context. This is done to hold as many contexts as possible unchanged on the array with the goal to reduce the amount of reconfiguration data that has to be downloaded onto the RU.

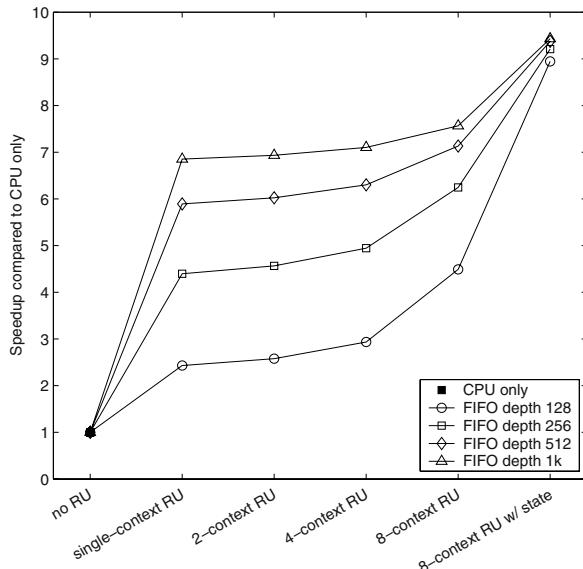
5 Results and Discussion

Figure 5 illustrates the results of the experiments as functions of the device architecture (shown on the horizontal axis) and the FIFO buffer size. The execution time of the filter for the CPU only is 110.65 million cycles. Figure 5(a) shows the speedups relative to this computation time and Fig. 5(b) presents the CPU load. We assume a real-time system that filters blocks of data samples at a given rate. When the filter computation is moved from the CPU to the reconfigurable array, the CPU is relieved from these operations and can use this capacity for running other tasks. However, the CPU still has to transfer data to and from the FIFOs, write contexts to the RU on demand, and control the context switches. The load given in Fig. 5(b) determines the spent CPU cycles normalized to the CPU only system. We point out the following observations:

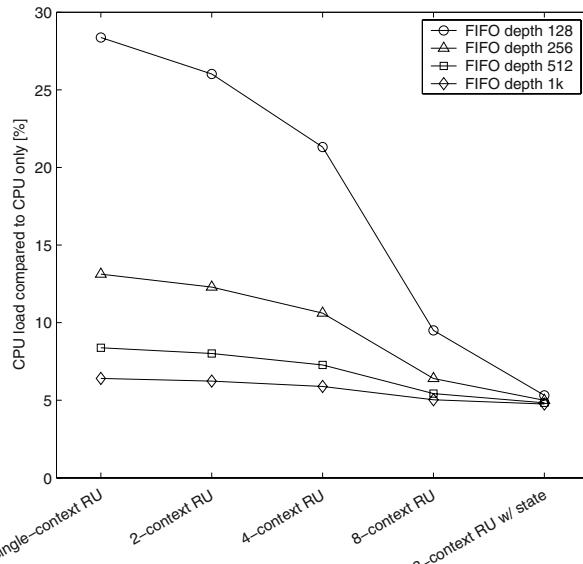
- Hardware virtualization is an extremely useful concept. We were able to run the same filter implementation on reconfigurable array models with different features without resynthesizing the application.
- Using an RU we achieve significant speedups, ranging from a factor of 2.4 for a 128 word FIFO single-context device up to a factor of 8.9 for an 8-context RU that restores context state.
- The performance of the system in terms of speedup and CPU load depends on the length of the FIFO buffers. Enlarging the FIFOs increases the performance and at the same time the filter delay. Practical applications could limit these potential gains by imposing delay constraints. For instance, a 2-context RU using a FIFO with 1k words instead of 128 words improves the speedup by a factor of 2.7, while increasing the latency by a factor of 8.
- Figure 5 shows that a multi-context array storing the context states greatly benefits our application if we can store *all* logical contexts. In this case, we can avoid the overlapping of data blocks. For an 8-context array with a 128 word FIFO the speedup increases by factor of 2.0. In addition, as no reconfiguration is required, the speedup becomes almost independent of the FIFO size.
- Employing a reconfigurable coprocessor not only speeds up the computation but also lowers the CPU load significantly. As Figure 5(b) displays, for a single-context RU the CPU load drops from 100% to 28.4% for a 128 word FIFO and to 6.4% for a 1k word FIFO. Increasing the number of physical contexts the load approaches the asymptotic value of 4.8%, because the CPU task reduces to data transfer and context switches.

6 Summary and Future Work

In this paper, we have discussed the concept of hardware virtualization and the use of multi-context architectures to achieve it. We have presented a co-simulation framework based on a hybrid system model consisting of a reconfigurable unit attached to a CPU. As a case study, we have mapped an FIR



(a) Speedup



(b) CPU load

Fig. 5. Performance figures in comparison to the CPU only case

filter to our virtualized hardware and run it on various architectures by cycle-accurate simulation. The results show that hardware virtualization is a valuable concept and that multi-context features can be successfully employed. Further work includes:

- Implementation of application types that require more complex context sequences (control flow).
- Integration of a dedicated RU memory port.
- Investigation of context prediction and prefetching techniques.
- Development of an area model for the reconfigurable unit in order to quantify the hardware overhead introduced by the multi-context features.

References

1. Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R.R.: PipeRench: A reconfigurable architecture and compiler. *IEEE Computer* **33** (2000) 70–77
2. Trimberger, S., Carberry, D., Johnson, A., Wong, J.: A time-multiplexed FPGA. In: Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM). (1997) 22–28
3. Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J., DeHon, A.: Stream computations organized for reconfigurable execution (SCORE). In: Field-Programmable Logic and Applications (Proc. FPL), LNCS 1896, Springer-Verlag (2000) 605–614
4. Schmit, H., Whelihan, D., Moe, M., Levine, B., Taylor, R.R.: PipeRench: A virtualized programmable datapath in 0.18 micron technology. In: Proc. 24th IEEE Custom Integrated Circuits Conf. (CICC). (2002) 63–66
5. DeHon, A.: DPGA utilization and application. In: Proc. 4th ACM Int. Symp. on Field-Programmable Gate Arrays (FPGA). (1996) 115–121
6. Baumgartne, V., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP – a self-reconfigurable data processing architecture. In: Proc. 1st Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA). (2001) 64–70
7. Singh, H., Lee, M.H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Chaves Filho, E.M.: MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers* **49** (2000) 465–481
8. Enzler, R., Plessl, C., Platzner, M.: Co-simulation of a hybrid multi-context architecture. In: Proc. 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA). (2003)
9. Hauser, J.R., Wawrzynek, J.: Garp: A MIPS processor with a reconfigurable co-processor. In: Proc. 5th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM). (1997) 12–21
10. Austin, T., Larson, E., Ernst, D.: SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer* **35** (2002) 59–67

A Dynamically Adaptive Switching Fabric on a Multicontext Reconfigurable Device

Hideharu Amano¹, Akiya Jouraku¹, and Kenichiro Anjo²

¹ Dept. of ICS, Keio University,

² NEC Electronics,

drp@am.ics.keio.ac.jp

Abstract. A framework of dynamically adaptive hardware mechanism on multicontext reconfigurable devices is proposed, and as an example, an adaptive switching fabric is implemented on NEC's novel reconfigurable device DRP(Dynamically Reconfigurable Processor).

In this switch, contexts for the full crossbar and alternative hardware modules, which provide larger bandwidth but can treat only a limited pattern of packet inputs, are prepared. Using the quick context switching functionality, a context for the full crossbar is switched by alternative contexts according to the packet inputs pattern. Furthermore, if the traffic includes a lot of packets for specific destinations, a set of contexts frequently used in the traffic is gathered inside the chip like a working set stored in a cache.

4×4 mesh network connected with the proposed adaptive switches is simulated, and it appears that the latency between nodes is improved three times when the traffic between neighboring four nodes is dominant.

1 Introduction

Techniques on dynamically reconfigurable systems have been widely researched especially for mobile terminals whose hardware resources are strictly limited. Advanced dynamically reconfigurable systems or devices whose functions can be changed so as to adapt for surrounding conditions have been reported[3][4]. Such systems are also useful for quick delivery of an improved version of hardware which supports extended services. By further extension of such technologies, there becomes a possibility of a dynamically adaptive hardware, which modifies its structure automatically to fit the surrounding conditions.

Unfortunately, the configuration time and consuming power required for changing hardware structure of commodity reconfigurable devices are so large that the benefits of dynamically reconfiguration are lost in most cases. Thus, target applications of the dynamically adaptive hardware are mainly limited to specific functions of mobile devices.

However, recent advanced reconfigurable devices[1][2] which support quick and run-time reconfiguration can extend the target field of dynamically adaptive hardware drastically. For example, DRP(Dynamically Reconfigurable Processor)[1] developed by NEC in 2002 provides 16 hardware contexts inside the chip, and

can switch them with a clock cycle. The chip is partitioned into several regions where context switchings can be controlled independently. That is, a partial context switching is possible. Run-time reconfiguration from outside the chip to the context which is not currently used is also supported.

Here, by making the best use of such facilities, a novel framework of dynamically adaptive hardware is proposed. As a design example, a dynamically adaptive switching fabric is implemented on DRP, and the performance is evaluated.

2 Dynamically Adaptive Hardware

2.1 Concept of Dynamically Adaptive Hardware

Fig. 1A shows an execution unit of dynamically adaptive hardware proposed here. It consists of a fixed region and a multicontext reconfigurable region[7] which switches the context with a clock. The target circuit for adaptation is implemented on the multicontext reconfigurable region, while the context switching is managed with the scheduler on the fixed region. Here, a multicontext device which provides tens of contexts inside the chip is assumed. As described later, NEC's DRP provides sixteen contexts. The configuration data outside the chip can be loaded to currently unused context without disturbing the current available context like a virtual hardware mechanism[5].

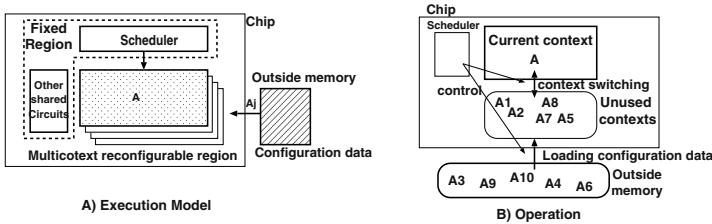


Fig. 1. Concept of dynamically adaptive hardware

Assuming that A is fully equipped hardware module which can execute every required function. A_1, A_2, \dots, A_n are alternative hardware modules which can execute only a part of A 's functions with higher speed or lower power consumption. Generally, if required functions are limited, it is not difficult to design a circuit whose performance or power consumption is better than A .

When the system starts, A is executed on the multicontext region, and other specialized hardware modules A_j are held in unused contexts. Some of A_j which cannot be stored inside the chip are located in the memory outside the chip as shown in Fig. 1B. The scheduler in the fixed region inspects inputs and states of the executing logic. If the scheduler judges that another hardware module A_j

with better performance/power consumption can be used in the current situation, the context is replaced to that for A_j . If the situation changes and functions which A_j cannot treat are required, another context A_k which can treat the situation comes up. If there is no other candidates, fully equipped A is used again.

If the context corresponding to A_j is not inside the chip, A is used until the configuration data for A_j is loaded from outside the chip, and then the context is switched. The over-written context number by the configuration data from outside the chip is selected with the LRU (Least Recently Used) algorithm. The fully equipped A is assigned into the context 0, and never be over-written. If there are a lot of alternative hardware modules, only frequently used candidates are loaded inside the chip like a cache mechanism of common computers. That is, the system is adopted for the frequently occurring situations. Note that the system does not stop during loading configuration data for A_j , since fully equipped A is continuously working on the current context.

There are some open problems in this framework. First is the algorithm for selecting alternative hardware modules with limited functions A_1, \dots, A_n . If circuits for evaluating the execution performance or power consumption can be provided in the scheduler, learning algorithms developed in the artificial intelligence can be applied. However, such a complicated scheduler will increase the chip area and power consumption. Thus, for most applications, a simple learning mechanism is useful.

The next problem is the timing of the context switching and data communication between them. This is a common problem of multicontext reconfigurable devices, and a design methodology as a solution is proposed[7]. As described later, DRP provides distributed memory modules which are shared with all contexts. Communication between contexts can be done using such shared memory modules.

If the target system consists of several components, and the target device enables partial context switching, this framework can be applied to each component independently. In general, components are often designed by using IPs (Intellectual Properties). For such adaptive systems, an IP which includes a set of configurations consisting of the fully equipped A , other candidates A_1, \dots, A_n , and corresponding conditions for application is desirable.

2.2 A Dynamically Adaptive Switch

As an example, a simple packet switching fabric with four inputs/outputs for system area network or parallel machines is designed. A 64bit width packet buffer which can store two full-size packets at maximum is provided to each input as shown in Fig. 2. Each link is 32bit width, and the transmission bit-late on the link is assumed to be twice as that of the clock frequency inside the chip. A flexible size packet with 256 flits at maximum is transferred in the asynchronous wormhole manner¹. A header flit includes the destination and size of the packet.

¹ In a real switch, the link transfer rate becomes often eight times that of inside the switch[8]. Here, from the restriction of the DRP chip, the rate is set to be double.

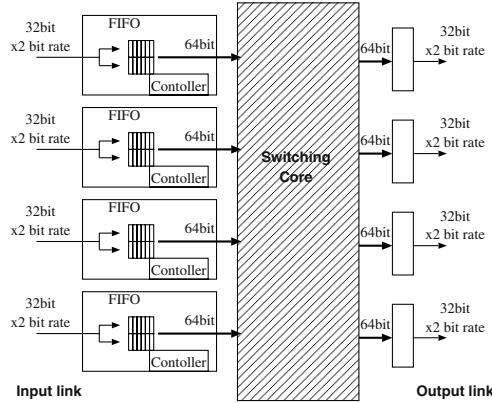


Fig. 2. Diagram of the switching fabric

The input port controller including packet buffer is assigned into the fixed region, while the multicontext region is used for the switching core. Here, fully equipped A for the switching core is full size (4×4) crossbar with 32bit bandwidth shown in Fig. 3A. However, if packets arrive at only an input port, the crossbar can be replaced by a configuration with only wires as shown in Fig. 3B. We call this “0-to-1 wire” configuration, and in general, “ n -to- m wire” configuration is possible for ($n < 4, m < 4$). Compared with the full size crossbar, “0-to-1 wire” configuration requires only area for wires. That is, 64bit width wires, the double width of 4×4 crossbar can be used. That is, this configuration is possibly advantageous both in the performance and power consumption.

Similarly, a combination of wires (Fig. 3C) and a small size 2×2 crossbar shown in Fig. 3D can be used when packets are arrived at two input ports. Since the area required by these configurations is small compared with the full size crossbar, double bandwidth wires (64bits) also can be used. Thus, the design shown in Fig. 3B-D can be treated as alternative hardware modules with limited functions A_j .

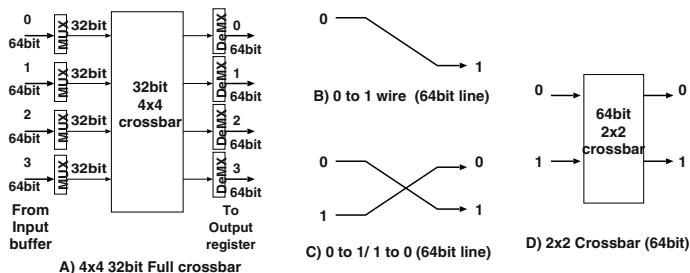


Fig. 3. Various Configurations for switching core

Since such alternative hardware modules can be formed for every combination of input/output, a lot of hardware modules A_j can be prepared for the switching core. If the context corresponding to requested alternative hardware module is not inside the chip, it is loaded from outside chip to currently unused context memory, then replaced with the full size crossbar. If the traffic includes a lot of packets for specific destinations, a set of contexts frequently used in the traffic is gathered inside the chip like a working set stored in a cache. That is, the switch can adapt to the traffic.

3 Implementation on DRP-1

3.1 DRP Overview

DRP is a coarse-grain reconfigurable processor core, which can be integrated into ASICs and SOCs. The primitive unit of DRP Core is called 'Tile', and DRP Core consists of arbitrary number of Tiles. The number of Tiles can be expandable, horizontally and vertically.

The primitive modules of Tile are processing elements(PEs), State Transition Controller(STC), 2-ported memories (VMEMs: Vertical MEMories), VMEM Controller(VMCtrl) and 1-ported memories (HMEMs: Horizontal MEMories). The structure of Tile is shown in Fig. 4.

There are 8x8 PEs located in one Tile. The architecture of PE is shown in Fig. 5. It has an 8-bit ALU, an 8-bit DMU, an 8-bit \times 16-word register file, and an 8-bit flip-flop. Those units are connected by programmable wires specified by instruction data. PE has 16-depth instruction memories and supports multiple context operation. Its instruction pointer is delivered from STC.

STC is a programmable sequencer in which certain FSM (Finite State Machine) can be stored. STC has 64 states, and each state is associated with the

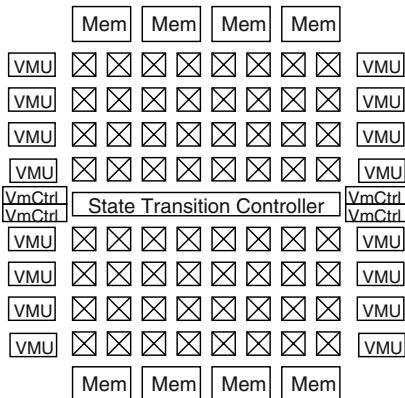


Fig. 4. Structure of a Tile

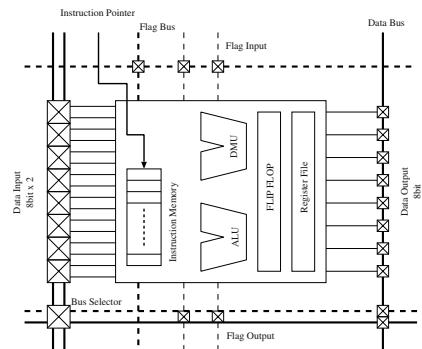


Fig. 5. Structure of a PE

instruction pointer. FSM of STC operates synchronized with the internal clock, and generates the instruction pointer for each clock cycle according to the state. Also, STC can receive event signals from PEs to branch conditionally. The maximum number of branch is four.

As for the memory units, Tile has eight 2-ported VMEMs on right and left sides, and four 1-ported HMEMs on upper and lower boundary. The capacity of a VMEM is 8-bit \times 256-word, and four VMEMs can be handled as a FIFO, using VMCtrl. HMEM is single-ported, thus has large capacity compared with VMEM. It has 8-bit \times 8K-word entries. Contents of these memories, flip-flops, register files of PE are shared with the datapath of all the contexts.

DRP Core, consisting of several Tiles, can change its contexts every cycle by instruction pointer distribution from STCs. Also, each STC can run independently, by programming different FSMs.

DRP-1 is the prototype chip, using DRP Core with 4 \times 2 Tiles. It is fabricated with 0.15- μ m 8-metal layer CMOS processes. It consists of 8-Tile DRP Core, eight 32-bit multipliers, an external SRAM controller, a PCI interface, and 256-bit I/Os. The maximum operation frequency is 100-MHz.

3.2 Dynamically Adaptive Switch on DRP-1

A simple dynamically adaptive switching fabrics with 4-input/output shown in Fig. 3A-D is implemented on DRP-1.

In this implementation, a left four Tiles are assigned into the fixed region including input packet buffers and controllers as shown in Fig. 6a). Remaining four Tiles are used for the multicontext reconfigurable region where the various switching cores are placed. In this implementation, the logic overhead for controlling context switch is almost included in the arbiter, and the switching core is a combinatorial circuit except for the final buffer. Thus, there is almost no logic overhead for management of context switching.

ALUs and DMUs in the fixed region are almost fully used, and about a half of them are used for wiring. On the contrary, a lot of empty PEs are remained

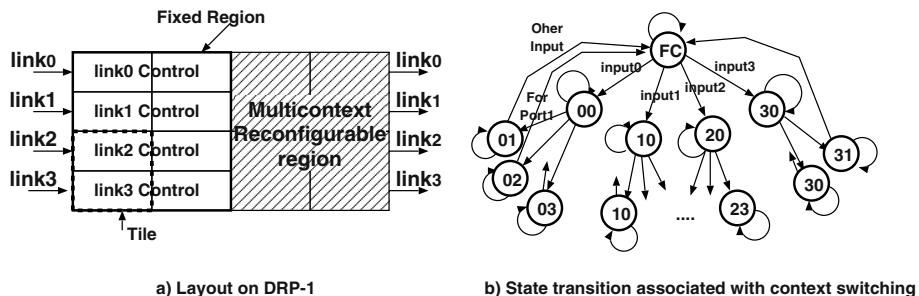


Fig. 6. Layout and State transition on DRP-1

in dynamically reconfigurable part especially for 1 to 1 wiring (Fig. 3B and 2×2 crossbars (Fig. 3D).

The evaluated maximum frequency is about 32MHz independent from the configuration in the switching core, since the packet controllers in the fixed region make critical paths in the total design. However, since the transfer bandwidth of Fig. 3B, Fig. 3C and Fig. 3D is enhanced compared with the fully equipped hardware module shown in Fig. 3A), the performance can be improved by replacing the configuration.

In DRP, STC controls the context switching by a state transition in which each state is associated into a context. Unfortunately, with the restriction in DRP-1, only four states can be designated as destinations from a state.

In this implementation, sixteen states each of which is corresponding to a context are used as shown in Fig. 6b). When the switch is initialized, the state “FC” corresponding to the full crossbar is used.

The control logic in the fixed region generates a signal (input0) when a packet arrives at input port 0. Detecting the signal as an event, STC moves its state into “00” corresponding to “0-to-0-wire” configuration. During a clock for the context switching, the arbiter in the fixed region checks the packet header, and decides the destination output port. If the destination output port of the packet is 1, 2 or 3, the state moves to “01” for “0-to-1-wire”, “02” for “0-to-2-wire”, or “03” for “0-to-3-wire”, respectively. Otherwise (that is output port 0 is selected), the state “00” is used without context switching. That is, the state transition for context switching is overlapped with the arbitration sequence, and no extra cycles are required.

In this implementation, sixteen contexts are assigned into full crossbar and “n-to-m-wire” configurations, where n and m is 0,1,2 or 3 except “3-to-3-wire” configuration which cannot be included by the limitation of the contexts. That is, if the arbiter detects the destination of the packet is output port 3 in “30” state, the state moves back to “FC” where the full crossbar is used.

A virtual hardware mechanism, which enables to use a full set of configurations including “ 2×2 crossbars”, is under designing now, since it requires supporting hardware outside the chip. In this mechanism, state number is extended to hold contexts which cannot be held inside the chip. The fixed part checks whether the context is inside of the chip or not by referring the mapping table. If the context corresponding to the destination state is not inside the chip, the configuration data is loaded to unused context selected with the LRU algorithm.

4 Performance Evaluation

4.1 Evaluation Conditions

We evaluated the performance of the proposed dynamically adaptive switch with a flit level network simulator. The structure of the switching fabrics is almost the same as the implementation on the DRP shown in the previous section except with the size of switch. That is, the number of ports is extended to 5×5 for

forming a two dimensional mesh structure in the simulation. A port is used for connecting with a host processor and the rest ports are connected to four neighboring switches.

A virtual hardware mechanism now under development is assumed to be available in this simulation, that is, runtime configuration is triggered when the required configuration is not inside the chip. Although several ways of configurations are supported in DRP-1, the most commonly used is configuration through the PCI interface. Configuration data corresponding to PEs in all Tiles, memory modules and other interfaces is mapped into a single logical address with 20bit address and 32bit data. It can be accessed as a simple byte-addressed 32bit memory from the host, and the configuration data is transferred through PCI bus usually in the burst mode. In the current implementation, the context switching can be done at any time, but requires a clock delay. After loading the configuration, an extra clock delay is assumed for re-writing the state transition table.

The size of configuration data for dynamically reconfigurable parts (4 Tiles) is dependent on the design. Since both designs of “n-to-m wire” and “2×2 crossbar” are simple, the runtime configuration requires about 200 clock cycles in total. Other simulation parameters are shown in Table 1.

Table 1. Simulation Parameters

Simulation time	3,000,000 clocks (ignore the first 50,000 clocks)
Topology	4 × 4 two dimensional mesh
Packet length	256 flits
Flow control	virtual cut-through
Routing	e-cube routing with a virtual channel
Traffic pattern	uniform/localized

4.2 Evaluation Results

Fig. 7 shows the throughput versus latency under the uniform traffic, that is, the destination of packets are randomly distributed. The line marked “full crossbar” is corresponding to the case that only fully equipped configuration (5×5 crossbar in this case) is used.

“full crossbar + n-to-m wire” uses “n-to-m wire” configurations shown in Fig. 3B. The latency is slightly improved when the traffic is not severe, but becomes worse when the traffic becomes severe. This comes from the overhead for context switching and an extra overhead after the loading configuration.

“full crossbar + 2×2 crossbar” uses 2×2 crossbar configuration shown in Fig. 3D when arriving packets are stored in two input ports. In this case, the latency is reduced independent on the traffic load. Although the possible configurations of “2×2 crossbar” increases to 100, the number of traffic patterns

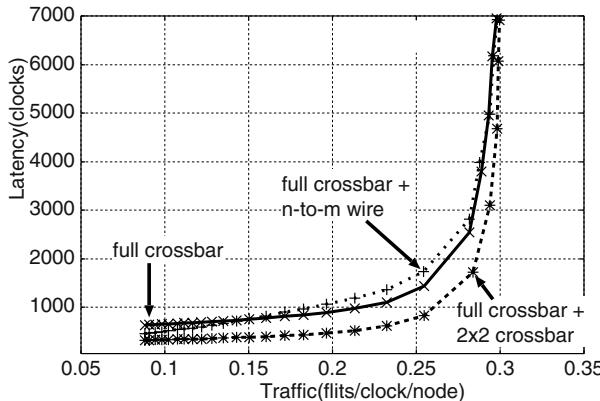


Fig. 7. Bandwidth versus Latency (Uniform traffic)

which a “ 2×2 crossbar” can treat is four times that of “n-to-m wire”, and the performance is improved.

When a lot of scientific applications including Partial Differential Equations are executed on mesh-connected parallel machines, a large part of communication is done between neighboring four nodes. For such a condition, we used the localized traffic in which 90% of packets are to the neighboring nodes and the rest is randomly distributed. Fig. 8 shows the evaluation results under this localized traffic. In this case, both “n-to-m wire” and “ 2×2 crossbar” improves bandwidth as well as latency, since it does not require run-time loading of configuration. Especially, “ 2×2 crossbar” improves three times in latency and 9% in bandwidth.

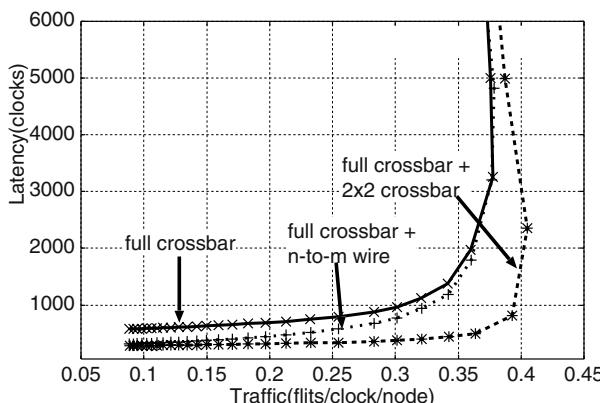


Fig. 8. Bandwidth versus Latency (Localized Traffic)

5 Related Work and Conclusion

A framework of dynamically adaptive hardware is proposed, and as an example, a dynamically adaptive switching fabric is implemented on NEC's multicontext device DRP.

The configuration management in the proposed mechanism is similar to virtual hardware mechanisms[5][9] with multicontext reconfigurable devices including our previous work. The cache effect of configuration data in reconfigurable devices is also discussed in [10]. However, the system never stop during loading of configuration data for A_j , since fully equipped A is continuously working on the current context. Although the approach in dynamic hardware plugins[4] is similar to our approach in some points, their approach focuses on the functional adaptation. Researches called “evolvable hardware”[11], which combines reconfigurable logics and hardware genetic algorithm engines have been exerted. Although genetic algorithm may be useful for selection of alternative logics, the functions supported in our mechanism do not change beyond the design.

References

1. M.Motomura:”A Dynamically Reconfigurable Processor Architecture,” Microprocessor Forum, Oct. 2002.
2. P.Master: ”The Age of Adaptive Computing Is Here,” Proc. of FCCM, pp.1-3 (2002).
3. G.J.M.Smit, P.J.M.Havinga, L.T.Smit, P.M.Heysters: “Dynamic Reconfiguration in Mobile Systems,” Proc. of FPL2002, pp.162-170, (2002).
4. E.L.Horta, J.W.Lockwood, D.Partour, “Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration.” Proc. of DAC2002, June (2002).
5. X.-P. Ling, H. Amano, “WASMII: A Data Driven Computer on a Virtual Hardware” Proc. FCCM, pp. 33–42, (1993).
6. S. Trimberger, D. Carberry, A. Johnson and J. Wong “A Time-Multiplexed FPGA” Proc. FCCM pp.22-28, (1997).
7. N.Kaneko, H.Amano, “A General Hardware Design Model for Multicontext FPGAs,” Proc. of FPL, pp.1037–1047, (2002).
8. S.Nishimura, et al., “High-speed network switch RHiNET-2/SW and its implementation with optical interconnections”, Hot Interconnects 8, pp.31–38, Aug. (2000).
9. G.Brebner, “The Swappable Logic Unit: a Paradigm for Virtual Hardware,” Proc. of FCCM, pp.82–91, (1997).
10. Z.Li, K.Compton, S.Hauck, “Configuration Caching Management Techniques for Reconfigurable Computing,” Proc. of FCCM, pp.22–36, (2000).
11. T.Higuchi and N.Kajihara, “Evolvable hardware chips for industrial applications,” Commun. ACM, Vol.42, No.3, pp.60–66, (1999).

Reducing the Configuration Loading Time of a Coarse Grain Multicontext Reconfigurable Device

Toshiro Kitaoka¹, Hideharu Amano¹, and Kenichiro Anjo²

¹ Dept. of ICS, Keio University,

² NEC Electronics,

`drp@am.ics.keio.ac.jp`

Abstract. High speed and low cost configuration loading methods for a coarse grain multicontext reconfigurable device DRP(Dynamically Reconfigurable Processor) are proposed and implemented. In these methods, the configuration data is compressed on the host computer before loading, and decoded at the time of loading by circuits implemented on a part of logics. Unlike conventional reconfigurable device, the logic for decoder circuits is switched with application circuits immediately after loading in multicontext reconfigurable devices. Thus, the circuit does not use a real estate of the chip during the execution. Two compression methods LZSS-ARC and Selective coding are implemented and evaluated. LZSS-ARC achieves better compression ratio, while Selective coding can work at the same frequency of the data loading.

1 Introduction

A run-time reconfigurable system, which changes its structure dynamically, has been widely researched especially for mobile terminals whose hardware resources are strictly limited. Recent advanced reconfigurable devices[2][3] which support quick and run-time reconfiguration can extend target fields of dynamic reconfigurable systems drastically. A coarse grain cell is adopted as an element in such devices for efficient use of the streaming applications. Some of these chips include multi-context functionality which provides several configuration data sets inside the chip, and changes them quickly.

However, even using such advanced devices, the time for loading of the configuration data from outside the chip often bottlenecks the system performance for some dynamically reconfigurable applications. Reducing the amount of configuration data with compression techniques is one of hopeful approaches to improve the configuration speed. Efficient runtime decoding techniques have been researched for traditional FPGAs[9][10].

Unfortunately, such compression methods are not efficient for recent coarse grain reconfigurable devices because of the complicated cell architecture. In this paper, we propose and evaluate runtime decoding methods of the beforehand compressed configuration data for a recent multi-context reconfigurable device,

NEC's DRP(Dynamically Reconfigurable Processor)[2]. This method enables high-speed loading of configuration data with a small overhead of the decoder hardware by making the best use of the multi-context facility.

2 Configuration Code Compression and Runtime Decoding

2.1 Code Compression Techniques for Conventional FPGAs

For the SRAM type FPGAs/CPLDs, which are now popularly in use, the size of configuration data is sometimes more than 5 Mbit and they need an msec-order time for loading. From technical point of view, it is possible to make the configuration time almost equal to that of the access time of common static RAMs. Even using such a high speed configuration circuit which can load 64 bit data at every 50 MHz clock, it takes 1.6 msec to load 5 Mbit configuration data.

Since the configuration of common FPGAs/CPLDs includes a large amount of data corresponding to unused LUT(Look Up Table)s, wires and switches, common compression techniques are efficiently applied. Several techniques are proposed for compression and decoding configuration data of common FPGAs/CPLDs[9][10]. In these methods, configuration data is prepared in a beforehand generated by certain software after place and routing. In such researches, an FPGA/CPLD assumed to provide hardware which decodes compressed configuration data and writes them into the configuration memory directly. This approach requires dedicated decoding hardware inside the FPGA chip as shown in Fig. 1, and requires a certain chip area. Thus, researches focus on the decoding circuits which enables runtime decoding with small hardware cost as possible.

Above compression algorithms for conventional FPGA with LUTs are not suitable for recent coarse grain reconfigurable devices. Table 1 shows the ratio of compressed data size to the original one when LZ77 and Huffman compression used in traditional compression methods are applied to the configuration data of two different reconfigurable devices. One is NEC's coarse grain reconfigurable device DRP, and another is its previous version called DRL[1] which uses LUTs. It appears that they can only reduce about 30% data size, while the configuration data of LUT based device can be compressed to less than a half size of original.

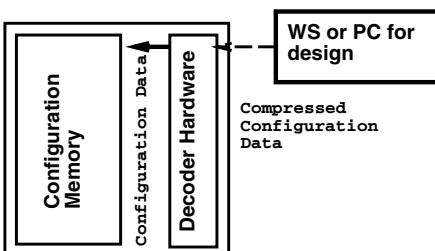


Fig. 1. FPGA with a decoder

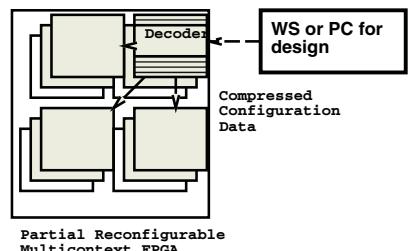


Fig. 2. partially multicontext reconfigurable device

Table 1. Ratio of the compressed data size (Neural Net)

	Coarse grain (DRP) (%)	LUT based device (DRL) (%)
LZ	72	27
Huffman	73	28

Although a coarse grain logic element requires less configuration data size than common FPGAs with LUTs, it does not include data with a long sequence of '0's or '1's. This is the main reason why the traditional LZ or Huffman techniques are not so efficient.

2.2 Combination with a Multicontext Reconfigurable Device

On the other hand, some of recent reconfigurable devices are equipped with a multicontext facility which can be efficiently used for decoder of the compressed configuration.

Multicontext reconfigurable devices[5][11][1] can store multiple sets of configuration data. By switching the output of each configuration memory by the multiplexor, the configuration can be immediately switched. Configuration data on each configuration memory is called a context and switching them for reconfiguration by the multiplexer is called context switching. In many multicontext devices, context switching requires only a clock, the hardware structure can be changed quickly. Now, DRP, a multicontext device with sixteen contexts developed by NEC[2] is available, and Quicksilver's ACM[3] and IPFlex's DNA chip[4] also provide the similar facility.

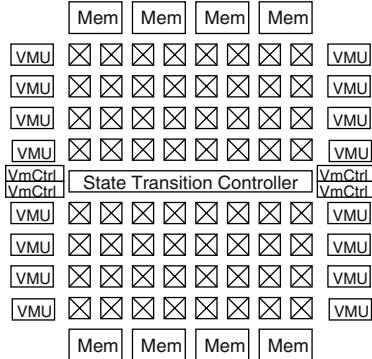
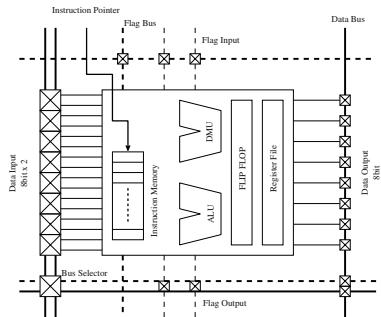
For high-speed but cost effective configuration, we propose to implement decoding hardware for compressed configuration data in one or small number of the contexts. After configuration is done with the decoding hardware, the context is switched and the loaded circuit starts quickly.

Furthermore, in DRP, a partial context switching mechanism is available. In such a device, as shown in Fig. 2, it is possible to implement the decoding hardware in a partition and to rewrite the configuration data of unused contexts in other partitions. By scheduling context-rewriting in each partition, it is expected to cover up the time for configuration from outside the chip. This mechanism enables high speed run-time configuration for dynamically reconfigurable applications[6].

3 DRP and Its Configuration

3.1 DRP Overview

DRP is a coarse-grain reconfigurable processor core, which can be integrated into ASICs and SOCs. The primitive unit of DRP Core is called 'Tile', and DRP Core consists of arbitrary number of Tiles. The number of Tiles can be expandable, horizontally and vertically.

**Fig. 3.** Structure of a Tile**Fig. 4.** Structure of a PE

The primitive modules of Tile are processing elements(PEs), State Transition Controller(STC), 2-ported memories (VMMEMs: Vertical MEMories), VMEM Controller(VMCtrl) and 1-ported memories (HMEMs: Horizontal MEMories). The structure of Tile is shown in Fig. 3.

There are 8x8 PEs located in one Tile. The architecture of PE is shown in Fig. 4. It has an 8-bit ALU, an 8-bit DMU, an 8-bit×16-word register file, and an 8-bit flip-flop. Those units are connected by programmable wires specified by instruction data. These bitwidths are ranging from 8B to 18B according to the location. PE has 16-depth instruction memories and supports multiple context operation. Its instruction pointer is delivered from STC.

STC is a programmable sequencer in which certain FSM (Finite State Machine) can be stored. STC has 64 states, and each state is associated with the instruction pointer. FSM of STC operates synchronized with the internal clock, and generates the instruction pointer for each clock cycle according to the state. Also, STC can receive event signals from PEs to branch conditionally. The maximum number of branch is four.

As for the memory units, Tile has eight 2-ported VMMEMs on right and left sides, and four 1-ported HMEMs on upper and lower boundary. The capacity of a VMEM is 8-bit×256-word, and four VMEMs can be handled as a FIFO, using VMCtrl. HMEM is single-ported, thus has large capacity compared with VMEM. It has 8-bit×8K-word entries. Contents of these memories, flip-flops, register files of PE are shared with the datapath of all the contexts.

DRP Core, consisting of several Tiles, can change its contexts every cycle by instruction pointer distribution from STCs. Also, each STC can run independently, by programming different FSMs.

DRP-1 is the prototype chip, using DRP Core with 4×2 Tiles. It is fabricated with 0.15-um 8-metal layer CMOS processes. It consists of 8-Tile DRP Core, eight 32-bit multipliers, an external SRAM controller, a PCI interface, and 256-bit I/Os. The maximum operation frequency is 100-MHz.

3.2 DRP Configuration

Although several ways of configurations are supported in DRP-1, the most commonly used is configuration through the PCI interface. Configuration data corresponding to PEs in all Tiles, switches, memory modules and other interfaces is mapped into a single logical address with 20bit address and 32bit data. It can be accessed as a simple byte-addressed 32bit memory from the host, and the configuration data is transferred through PCI bus usually in the burst mode. Although the address is not continuously used in all the 20-bit address space, uppermost 5bit of address is the same for a long fixed sequence. That is, address can be easily compressed into 15bit.

In the implementation shown here, a Tile is assigned for decoder circuits of compressed configuration data transferred from the PCI interface. Unfortunately, the configuration path and data path are completely separated in DRP-1, and the compressed data cannot transferred to the configuration memory. A simple bus from the output data of the Tile to the configuration path is assumed here.

4 Coding Methods and Decoder Implementation

Following properties are required for coding methods for compression of configuration data. (1) The decoding must be done quickly at the time of loading the configuration, while a complex encoding by the software is allowed. (2) The decoding must be done with small amount of hardware which can be implemented on a small part of a multicontext reconfigurable device. Methods which require large amount of buffers or logics cannot be adopted.

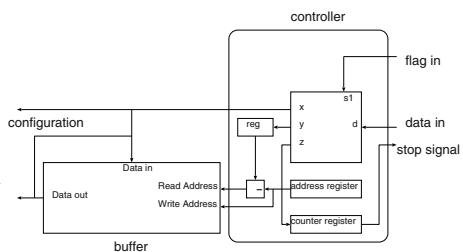
To satisfy the above conditions, we adopted LZSS[8] and modified it so as to fit the configuration data of DRP. This method is called LZSS with address run-length compression or LZSS-ARC. We also proposed another method which changes coding methods by switching decoding hardware based on the properties of the target context. This method is called Selective coding.

4.1 LZSS with Address Run-Length Compression

Coding Method. The LZ77 coding is a data-coding method which was proposed by J.Ziv and A.Lempel in 1977[7]. In this method, the dictionary for coding is not prepared beforehand but built dynamically while reading symbols for coding.

The sequence of symbols which is to be coded is replaced with special symbols such as (position, length), if the sequence appeared before. For example, in the sequence “ABCPQABCRS”, ABC which appeared for the second time agrees with the 3-letter symbol, five symbols preceding, and this sequence is coded as ABCPQ(5,3)RS. LZSS is modified LZ method proposed by Storer and Szyman-ski in 1982[8]. In LZ77, an uncoded sequence and the tag (position, length) is alternatively appears, while LZSS uses flag bits to identify the tag or uncoded sequence. That is, the above example is coded to a sequence ABCPQ53RS and flag bits 000001100, whose bits corresponding to ’53’ are set.

Address Data	base address
5c0c0 a0	5c0c0
5c0c1 30	
5c0c2 73	+
5c0c3 30	→ runlength
5c0c4 30	8
5c0c5 d0	+
5c0c6 30	data sequence
5c0c7 c1	a0,30,73,30,30,d0,30,c1
5c0d4 26	
...	...

Fig. 5. Address run-length compression**Fig. 6.** LZSS-ARC decoder circuit

However, direct use of LZSS is not efficient for the configuration data of DRP, since it consists of pairs of 20bit address and 32bit data. So, we propose to apply a pre-processing before LZSS compression, and call it LZSS with address run-length compression or LZSS-ARC. In this method, byte addressing data sequence is represented with a base address, run-length and a sequence of byte data as shown in Fig. 5. In DRP-1, a byte configuration data is used for each element (ALU, DMU, flip-flop and register files) of PE, and the same configuration is used for the same functions. Thus, the ratio of coded part in the data sequence is increased as well as the data size itself is reduced with this method.

Design of the LZSS-ARC Decoder. At the input stage of decoder, the address run-length compression is decoded, and the sequence of data and flag are transferred to the LZSS-ARC decoder circuit.

In the process of LZSS-ARC decoding, the sequence of symbols which is already decoded is stored in the buffer, and if a special symbol appears, the symbol-sequence of its length is output from the position in the buffer. As shown in Fig. 6, the LZSS-ARC decoder consists of a controller and a symbol buffer. In DRP, Vmem modules, which is 8-bit × 256 dual-port memory, is used as a symbol buffer. The controller provides a counter for counting output symbols and a register used as a pointer of the window.

1. The flag bit indicating whether the data is encoded or not is checked.
2. If input symbol is not encoded, it is directly sent to the output.
3. If a special symbol (position, length) comes, the decoder starts operation and input is suspended. First, the length is set into the counter. Symbols in the buffer are sent to the output and the counter is decremented. At the same time, the encoded symbol itself is also added to the buffer. When the counter becomes zero, the decoding is finished, and the next symbol starts being inputted.

4. The value of the address register increases every time when a new symbol is added to the buffer. When the buffer becomes full, the address register returns to the first position, and the content of the buffer is cleared.

The total circuits are divided into four blocks: Uncoded data input, Matching position input, Matching symbol length, and Output. Each block is enough small to be assigned into a Tile, and they are implemented in four contexts in the same Tile. Since input data must be suspended during decoding operation, an FIFO is required, and another Vmem is used. Thus, this implementation requires two Vmem modules ($8\text{bit} \times 512$ in total) and four contexts assigned into the same Tile. DRP-1 provides 16 contexts with 4×2 Tiles and 80 Vmem modules, that is, the decode only requires 3% of PEs and 2.5% of Vmem modules. From the critical path analysis, the maximum frequency of the decoder is 200MHz. That is, configuration data loading with 33MHz clock can be done almost without suspending.

4.2 Selective Coding

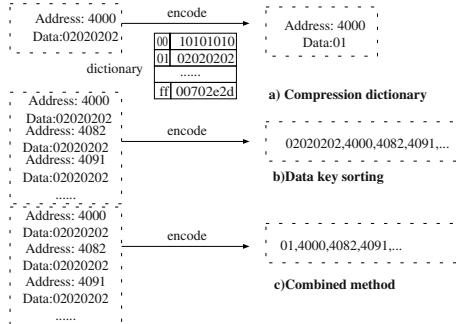
Coding Method. The loading method with LZSS-ARC decoder requires an internal clock whose frequency which is a multiple of loading data rate. So, it can only accept slow loading clock frequency due to the maximum boundary of the internal clock frequency. We propose another method which can work with the same speed of the loading rate.

As mentioned before, configuration data for DRP includes a lot of similar data patterns. Thus, the following two compression techniques are useful. In both techniques, 20bit address is compressed into 15bit, for uppermost 5bit is the same for a long fixed sequence in DRP.

- Compression directory: A dictionary which enumerates frequent appearing 32bit data pattern referred by 8bit index is generated from statistics of several DRP design examples. As shown in Fig. 7a), 32bit data is compressed into 8bit index if the pattern is included in the directory.
- Data key sorting: When a single data appears many times in the target compress data, corresponding addresses are sorted as shown in Fig. 7b). This technique can be applied with the former technique as shown in Fig. 7c).

In this method, two compression methods are selectively used for the property of the compressed target. Data to which both techniques cannot be used is transferred as it is.

Design of the Decoder for Selective Coding. We designed three decoders for compressed data with Compression directory, Data key sorting, and the combined method. The decoder for Compression directory consists of a simple controller and directory implemented with five Vmem modules. The decoder for Data key sorting is also simple, since configuration data can be written into configuration memory of DRP in any order. Nine contexts are used in total, and

**Fig. 7.** Selective coding

assigned into the same Tile. Appropriate context corresponding to each technique is selectively applied depending on the coding method, and we call this method Selective coding. This implementation requires 7% of PEs and 6.25% Vmem modules of DRP-1 in total. From the critical path analysis, the maximum frequency of the decoder is 156MHz.

5 Evaluation

5.1 Compression Ratio

The compression data size of several DRP applications with LZSS-ARC and Selective coding are shown in Table 2.

The compression ratio of LZSS-ARC is slightly better than that of Selective coding especially for encryption algorithms in which various kind of functions of DMU/ALU are used. Van del Pol uses a lot of adders, and Wavelet uses a lot of data selector. This is the reason why Selective coding is effective for such designs.

Table 2. Compressed data size

Application	Config. data size	size with LZSS-ARC (ratio)	size with Selective coding (ratio)
Neural Net	78104 bit	36944 bit (47%)	41910 bit (53%)
Van del Pol	48080 bit	31888 bit (66%)	21544 bit (44%)
AES	69296 bit	39352 bit (56%)	48758 bit (70%)
DES	56920 bit	31752 bit (55%)	40629 bit (71%)
MD5	69296 bit	42776 bit (61%)	49663 bit (71%)
Wavelet	116720 bit	76440 bit (65%)	69349 bit (59%)

5.2 Loading Clock Cycles

Fig. 8 shows the required configuration time for loading configuration data, when the 33MHz PCI interface of DRP-1 is used. Note that, the loading of configuration for decoder hardware itself is not included in this figure, since it is only done at initializing the system. The required clock cycles are normalized to the case when the data is not compressed. LZSS-ARC is assumed to run at 198MHz (6 times of 33MHz), and in this case, the stall of the PCI rarely occurs. That is, LZSS-ARC achieves better performance. However, the Selective coding method allows to use the same clock frequency for both internal and loading, thus it can exploit the maximum throughput of the decoder inside. Thus, Selective coding becomes advantageous when loading frequency is increased in the future.

Here, the absolute clock cycles required for DRP configuration are compared with conventional FPGA with fine grain structure. The same circuits for DES encryption is also implemented on Xilinx's Virtex XC2V250[11], and its configuration data is compressed with LZSS. Ths size of data and clock cycles for loading are shown in Table 3.

Although the compression ratio of LZSS-ARC for DRP is worse, the absolute clock cycles are much less than that of LZSS for XC2V250. That is, the configuration speed is almost 80 times of the traditional FPGA with fine grain structure.

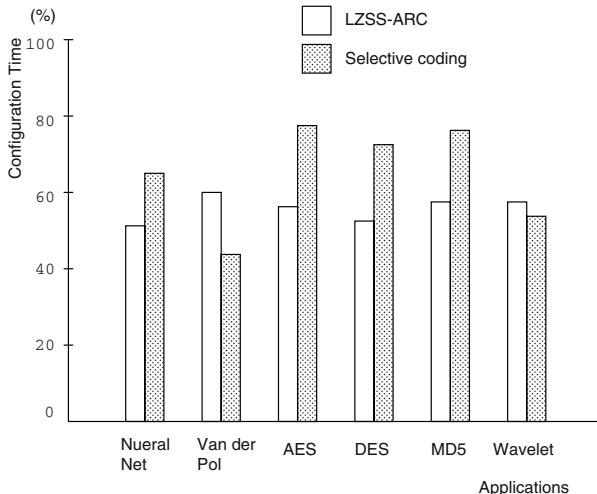


Fig. 8. Configuration time through PCI bus

Table 3. Compressed data size

Device	Config. data size	size with LZSS/LZSS-ARC (ratio)	Clock cycles for loading
DRP	56920 bit	31752 bit (55%)	992 cycles
XC2V250	1697736 bit	662112 bit (39%)	82746 cycles

In this evaluation, only decompression speed is considered, since the compression is done beforehand in the host PC. All compression methods described here require only a few seconds for compression on recent PCs with Pentium III, and it is negligible considering the time required for place-and-routing stage in the FPGA design.

6 Summary

Two compression methods LZSS-ARC and Selective coding suited for a coarse grain device DRP are proposed and implemented. Using these methods, about 80 times high speed configuration compared with conventional fine grain FPGA is achieved. Applications implemented on DRP sometimes use multiple contexts with similar structure. By using the similarity between contexts, another configuration compression method can be designed, and it is our future work.

References

1. T. Fujii et. al. "A Dynamically Reconfigurable Logic Engine with a Multi-Context/Multi-Mode Unified-Cell Architecture" Proc. International Solid State Circuit Conference 1999.
2. M.Motomura:"A Dynamically Reconfigurable Processor Architecture," Microprocessor Forum, Oct. 2002.
3. P.Master: "The Age of Adaptive Computing Is Here," Proc. of FCCM, pp.1-3 2002.
4. <http://www.ipflex.com/>.
5. X.-P. Ling, H. Amano, "WASMII: A Data Driven Computer on a Virtual Hardware" Proc. FCCM, pp. 33-42, 1993.
6. E.L.Horta, J.W.Lockwood, D.Partour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration." Proc. of DAC2002, June, 2002.
7. Jacob Ziv and Abraham Lempel. "A universal algorithm for sequential data compression" IEEE Transactions on Information Theory, IT-23(3):337-343, 1977.
8. J.A.Storer, T.G.Szymanski, "Data compression via textual substitution," J.of ACM, 29 (4), pp.928-951, 1982.
9. D.William, S.Wilson, S.Hauck, "Runlength Compression Techniques for FPGA Configurations," Proc. FCCM, pp.276-277, 1999.
10. Z.Li, S.Hauck, "Configuration Compression for Virtex FPGA," Proc. FCCM, pp.143-154, 2001.
11. <http://www.xilinx.com/>.

Design Strategies and Modified Descriptions to Optimize Cipher FPGA Implementations: Fast and Compact Results for DES and Triple-DES

Gaël Rovroy, François-Xavier Standaert,
Jean-Jacques Quisquater, and Jean-Didier Legat

UCL Crypto Group, Laboratoire de Microélectronique,
Université catholique de Louvain,
Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium,
{rouvroy, standaert, quisquater, legat}@dice.ucl.ac.be

Abstract. In this paper, we propose a new mathematical DES description that allows us to achieve optimized implementations in term of ratio *Throughput/Area*. First, we get an unrolled DES implementation that works at data rates of 21.3 Gbps (333 MHz), using Virtex-II technology. In this design, the plaintext, the key and the mode (encryption/decryption) can be changed on a cycle-by-cycle basis with no dead cycles. In addition, we also propose sequential DES and triple-DES designs that are currently the most efficient ones in term of resources used as well as in term of throughput. Based on our DES and triple-DES results, we also set up conclusions for optimized FPGA design choices and possible improvement of cipher implementations with a modified structure description.

Keywords: cryptography, DES, FPGA, efficient implementations, design methodology.

1 Introduction

The rapid growth of secure transmission is a critical point nowadays. We have to exchange data securely at very high data rates. Efficient solutions, with huge data rate constraints, have to be hardware implemented and flexible in order to evolve with the permanent changes in norms. FPGA (Field Programmable Gate Arrays) implementations of the triple-Data Encryption Standard (triple-DES) efficiently meet these constraints. Triple-DES is based on three consecutive DES¹. As detailed in [3, 6–8], DES is very well suited for FPGA solution. Accurately, we have first to focus our attention to achieve very fast DES designs with limited hardware resources.

Some high-speed DES hardware implementations have been published in the literature. These designs unroll the 16 DES rounds and pipeline them. Patterson

¹ Without intermediate *IP* and *IP-1* permutations.

[7] made a key-dependent data path for encryption in an FPGA which produces a bitstream of about 12 Gbps. Nevertheless the latency to change keys is tenth of milliseconds. A DES implementation is also downloadable from FreeIP [6] and encrypts at 3.05 Gbps. Last known implementations were announced by Xilinx company in [3, 8]. They were FPGA implementations of a complete unrolled and pipelined DES encryptor/decryptor. The 16-stage and 48-stage pipelined cores could achieve data rates of, respectively, 8.4 Gbps and 12 Gbps². The 48-stage pipelined version could also produce a throughput of 15.1 Gbps on Virtex-II. It also allowed us to change the plaintext, the key and the encryption/decryption mode on a cycle-by-cycle basis.

In this paper, based on our new mathematical DES description [17], we finally get also an optimized complete unrolled and pipelined DES design that encrypts with a data rate of 21.3 Gbps with 37 cycles of latency³. This design is the best currently known one in term of ratio *Throughput/Area*.

Concerning sequential designs, some DES and triple-DES FPGA implementations have been published in the literature. These designs encrypt or decrypt most of time every 16 cycles. In academic literature, we mainly found the paper of C. Paar [16] with a DES data rate of 402.7 Mbps. The most recent publication (October, 2001) comes from K. Gaj et al. [15]. They propose a triple-DES design with a bitstream of 91 Mbps with less resources, using Virtex1000 component. For commercial Xilinx IP cores, different DES and triple-DES implementations are also available from Helion Technology, CAST, and InSilicon [10–14]. The best triple-DES one in term of ratio *Throughput/Area* comes from CAST and gives a throughput of 668 Mbps on Virtex-II FPGA, using 790 slices (January, 2002).

Concerning our DES and triple-DES sequential implementations, we get designs that can encrypt or decrypt every 18 cycles. Our triple-DES design uses 604 slices and produces a throughput of 917 Mbps. This solution is the best sequential one known nowadays in term of resources used as well as in term of throughput.

Based on our DES and triple-DES results, we also set up conclusions for accurate design choices. We also mention the importance of cipher modified structure descriptions.

The paper is organized as follows: section 2 refers the Data Encryption Standard; section 3 explains our previous published new mathematical DES description; section 4 describes our unrolled and pipelined DES implementations and proposes a comparison with the previous Xilinx implementations; section 5 details our sequential DES and triple-DES implementations and also shows comparisons; our design strategies to optimize cipher FPGA implementations are explained in section 6; finally, section 7 concludes this paper.

² These results were obtained with Virtex-E technology.

³ Using Virtex-II technology.

2 The DES Algorithm

In 1977, the Data Encryption Standard (DES) algorithm was adopted as a Federal Information Processing Standard for unclassified government communication. It is still largely in use. DES encrypts 64-bit blocks with a 64-bit key, of which only 56 bits are used. The other 8 bits are parity bits for each byte. The algorithm counts 16 rounds. DES is very famous and well known. For detailed information about DES algorithm, refer to the standard [5] or our preceding work [17].

3 New Mathematical DES Description

The original DES description is not optimized for FPGA implementations regarding the speed performance and the number of LUTs used. It is not an efficient way to fit as well as possible into the Virtex CLBs. In order to achieve a powerful design, we explain a new mathematical description of DES previously detailed in our work [17]. The aim is to optimize its implementations on FPGA, reducing resources used as well as the critical path of one DES round.

An FPGA is based on slices composed by two 4-bit input LUTs and two 1-bit registers. Therefore, an optimal way to reduce the LUTs use is to regroup all the logical operations to obtain a minimum number of blocks that take 4-bit input and give 1-bit output. In addition, it is important to mention that all permutation and expansion operations (typically P , E , IP , $IP-1$, $PC-1$ and $PC-2$) do not require additional LUTs, but only wire crossings and fanouts (pure routing).

First, we transform the round function of the enciphering computation. This transformation has no impact on the computed result of the round. Figure 1 shows a modified round representation, where we move the E box and the XOR operation.

This involves the definition of a new function denoted R (like reduction):

$$\begin{aligned} R &= E^{-1}, \\ \forall x, R(E(x)) &= x, \\ \exists y \mid E(R(y)) &\neq y. \end{aligned} \tag{1}$$

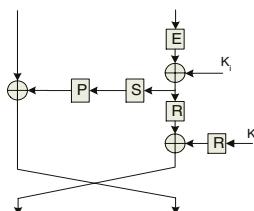


Fig. 1. Modified description of one DES-round.

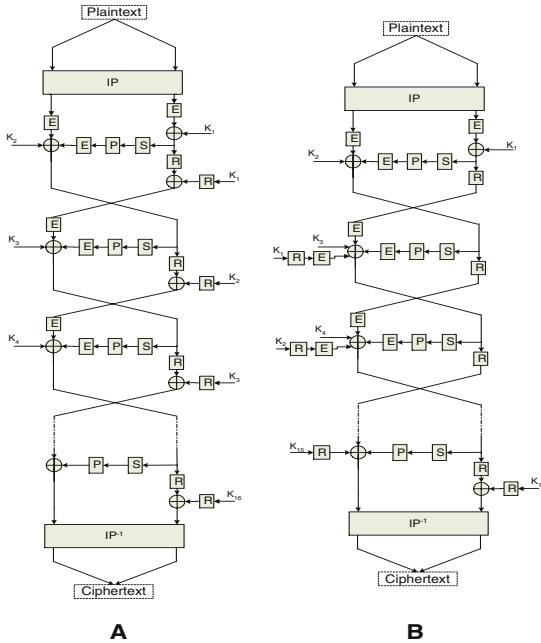


Fig. 2. Two modified descriptions of the DES algorithm.

Now, if we change all the enciphering parts of DES with this modified round function and if we combine the E and XOR block with the left XOR block of the previous round, we get the **A** architecture detailed in Figure 2. Another more efficient solution is to move the R and XOR of the right part of the round into the left XOR operator of the next round. As a result, we obtain the **B** architecture shown in Figure 2.

In the **B** arrangement, the first and last rounds are quite different from intermediate ones. Therefore, we obtain an irregular DES description. In addition, we increase the number of E and R blocks, which will not alter the number of LUTs consumed. We also keep exactly the same number of S -boxes that is the expensive part of the architecture. Concerning the modulo two sum operators, we really decrease their number. We spare 15×32 2-bit XOR s comparing with the **A** architecture⁴. We can directly conclude that this design will consume less logic than traditional implementations.

Therefore, modifying the original DES description according to the **B** architecture of Figure 2, we are able to significantly reduce the number of resources used. In addition, we also reduce the critical path of one round function. These transformations allow us to obtain very efficient DES FPGA implementations as detailed in the next sections.

⁴ The design exactly counts 14×48 4-bit XOR s, 1×48 3-bit XOR s, 1×48 2-bit XOR s, 1×32 3-bit XOR s and 1×32 2-bit XOR s.

4 Our Unrolled and Pipelined DES Implementations

To be speed efficient, we propose designs that unroll the 16 DES rounds and pipeline them, based on the B mathematical description of Figure 2. In addition, we implemented solutions that allow us to change the plaintext, the key and the encryption/decryption mode on a cycle-by-cycle basis, with no dead cycle. We can achieve very high data rates of encryption/decryption.

The left part of Figure 3 illustrates how the critical path, in our solution, is hugely decreased. We only keep one S-boxes operator and one XOR function (= two LUTs + one F_5 function + one F_6 function)⁵. With this solution we obtain a 1-stage pipeline per round. Due to the irregular structure of our design, we have to add an additional stage in the first round. To be speed efficient for implementation constraints, we also put a 2-stage pipeline respectively in the input and in the output. This approach allows an additional degree of freedom for the place and route tool. As mentioned in the figure, first and last registers are packed into IOBs. Therefore, we obtain a 21-stage pipeline.

In the right part of Figure 3, we put an extra pipelined stage in each round in order to limit the critical path to only one S-box. As a consequence, we finally get a 37-stage pipeline. Table 1 shows our 21-stage and 37-stage pipelined results.

Comparing to the preceding efficient Xilinx implementations [3, 8], our 21-stage gives better results in terms of speed and logical resources. Nevertheless, it is quite more registers consuming. For the 37-stage pipeline, we use less LUTs again. We also reduce the registers needed. This is due to the fact that we only have a 2-stage pipeline per round. In addition, this design uses shift registers for the key schedule calculation. In Virtex FPGAs, *SRL16* blocks can directly

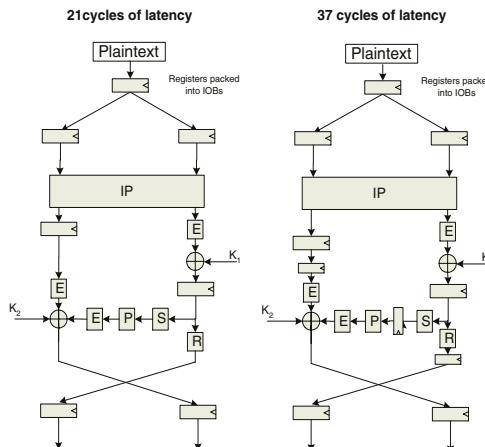


Fig. 3. Our two unrolled and pipelined DES designs.

⁵ E and R operators do not increase the critical path.

Table 1. Final results of our pipelined implementations.

Pipeline	Xil. 16-stage	Xil. 48-stage	Our 21-stage	Our 37-stage
LUTs used	4216	4216	3775	3775
Registers used	1943	5573	2904	4387
Frequency in XCV300-6	100 MHz	158 MHz	127 MHz	175 MHz
Data rate in XCV300-6	6.4 Gbps	10.1 Gbps	8.1 Gbps	11.2 Gbps
Frequency in XCV300E-6	132 MHz	189 MHz	176 MHz	258 MHz
Data rate in XCV300E-6	8.4 Gbps	12.0 Gbps	11.2 Gbps	16.5 Gbps
Frequency in XC2V1000-5	/	237 MHz	227 MHz	333 MHz
Data rate in XC2V1000-5	/	15.1 Gbps	14.5 Gbps	21.3 Gbps

implement a 16-bit shift register into one LUT. This design uses 484 extra LUTs for the key schedule calculation⁶.

The reason why we have a better speed result for the 37-stage pipeline is quite strange. Obviously, in their design, they do not put registers into IOBs and an additional pipelined stage before and after encryption. Without such registers, the critical path is in the input and output paths.

In addition, after checking and simulating their available source code on the web, we found two errors. First, they forgot to put a 1-stage pipeline after the XOR between key and R part. Actually, Xilinx implemented this 1-stage pipeline but they sent the XOR directly between key and R part into S-boxes, instead of the corresponding registered value. They also forgot to register the key just before the XOR function. Therefore, their critical path is quite longer. Finally, their solutions do not implement a correct DES that can encrypt every cycle.

To conclude, we propose two efficient and different solutions in terms of space and data rate. Depending on environment constraints, we really believe that one of the two designs should be well appropriate. Especially for high ratio *Throughput/Area*, the 37-stage pipelined solution is very efficient in terms of speed and slices used. Table 2 compares ratio *Throughput/Area* between 48-stage Xilinx implementation [3, 8] and our 37-stage implementation. We directly see the significative improvement.

Table 2. Comparisons with Xilinx implementation on Virtex-II-5

DES Version	Xil. 48-stage	Our 37-stage
Slices used	3900	2965
Data rate	15.1 Gbps	21.3 Gbps
Throughput/Area (Mbps/slices)	3.87	7.18

⁶ Estimation of the corresponding Xilinx implementation gives the use of about 900 LUTs for shift registers. No accurate values are given in [3, 8].

5 Our Sequential DES and Triple-DES Implementations

To be space efficient, we propose one DES and triple-DES sequential design, based on the B mathematical description of Figure 2. Instead of keeping a critical path of one LUT (as the best one of our pipelined designs), we prefer to limit the critical path to two LUTs (+ one F_5 function + one F_6 function). Indeed, if we limit the critical path to one LUT, we will encounter some critical problems with the routing part of the control signals. For example, to select the input data of a round with MUX functions, we will need 64 identical control signals. An unique signal will be generated with the control part. Then, this signal will be extended to all MUX operators. The propagation delay, of this high fanout signal, will approximatively correspond to one LUT delay. Therefore, we will have an critical path of about two LUTs. So, our sequential DES and triple-DES are designed to have a critical path of two LUTs.

First, we obtain one DES design that encrypts/decrypts, with possible different keys, every 18 cycles with no dead cycles. Our solution proposes almost the same interface as commercial cores [10, 12, 14]. Indeed, they propose modules with 16 cycles for encryption/decryption. Nevertheless, we get better throughput because of higher work frequency.

Figure 4 explains the key scheduling part for the computation of $K1$ and $K2$, needed in the data path. The selection of the input key is carried out with XOR operations. To control the correct functionality, all registers have to be cleared in the right cycles. The $reset_regKIN$ and $reset_K1$ signals ensure this constraint.

As shown on the B scheme of Figure 2, due to the irregular new DES description, we also need additional XOR operators in the first and last round. To carry out these operations, we use the data path part where we clear the output regis-

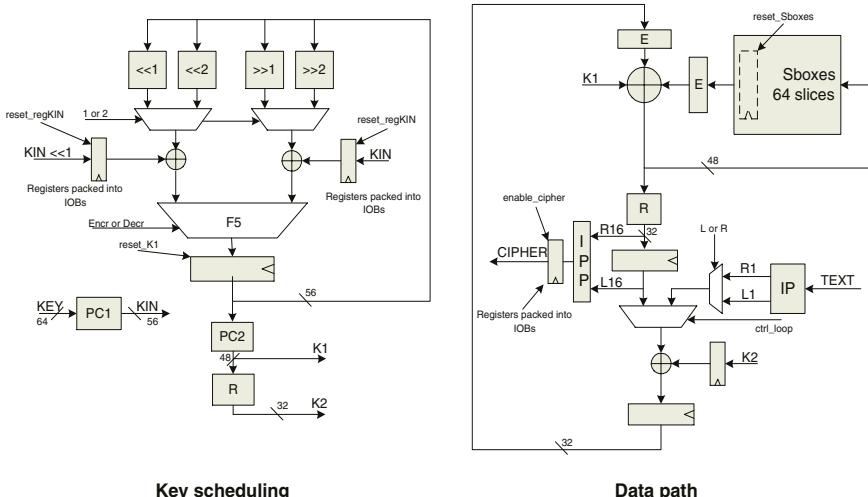


Fig. 4. Our sequential DES encryptor/decryptor.

ters of S-boxes, in the first and 18th cycles. This is done using the *reset_Sboxes* signal. If we decide to spare these XOR operators, we loose 2 cycles for the encryption/decryption. This is why we get a 18-cycle encryptor/decryptor.

In addition, we mainly focus our attention to maximize the utilization of resources in a slice. For example, we prefer to register the S-boxes outputs instead of other signals because it does not consume additional slices. If we do not this, using these slice registers for other signals will not allowed. Therefore, we will waste these registers. To manage all the signals, we develop a control unit that produces registered signals.

On the other hand, we also obtain one triple-DES design that encrypts/decrypts, with possible different keys, every 18 cycles with no dead cycles. This circuit is obtained using three of our DES sequential implementations. Our solution proposes almost the same interface as commercial cores [10, 11, 13]. Indeed, they mainly propose modules with 16 cycles for encryption/decryption. Nevertheless, we get better throughput because of higher work frequency.

Table 3 summarizes our result for DES and triple-DES sequential designs. In table 4 and 5, we compare our sequential DES and triple-DES with other existing implementations, in term of ratio *Throughput/Area*.

Table 3. Final results of our sequential DES and triple-DES implementations.

Sequential	DES	triple-DES
LUTs used	365	1103
Registers used	202	672
Slices used	189	604
Latency (cycles)	20	58
Output every (cycles)	1/18	1/18
Freq. in XCV300E-6	176 MHz	165 MHz
Freq. in XC2V1000-5	274 MHz	258 MHz

Table 4. Comparisons with other sequential DES implementations on Virtex-II -5.

Sequential DES	CAST	Helion	Ours
Slices used	238	$\simeq 450$	189
Throughput (Mbps)	816	640	974
Throughput/Area (Mbps/slices)	3.43	1.42	5.15

Table 5. Comparisons with other sequential triple-DES implementations on Virtex-II-5.

Sequential 3-DES	CAST	Gaj et al.	Ours
Slices used	790	614	604
Throughput (Mbps)	668	91 (XCV1000)	917
Throughput/Area (Mbps/slices)	0.85	0.15	1.51

6 Design Strategies to Optimize Cipher FPGA Implementations

FPGAs allow computing in parallel with high work frequencies. Nevertheless, wrong uses of their CLB structure can dramatically increase the speed efficiency as well as the resources used. Accurate slice mapping will permit to get fast and compact designs.

In this section, we set up conclusions for accurate cipher FPGA⁷ designs. Our conclusions are directly based on our DES and triple-DES implementations. We also mention the possible improvement of cipher implementations with a modified structure description. We propose systematic design rules to achieve very fast and compact cipher FPGA implementations, depending on the environment constraints. The following methodology is proposed:

1. Analyze and Modify the Mathematical Cipher Description:

Most block ciphers have a regular structure with a number of identical repeated round functions, depending on different subkeys. Subkeys are also generated using a number of identical repeated keyround functions. This regular and repeated round structure allows achieving efficient FPGA designs. Regular cipher structures permit very suitable sequential designs. Nevertheless, original cipher descriptions are not always optimized regarding the speed performance and number of slices used. An FPGA is based on slices composed by two 4-bit input LUTs and two 1-bit registers. Therefore, an optimal way to reduce the LUTs used is to regroup all the logical operations to obtain a minimum number of blocks that take 4-bit input⁸ and give 1-bit output.

Therefore, the aim of this step is to reduce the number of LUTs used as much as possible, modifying the structure description without changing the mathematical functionality. The following methodology has to be applied:

- (a) In round and keyround descriptions, try to move and/or inverse and/or merge blocks as well as possible.
- (b) Ignore round and keyround descriptions, and try to regroup blocks from different rounds. Try to keep a regular structure as much as possible for sequential designs.

If no improvements are possible, go to the next point.

2. Choose Your Design Based on Data Rate Constraints:

Depending on the data rate constraints, adopt one of the following designs:

- (a) For more than ± 10 Gbps constraints, choose a complete unrolled and pipelined implementation.
- (b) For less than ± 1 Gbps constraints, choose a sequential design.
- (c) For other data rate constraints, choose an incomplete unrolled and pipelined design or multiple sequential designs.

⁷ For Virtex technologies.

⁸ Sometimes 5-bit input.

3. Fit as Well as Possible into CLB Slices:

- (a) **For a complete unrolled and pipelined design**, it is important to achieve huge data rates. Therefore, we will adopt designs where we limit the critical path inside one slice (one LUT + one F_5 function + one F_6 function at most). Left part of Figure 6 shows some typical slice uses for one slice critical path designs.
- (b) **For sequential design**, it is important to achieve compact design. Due to control signals, it is better to limit the critical path to two LUTs (+ one F_5 function + one F_6 function). Indeed, if we limit the critical path to one LUT, we will encounter some critical problems with the routing part of the control signals. For example, to select the input data of a round with MUX functions, we will need β^9 identical control signals. A unique signal will be generated with the control part. Then, this signal will be extended to all MUX operators. The propagation delay, of this high fanout signal, will approximatively correspond to one or more LUT delay.

Therefore, we recommend to limit the control signals to one LUT and routing. The routing part generates additional delays. With an average 60-fanout signal, the delay corresponds to more and less one LUT delay. For the data path part, if we do not use fanout more than three, we can limit the critical path to two LUTs. Figure 5 summarizes our advices. Figure 6 shows some typical slice uses for two slice critical path designs. We show how to use slice flip flops to register input signals. Figure 6 does not illustrate all the slice configuration possibilities. It is again important to mention that we need to describe the VHDL code at a slice level in order to avoid bad slice mapping and extra slice uses.

- (c) **For an incomplete unrolled and pipelined design or multiple sequential designs**, previous advices are always valid depending on the pipelined or sequential choice.

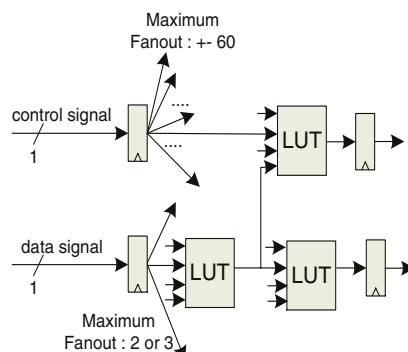


Fig. 5. Recommended architecture for sequential designs.

⁹ Typically β equals 64, 128 or 256.

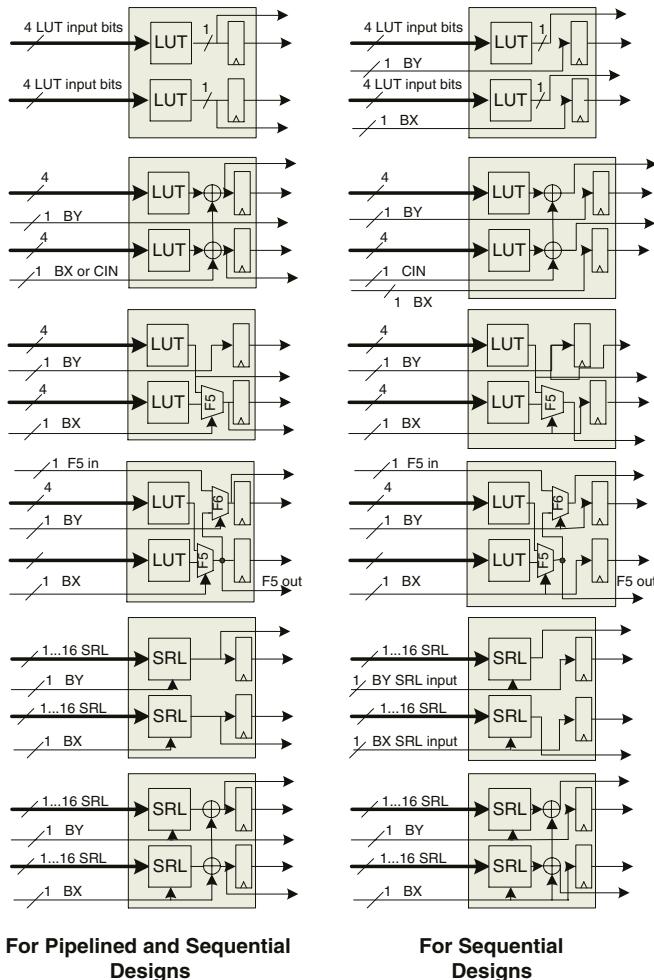


Fig. 6. Slice uses for different designs.

4. Deal with Input and Output Buffers:

To achieve good frequency results, input and output registers have to be packed into IOBs. In addition, we also recommend to put one more register stage in the input and output paths. This approach allows an additional degree of freedom for the place and route tool.

The previous explained rules set up a methodology to implement cipher algorithms. We do not affirm that this methodology is the most optimized systematic method. Nevertheless, it gives efficient results for DES and triple-DES algorithms.

7 Conclusion

We propose a new mathematical DES description that allows us to achieve optimized FPGA implementations. We get one complete unrolled and pipelined DES, and sequential DES and triple-DES implementations. Finally, we get the best DES and triple-DES FPGA implementations known nowadays in term of ratio *Throughput/Area*. The fastest DES gives a data rate of 21.3 Gbps (333 MHz). In this design, the plaintext, the key and the mode (encryption or decryption) can be changed on a cycle-by-cycle basis with no dead cycles. Our sequential triple-DES computes ciphertexts every $\frac{1}{18}$ cycle and produces a data rate of 917 Mbps, using 604 slices.

Based on our DES and triple-DES results, we also set up conclusions for optimized cipher FPGA designs. We recommend to modify the mathematical cipher description. Depending on our data rate constraints, we advice to choose unrolled and pipelined designs or sequential designs. We also show how to fit in slices as well as possible and how to deal efficiently with input and output buffers.

References

1. J.M. Rabaey. *Digital Integrated Circuits*. Prentice Hall, 1996.
2. Xilinx. Virtex 2.5V field programmable gate arrays data sheet. available from <http://www.xilinx.com>.
3. Xilinx, V. Pasham and S. Trimberger. High-Speed DES and Triple DES Encryptor/Decryptor. available from <http://www.xilinx.com/xapp/xapp270.pdf>, Aug 2001.
4. B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.
5. National Bureau of Standards. *FIPS PUB 46*, The Data Encryption Standard. U.S. Departement of Commerce, Jan 1977.
6. FreeIP. <http://www.free-ip.com/DES/index.html>.
7. C. Patterson. High performance DES encryption in Virtex FPGAs using Jbits. In *Proc. of FCCM'01*, IEEE Computer Society, 2000.
8. S. Trimberger, R. Pang and A. Singh. A 12 Gbps DES encryptor/decryptor core in an FPGA. In *Proc. of CHES'00*, LNCS, pages 156–163. Springer, 2000.
9. M. Davio, Y. Desmedt, M. Fosséprez, R. Govaerts, J. Hulsbosch, P. Neutjens, P. Piret, J.J. Quisquater, J. Vandewalle and P. Wouters. Analytical Characteristics of the DES. In David Chaum, editor, *Advances in Cryptology - Crypto '83*, pages 171–202, Berlin, 1983. Springer-Verlag.
10. Helion Technology. High Performance DES and Triple-DES Core for XILINX FPGA. available from <http://www.heliontech.com>.
11. CAST, Inc. Triple DES Encryption Core. available from <http://www.cast-inc.com>.
12. CAST, Inc. DES Encryption Core. available from <http://www.cast-inc.com>.
13. inSilicon. X.3 DES Triple DES Cryptoprocessor. available from <http://www.insilicon.com>.
14. inSilicon. X_DES Cryptoprocessor. available from <http://www.insilicon.com>.

15. P.Chodowiec, K. Gaj, P. Bellows and B. Schott. Experimental Testing of the Gigabit IPSec-Compliant Implementations of RIJNDAEL and Triple DES Using SLAAC-1V FPGA Accelerator Board. In *Proc. of ISC 2001: Information Security Workshop*, LNCS 2200, pp.220-234, Springer-Verlag.
16. J.P. Kaps and C. Paar. Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine. In *Proc. of SAC'98 : Selected Areas in Cryptography*,LNCS 1556, pp. 234-247, Springer-Verlag.
17. G. Rovroy, FX. Standaert, JJ. Quisquater, JD. Legat. Efficient Uses of FPGA's for Implementations of DES and its Experimental Linear Cryptanalysis. Accepted for publication on April 2003 in IEEE Transactions on Computers, Special CHES Edition.

Using Partial Reconfiguration in Cryptographic Applications: An Implementation of the IDEA Algorithm

Ivan Gonzalez, Sergio Lopez-Buedo, Francisco J. Gomez, and Javier Martinez

Escuela Politecnica Superior, Universidad Autonoma de Madrid,
Cantoblanco E-28049
Madrid, Spain

{Ivan.Gonzalez, Sergio.Lopez-Buedo, Francisco.Gomez,
Javier.Martinez}@uam.es

Abstract. This paper shows that partial reconfiguration can notably improve the area and throughput of symmetric cryptographic algorithms implemented in FPGAs. In most applications the keys are fixed during a cipher session, so that several blocks, like module adders or multipliers, can be substituted for their constant-operand equivalents. These counterparts not only are faster, but also use significantly less resources. In this approach, the changes in the key are performed through a partial reconfiguration that modifies the constants. The International Data Encryption Algorithm (IDEA) has been selected as a case-study, and JBits has been chosen as the tool for performing the partial reconfiguration. The implementation occupies an 87% of a Virtex XCV600 and achieves a throughput of 8.3 GBits/sec.

1 Introduction

Most operations in the IDEA algorithm [1-4] involve data coming from both the plaintext and the key. This is not a peculiarity of IDEA: most symmetric algorithms work in the same way, like DES [5,6] and Rijndael [7]. Thus, if the key were fixed, then a major portion of the algorithm operations will be by constant. This looks very appealing, as it will imply a benefit in both area and performance. It is widely known that the operations by constant not only are faster, but also occupy less area [8,9].

Unfortunately, a ciphering hardware that has a fixed key does not make too much sense, because it would offer a poor security. However, the key remains constant during the ciphering sessions, which are relatively long periods. For example, in a Virtual Private Network connection, once the encryption key has been negotiated, it is maintained during all the session. Then, in an FPGA environment, one could think of using a fixed-key hardware that would be reconfigured each time the key changes. This will be the optimal solution, because it will take advantage of the benefits of a constant-key hardware, but anyway the key can be changed by the user. This is right as long as the reconfiguration is fast enough not to stop the ciphering for an unacceptable amount of time. Therefore, the hardware should be designed in such a way that

the key can be modified just by altering the contents of a few LUTs, so the changes can be done fast and easily. This approach has been followed in this paper, using JBits for performing the reconfiguration. As it is detailed in the results, it provides a significant improvement in both area and performance when compared to a conventional (variable key) solution.

2 The IDEA Algorithm

IDEA is a symmetric algorithm [1-4], that is, the same key is used for both encryption and decryption. IDEA encrypts 64-bit blocks with a 128-bit key, using three simple operations: or-exclusive, module 2^{16} addition and module $2^{16}+1$ multiplication.

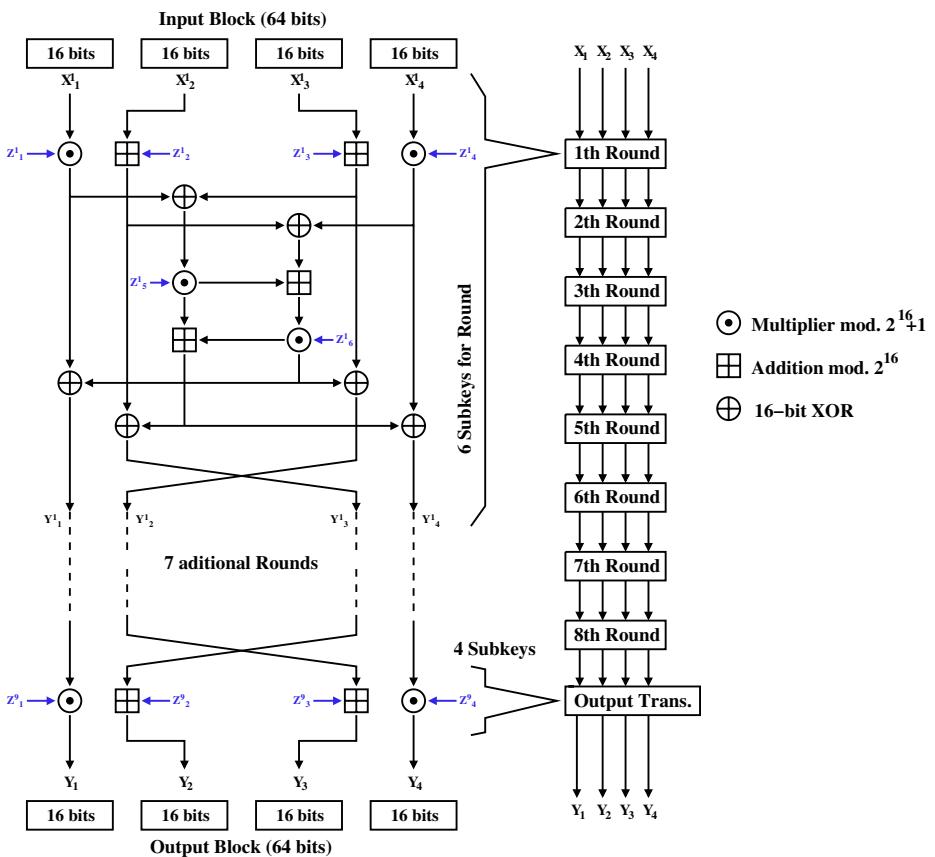


Fig. 1. The IDEA Algorithm.

As it can be seen in Fig.1, the input data passes through 8 similar stages, called rounds, and there is a last one, called output transformation. The only difference in the rounds is the 16-bit subkeys, noted in the figure as Z_j^l . The first 8 subkeys are

obtained directly from the original 128-bit key. Then, this key is rotated 25 times to the left to obtain the second set of 8 subkeys, and so on, till all the 52 subkeys are created. That is, six subkeys per stage plus four subkeys for the last transformation. This process needs to be done only when the key is changed. Finally, the only difference between encryption and decryption is the subkey generation; the same datapath depicted in Fig. 1 is used for both actions.

3 Design Methodology

The approach taken in this paper for implementing IDEA is replacing all the operational units involving the key with its constant-operand equivalents; as Fig. 1 shows, these units are the module 2^{16} addition and the module $2^{16}+1$ multiplication. The benefits from this methodology come from two sides. First, as it was stated in the introduction, arithmetic or logic operators by constant are faster and occupy less area, and this is especially true in FPGAs. And second, the subkey generation hardware is eliminated [4,10], because this operation is done in an associated microprocessor which calculates the new constants and reconfigures the FPGA.

For this methodology to be useful, the modifications to be done in the circuit when the key is changed must be easy to perform. This certainly implies that the changes will involve only the contents of the LUTs that build up the operators. Changing the routing or adding new hardware is a complex task, and few tools support it. Another advantage of limiting the changes to the LUT contents is that the circuit can be created using the traditional tools; in this case, it was created using VHDL. The only aspect that has to be taken into consideration is that the constant-operand module adders and multipliers must be designed in such a way that they can be changed only by reconfiguring a few LUTs.

Thus, the operation of the circuit can be summarized as follows. First, the FPGA is loaded with the IDEA using an initial session key. When the key has to be changed, an associated microprocessor calculates all the subkeys and the new LUT values, and sends the new configuration to the FPGA. Here, this process has been done using JBits, but any other tool could have been used, or even the bitstream could have directly been altered, using the information provided by Xilinx [11]. The need of a microprocessor might appear as a drawback. But nowadays, nearly all systems include at least one. It may even be embedded on the FPGA, so no additional devices would be needed. Moreover, as the complexity of the subkey generation and the reconfiguration is very low, a little overhead will be imposed to the existing microprocessor.

3.1 JBits

Basically, JBits [12,13] is a Java API (application program interface) that allows the designer to describe digital circuits using Java. Currently, it is available only for Virtex FPGAs. The main advantage of JBits is that the designs can be parameterised

at run-time. That is, the circuit is generated by running a Java program, which at run-time takes the decisions in order to adapt the generated circuit to the current circumstances: area available, device present... This methodology is called run-time parameterizable (RTP) cores [14].

Another advantage of JBits, closely related to the previous one, is its support for partial reconfiguration [15]. The interface for handling it is straightforward; only a few lines of code are necessary to load a new bitstream and reconfigure the FPGA. JBits automatically infers the differences between them current bitstream and the new one, and only reconfigures the frames that have changed. That is the main reason why JBits has been selected in this paper.

3.2 Addition Module 2^{16}

In this case there is not a significant improvement regarding the non-constant operand version. Only the register needed for storing the subkey is saved. The interest of the proposed solution is to give an example of how to avoid undesirable optimizations that could be carried out by the synthesis tool.

The basic block of a generic n-bit adder is a two bits adder with 1 bit carry. A structural description in VHDL using the primitives of the Virtex slice was developed. In this implementation the XOR operation to add the two input bits is mapped into a 2 input LUT.

The version with a constant-operand is obtained by transforming the previous LUT into an 1 input LUT. The LUT operation inverts the input when the bit of the constant is 1 or it transfers the input to the output when the bit of the constant is zero. As the value of the constant may be modified at execution time by partial reconfiguration, this LUT should be configured by default like an inverter, avoiding undesirable optimization, and later proceed to change its functionality by means of JBits, like it is shown in the following code:

```

int[] f_zero = Expr.F_LUT("~-F1");
int[] f_one = Expr.F_LUT("F1");

if(((constant >> bit)&0x1) == 0)
    jbits.set(row,column,LUT.SLICE0_F,f_zero);
else
    jbits.set(row,column,LUT.SLICE0_F,f_one);

```

3.3 Multiplier Module $2^{16}+1$

A constant (k) coefficient multiplier (KCM) [8,9] has been considered because its size and layout are independent of the constant encoded within it, and the mechanism for implementing the multiplication is contained in look-up tables. The block diagram of a 16-bit KCM multiplier is shown in figure 2.

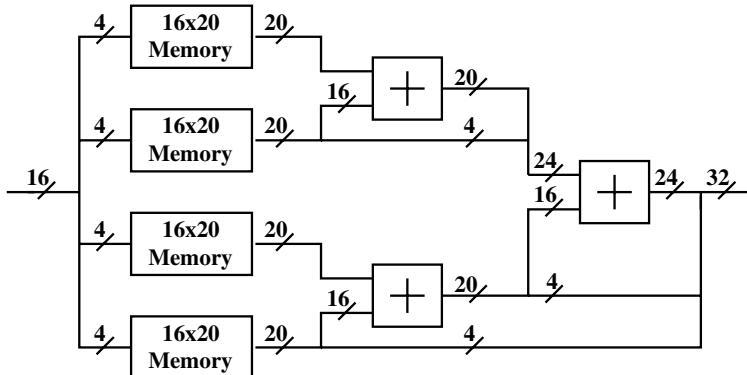


Fig. 2. A 16-bit KCM Multiplier.

The 16-bit version of a KCM holds 4 copies of the constant (k) multiplication table, each of which contains 16 entries. In a KCM implemented in Virtex, when multiplying an n -bit operand by a constant, $n/4$ copies of the multiplication table are required (because the LUTs have 4 inputs). Each group of $n/4$ bits of the operand is used to address a table, and the partial products thus obtained are added to get the product. This KCM multiplier is between 3 and 3.8 [8] times smaller than a generic one.

It can be seen in figure 2 that the overall structure of the KCM is independent of the constant encoded within it. Only the contents of the look-up tables vary, and therefore they have to be reconfigured when the constant is changed. A first approach would be implementing these memories as LUT ROMs. However, there is a potential problem: the place and route tool might swap the address lines to simplify the routing. This can be solved by using the SRL16 primitive [9] instead of a ROM. This primitive corresponds to a shift register, not to a memory. But if the write enable is tied low, and the contents of the register are initialized correctly, it works in the same way as memory. But, as it is not strictly a memory, the routing tool can not swap the input lines.

The calculation of the module $2^{16}+1$ is based in the Low-High algorithm:

```

uint16 modmult(uint16 operand) {
    word32 p;
    uint16 c, d;

    if(k) {
        if(operand) {
            p = (word32)k * operand;
            c = low16(p);
            d = p >> 16;
            return c - d + (c < d);
        }
        else return 1 - k;
    } else return 1 - operand;
}

```

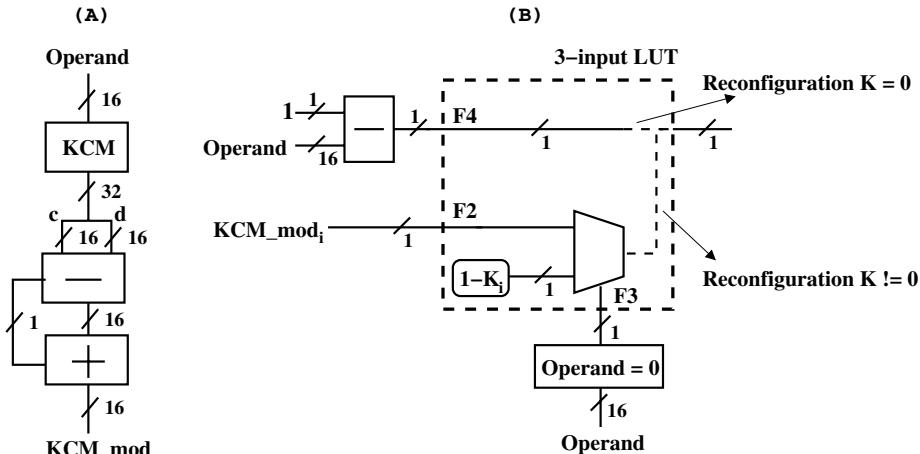


Fig. 3. Module $2^{16}+1$ multiplier. (A) KCM module $2^{16}+1$ with non zero operands. (B) Output selector.

An adaptation of this algorithm for reconfigurable hardware is shown in figure 3. As it can be seen in Fig. 3A, the output carry of the ‘c-d’ subtraction is used to perform the ‘c<d’ comparison.

An output selector (Fig. 3B) using LUT reconfiguration has implemented the multiple-choice selection in the Low-High algorithm. At configuration time, there are two possibilities to be considered according to the constant value. When the constant is zero the LUT works as a buffer that translates their input F4 to the output. In this case the LUT input F4 is connected to the result of the (1-Operand)_i subtraction.

Otherwise, with a non-zero constant, the (1-K)_i value are hardwired into the LUT and also, the LUT is configured as a multiplexer. When the operand is zero the output is (1-K)_i, otherwise the value of the KCM_mod block connected to the input LUT (F2).

This design also presents the problem that the LUT inputs might be swapped by the routing tool, and it should be implemented using SRL16 components.

The JBits code for this reconfigurable output selector is the following:

```

int aux = (1-constant)&0xFFFF;
int[] f_operand = Expr.F_LUT("~F4");
int[] f_function1 = Expr.F_LUT("(~(~F3 | (F3 & F2))");
int[] f_function0 = Expr.F_LUT("~(F3 & F2)");

if(constant == 0)
    jbits.set(row,column,LUT.SLICE0_F,f_operand);
else if((aux >> bit)&0x1) == 0)
    jbits.set(row,column,LUT.SLICE0_F,f_function0);
else
    jbits.set(row,column,LUT.SLICE0_F,f_function1);

```

3.4 Relative Location Constrains

In order to reconfigure a component, JBits must know where it is located inside the FPGA. Using placement attributes in the VHDL code, and particularly, LOC and RLOC, this problem is resolved completely or partially. LOC establishes the fixed position of each component, so it is possible to know a-priori where they are found, but it is a hard constraint for the place&route tool. RLOC is a soft constraint, because it permits the user to give a relative position between the basic elements of each component. Then, the tool can choose the component position. Generally, better results are obtained because RLOC is more flexible than LOC attribute. The drawback is that, after the implementation, the reports have to be examined to find the position of the different components to reconfigure.

These relative location attributes are a slice level constrains. If the two LUTs of the slice are used, the BEL attribute indicates which element goes in the F LUT and which one in the G LUT.

All reconfigurable components have been relatively placed in columns using RLOC. This location constrain decreases the number of frames that are necessary to reconfigure during the process, therefore minimizing the reconfiguration time.

4 Pipelined Algorithm

Pipelining is necessary to increase the throughput. In the IDEA case it is straightforward, because if Electronic CodeBook (ECB) chaining mode of operation is employed, the datapath has no feedbacks (Fig. 1). In a first approach, the segmentation registers will be added after every basic operation. Then, the clock period will be limited by the delay of the most complex operator, the module $2^{16}+1$ multiplier. But the throughput attainable using this degree of pipelining is fairly poor, because of the complexity of the multiplier. Thus, in the final design the multiplier has been pipelined again.

The only drawback of pipelining ECB-mode IDEA is that it causes a large area increment because has many nets which cross the segmentation boundaries. In FPGAs, pipelining after a logic operator does not generally cause an area increment, because there are as many LUTs as flip-flops. But if the pipelining crosses a net, the flip-flops have to be obtained from new CLBs, which probably will have their LUTs unused.

4.1 Pipelined Module $2^{16}+1$ Multiplier

The module multiplier is the most complex component of IDEA, having a significantly greater delay than the other operations. In order to reduce this delay, it has been segmented in 4 stages. The first three stages correspond to the pipelining of the KCM multiplier, adding registers after the 16x20 memories and the two adder stages (Fig. 4A). This segmentation affects directly to the rest of the module $2^{16}+1$ multiplier, so it is necessary to spread it to the other blocks (Fig. 4B). A fourth stage is

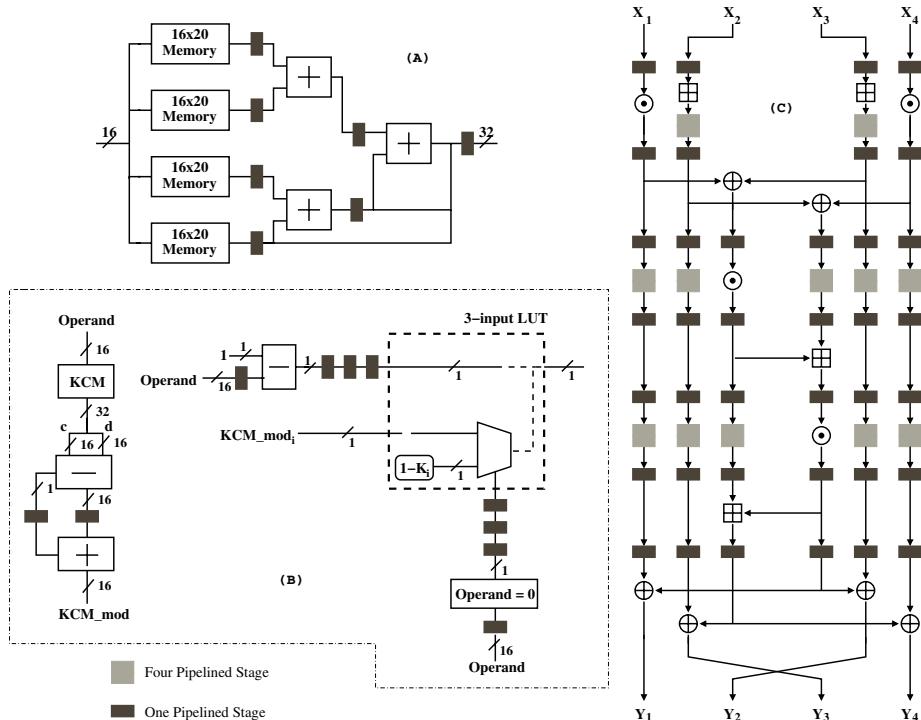


Fig. 4. (A) Pipelined KCM. (B) Pipelined Module Multiplier (C). Pipelined IDEA Round.

added before the adder generating KCM_mod. Finally, another segmentation stage is added after the multiplier (Fig. 4C).

4.2 Final Design for the Pipelined IDEA

The final design of the IDEA algorithm uses 19 pipelined stages per round and 6 pipelined stages for the output transformation. It implies a total of $(8 \times 19) + 6 = 158$ pipelined stages. So 158 clock cycles are necessary for a 64-bit input plaintext block to arrive at the output. This latency is very high for applications that encrypt/decrypt small quantities of data, but usually the cryptographic algorithms work on a stream of bytes, so it represents no problem.

5 Results

The fully pipelined IDEA algorithm uses only an 87% of a XCV600 and works at 131 MHz, obtaining a remarkable throughput of 8.3 Gbits/sec. This result is compared in table 1 with several implementations. By comparing with the alternative pipelined at

module multiplier level [16], this implementation shows an area reduction of 18%, while the throughput is five times better. The best FPGA implementation of IDEA that has been referenced [2] has a throughput of 6.8 GBits/sec. This throughput has been improved in the results obtained in this paper, and the area used is smaller, nearly 33% less. The implementation presented in [4] also uses reconfiguration, but it is not comparable with the solution offered in this paper, because it is not a fully unrolled implementation of IDEA.

Table 1. Results.

Implementation	Slices	Device	Frequency	Throughput	Latency
This paper	6078	87% XCV600-6	131.1 MHz	8.3 GBits/sec	1.20 us
Deeply Pipelined [2]	9052	73% XCV1000-6	105.9 MHz	6.8 GBits/sec	1.24 us
Cheung et al. [4] 2 round cores	2444	79% XCV300-6	82 MHz	1.2 GBits/sec	2.13 us
Pipelined [16]	7410	60% XCV1000-6	24.5 MHz	1.5 GBits/sec	2.37 us
Combinational [16]	6863	55% XCV1000-6	1.4 MHz	0.8 GBits/sec	0.71 us

6 Conclusions

The results show that using key replacement through partial reconfiguration can significantly improve the throughput and area of FPGA implementations of cryptographic algorithms. The selected case-study, fully unrolled IDEA implementation, using other design methodologies had to be implemented on a XCV1000 [2,16]. But if this approach is employed, it fits in a XCV600 because it saves a 33% in area. This method not only provides area savings: the throughput is improved by 23% in comparison to the deeply pipelined solution. The area comparison is not so favourable in less pipelined implementations (only 18% less), but in this case the throughput is much higher, more than 500%. Although the key can only be changed using reconfiguration, this can not be considered a problem, because the keys remain constant during the ciphering sessions, and the partial reconfiguration can be performed very fast, in less than 4 ms for the example used in this paper. Finally, the design was made with the standard design tools and VHDL. By employing a set of techniques which have been described in this paper, the encryption key can be changed by altering the contents of only a few LUTs. This has successfully been done using JBits.

Acknowledgement

This work has been partially supported by the Spanish Government under project number TIC2000-0464.

References

1. J. L. Beuchat, J.O. Haenni, H.F. Restrepo, C. Teuscher, F. J. Gomez, and E. Sanchez, "Approches matérielles et logicielles de l'algorithme de chiffrement IDEA", *Technique et Science Informatiques (TSI)*, vol. 1, pp. 203-224, 2001 (in French).
2. Antti Hämäläin, Matti Tommiska, and Jorma Skyttä, "6.78 Gigabits per Second Implementation of the IDEA Cryptographic Algorithm", *Proc. 12th International Conference FPL 2002*, pp. 760-769, Montpellier, France, September 2002. Berlin: Springer Verlag, 2002.
3. M. P. Leong, O. Y. H. Cheung, K. H. Tsoi, and P. H.W. Leong, "A Bit-Serial Implementation of the International Data Encryption Algorithm IDEA", *Proc. 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 122-131, Napa, California, 2000.
4. O.Y.H. Cheung, K.H.T. Soi, P.H.W. Leong, and M.P. Leong, "Tradeoffs in Parallel and Serial Implementations of the International Data Encryption Algorithm IDEA", *Proceedings of CHES*, pp. 333-347, Paris, 2001
5. Cameron Patterson, "High Perfomance DES Encryption in Virtex FPGAs using JBits", *Proc. 2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 113-121, Napa, California, 2000.
6. J. Leonard and W. Magione-Smith, "A Case Study of Partially Evaluated Hardware Circuits: Key-Specific DES", *Proc. Seventh International Workshop on Field-Programmable Logic and Applications FPL '97*, London, UK, 1997. Berlin: Springer Verlag, 1997.
7. J. Daemen and V. Rijmen, "AES Proposal: Rijndael, NIST AES Proposal", 1998.
8. Xilinx XAPP 054 (<http://www.xilinx.com/xapp/xapp054.pdf>)
9. Philip James-Roxby and Brandon J. Blodget, "A Study of high-perfomance reconfigurable constant coefficient multiplier implementations", Xilinx Inc., Tech Notes Archive Chipcenter (<http://www.chipcenter.com/pld/images/pldf085.pdf>)
10. P.H.W. Leong and K.H. Leung, "A Microcoded Elliptic Curve Processor using FPGA Technology", *IEEE Transactions on VLSI Systems*, accepted for publication, 2002
11. Xilinx XAPP 290 (<http://www.xilinx.com/xapp/xapp290.pdf>)
12. S. A. Guccione and D. Levi, "JBits: A Java-based Interface to FPGA Hardware", San Jose, California: Xilinx Inc., 1998.
13. Steven A. Guccione, Delon Levi, and Prasanna Sundararajan, "JBits: A Java-based Interface for Reconfigurable Computing", *Proc. 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*.
14. S. A. Guccione and D. Levi, "Run-Time Parameterizable Cores", San Jose, California: Xilinx Inc., 1999.
15. S. McMillian and S. A. Guccione, "Partial Run-Time Reconfiguration using JRTR", San Jose, California: Xilinx Inc., 2000.
16. I. Gonzalez, "Codiseño en Sistemas Reconfigurables basado en Java", *Internal Technical Report*, UAM, Madrid, 2002 (in Spanish).

An Implementation Comparison of an IDEA Encryption Cryptosystem on Two General-Purpose Reconfigurable Computers

Allen Michalski¹, Kris Gaj¹, and Tarek El-Ghazawi²

¹ ECE Department, George Mason University
4400 University Drive, Fairfax, VA 22030, U.S.A.
`{emichals, kgaj}@gmu.edu`

² ECE Department, The George Washington University
801 22nd Street NW, Washington DC 20052, U.S.A.
`tarek@seas.gwu.edu`

Abstract. The combination of traditional microprocessors and Field Programmable Gate Arrays (FPGAs) is developing as a future platform for intensive computational computing, combining the best aspects of traditional microprocessor front-end development with the reconfigurability of FPGAs for computation-intensive problems. Several prototype PC-FPGA machines have demonstrated significant speedups compared to standalone PC workstations for computationally intensive problems. Cryptographic applications are a clear candidate for this type of platform, due to their computational intensity and long operand lengths. In this paper, we demonstrate an efficient implementation of IDEA encryption, using two of the leading reconfigurable computers available, SRC Computers' SRC-6E and Star Bridge Systems' HC-36. We compare the hardware architecture and programming model of these reconfigurable computers, and the implementation of a common IDEA encryption architecture in both platforms. Detailed analyses of FPGA resource utilization for both systems, data transfer and reconfiguration overheads for the SRC system, and a comparison between SRC and a public domain software implementation are given in the paper.

1 Introduction

The need for faster computational processing methods has grown along with the desire to process large amounts of data in shorter periods of time. Approaches to this problem have typically involved microprocessor-based solution, utilizing concurrent processing with multiple processors within a single machine or in a cluster of workstations over a network. Since their introduction, FPGAs have emerged as a low cost complement to traditional microprocessor software and hardware solutions for computationally intensive applications, due to their hardware reconfigurability [1].

Several workstation-based FPGA systems are available today for commercial applications. These “reconfigurable computers” make use of workstation microprocessor(s) for front-end processing, and provide one or more FPGAs for computations less suited for a microprocessor. The choice of how a design can be divided between a microprocessor and an FPGA depends on the development

environment available for the system. In this paper, we explore the efficient implementation of pipelined IDEA encryption within two such available reconfigurable computers: the SRC-6E from SRC Computers, and the HC-36 from Star Bridge Systems. This paper discusses the hardware architectures and development tools of both systems in detail, and provides FPGA timing and utilization results for both systems. A summary including the state of development of both systems is also presented.

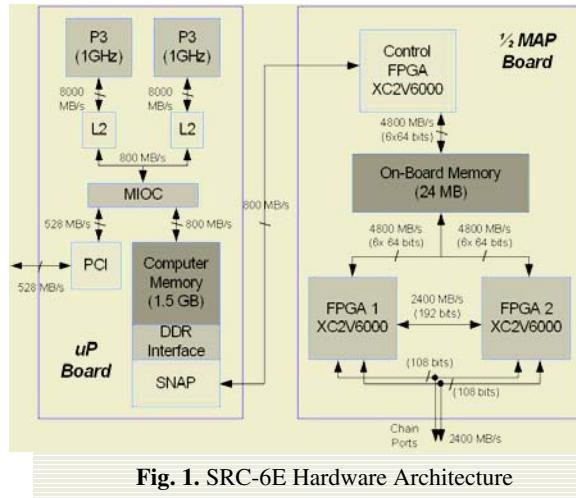


Fig. 1. SRC-6E Hardware Architecture

2 FPGAs and Reconfigurable Computing

An FPGA is a regular structure of basic modules called Configurable Logic Blocks (CLBs), which can be interconnected to provide hardware implementations of algorithms required by a designer. FPGA interconnects between modules are under the designer's complete control [2]. The FPGAs used in both the SRC and Star Bridge systems are the Xilinx Virtex II 6000 series, which have a capacity of six million system gates, and provides dedicated logic for fast carry propagation for addition operations, multipliers that handle operand sizes up to 18 bits, and block RAMs for local memory access. FPGAs are reconfigurable, meaning that the FPGA device can be configured to carry out a specific function, and can be reconfigured to carry out a different function at a later time.

The term “reconfigurable computing” is used to describe a combination of microprocessor systems with FPGAs to provide a reconfigurable hardware environment. Both the SRC and Star Bridge systems are examples of reconfigurable computing platforms. Reconfigurable computers offer benefits over microprocessor-based hardware solutions because FPGAs can more easily exploit computational precision and operand sizes required by the design, and can implement operation pipelining and parallelism specific to the needs of the application being developed. Microprocessor instruction sets have fixed operand lengths that may not match

operand sizes specific to the design, and the implementation of algorithms typically involves multiple-instruction executions for one algorithmic operation. In addition, much of the microprocessor's capability is unutilized, since a general-purpose microprocessor is designed to implement a wide variety of operations specific to a workstation computing environment, whereas most design requirements for cryptographic systems only use a subset of the full capabilities of a microprocessor. All of the resources of an FPGA can be dedicated to the needs of a design, which provides a more efficient implementation versus a single or multi-processor-based design solution.

3 The SRC-6E

3.1 SRC-6E Hardware Overview

The SRC-6E system architecture frontend consists of two dual Intel processor motherboards. Each motherboard contains two Intel P3 Xeon processors and 1.5 GB of memory. Each system hosts a multi-processor version of the Linux operating system, and provides two distinct Linux-based microprocessor-FPGA reconfigurable computers.

An SRC MAP® processor is attached to each Intel motherboard, as shown in Fig. 1. Each MAP processor consists of two Xilinx Virtex II 6000 FPGA chips available for user logic, and a control processor, which is also a Xilinx Virtex II 6000 FPGA, all running at a clock rate of 100 MHz. The control processor implements fixed control logic, and is responsible for direct memory access (DMA) transfers between Intel system memory and the onboard memory of the MAP processor. The user logic and control FPGAs have access to six banks of dual port 512k x 64 bit static RAM providing a total of 24MB of memory external to the FPGAs. The MAP control processor communicates with the Intel processors through a SNAP interconnect. The SNAP interconnect is a high speed, low latency interface which functions as a Double Data Rate (DDR) memory interface, and plugs into a DDR SDRAM slot on the motherboard. SNAP provides higher data throughput between the Intel processor and the MAP processor versus component interfacing using the PCI-X bus.

3.2 SRC-6E Programming Environment

SRC has created a development environment that uses traditional programming paradigms in addition to hardware description language (HDL) development for implementing a design with the MAP FPGAs, as shown in Fig. 2. The SRC environment provides the ability to implement FPGA user logic using either C or FORTRAN sourcecode alone, or in combination with HDL sourcecode. GNU compilers are provided for C or FORTRAN sources that target the Intel processors, and SRC provides its own C or FORTRAN compilers that target the MAP processors. The MAP processor compilers produce Verilog code which is then synthesized using Synplify Pro, and Xilinx tools perform map, place and route. The GNU compilers allow the use of ANSI-compliant C or FORTRAN code and libraries, which allows for integration with other existing UNIX applications.

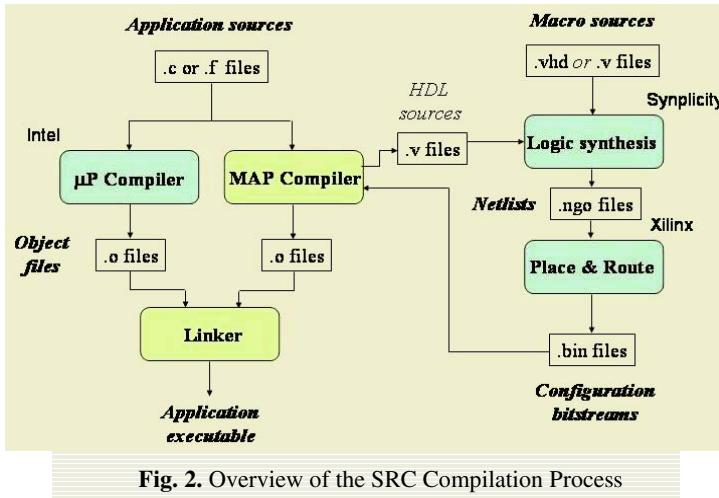


Fig. 2. Overview of the SRC Compilation Process

The design environment is hosted in each Linux platform within the SRC-6E. The compilation process consists of compilation of user logic HDL files, if present, that will execute on the MAP processor, compilation of C or FORTRAN code that will also execute on the MAP processor, and compilation of the C or FORTRAN code that will execute on the Intel processors. The compilation process places wrapper code around logic that resides within the FPGAs to facilitate data transfer and control synchronization between the microprocessor and MAP processor. The binary files produced by each compile process are combined into one single executable for the host Intel platform, which is responsible for loading the Intel and MAP processors with the code compiled for each.

3.3 Designing within the SRC-6E

Designs in SRC can have required operations performed on either the Intel or MAP processors. Within the MAP processor, FPGA designs can be implemented using a C or FORTRAN source alone, a C or FORTRAN source in combination with HDL, or completely in HDL, using a C or FORTRAN source to provide data transfer services. A simple API is available in the high-level language (HLL) to provide control functionality and data transfer functions between the Intel and MAP processors, and the passing of data to and from included user HDL designs. MAP C sourcecode data types are limited to 32 and 64 bit types.

HDL source that is targeted for the MAP processor is called a macro. SRC supports either VHDL or Verilog sources within macros. A macro's primary characteristics are defined as functional, stateful, or external. Additional characteristics define latency of the design and whether the design is pipelined. An “info” file is used to describe these characteristics. A functional macro is one that carries no state information, and therefore doesn't require the reset of a state machine. A stateful macro is one that carries state information and may need to be reset during a data cycle. An external macro is one that needs direct hardware access to memory, versus using HLL calls to read memory and supply data to the design. Latency defines

how many clock cycles are needed before a result is available, and the pipeline attribute defines whether the macro can take data on each clock cycle. The combination of all these attributes defines the HDL macro supplied by the user.

If a C or FORTRAN source is used alone or in combination with HDL source to implement the design within the MAP processor, the compiler attempts to extract the maximum parallelism from the code and generate pipelined hardware logic for instantiation in the MAP FPGAs [3].

3.4 Debugging within the SRC-6E

Debugging within SRC can be attained by traditional functional testing of HDL outside of SRC and traditional HLL debugging techniques, or using SRCs MAP debugger. The MAP debugger provides hardware debugging of code compiled for the MAP processor, and emulation debugging of emulation or simulation code. Emulation is based on specially formatted “data flow graph” C routines, provided by the compiler from existing HLL source if that is used to implement logic, or by the user in an additional file if HDL is also used, that emulates the functionality of the design being implemented in the MAP processor. Simulation mode actually simulates the HDL produced by the compiler, which requires a license for a Verilog simulator not included in the development environment.

4 The Star Bridge HC-36

4.1 HC-36 Hardware Overview

The Star Bridge HC-36a consists of a Tyan S2720 dual Intel P4 Xeon processor motherboard with 4 GB of memory. Attached to the motherboard through the PCI-X bus is a PCI card containing two Virtex II 4000 FPGAs for dedicated PCI and hardware control, one Virtex II 6000 FPGA allocated for user-designed FPGA control functions, and 4 Virtex II 6000 FPGAs available for user logic, as shown in Fig. 3. FPGA clocks are set to the PCI-X clock speed, with the FPGA digital clock manager used to provide clock speeds other than the PCI-X clock speed. Star Bridge uses the name “HyperComputer®” to refer to this combination of FPGAs, an Intel-based frontend, and Star Bridge’s VIVA development environment [4]. The environment is hosted in Microsoft’s Windows 2000 Server operating system.

FPGAs available for user logic are referred to as PE1 to PE4. These four PEs (Processing Elements) have dedicated 50-bit connections to each other. Each PE, along with the XPoint FPGA mentioned below, has four banks of memory, each bank having 512 MB of RAM with a 64-bit PE interface. This gives a total of 10 GB of memory with 20 64-bit independent memory channels within the user FPGAs.

Control FPGAs available to the user consist of an FPGA known as XPoint and a router FPGA. XPoint connects to the four user FPGAs using dedicated 32-bit data connections, and is allocated to provide user-defined control interfacing to the user FPGAs. The router FPGA has dedicated 94-bit connectivity to the user FPGAs, and provides additional user FPGA connectivity.

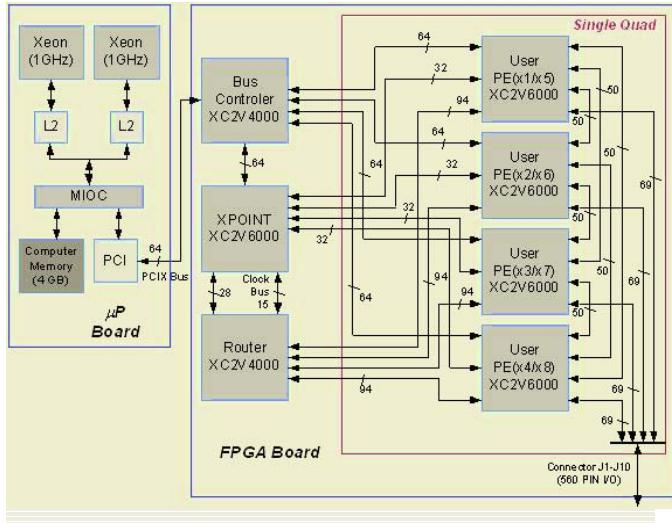


Fig. 3. Star Bridge HyperComputer FPGA Configuration

The base grouping of four PEs is called a Quad Element. Only one Quad Element is available in the HC-36a, however each XPoint control FPGA can be used to control two Quad Elements. This structure of one XPoint control FPGA along with two Quad Elements can be replicated to create larger FPGA arrays, and is used to create larger Star Bridge platforms [4].

4.2 The Star Bridge Programming Environment

VIVA® is Star Bridge's proprietary development environment. It provides a graphical user interface based on "drag and drop" design principles. VIVA's design language, "Implementation Independent Algorithm Description Language" or IIADL, is object-based and uses object attributes to specify different options within the design. Fig. 4 shows the VIVA Integrated Development environment (IDE).

Designs within VIVA can be allocated to execute on either the Intel platform or the FPGA array. Objects that execute on the Intel processor are implemented using pre-built VIVA libraries within Windows. VIVA is able to use Windows COM (Component Object Model) to communicate with other Windows applications, allowing Windows applications that make use of COM methods to be integrated in with VIVA FPGA designs.

VIVA provides its own proprietary synthesis tool for designs that will run on user FPGAs. The target of the synthesis tool is defined by a system description. VIVA includes system descriptions for the Pentium x86, which allows simulation of the design in an x86, and for specific PEs in the FPGA array. VIVA's system descriptions allow VIVA to abstract out the target hardware description from the design to be implemented, allowing VIVA to target hardware other than its own.

If the design is targeted for a PE, the synthesis EDIF output is fed into Xilinx place and route tools to produce a final binary, which VIVA then loads onto the Intel processors and the FPGA array. After initial execution, the VIVA binary can be saved

in a file format that can be loaded using VIVA command-line tools or within the VIVA development environment.

4.3 Designing within VIVA

VIVA defines basic data types, from which objects can be built. The basic types are Bit, Variant, Vector, NULL, LSB, MSB and BIN. Bit is the only non-abstract fundamental data set. Within VIVA, data sets are basically recursive combinations of Bits. Each type therefore has two basic objects associated with it: an Exposer and a Collector. A data set Exposer takes the data set and outputs its two children data sets. A collector does the opposite: it takes two child data sets and combines them into one parent data set. For instance, a BIN016 (16-bit binary) data set, when exposed, will produce two BIN008 (8-bit binary) data sets. A MSB016, when Exposed, will produce a MSB001 and a MSB015. This is equivalent to splitting out the most-significant bit of a std_logic_vector(15 downto 0), and returning a std_logic and a std_logic_vector(14 downto 0).

The above definition of types allows for recursion of large data sets into small data sets, which is useful when a large operation is comprised of smaller operations of the same type. For instance, a 16-bit AND gate can be built using a tree of 2-bit AND gates, which can be built recursively. In order for recursion of an operation to be properly implemented, two objects for the operation must be created: a base object of the smallest acceptable operand size, and a recursive version of the object, which tells VIVA how to recurse down to the base operand object. Operands can be overloaded by simply copying the operand object design and changing the input data types, effectively overloading the operand type to handle different data sets. Objects can incorporate other types of objects, which allows for the hierarchical building of basic objects.

VIVA provides basic operands such as AND, OR, and INVERT, which are standard to VIVA. These basic objects can be implemented in multiple system descriptions. VIVA objects, in general, can be designed to be implemented in multiple system descriptions or in a specific system description only. VIVA provides a library of additional operands, and data types built from the standard data types. This library, Corelib, has objects ranging from registers through state machines used for design control to file I/O objects for data transfer from and to a Windows file (see Fig. 4). This library is meant to be the basis for designs built in VIVA.

4.4 Debugging within VIVA

Debugging a design within VIVA is performed by loading the x86 system description into a design simulate design components within the x86 environment. There is no in-place hardware debugger, although one is planned for future release. The simplest method of input and output testing is displaying inputs and outputs using widgets, which can be displayed using scrollbars, textboxes, graphics, and a number of other selections, as in Fig. 4.

The clock can be manually single-stepped or continuously run. Windows COM objects can be integrated in to provide advanced debugging of the data output of a design.

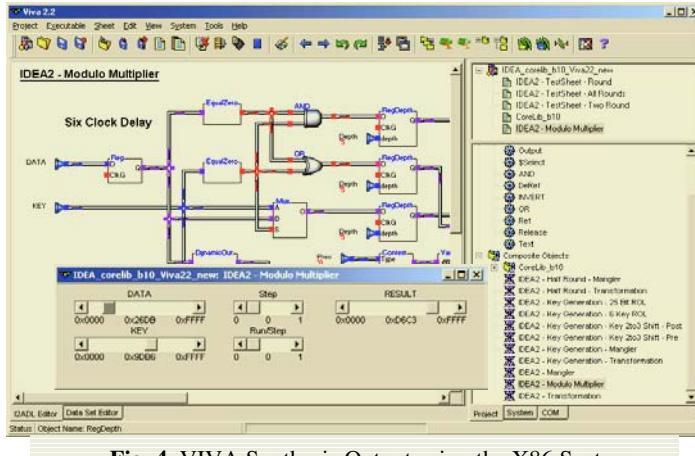


Fig. 4. VIVA Synthesis Output using the X86 System

5 IDEA

5.1 The IDEA Algorithm

The International Data Encryption Algorithm (IDEA) is a symmetric block cipher developed by Xuejia Lai and James Massey of the Swiss Federal Institute of Technology and published in 1990[5]. At that time it was suggested as a candidate to replace DES, however its widest adoption has been in PGP, which has insured widespread use of the algorithm.

IDEA uses a 128-bit key to encrypt data blocks of 64 bits. IDEA consists of eight rounds followed by a Transformation half-round that provides a 64-bit encrypted output. A round consists of a Transformation half-round and a Mangler half-round, of which an excellent description is provided by William Stallings in the book "Cryptography and Network Security: Principles and Practice" [6]. IDEA makes use of three basic operations to carry out encryption and decryption: a 16-bit XOR operation, a 16-bit modulo addition operation ($\text{mod } 2^{16}$), and a 16-bit modulo multiplication operation ($\text{mod } 2^{16} + 1$). In addition, an all-zero operand input to a modulo multiplier within IDEA equals 2^{16} for internal calculations.

Each round requires six keys: four for the Transformation half-round and two for the Mangler half-round. For the total 8.5 rounds required for IDEA encryption, 52 16-bit keys are required. The first eight keys are provided by the input key. Each additional set of eight keys is generated by performing a circular left shift of 25 bits of the previous eight key set.

5.2 A Common Design Choice for IDEA Implementation within SRC and Star Bridge

The XOR and modulo addition operations of IDEA can be easily implemented. The most difficult operation, multiplication mod ($2^{16} + 1$) with the input of 0 = 2^{16} , can be

broken down into three cases: multiplication of two nonzero inputs, multiplication where one input is zero and multiplication where both inputs are zero. For multiplication of two nonzero numbers, the following rule is used [6]:

$$\begin{aligned} ab \bmod (2^n + 1) &= (ab \bmod 2^n) - (ab \div 2^n) && \text{if } (ab \bmod 2^n) \geq (ab \div 2^n) \\ ab \bmod (2^n + 1) &= (ab \bmod 2^n) - (ab \div 2^n) + 2^n + 1 && \text{if } (ab \bmod 2^n) \leq (ab \div 2^n) \end{aligned}$$

For the remaining two cases, zero tests are used to mux in appropriate nonzero results and constants to provide the correct answer. To minimize latency, the use of a Virtex II hardware multiplier was desired to implement the two equations above. Since a 16x16 multiply operation using a single Virtex II 6000 block multiplier requires over 10 ns, a two-level three pipeline “divide-and-conquer” strategy was used to meet the 10 ns timing constraint. This method requires four 8x8 multiplies along with column additions [7].

Design choices for IDEA centered on making each half-round modular to create a repetitive instantiation, therefore key scheduling is broken into a unit that can be implemented in a modular round. The solution for key scheduling requires the generation of six keys for each round. After eight keys have been consumed, though, a 25-bit rotate left operation is required. To accommodate the above two constraints, a unique constant is input to each round’s key scheduler, which selects a mux that determines where the key rotate operation occurs within that round, if required.

The design was pipelined in order to achieve high throughput. Both data and key scheduling were pipelined, which allows this core to be used in IDEA breaking or encryption since different data blocks can be introduced with either the same or a different key at each clock cycle. Pipeline placement was chosen based on synthesized VHDL and Xilinx place and route results for a target Xilinx II 6000 FPGA and a timing constraint of 10 ns (100 MHz). The final design has a pipeline latency of 116 clocks: each Transformation half-round requires four clock cycles and each Mangler half-round requires ten clock cycles. Fig. 5 shows a block diagram of the design.

6 IDEA within SRC

6.1 Design Implementation within SRC

To implement an algorithm within SRC, at least two source files are required. *main.c* is responsible for reading in data and calling a user-defined function that loads a user-logic bitstream into the MAP processor. The second source *IDEA_test.mc* implements the MAP function that is called from *main.c*, and describes a bitstream to be loaded into the MAP processor. This MAP function calls SRC functions that control the data transfer between the Intel platform and the MAP processor, and uses a C for-loop structure for passing data to C commands that implement data processing within the MAP processor and to user HDL macros (see Table 2) that are also loaded within the MAP. In addition, an “info” file is required to specify attributes of the IDEA VHDL instantiation for the MAP compiler.

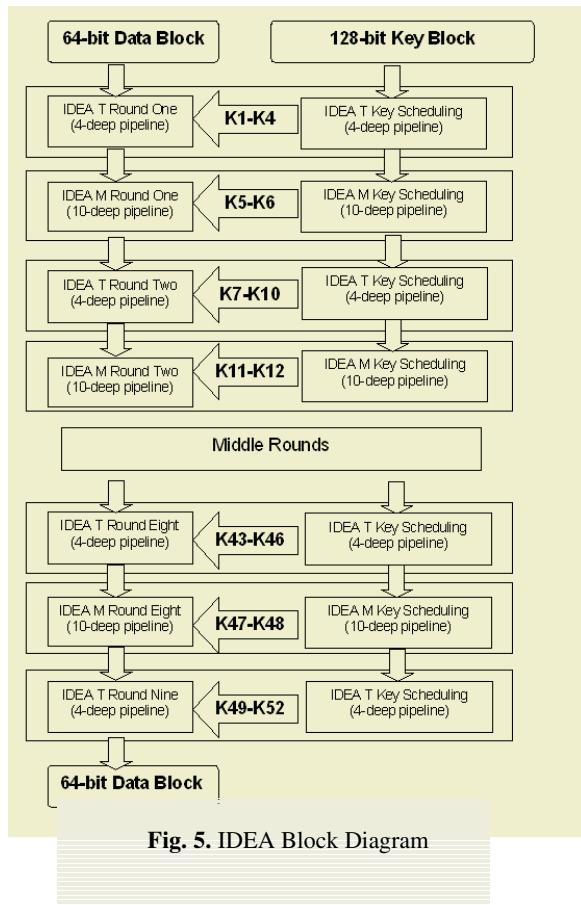


Fig. 5. IDEA Block Diagram

Two test cases were implemented within SRC, as shown in Tables 1 and 2. The first test case instantiates the whole IDEA algorithm within VHDL, and uses the MAP C function to pass data. The SRC info file for case one defines the VHDL user macro as a functional pipelined macro with a 116-clock latency. The second case instantiates half-rounds within VHDL, and uses the MAP C function to instantiate the 8.5 rounds required by IDEA. The SRC info file for case two describes two VHDL macros, both being defined as functional pipelined macros with a 4 or 10-clock latency. These test cases are representative of SRC design options, and allow for an FPGA resource and timing comparison between the two methods.

To test for latency results, a Unix high-resolution time structure was used to time of the HLL function call within *main.c*, while SRC timer calls were used within *IDEA_test.mc* to time the input and output DMA data transfers and the time to process the data. HLL time reads are in *us*, while timer reads within SRC provide the current clock tick. A conversion to ms was performed to arrive at the final timing calculations.

Table 1. Portions of IDEA Rounds Instantiated using VHDL

```
-- Round 8.
signal outkeyhigh_half_round8,    outkeylow_half_round8,
       outblockdata_half_round8, outkeyhigh_round8,
       outkeylow_round8,         outblockdata_round8:
                           std_logic_vector(63 downto 0);

begin
  -- Round 1.
  round1_T: component IDEARoundT
  port map (clk => clk, inkeyhigh => inkeyhigh, inkeylow => inkeylow,
            mux_sel => X"00000000", inblockdata => inblockdata,
            outkeyhigh => outkeyhigh_half_round1,
            outkeylow => outkeylow_half_round1,
            outhalfblockdata => outblockdata_half_round1);

round1_M: component IDEARoundM
port map (clk => clk, inkeyhigh => outkeyhigh_half_round1,
          inkeylow => outkeylow_half_round1, mux_sel => X"00000000",
          inhalfblockdata => outblockdata_half_round1,
          outkeyhigh => outkeyhigh_round1,
          outkeylow => outkeylow_round1,
          outblockdata => outblockdata_round1);
```

Table 2. Portions of IDEA Rounds Instantiated using C

```
/* Round 8 */
uint64_t outkeyhigh_half_round8, outkeylow_half_round8,
         outblockdata_half_round8, outkeyhigh_round8,
         outkeylow_round8, outblockdata_round8;
.....
int i;
.....
cm2obm_c(c, data_in, size*8);
wait_server_c();
read_timer(start_loop);

for (i=0; i<size; i++) {
  keyhigh = a[i];
  keylow = b[i];
  data = c[i];

  /* Round 1 */
  IDEARoundT(keyhigh, keylow, (uint32_t)0x0, data,
             &outkeyhigh_half_round1, &outkeylow_half_round1,
             &outblockdata_half_round1);
  IDEARoundM(outkeyhigh_half_round1, outkeylow_half_round1,
            (uint32_t)0x0, outblockdata_half_round1,
            &outkeyhigh_round1, &outkeylow_round1,
            &outblockdata_round1);
```

6.2 Results of SRC Timing

Results of the SRC implementation of IDEA are summarized in Tables 3 and 4. The timing results were independent of which of the two SRC implementations of IDEA described in Section 6.2 was used for the measurements.

Table 3. FPGA Processing and Configuration Times

Data Size	PC End-to-End Time (ms)	MAP FPGA Conf Time (ms)	MAP Total Processing Time (ms)	MAP Data Transfer In (ms)	MAP FPGA Processing (ms)	MAP Data Transfer Out (ms)
5 MB	185.6	102.0	83.6	53.0	6.6	24.0
10 MB	269.7	102.0	167.7	104.4	13.1	50.1
15 MB	350.8	102.2	248.7	153.5	19.7	75.4
20 MB	432.6	101.8	330.8	203.0	26.2	101.5

Table 4. FPGA Throughput Results

Data Size	MAP Total Processing Throughput (MB/s)	MAP Transfer In Throughput (MB/s)	MAP Transfer Out Throughput (MB/s)	MAP FPGA Processing Throughput (MB/s)
5 MB	59.8	282.9	207.9	761.1
10 MB	29.8	287.3	199.4	762.0
15 MB	20.1	293.1	198.8	762.3
20 MB	15.1	295.6	196.9	762.5

In Table 3, the execution times are provided for four input data sizes ranging from 5 MB to 20 MB. In each case, the encryption was accomplished by ten calls to the MAP function with a proportional amount of input data to the total data processed. An end-to-end time includes a single MAP configuration time in the range of 102 ms, and the total MAP processing time, which is proportional to the amount of data being encrypted. Since the reconfiguration of the MAP FPGA can be performed before any input data becomes available for processing, this configuration may be treated as a part of a one-time set up routine.

It is worth noting that the time spent for processing data inside of the MAP FPGA (MAP FPGA Processing Time) constitutes less than 8% of the Total MAP Processing time. Instead, the majority of the time is spent for transferring data to and from the MAP using the DMA transfer between the microprocessor's Computer Memory and the MAP's On-Board Memory. The MAP Transfer In Time is greater than the MAP Transfer Out Time because in our implementation of IDEA, each input consists of both a data block (64 bits) and the corresponding key (128 bits), while an output includes only an encrypted data block (64 bits).

In Table 4, the corresponding MAP Processing and Data Throughputs are calculated. All throughputs are for the most part independent of the amount of data being processed. The MAP FPGA Processing Throughput approaches the theoretical maximum of 64 bits per clock cycle = 800 MB/s. The only reasons for a slightly smaller value of this throughput are the latency of the pipelined architecture of IDEA (116 clock cycles) and a control overhead associated with implementing a loop structure within the MAP function, measured to be equal to 47 clock cycles [13]. The MAP FPGA Processing time expressed in the number of clock cycles is equal to:

$$\begin{aligned} & \text{Encryption Unit Latency} + (\text{Number of Data Blocks Processed} - 1) \\ & + \text{Loop Control Overhead} \end{aligned} \quad (1)$$

The MAP Transfer In Throughput is greater than the MAP Transfer Out Throughput because of the larger number of 64-bit words being processed without changing the OBM address. Finally, the Total MAP Processing Throughput (not including reconfiguration) is in the range of 60 MB/s, which is only 7.5% of the theoretical maximum of 800 MB/s. The limited data transfer bandwidth and a lack of overlapping between data transfers and data computations inside of the MAP FPGA contribute to this considerable slow down.

Table 5. SRC PAR Results

Test Case	Slices	Slice FFs	LUTs	Mult 18x18	Clock Period
VHDL Only	9006 (26%)	13221 (19%)	10111 (15%)	136 (94%)	11.579 ns
VHDL-C	11442 (33%)	18460 (27%)	10435 (15%)	136 (94%)	11.379 ns

6.3 Results of SRC FPGA Timing and Resource Utilization

SRC v1.4 compilers were used for MAP compilation. Synopsys v7.2 was used for synthesis, and Xilinx v5.2 was used for map, place and route. For the VHDL-only case, synthesis times averaged around two minutes, map times averaged around three minutes, while place and route (PAR) times averaged approximately 66 minutes. For the VHDL-C case, synthesis times averaged around two minutes, map times averaged around four minutes, while place and route (PAR) times averaged approximately 94 minutes. Table 5 shows Xilinx PAR results for both the SRC VHDL and VHDL-C instantiation cases. Using a VHDL-C combination to instantiate the higher level hierarchy of IDEA shows an 8% increase in the number of FPGA slice flip-flops used. This is due to SRC control logic that is inserted in its instantiation of the IDEA rounds. Clock timing differences were negligible.

7 IDEA within VIVA

VIVA was used to build up a design that matches the VHDL implementation. Fig. 6 shows portions of the Mangler half-round, and Fig. 7 shows portions of an IDEA test sheet. The VIVA PE system descriptions are not complete; a limitation of this is that the full I/O capabilities of the FPGA were not available. As a result, a limitation in this design is that the key inputs are constant, as the minimum number of inputs required is 192 bits (64 data + 128 key), while only 128 bits are available for inputs in the system description. Functional verification was performed using the same IDEA data file as in SRC.

Measuring of timing results of data transfer or data processing on the Intel side requires using a COM wrapper around a Windows timer, as a standard VIVA timer object is not available within the current library. A COM object was developed for timing; however, the CoreLib library's beta state of development prevented its integration with CoreLib's I/O objects.

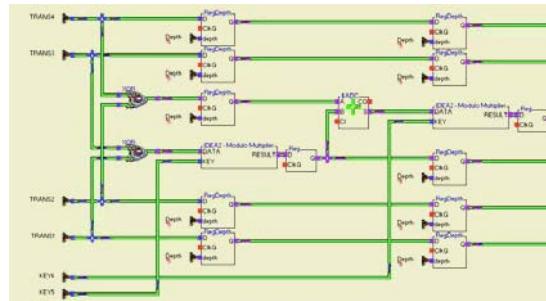


Fig. 6. VIVA Half Round Mangler Design

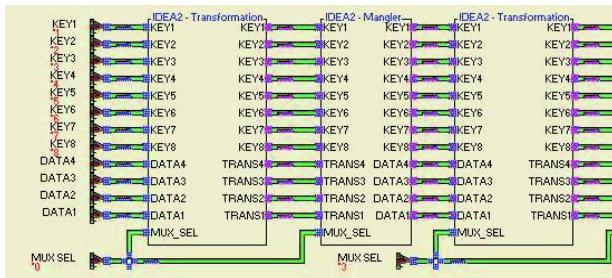


Fig. 7. IDEA in VIVA

Table 6. Star Bridge PAR Results

Test Case	Slices	Slice FFs	LUTs	Mult 18x18	Clock Period
VIVA	17292 (51%)	16118 (23%)	7639 (11%)	136 (94%)	12.191 ns

7.1 Results of Star Bridge FPGA Timing and Resource Utilization

VIVA v2.2 was used for synthesis, and Xilinx v4.2 was used for map and PAR. VIVA synthesis and PAR times scaled exponentially depending on the number of rounds synthesized: one round took 25 minutes, five rounds took approximately 7 hours, and the full 8.5 rounds took approximately 36 hours, due to the alpha state of their synthesis algorithms. The Xilinx map time for the full round implementation was 14 minutes, and PAR for the full round implementation was 85 minutes.

Table 6 shows Xilinx PAR results. Compared to the SRC VHDL-only design, the number of Slices used was 24% greater, while the number of Slice Flip Flops used was only 3% greater. An explanation for this discrepancy is that VIVA may be using only one LUT per CLB slice. The total number of LUTs used was 4% smaller. Clock timing was slightly longer. This comparison is tentative since the key inputs are constant versus the SRC design.

8 Crypto++ Software Implementation versus SRC

Crypto++ v5 was used to implement IDEA encryption within software on SRC's Linux-based workstation, to normalize the microprocessor capability between the software and the software-FPGA implementations. 20 MB of data was processed using ten calls to the Crypto library's IDEA encryption algorithm, using the same input data as SRC and VIVA. Timing measurements for IDEA encryption processing of the data were taken using a UNIX high resolution timer.

The time required to process 20 MB of data was approximately 8 seconds, which corresponds to a throughput of 2.5 MB/s. When compared to SRC's MAP Processing Throughput for IDEA, the SRC design is 24 times faster than the software implementation. When compared to the time including the FPGA configuration time, the SRC design was 18.5 times faster. Given continuous processing of large data blocks, the throughput advantage of SRC will approach the 24-times speed increase maximum.

9 SRC and Star Bridge Limitations

Both systems have limitations, which should be noted before conclusions can be understood.

SRC's compiler can only handle certain HLL constructs within a MAP processor subroutine. It is not a limitation in this design, and SRC has been adding significant functionality with each release.

Star Bridge's synthesis technology is in an alpha state. Significant improvements have been seen with the last few product releases; however the product is still in its early development stages. Star Bridge's CoreLib library is in a beta state, and design elements within the library are not final.

Both systems plan on implementing multiple FPGA use within their next software releases. SRC and Star Bridge currently cannot make use of more than one FPGA.

10 Conclusion

Reconfigurable computing is a platform that can provide larger and more efficient computing resources for implementing computationally intensive design solutions. Both SRC and VIVA have unique reconfigurable computer environments that provide complimentary ways achieving this goal. This paper shows how the designer can use SRC's development environment to implement design objectives using different combinations of traditional HLL development with HDL development, and how Star Bridge's VIVA environment provides a new paradigm for FPGA design entry and synthesis. An IDEA encryption algorithm is implemented in both environments for comparison.

Both SRC and Star Bridge take a different approach in their design environment. The SRC system is based on traditional programming languages, builds upon known development paradigms, and can incorporate existing HLL source code for designs

targeted at FPGAs. Star Bridge's VIVA provides a new development paradigm that has the potential to provide an easier method of design entry, and can target several different architectures in addition to its own.

The combination of a PC with FPGAs provides a high-speed design alternative to designs implemented solely in PC software. This paper shows that the FPGA's maximum throughput can be significantly reduced due to time required to configure the FPGA and data transfer between PC memory and memory available to the FPGA. While the effect of reconfiguration can be reduced with large data processing, the effect of data transfer is a consistent bottleneck to maximum throughput if large amounts of data need to be transferred.

References

1. Singleterry, R., Sobiesczanski-Sobieski, J. Brown, S., "Field-Programmable Gate Array Computer in Structural Analysis: An Initial Exploration", available at <http://www.starbridgesystems.com>
2. Parnell, K., Mehta, N., "Programmable Logic Design Quick Start Handbook", available at <http://www.xilinx.com>
3. SRC Inc. Web Page, <http://www.srccomp.com>
4. Star Bridge Systems Web Page, <http://www.starbridgesystems.com>
5. Lai, X., Massey, J., "A Proposal for a New Block Encryption Standard", Proceedings, EUROCRYPT '90, 1990.
6. Stallings, W., "Cryptography and Network Security: Principles and Practice", 2nd Edition, pgs. 102-109, 128, 1999.
7. Parhami, B., "Computer Arithmetic Algorithms and Hardware Designs", pgs 191-192, 2000.
8. Dai, W., Crypto++ v. 5, <http://www.cryptopp.com>, Sep. 2002.
9. VIVA User's Guide, Version 2.2
10. The SRC-6E C Programming Environment Guide, v1.2, 2003, available from SRC Computers.
11. The SRC-6E MAP Hardware Guide, 2003, available from SRC Computers.
12. 'Reconfigurable, Inherently Parallel "Hypercomputing" ', PowerPoint presentation, 2002, available from Star Bridge Systems.
13. Fidanci, O.D., Diab, H., El-Ghazawi, T., Gaj, K., Alexandridis, N., "Implementation trade-offs of Triple DES in the SRC-6e Reconfigurable Computing Environment," 2002 MAPLD International Conference, Sep. 2002.
14. Fidanci, O.D., Poznanovic, D., Gaj, K., El-Ghazawi, T., Alexandridis, N., "Performance and Overhead in a Hybrid Reconfigurable Computer," Reconfigurable Architecture Workshop, RAW 2003, Apr. 2003.

Data Processing System with Self-reconfigurable Architecture, for Low Cost, Low Power Applications

Michael G. Lorenz, Luis Mengibar, Luis Entrena, and Raul Sánchez-Reillo

Electronic Technology Department
Universidad Carlos III de Madrid. Spain.

{lorenz, mengibar, entrena, rsreillo}@ing.uc3m.es

Abstract. In this paper, a low cost self-reconfigurable data processing system with a USB interface is presented. A single FPGA performs all processing and controls the multiple configurations without any additional elements, such as microprocessor, host computer or additional FPGAs. This architecture allows high performances at very low power consumption. In addition, a hierarchical reconfiguration system is used to support a large number of different processing tasks without the penalty in power consumption of a big local configuration memory. Due to its simplicity and low power, this data processing system is specially suitable for portable applications.

1 Introduction

There are several examples of architectures for data processing based on reconfigurable platforms. Usually, a PCI interface is found [1], or a custom system like in SPLASH-2 [2], DISC [3]. However, the cost and the complexity of this kind of systems are high because they are intended for investigation or high-end applications. More examples can be seen in [4],[5].

Most of these systems require plenty of resources, one or more FPGAs with several hundred (or millions) of gates each, a high-end personal computer or a workstation in order to implement the dynamic reconfiguration. In this scenario, it is difficult to think in a low power consumption product, since only the quiescent power dissipation in these systems can be quite important [6].

This paper proposes a completely different system, very simple, with only a commercial FPGA, an ATTEL AT40K family FPGA [7]. The control of the data processing system and the reconfiguration flow is performed by the FPGA itself; this approach eliminates the need for additional hardware, another FPGA [8],[9] CPLD ,[10],[11], or microprocessor [12],[13], to perform this task, which would increase the cost and the power dissipation of the system. On the other hand, this approach can be implemented by exploiting the partial reconfiguration capability of the FPGA [14]. It results in a powerful system for signal processing and at a very low cost, less than 50€ instead of thousand of Euros.

Besides the FPGA, the system contains only a few more elements: a non-volatile reconfiguration memory and a block of data memory. The link to the external world is made with a USB interface, easy to find in many systems. Most of the data is stored locally in the board. This makes unnecessary to send a huge amount of data through

the connection link, so that the USB interface does not slow down the data processing, because all the data processing is made inside this board. The signal processing application chosen to test the system is a simple pattern recognition system.

In the following sections, the architecture of the system, two levels of hierarchy of the reconfiguration mechanism, the simple operating system, and the example chosen to evaluate the data processing system, are described in detail. Finally, we summarize the results, in performance and power dissipation, and present the conclusions of this work.

2 Reconfigurable Architecture

In the following paragraphs the hardware of the system, the reconfigurable data memory and the operating system designed are described in detail.

2.1 Hardware Architecture

Figure 1 shows the system's architecture diagram. The system is built around a dynamic partial reconfigurable FPGA (AT40K40 from ATMEL). The rest of the system consists of a reconfiguration cache memory, a data memory, a USB interface and a programmable expansion port. The reconfiguration cache memory is a FLASH memory devoted to store FPGA configuration data. The size of the reconfiguration cache memory allows to store up to eight different configurations for the FPGA, which correspond to up to eight different data processing tasks.

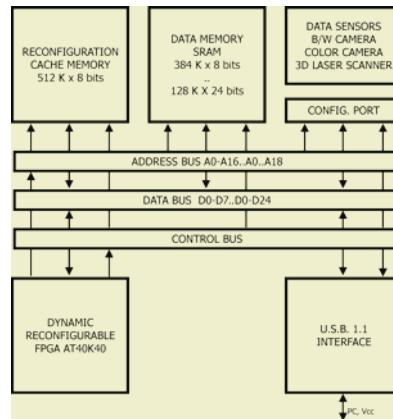


Fig. 1. Architecture of the Data Processing System with reconfigurable data memory

The data memory is a fast SRAM memory provided to allocate temporal data. It can be reconfigurable; both in data bus size and address bus width. The USB interface is intended to send data or commands to and from external equipment. In particular, it allows exchanging process data, and configuration data, with a host computer

attached externally, which can be a PC or a simple microcontroller with USB interface.

A configurable expansion port for data acquisition purposes is provided. Several different sensors can be attached to this port, such as infrared sensors, ultrasound sensors, 3D laser scanner sensors, linear cameras, digital cameras, etc...

2.1.1 Reconfiguration Mechanisms

The reconfiguration process is built in two levels of hierarchy, in a similar way to a computer cache memory system. With this approach a versatile system can be developed, with enough reconfiguration speed for critical tasks, but also with the ability to support a large number of different tasks. The first level has been named *Reconfiguration Cache* and the second level has been named *Reconfiguration Library*. The reconfiguration cache is implemented in the board, near the FPGA. The reconfiguration library is stored in the attached host computer, if necessary.

2.1.2 Reconfiguration Cache Memory

The reconfiguration data is stored in a non-volatile FLASH memory, which can allocate up to eight different tasks [15]. The first one, cache page zero, manages the reconfiguration flow, the reconfiguration cache control, and controls the data processing and the task downloads from the reconfiguration library, from the host computer. This module remains resident in the FPGA during all the time the data processing system is active. The rest of the tasks present in the reconfiguration cache, pages one to seven, are intended for different data processing purposes.



Fig. 2. Configuration Program

In order to setup the self-reconfigurable data processor for a particular data processing application, it is necessary to download the data processing tasks from the reconfiguration library. For this purpose, a simple download program (Figure 2) has been developed.

These tasks are loaded into the FPGA, in a sequential way, using a partial dynamical reconfiguration scheme. The execution order is previously determined, during the setup of the system, and implemented in the resident control task to obtain the desired data processing flow. Only if the seven processes are not enough, the

resident control task can recall new tasks from the reconfiguration library, in the same way as in a second level cache memory. These new tasks are downloaded through the USB interface into the reconfiguration cache. Only in the case the system requires more than seven different data processing tasks, it reverts to a non-autonomous system.

2.1.3 Reconfiguration Library

The reconfiguration library is stored in a database located in the host computer. The system's functionality can be increased with this database, as follows. If the seven different tasks are not enough, or it is necessary to adjust the processing sequence, the control task, permanently resident in the FPGA, requests to the PC the configuration bit-stream corresponding to the task needed, from up to 256 available. This command request is sent to the host computer through the USB interface. The control process task erases one of the seven local tasks stored in the reconfiguration cache memory, replacing it with the new process task received. Finally, the control task generates the signal to start a partial dynamical reconfiguration process for the new task. Both the data memory and control process are not altered during this reconfiguration process. The new task will be kept in the reconfiguration cache for future use and can be eventually removed to allocate other tasks. The number of processes stored in the reconfiguration library is currently limited to 256. This number is enough to store any task needed in the image processor system and it is easy to implement with an eight-bit wide command.

2.1.4 Reconfigurable Data Memory Map

To improve the performance of the system, it is possible to reconfigure the data memory map to adapt its data width to the more advantageous size in each case. Due to the dynamical reconfiguration possibilities of the FPGA, it is possible to do this *on the fly*, without disturbing the normal process of the system.

In some applications the size is implemented with only 8 bits wide. This is the case of image capture with a black and white camera. The data captured is of the same size, and the capture is easily implemented, without wasting any portions of the data memory.

In processes like convolution, correlation, filtering etc... it is necessary to get a large amount of data from the memory. This part of the process can be improved if the system is able to get more than a single data value each clock cycle. In this case the system reconfigure itself to a 24 bits data width. At this data width, it is possible to get up to three single data values at the same time, making the system almost three times faster.

In table 1 the different memory configurations provided in the systems are shown. They depend on the width of the data collected from the sensor port, (e.g. 16 bits for a 3D laser scanner or 8 bits for each primary color in a color camera), and also on the active process. For example in a convolver the data memory is arranged in 24 bits data mode. This approach permits a wider bandwidth in the data processing stage, and a better management of the data memory during the acquisition and communications stages. This allows a lower memory size, minimizing the overall power dissipation of the data processing system.

Table 1. Different Data Memory Configurations

	Data Bus	Address Bus
B/W Camera	8 bits	A0..A18
Colour Camera RGB	24 bits	A0..A16
Stereoscopic Camera	2 x 8 bits	A0..A17
3D Laser Scanner	16 bits	A0..A17
Ultrasonic Sensor	8 bits	A0..A18
Send Data to Host Comp.	8 bits	A0..A18
Convolver (Sobel 3x3)	24 bits	A0..A16

2.2 Operating System

In order to control the system a simple operating system with a small instruction set has been defined. The complete set of instructions can be seen in Table 2.

Table 2. Set of Commands defined in the operating system.

Command Definition			
Host to Processing System			
Configuration memory commands (Level 1 cache)			
Codification	Designation	Operand	More
00000001.. 00000111	Record Config Memory Block	Block Nr. 1.. Block Nr. 7	Data
00010001.. 00010111	Read Config Memory Block	Block Nr. 1.. Block Nr. 7	
00100001.. 00100111	Erase Config Memory Block	Block Nr. 1.. Block Nr. 7	
Secuence control Commands			
Codification	Designation	Operand	More
0100XXXX	Stop Process		
0101XXXX	Start Process		
0110XXXX	Next Process		
0111XXXX	Last Process		
1000XXXX	Goto first Process		
10010000.. 10010111	Goto Process Nr. (oper)	Process Nr. 0.. Process Nr. 7	
1010XXXX	Read Process		
1011XXXX	RESERVED		
Processing System to Host			
Codification	Designation	Operand	More
1100XXXX	Request Process		(0..255)
11010000.. 11011111	Send Data	Data Type 0.. Data Type 15	Data
1110XXXX	RESERVED		
Headers			
Codification	Designation	Operand	More
1111XXXX	Data		
0011XXXX	Configurations		

The first group of instructions is only intended for the setup of the system. It is intended to provide a simple way to properly configure the data processing system. With these instructions it is possible to easily change all data processing tasks, and to accomplish the different data processing operations. The configuration program shown in figure 2 uses these instructions.

The second group of instructions is intended for use during the normal operation of the data processing system. These instructions are designed to accomplish interchange of data between the system and the host computer. These data can be process data or reconfiguration data, in order to perform more than the eight different data processing tasks stored in the reconfiguration cache memory.

3 Example of the Data Processing System Operation

To test the system, the system was configured to implement a complete pattern recognition process, having several image processing steps consecutively. The first step is usually the image capture. Then, a sequence of image processing steps is performed by partially reconfiguring the FPGA for each step. Typical image processing steps include a median filter to eliminate the image's noise, image convolution with an edge detection kernel (Sobel 3x3 mask size), and image correlation (18x18 size mask) [16]. Finally, in the last step the system sends the image processing results, reduced to a 1 bit data per pixel, to the host computer. Additional post-processing or display can be performed in the host computer, if needed.

The image capture is done with a programmable digital camera, based on a CMOS sensor chip from Omnivision [17]. The image size is programmable, up to a maximum of 384 x 288 pixels, and has a B/W digital output with 256 gray levels. In this case, in order to improve the data memory usage, a lower resolution 256 x 192 pixels has been chosen. The camera has low power consumption with only 100 milliwatts during the capture stage and 500 microwatts in stand-by mode. The captured images are stored in the data memory of the system.

4 Experimental Results

Figure 4 shows a picture of the system prototype. The configurable sensor port is shown in front. Several types of data sensors can be attached to this port, by only properly configuring this expansion port. The system is able to implement a median filter or a 3x3 convolution, with the image of 256x192 pixels, at a rate of nearly seven mega pixels per second. Pattern recognition with an 18 x 18 pixel mask is made in about 135 milliseconds, including the time used to send the results through the USB interface to the host computer. All these results have been obtained with a system clock of just 10 MHz. By using SRAM with 45ns instead of the installed memory of 90 ns, it is possible to double the system clock, and the performance of the system. The image processing system is used for navigation of an autonomous robot.

Table 3. Performance of the System.(System Clock 10 MHz).

	Configuration Time ms	Processing Time ms	Total Time ms	Images per Second
Image Capture (256x192 Pixels)	0.84	3.80		
Filter (Median)	0.90	19.42		
Convolver (Sobel 3x3)	1.08	19.42		
Correlation (18x18 Mask)	2.85	84.82		
Image Send USB (1 bit Data)	0.79	4.90		
	6.46	132.36	138.82	7.20
Time Percent	4.65%	95.35%	100.00%	

The time required to completely configure the FPGA is about 6 milliseconds, with a system clock of 10 MHz. This time includes configuration of all the 2300 FPGA's cells. However, if a partial reconfiguration is made, as in this case, the time required is proportional to the number of cells to be reconfigured, at a rate of three μ secs per cell approximately. In conclusion, reconfiguration time is just a small fraction of the total processing time. These configuration times, for each task in the sample, can be seen in table 3 and table 4. The data shown in table 3, correspond to the system arranged without data memory reconfiguration. In each clock cycle, the system can get only a pixel from memory (data bus eight bits wide).

Table 4. Performance of the System with Dynamical Memory Data Reconfiguration.(System Clock 10 MHz).

	Configuration Time ms	Processing Time ms	Total Time ms	Images per Second
Image Capture (256x192 Pixels)	0.84	3.80		
Filter (Median)	1.18	7.17		
Convolver (Sobel 3x3)	1.21	7.17		
Correlation (18x18 Mask)	3.84	33.90		
Send Image USB (1bit Data)	0.79	4.90		
	7.86	56.94	64.80	15.43
Time Percent	12.13%	87.87%	100.00%	

The data shown in table 4, correspond to the system arranged with data memory reconfiguration. In each clock cycle, the system can get up to three pixels from memory (data bus twenty four bits wide). The system performance was increased by a figure of 2.7 times, but with a higher overhead in total power dissipation, design size, and reconfiguration time. However the overall power dissipation per frame processed is lowered due to a better usage of memory access; 28 mW per frame without any reconfiguration of the data memory and 22 mW per frame processed with data memory reconfiguration.

Figure 3 shows the supply current to the self-reconfigurable data processor system during the dynamic reconfiguration of the FPGA. Note that the current scale is set in a negative range. Measures that are more accurate have been made using the procedure discussed in [18]. With this method, it is possible to measure power consumption even in each clock cycle.

The actual mechanism of FPGA reconfiguration is as follows. The process begins with the configuration of the look up tables, multiplexers and the flip-flops inside each individual cell. In a second stage all the interconnections paths are configured. From Figure 3 we can see that most of the energy is wasted in this second stage.

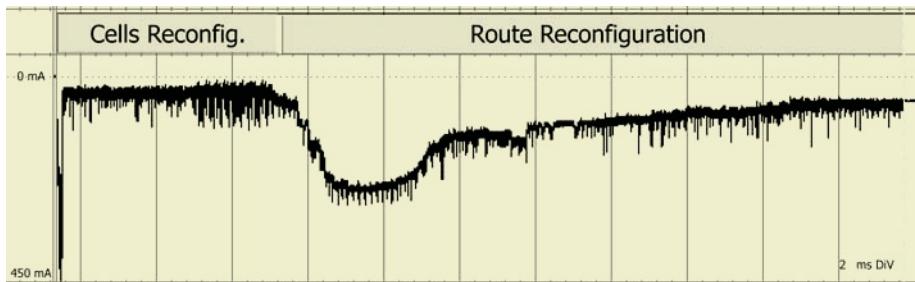


Fig. 3. Power consumption during dynamic partial reconfiguration (Atmel AT40K40 FPGA)

In Table 5 and 6 the measurements of system's power dissipation are provided. The data collected in Table 5 correspond to the system without any data memory reconfiguration, and a data memory width of eight bits. The last column represents the average power dissipation of the self-reconfigurable processing system.

Table 5. Power Consumption of the System.

	Reconfiguration Power μ Ws	Operation Power μ Ws	Average Power mW
Image Capture	33	53	
Filter (Median)	75	1457	
Convolver (Sobel 3x3)	111	1942	
Correlation (18x18 Mask)	972	23326	
Send Image USB (1bit Data)	36	59	
	1226	26836	202
Percent Consumption	4.37%	95.63%	

The data shown in Table 6 correspond to the system endowed with data memory reconfiguration. An increment in power dissipation during the configuration phase can be observed, that is due to the necessary pipelining of the data.

From this point of view, the system can be used for many data processing applications, despite its simplicity. It is remarkable that the system due to its low power consumption can be powered through the USB interface, making unnecessary any additional power supply.

Table 6. Power Consumption of the System with Data Memory Reconfiguration.

	Consumption Reconfiguration μ Ws	Consumption Operation μ Ws	Average Power mW
Image Capture	33	53	
Filter (Median)	98	1040	
Convolver (Sobel 3x3)	125	1470	
Correlation (18x18 Mask)	1309	18137	
Send Image USB (1bit Data)	36	59	
	1600	20758	345
Percent Consumption	7.16%	92.84%	

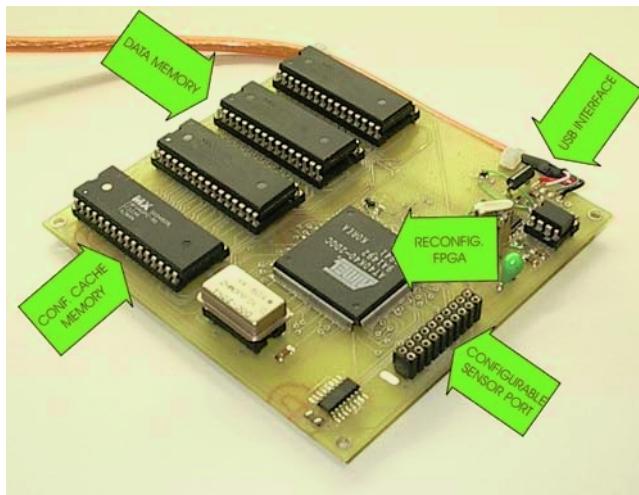


Fig. 4. Prototype Board.

5 Conclusions

In this paper we have described a low power self-reconfigurable data processing system that is able to make high speed data processing using dynamic reconfigurations to improve the system's versatility. The system is built with only one FPGA, some memory modules and a USB interface. Despite of the simplicity of the system, a large variety of processing tasks can be supported thanks to the use of a hierarchical reconfiguration scheme. The reconfiguration memory is used as a cache of the reconfiguration library, allowing the FPGA to be reconfigured for a large number of tasks while keeping the average reconfiguration time to a small fraction of the processing time.

The control of the data processing system and the reconfiguration flow is performed by the FPGA itself. This approach eliminates the need for additional hardware, which would increase the cost and the power requirements of the system. The system can be equipped with larger memories, enough for the 256 tasks supported, but at the expense of a higher cost and higher power consumption. The system has been evaluated with a classical pattern recognition but it can be used for many other digital signal processing applications, by simply reassigning the configurable expansion port to other types of sensors or interfaces.

References

- [1]. F.Lisa. F.Cuadrado. D. Rexachs. J. Carrabina, "A Reconfigurable Coprocessor for a PCI-Based Real Time Computer Vision System", Field-Programmable Logic and Applications, 1997, 392-399.
- [2]. D. Buell, J. Arnold et al. "SPLASH-2" Proceedings of the 4th Annual ACM Symposium on parallel Algorithms and architectures, June 92, 316-324.

- [3]. M.J.Wirthling, B.L. Hutchings. "A Dynamic Instruction Set Computer". IEEE Workshop on FPGAs for Custom Computing Machines. Napa, CA, April 95. 99-107.
- [4]. Hartenstein, R.; "A decade of reconfigurable computing: a visionary retrospective" Design, Automation and Test in Europe, 2001. 642 -649
- [5]. K. Compton, S. Hauck, "Reconfigurable Computing A Survey of System and Software", ACM computing surveys, Vol 34, No 2, June 2002. 171-210
- [6]. L. Benini, G. de Micheli, E. Macii, "Designing Low-Power Circuits: Practical Recipes", IEEE Circuits and Systems Magazine, Vol 1(1) First Quarter 2001, 6-25.
- [7]. "Configurable Logic:Design and Application Book" Atmel, San Jose, CA. 1998
- [8]. Haenni J.O., Beuchat J.L. and E Sanchez. "RENCO: A reconfigurable Network Computer". Sixth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'98) April 21-23, 1998 Napa California
- [9]. M.A.Aguirre, J. Tombs. A. Torralba and L.G. Franquelo, "Experience on VLSI for Digital Signal Processing Using Adavnced FPGAs: The Unshades Framework", 4th European Workshop on Microelectronics Education, Vigo May 2002, 193-196.
- [10]. C. Carmichael."Configuring Virtex FPGAs from Parallel EPROMs with a CPLD" Xilinx Application Note 137, 1999.
- [11]. J. Khan, J. Handa and R. Vemuri, "iPACE-V1 A portable adaptative computing engine for real time applications", Field-Programmable Logic and Applications , 2002, 69-78..
- [12]. "AT94K Field programmable system level Integrated circuit FPSSLIC Data sheet" Atmel, San Jose, CA. 2002
- [13]. J. Faura, J.M. Moreno et al, "Multicontext Dynamic reconfiguration and real time probing on a novel mixed signal programmable device with on chip microprocessor", Proceedings 7th International Workshop FPL97 London Sept 97. 1-10
- [14]. Lysaght P., Dunlop J.; "Dynamic reconfiguration of FPGAs" European FPL'93 Oxford, 1993. 82-94
- [15]. Huesung Kim; Somani, A.K.; Tyagi, A. "A reconfigurable multifunction computing cache architecture" Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , Volume: 9 Issue: 4, Aug. 2001, 509-523
- [16]. Derbyshire A. And Luk W.:Combining Serialisation and Reconfiguration for Convolver designs Eighth Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'00) April 17-19, 2000 Napa, California
- [17]. "OV5510 Data sheet". Omnivision Technologies Inc. July 1998
- [18]. L. Mengibar, M.G. Lorenz. et al, "Experiments in FPGA characterization for low power design", XIV Design of Circuits and Integrated Systems Conference, Mallorca Nov 1999. 385-390.

Low Power Coarse-Grained Reconfigurable Instruction Set Processor

Francisco Barat¹, Murali Jayapala¹, Tom Vander Aa¹, Rudy Lauwereins^{1,3},
Geert Deconinck¹, and Henk Corporaal²

¹ ESAT K.U.Leuven, Kasteelpark Arenberg 10, B-3001 Leuven–Heverlee, Belgium,
`firstname.lastname@esat.kuleuven.ac.be`

² TUEindhoven, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

³ Imec, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract. Current embedded multimedia applications have stringent time and power constraints. Coarse-grained reconfigurable processors have been shown to achieve the required performance. However, there is not much research regarding the power consumption of such processors. In this paper, we present a novel coarse-grained reconfigurable processor and study its power consumption using a power model derived from Wattch. Several processor configurations are evaluated using a set of multimedia applications. Results show that the presented coarse-grained processor can achieve on average 2.5x the performance of a RISC processor with an 18% increase in energy consumption.

1 Introduction

Current and future multimedia applications such as 3D rendering, video compression or object recognition are characterized by computationally intensive algorithms with deeply nested loop structures and hard real time constraints. Implementing this type of applications in embedded systems (e.g. multimedia terminals, digital assistants or cellular phones) leads to a power-optimizing problem with time and area constraints. By adequate use of the high parallelism available in the inner loops of these applications, it has been shown that reconfigurable instruction set processors, composed of a standard instruction set processor tightly coupled to some sort of reconfigurable logic, have the required computational power to fulfill the time constraints [4, 6, 7]. What has not been shown is whether some of these reconfigurable processors can also meet the power consumption requirements of these applications.

This paper presents CRISP, a coarse-grained reconfigurable instruction set processor designed for multimedia applications that can accelerate multimedia applications in a power efficient manner. The power of this architecture lies in the reconfigurable logic, which is composed of complex blocks such as ALUs or multipliers, that operate on the data sizes typically found in multimedia applications (8 to 32 bits), and is divided in independently enabled slices in order to reduce overall energy consumption and reconfiguration times. Also important is the tight coupling to the main microprocessor (the reconfigurable logic is seen

as an extra functional unit) that allows quick control and data communication between the processor and the reconfigurable logic.

The proposed processor architecture has been evaluated using a variation of the Wattch power estimation framework [3]. Several CRISP processors with different amounts of reconfigurable hardware are compared to a RISC processor. Results on a set of multimedia applications show that the reconfigurable processor is able to achieve on average 2.5 times the performance of a RISC processor with just an average of 18% energy increase.

This paper is organized as follows. The proposed CRISP architecture and the associated compilation/synthesis techniques are discussed in sections 2 and 3. Experimental setup and results are presented and discussed in sections 4 and 5, respectively. Related work is discussed in section 6 and the paper is closed with conclusions and future work.

2 A Low Power Reconfigurable Architecture

CRISP (Coarse-grained Reconfigurable Instruction Set Processor) is an instruction set processor composed of a main processor core tightly coupled to some coarse-grained reconfigurable logic. The coarse-grained reconfigurable logic is placed in a reconfigurable functional unit (RFU), and just like any other functional unit, an operation can be issued to it every clock cycle. The RFU reads/writes data from/to the main register file. The main processor can be any type of processor, though in this paper we will assume the processor is a simple RISC (Reduced Instruction Set Computer) processor.

Figure 1 presents the overall architecture of the complete processor. The main processor core reads its instructions from the level 1 instruction cache and obtains data via the level 1 data cache. Both caches are connected to a unified level 2 cache, which is in turn connected to an external memory. The reconfigurable fabric, in the center of the figure and directly controlled by the main processor core, contains configuration memory that is loaded via the unified level 2 cache (this allows reuse of configurations loaded from external memory and reduces reconfiguration times). The reconfigurable fabric can directly access the data cache via several data ports. Additionally, the reconfigurable logic communicates with the main processor core via a functional unit interface.

As shown in figure 1, the reconfigurable functional unit is divided in reconfigurable slices, one of which is shown with more detail on figure 2. Each slice contains several coarse-grained processing elements (PEs), a register file, interconnect, and a small configuration memory. Each processing element is either an ALU, shifter, multiplier or memory unit. Such complex processing elements are better suited than the traditional logic blocks based on look up tables (LUTs) for the execution of the operations typically found in multimedia applications, which are word-oriented and not bit-oriented. These complex PEs allow the reconfigurable logic to operate at higher frequencies with lower power consumption when compared to traditional FPGAs.

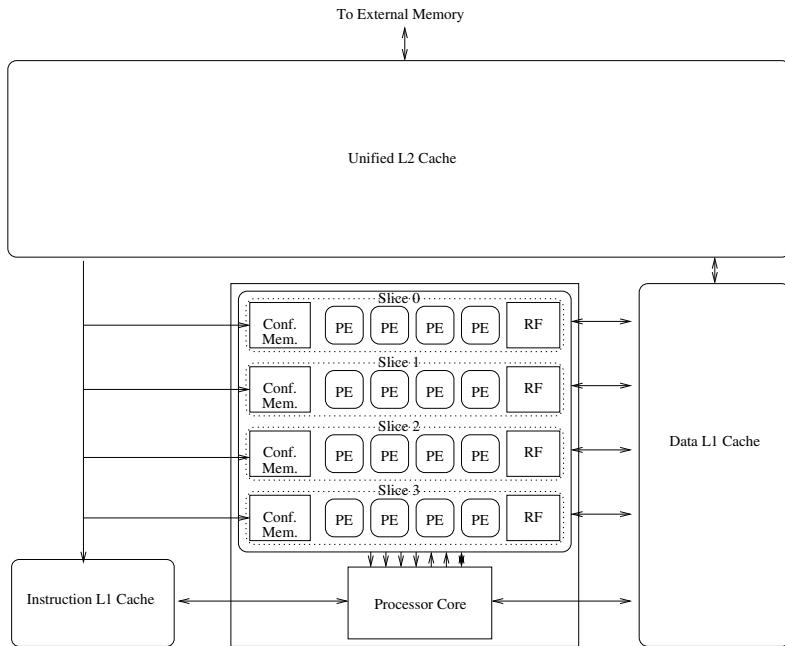


Fig. 1. Example CRISP instance (RFU: Reconfigurable Functional Unit, PE: Processing Element, FU: Functional Unit)

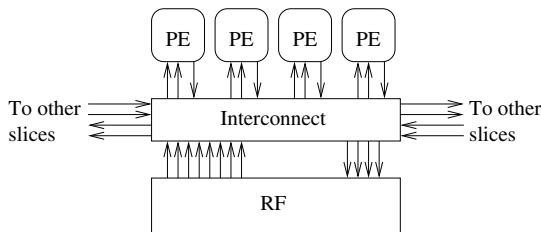


Fig. 2. Internals of a reconfigurable slice

The processing elements inside a slice are connected together through programmable interconnect. This interconnect is a full crossbar that operates on words and has the same complexity as the bypass network typically found in current VLIW (Very Long Instruction Word) microprocessors. This crossbar can connect the output of any processing element to the input of any other processing element. It also connects the processing elements to the register file and to the other slices. In most cases, each processing element writes its output to the register file of the slice, but this behavior can be optionally bypassed, just like in traditional FPGAs, and the result routed to a different processing element. By combining this optional register write and the interconnect crossbar,

it is possible to perform spatial computation such that elements in a data flow chain are connected together through the crossbar. The processing element at the end of the chain is connected to the register file.

The number of processing elements has to be kept small in order to reduce the complexity of the interconnect and register files. From the results of our simulations, four processing elements represent a good tradeoff between power consumption and performance (though for space reasons these results are not presented here).

Each reconfigurable slice also contains a configuration memory. This configuration memory stores the configuration for the slice's datapath components. Since the typical loop requires several configurations to be quickly alternated (as will be discussed in section 3), the configuration memory must be multi-contexted, (i.e. it must be able to store several configurations). Switching from one context (or configuration) to another takes just one clock cycle and is equivalent to reading from a shallow and wide memory. In the case of a slice with four processing elements, the width of this configuration memory is around 128 bits, much less than the bits required for a slice of a typical FPGA. The number of configurations in the configuration memory typically ranges between 8 and 32 contexts. Ideally, the number of contexts should be kept as small as possible to reduce the energy consumption of the configuration memory.

The reconfigurable functional unit is activated via a special reconfigurable instruction as shown in Figure 3. This reconfigurable instruction contains two main pieces of information. First, it contains a reconfigurable instruction identifier (RID) that specifies which of the many available configurations must be used. This identifier can select among a larger number of configurations than the number of contexts available in the configuration memory. If the required configuration is not currently loaded in the configuration memory, which behaves like a small cache indexed by this RID, the system is halted and the adequate configuration is loaded from the unified level 2 cache.

Aside from the RID, the reconfigurable instruction includes several fields of one bit length that specify which slices are going to be activated. Figure 3 shows these slice enable fields (named EN_x in the figure). For those parts of the application that require a small number of processing elements, only the first slice will be activated. For those parts with higher parallelism requirements, more slices will be used. This mechanism, which can be considered as a form of partial reconfiguration, reduces the size of the configuration stream that must be loaded in the case of a configuration miss. Additionally, the slices of the reconfigurable datapath that are not required can be switched off, thus providing an effective form of energy consumption control.

3 Compilation Techniques

Code generation for any reconfigurable instruction set processor involves main two steps: synthesis of the different configurations for the reconfigurable array and generation of the code for the main processor (not mapped to the recon-

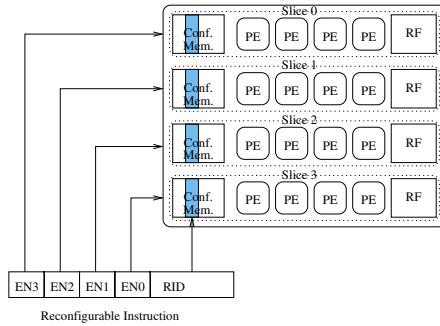


Fig. 3. Fields of a reconfigurable instruction. EN: slice enable bit, RID: reconfigurable instruction identifier

figurable array). In the case of CRISP, with processing elements of complexity similar to standard functional units, existing VLIW techniques have been reused.

On our research compiler (based on Trimaran [1]), code generation for loops is based on software pipelining. In software pipelining, iterations are initiated at regular intervals and execute simultaneously but in different stages of the computation. This allows mapping the available parallelism onto the number of resources of CRISP. With this technique [2], the code generated for a loop will contain as many configurations as the initiation interval of the loop (the number of cycles of the loop kernel). It is therefore important to check that an iteration does not last more than the number of available contexts in the configuration cache. If this was not the case, the generated code would need constant reconfiguration and performance would fall down.

Software pipelining can also be modified to exploit the ability to perform spatial computation by chaining operations [2]. This allows a reduction of the critical path length of inner loops, with the corresponding decrease in execution time. The process of code generation with spatial computation requires a proper model of the timing delay of the processing elements and the interconnect, since the process is similar to the place and route stage in FPGAs.

Additionally, our compiler studies for each loop the required number of slices. Only the necessary number of slices are used, in order to reduce both reconfiguration times and energy consumption.

4 Experimental Setup

In order to evaluate the performance and power consumption of the proposed architecture, we ran a set of multimedia applications on a simulated processor. The list of benchmarks can be seen on table 1. The applications were compiled using a modified version of the Trimaran [1] compiler that includes the compilation techniques described in the previous section. This compiler is able to exploit the coarse-grained reconfigurable unit present in the reconfigurable processor. Spa-

Table 1. Benchmarks

Benchmark	Application type
ADPCM decode	Audio
ADPCM encode	Audio
AES	Encryption
JPEG decode	Image
MPEG2 decode	Video

tial computation is not currently implemented in our prototype compiler and therefore was not used.

The compiled applications were simulated in a cycle accurate simulator that models the processor core, the reconfigurable array and the memory hierarchy, including all cache effects. This simulator provided a series of trace files that were later used to compute the power consumption of each application.

In order to calculate the power consumption of the reconfigurable processor, we have used Wattch [3] as a starting point for our power calculations. Wattch is a power consumption model for superscalar processors that can be customized to different architecture models. The power contribution of the different components of the processor is calculated using analytical models (for array structures like memories or register files), through empirical models (for the functional units) or via a combination of the two (for the clock distribution network).

Power consumption in a coarse-grained reconfigurable instruction set processor comes from two main sources: i) the main processor core, and ii) the reconfigurable unit. Since our main processor core is a RISC processor, it is straightforward to calculate the power consumption of the processor core using the Wattch building blocks. The power of the coarse-grained reconfigurable logic is modeled using these same Wattch blocks as shown in table 2.

Our reference processor is a RISC processor whose parameters can be seen on table 3. This RISC processor was extended with a reconfigurable functional unit with varying number of reconfigurable slices. The contents of each slice are shown in table 4. The number of slices was changed from 1 to 4.

The multibanked data memory was modeled as several independent SRAM memories. The performance and power consumption effect of the interconnect required in such a multibanked memory were not modeled.

Table 2. Mapping from reconfigurable logic components to Wattch power models

Reconfigurable logic component	Wattch power model
Register file	Register file
Processing element	Functional unit
Configuration memory	SRAM memory
Interconnect	Result bus
Clock	Clock

Table 3. RISC processor parameters

Parameter	Value
Instruction Level 1 Cache	2Kbytes, block size 16, 128 sets, direct mapped
Unified Level 2 Cache	64Kbytes, block size 32, 512 sets, 4 way set associative
External memory	SDRAM, 18 cycles latency
Registers in main processor core	32 32-bit registers
Data memory	Multi-banked SRAM 8Kbytes
Frequency	600MHz
Technology	0.35um

Table 4. Slice parameters

Parameter	Value
Processing elements	1 Multiplier, 1 ALU/shifter and 2 ALUs
Register file	32 32-bit registers
Configuration memory	32 configurations of 128 bits

5 Results

Figure 4 left shows the normalized execution time of the set of benchmarks in several configurations. RISC represents the baseline RISC processor and the other entries represent the RISC processor with the number of reconfigurable slices ranging from 1 to 4. From this graph we can see that as the number of slices is increased, the execution time drops until the curve saturates. After this saturation point adding extra slices does not improve the performance. The saturation point depends on characteristics of the benchmark. ADPCM encode saturates with just one slice, while others like ADPCM decode or mpeg2dec profit with 3 or more slices. From this figure we see that the average performance increase is 2.5 and the maximum is 5.

Figure 4 right shows the normalized energy consumption for the benchmark set. In general, the total energy consumption increases as more resources are added to the processor. The fact that the increase in energy consumption is not as fast as one might expect is derived from a better utilization of the processing elements of the processor as more slices are added. After the performance saturation point, the energy consumption is only increased by the extra load in the clock network (better clock gating would reduce this effect). The extra slices that are not using are basically shut off and hence their contribution to the energy consumption is negligible.

Figure 5 left shows the normalized energy delay product (calculated as application execution time in cycles times the application energy consumption). It can be observed from there that all reconfigurable processors have a better energy delay product than the baseline RISC. The reason for this is that the reconfigurable processors are able to exploit the parallelism in the application reducing the number of instructions that need to be executed.

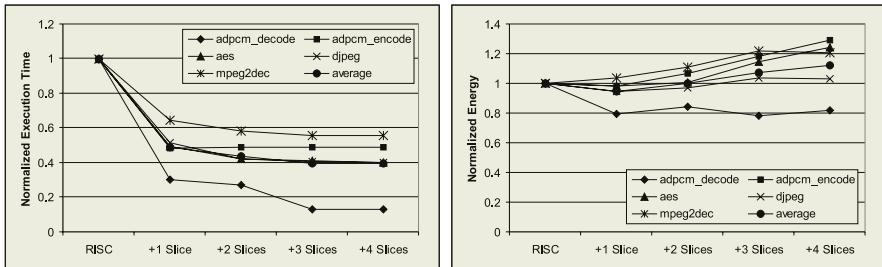


Fig. 4. Normalized execution time versus number of reconfigurable slices (left) and normalized energy versus number of reconfigurable slices (right)

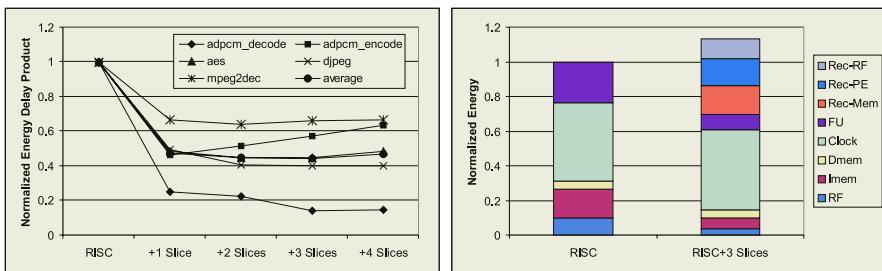


Fig. 5. Energy-delay product versus number of reconfigurable slices (left), and, normalized energy distribution for a RISC processor and a RISC processor with 3 slices for benchmark AES (right)

Figure 5 right shows the different components to the energy consumption of the RISC processor and a RISC processor with 3 slices for the benchmark AES. The benchmark AES runs at 2.5 the speed of the RISC processor and consumes only around 15% more energy consumption. We can see that a significant part of the energy consumption of the RISC processor is transferred to the reconfigurable components in the reconfigurable processor. Energy in the instruction memory decreases at an increase in the energy of the configuration memory. Energy from the functional units decreases and is transferred to the processing elements (resulting in almost the same energy consumption). In the case of the register file energy, the energy in all register files (processor core plus reconfigurable logic) is increased due to the usage of more power consuming units in the reconfigurable logic (multiported register files). Finally, the clock power is also increased, and from what can be seen from the figure, represents a significant amount of the energy consumption. Clock power reduction techniques would certainly reduce the power of both the RISC and reconfigurable processor.

As mentioned in the introduction, the proposed architecture is low power. The reasons for this are several:

- Coarse-grained datapath elements: they are much more energy efficient than bit level elements due to the small control overhead they require.
- Small and sliced configuration memory: Even though that constant switching in the configuration memory is required, the total power contribution of the configuration memory is marginal. Additionally, the fact that the configuration memory slices can be selectively enabled contributes to an even smaller energy consumption.
- Sliced datapath that limits the complexity and energy consumption of the different elements.
- Intelligent compiler that is energy aware and tries to minimize the number of active slices without compromising performance

6 Related Work

Coarse-grained reconfigurable processors like [4, 6, 7] have been shown to be extremely efficient for executing multimedia and DSP applications. [6, 7] present processors with a reconfigurable fabric similar in complexity to ours. One of the main differences are the way registers are spread over the array and the way the configuration memories are controlled. None of them provide for a mechanism to selectively load a configuration to only a subset of the reconfigurable array. [4] has a concept similar to the reconfigurable slices but is applied for a different reason, namely increasing the virtual size of the hardware. Another important difference of our work with previous others is that none have studied the power implications of their architectures.

Regarding the topic of power estimation for reconfigurable architectures, there has been work on modeling FPGA power consumption, like [8]. We are currently unaware of power models for coarse-grained processors, but fortunately, models for superscalar and VLIW processors can be reused. Some of them are [3, 5, 9].

7 Conclusions

This paper has presented a coarse-grained architecture designed for low power multimedia applications. The reconfigurable logic is divided in slices that can be independently activated to reduce the power consumption of the processor. The processor achieves an average 2.5 performance increase over a standard RISC processor with just an 18% energy overhead.

Future work will study in more detail the power consumption of the processor. Also, we will study the effects of spatial computation on the performance and power consumption of the processor.

Acknowledgements

This project is partially supported by the IWT through MEDEA+ project A502 MESA and by the Fund for Scientific Research -Flanders (FWO) through the postdoctoral fellowship of G.Deconinck.

References

1. Trimaran: An infrastructure for instruction level parallelism.
<http://www.trimaran.org>, 1999.
2. Francisco Barat, Murali Jayapala, Pieter Op de Beeck, and Geert Deconinck. Software pipelining for coarse-grained reconfigurable instruction set processors. In *Proc. ASP-DAC*, pages 338–344, January 2002.
3. David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. 27th Int'l Symp. Computer Architecture (ISCA 2000)*, pages 83–94, June 2000.
4. Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: a coprocessor for streaming multimedia acceleration. In *Proc. 26th Int'l Symp. Computer Architecture (ISCA 1999)*, pages 28–39, May 1999.
5. Weiping Liao and Lei He. *Compilers and Operating Systems for Low Power*, chapter Power Modeling and Reduction of VLIW Processors. Kluwer Academic Publishers, 2002.
6. Guangming Lu, Hartej Singh, Ming-Hau Lee, Nader Bagherzadeh, Fadi J. Kur-dahi, and Eliseu M. Chaves Filho. The morphosys parallel reconfigurable system. In *European Conference on Parallel Processing*, pages 727–734, 1999.
7. Takashi Miyamori and Kunle Olukotun. REMARC: Reconfigurable multimedia array coprocessor. In *Proc. 6th Int'l Symp. Field-Programmable Gate Arrays (FPGA98)*, February 1998.
8. Kara K. W. Poon, Andy Yan, and J. E. Steven Wilton. A flexible power model for fpgas. In *Proc. 12th Int'l Workshop Field-Programmable Logic and Applications (FPL 2002)*, September 2002.
9. W. Ye, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. The design and use of simplepower: a cycle-accurate energy estimation tool. In *Proc. DAC*, pages 340–345, June 2000.

Encoded-Low Swing Technique for Ultra Low Power Interconnect

Rohini Krishnan, Jose Pineda de Gyvez, and Harry J.M. Veendrick

Philips Research Laboratories,
Prof. Holstlaan 4, 5656AA Eindhoven, The Netherlands,
rohini.krishnan@philips.com

Abstract. We present a novel encoded-low swing technique for ultra low power interconnect. Using this technique and an efficient circuit implementation, we achieve an average of 45.7% improvement in the power-delay product over the schemes utilizing low swing techniques alone, for random bit streams. Also, we obtain an average of 75.8% improvement over the schemes using low power bus encoding alone. We present extensive simulation results, including the driver and receiver circuitry, over a range of capacitive loads, for a general test interconnect circuit and also for a FPGA test interconnect circuit. Analysis of the results prove that as the capacitive load over the interconnect increases, the power-delay product for the proposed technique outperforms the techniques based on either low swing or bus encoding. We also present the signal to noise ratio (SNR) analysis using this technique for a CMOS 0.13 μ m process and prove that there is a 8.8% improvement in the worst case SNR compared to low swing techniques. This is a consequence of the reduction in the signal switching over the interconnect which leads to lower power supply noise.

1 Introduction

As process geometries continue to shrink and we enter the nanometer era, the interconnects and the drivers and receivers associated with them belong to the major energy consumers on an integrated circuit. As more complex circuits are integrated in a single chip, with global busses, clock lines and timing circuits running across the chip, the fraction of energy consumed by the interconnect is ever increasing. The fraction of energy dissipated over interconnect and clock lines was found to be 40% for gate array based designs, 50% for cell-library based designs and 90% for traditional FPGA devices [7, 8].

Methods to reduce the amount of energy consumed by the interconnect have been extensively researched in the literature. In the past, encoding techniques like work zone encoding[9], bus invert coding[3] etc. have been proposed for inter and intra chip interconnects while low swing techniques [1] have been used for intra-chip interconnects. Encoding has been mainly applied only for I/O circuits due to the presence of huge external and parasitic capacitances at I/O pads. But with the advent of reconfigurable FPGAs, even on chip interconnect (with pass

transistors in connection boxes and switch boxes), have a rather large capacitive load, warranting the use of encoding techniques on-chip. By using bus-invert encoding as proposed in [3], the average power on the bus cannot be reduced by more than 25%. Other techniques have to be used to obtain larger energy reductions. Reducing the voltage swing of the signal on the wire has been the most efficient technique for reducing the power quadratically and power-delay product linearly. However low swing techniques suffer from lower noise immunity and reduced signal to noise ratios. Encoding technique for noise reduction has been proposed in [2] but it is not energy optimal.

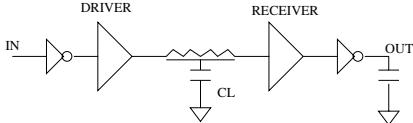
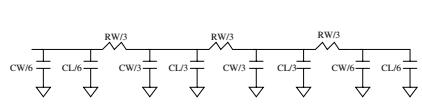
In this paper we present a novel encoded-low swing technique that combines the advantages of encoding and low swing. We present exhaustive simulation results over a benchmark test architecture (for general and for FPGA interconnect) which includes the driver and receiver circuitry. The analysis is done for a range of capacitive loads. We compare the proposed technique with existing techniques namely, “no encoding no low-swing” (full-swing CMOS) approach, “low swing” approach, and the “encoding” approach.

We examine the results for a random set of data to be transmitted over the interconnect keeping in mind that the best case energy savings for encoding is when the number of transitions over the interconnect is reduced to zero (ideal case when the data values are allowed to flip every clock cycle) and the worst case for encoding is when $N/2$ values over a N bit wide interconnect flip every cycle. We do our study for a particular low swing technique, namely Static Driver with Voltage Sense Translator, abbreviated as SDVST-II[6] and a particular encoding technique, bus-invert coding[3]. We chose to compare against these two techniques since we use the SDVST-II and bus-invert coding for generating the encoded-low swing signals as well (the same transmitters and receivers are used for generating the low-swing, encoded and encoded-low swing signals thus ensuring a fair comparison). The results are general and hold for other low swing and encoding techniques also.

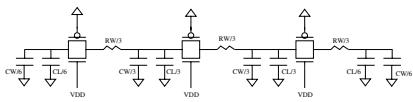
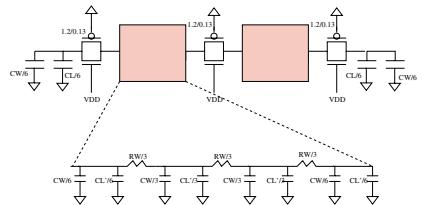
The paper is organized as follows. First, the benchmark circuits (general and FPGAs) used for all our simulations and comparisons are presented. The encoded-low swing technique and an efficient circuit implementation of the same is then presented. This is followed by a comparison with existing techniques. The effect of crosstalk and process variations over the schemes is also analysed. This is followed by a signal to noise ratio analysis. Finally the summary and the conclusions drawn are presented. Throughout this paper, we refer to the “no encoding no low-swing” approach by the acronym NENL, “encoded-low swing” approach by the acronym EL, “low swing” approach by the acronym L, the “encoding” approach by the acronym E.

2 Test Architecture

We use the test architecture of Fig. 1[1] for the purpose of evaluating the effectiveness of our approach for general interconnects. Fig. 1 shows the schematic of the benchmark interconnect circuit. An inverter prior to the driver and after

**Fig. 1.** Benchmark test architecture**Fig. 2.** General interconnect model

the receiver is added with 20fF capacitive load. Both the inverters are sized with $W_p=1.2\mu m$ and $W_n=0.6\mu m$. Fig. 2 shows the general interconnect line which is a metal-3 layer wire with a length of 1 mm modeled by a π_3 distributed RC model with an extra capacitive load, CL , distributed along the wire for fanout. We distribute CL since the exact location of the fanout load is not known, and so we cannot use a lumped capacitance. RW and CW stand for the wire resistance and wire capacitance respectively.

**Fig. 3.** FPGA interconnect model for short length**Fig. 4.** FPGA interconnect model for long length

We have developed the test architecture in Fig. 3 and Fig. 4 for FPGA interconnects. In the FPGA interconnect model, we incorporate pass transistor switches to replicate the path over which signals travel between logic blocks[5]. Note that we actually use complementary pass gates instead of NMOS pass gates to get rid of the threshold drop and simplify our analysis. Usage of complementary pass gates instead of NMOS pass gates increases the delay of the signal. But since we use the same test architecture for comparing all the techniques, the delay is equally degraded for all the schemes and our comparison remains fair. Fig. 3 shows the RC interconnect model which is used when the wire connecting the complementary pass transistor switches is short. When the wires are long, the RC model in Fig. 3 is replaced by a distributed RC model as shown in Fig. 4. In Fig. 4, CL' represents the capacitive load due to fanout between two complementary switches.

As already mentioned, we compare four cases, the NENL, EL, L and E approaches. In the NENL approach, the driver and the receiver are just buffers. In the EL approach, the driver converts the input signal into an encoded-low swing signal, which is converted back to its original value and level by the receiver. In the L approach, the driver is a low swing converter and the receiver is a level

restorer. In the E approach, the driver is an encoder and the receiver is a decoder. All circuit comparisons are based on CMOS 0.13 μm process parameters and spice models. The minimum drawn channel length for this process is set to 0.13 μm .

3 Encoded Low Swing Technique

In this section, a brief overview of the proposed technique is presented. In the proposed encoded-low swing scheme, the current values to be transmitted on the bus are compared with the previous state of the bus. When the number of bits flipping is greater than $\frac{N}{2}$ where N is the width of the bus, the decision to transmit the inverted signal values is made. In addition, an “invert” signal is also sent to the receiver to indicate whether the bus values are inverted or not. These encoded values are then converted into their low swing equivalents and transmitted. In this way, we ensure that the energy consumed over the interconnect is minimum. Our two pronged strategy not only reduces the probability of transitions over the interconnect but also transmits only low swing values to achieve tremendous energy reductions. This energy saving can only be achieved if we have an efficient driver and receiver circuit, which does not consume more energy than is saved over the interconnect. For that, an efficient circuit implementation has been developed.

We first theoretically estimate the energy savings that are possible using the proposed technique. We can estimate the average number of transitions using probabilistic analysis for a N bit wide bus. The dynamic switching energy of the bus is given by Eqn.1[1].

$$E_{dyn} = C_{average} V_{ref}^2 T \quad (1)$$

In Eqn. 1, T is the total number of transitions over the wire. Without encoding, the transitions, T_{NE} , for an average case for a N bit wide bus is

$$T_{NE} = \sum_{M=1}^{N} P(M).M \quad (2)$$

where T_{NE} denotes the number of transitions without encoding, P(M) denotes the probability that M bits flip in a N bit wide bus and is given by

$$P(M) = \frac{1}{2^N} C \left(\begin{array}{c} N \\ M \end{array} \right) = \frac{1}{2^N} \frac{N!}{(N-M)!M!}. \quad (3)$$

By using the bus-invert coding method , we compute the transitions for an average case for a N bit wide bus. We differentiate between the cases when N is odd and N is even. This is shown next.

Case a: When N is Odd. Using bus invert coding, the number of transitions is given by Eqn. 4. T_E indicates the number of transitions over the bus in the

presence of encoding. Here, when the number of bit flips exceeds $\frac{N+1}{2} - 1$, the decision to invert the data bits is made. Counting the extra transition due to the invert signal, the number of transitions over the bus, when $\frac{N+1}{2}$ data bits flip, is $N - \frac{N+1}{2} + 1 = \frac{N+1}{2}$.

$$\begin{aligned} T_E = & \frac{1}{2^N} [1C\binom{N}{1} + 2C\binom{N}{2} + \dots + \\ & + \left(\frac{N+1}{2}\right)C\binom{N}{\frac{N+1}{2}} + \left(\frac{N+1}{2} - 1\right)C\binom{N}{\frac{N+1}{2} + 1} + \\ & + \left(\frac{N+1}{2} - 2\right)C\binom{N}{\frac{N+1}{2} + 2} + \dots + 1C\binom{N}{N}] \end{aligned} \quad (4)$$

Case b: When N is Even. Here, when the number of bit flips is exactly $N/2$, there is no advantage in encoding. We then take the decision of inverting the values on the bus if it does not cause a transition over the “invert” signal itself. This means that when N is even, an extra state flip flop for storing the state of the “invert” signal is needed which is not the case when N is odd.

$$\begin{aligned} T_E = & \frac{1}{2^N} [1C\binom{N}{1} + 2C\binom{N}{2} + \dots + \left(\frac{N}{2} + 1\right)C\binom{N}{\frac{N}{2}} + \\ & + \frac{N}{2}C\binom{N}{\frac{N}{2} + 1} + \left(\frac{N}{2} - 1\right)C\binom{N}{\frac{N}{2} + 2} + \dots + 1C\binom{N}{N}] \end{aligned} \quad (5)$$

Substituting values in (1),(2),(4),(5) the expected average energy per unit capacitance curves (NENL, EL_average, L, E_average) for different values of N varying from 1 to 16 for the different techniques, is shown in Fig. 5. For the NENL technique and E technique, we use $V_{ref} = V_{DD} = 1.2V$. For the L and EL technique we use $V_{ref} = 0.8V$. The best case energy savings for the bus-invert coding technique, is when all the bits over a N bit wide bus flip.

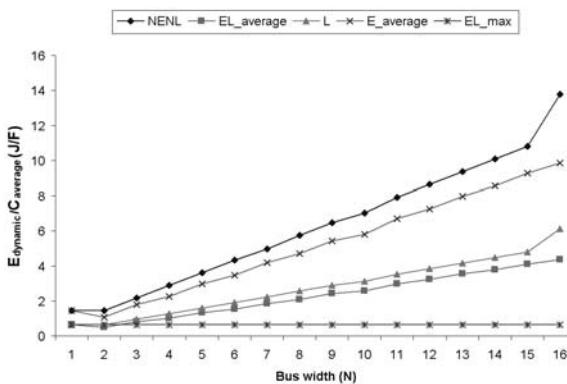


Fig. 5. Theoretical energy for different schemes

In this case, for the techniques which do not employ encoding, the number of transitions would be N , and for the techniques which employ encoding, the number of transitions would be 1 (due to the invert signal). So the lower bound for the energy/capacitance value is $V_{ref}^2 * T_E = V_{ref}^2 * 1$, this is indicated by the EL_max curve in Fig. 5. The curves in Fig. 5 do not show the energy consumed by the driver and receiver circuits. They only show the theoretical estimate of the amount of energy per unit capacitance that is consumed over the interconnect. The actual energy consumed, the delay and the power-delay curve inclusive of the contributions from the driver and receiver circuit are shown in the section on simulation results.

4 Circuit Implementation

We developed an efficient implementation of the driver for an 8 bit wide bus using an analog majority voter circuit as shown in Fig. 6. The receiver circuit is shown in Fig. 7. The current state of the bus (D_0T , D_1T , ..., D_7T , INV) is compared with the new values to be transmitted. If majority of the bits have flipped, the analog majority voter sets the INVB signal (shown in Fig. 6) to high. The advantage of using the analog majority voter circuit is that it is easily scalable to larger bus widths with very little extra area overhead. The encoded signal values are then converted into a low swing value using the NMOS only push-pull driver[1, 6]. The driver and receiver circuits consume very little power as is illustrated in our simulation results as well. In the driver, in the analog majority voter circuit, by using the clock as the gate signal for the PMOS transistors in the latch and for the NMOS transistor (at the bottom) acting as a current source, we ensure that there is never a path from the power supply to ground except during the clock transitions. In the receiver, since we use cascode circuitry and differential circuits, the short circuit current is reduced.

The receiver consists of a low-swing restorer and a decoder as shown in Fig. 7. The decoder consists simply of XOR gates, which uses the “invert” signal to either invert or not-invert the received values depending on whether the “invert” signal is 1 or 0.

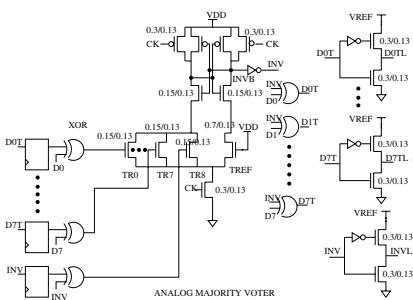


Fig. 6. Encoded low swing transmitter for 8 bits

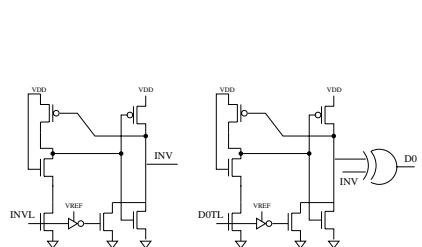


Fig. 7. Encoded low swing receiver

5 Simulation Results and Comparison

The set up that we use for comparing the four techniques, the NENL, EL, L and E was illustrated in Fig. 1. It has to be noted that the power-delay product that we obtain for the E scheme and the EL scheme are data dependent. The best case energy savings for E and EL is when all the bits flip, and the worst case is when $N/2$ bits flip. We take care to ensure that the data sequence does not consist of bits which flip every cycle, since this represents the best case for bus-invert coding and this would result in an unfair comparison. The simulation results that we obtain are averaged over a random sequence of data bits. All the results(power, delay and power-delay product) are inclusive of the transmitter and receiver circuits.

5.1 FPGA Interconnect for 8 Bits

We perform the simulations over the benchmark FPGA interconnect circuit shown in Fig. 3 and Fig. 4. The proposed EL technique is particularly interesting here due to the presence of large capacitive loads over FPGA interconnects. Even though each wire in a FPGA channel can have different sources and sinks depending on the configuration that is loaded, the proposed technique can be applied to FPGAs when the logic block is doing datapath operations where normally the granularity is higher like a 4 bit addition, 8 bit multiplication etc. The energy and delay over FPGA interconnects are larger than over general interconnects due to the presence of series complementary pass transistor switches which increase the resistance and capacitance over the path of the signal. This is confirmed in our simulation results as well. As the capacitive load begins to increase, the encoded-low swing technique consumes the lowest energy. By reducing the number of transitions, and subsequently the number of times the capacitance has to be charged and discharged, over the interconnect, the schemes employing encoding (E and EL) have lower delays than the L scheme, but higher than the NENL scheme. Fig. 8 shows the plot of power-delay product against capacitive load. For low capacitive loads ($CL \leq 100fF$), the L scheme has the best power-delay product. But, as CL increases, the EL scheme outperforms the rest and has the lowest power-delay product.

The above simulations were performed for a fixed length of interconnect ($L=1mm$) and the capacitive load due to fanout (CL) was varied from $0ff$ to $2pF$. We also performed simulations over the FPGA test interconnect circuit keeping CL constant at an average value of $400fF$ and varied the length of the interconnect from $0.1mm$ to $1mm$. The results are summarised in Fig. 9. Fig. 9 essentially shows how the power-delay product varies with wire resistance, RW , and wire capacitance, CW . It can be seen that the proposed technique has the best power delay product.

5.2 General Interconnect for 8 Bits

Analysis of Fig. 10 illustrates that power values increase almost linearly against capacitive load (CL) but with different slopes for different schemes. As the ca-

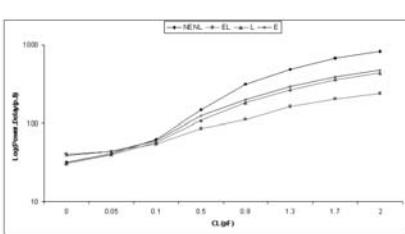


Fig. 8. Power-Delay vs CL for $L=1\text{mm}$ for FPGA interconnect

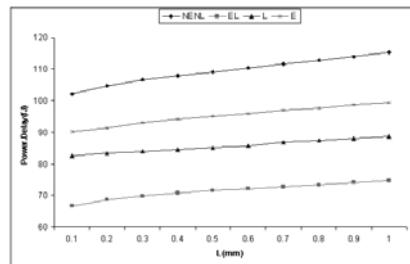


Fig. 9. Power-Delay vs Length for $CL=400\text{fF}$ for FPGA interconnect

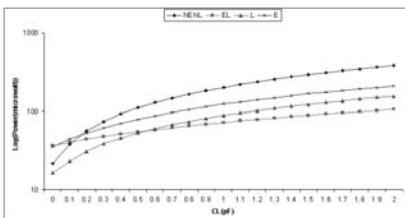


Fig. 10. Power vs CL for $L=1\text{mm}$ for general interconnect

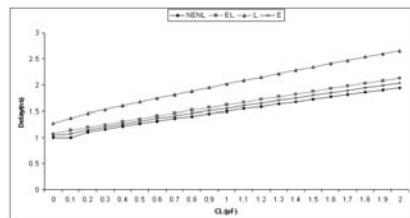


Fig. 11. Delay vs CL for $L=1\text{mm}$ for general interconnect

capacitive load begins to increase, the encoded-low swing technique outperforms the other techniques. It can be seen from Fig. 11 that the signal delay over the interconnect is the lowest for the NENL(full-swing CMOS) scheme and is the highest for the L scheme, as expected. By reducing the number of transitions, and subsequently the number of times the capacitance has to be charged and discharged, over the interconnect, the schemes employing encoding (E and EL) have lower delays than the L scheme. Fig. 12 shows the plot of power-delay product against capacitive load. For low capacitive loads ($CL \leq 200\text{fF}$), the L scheme

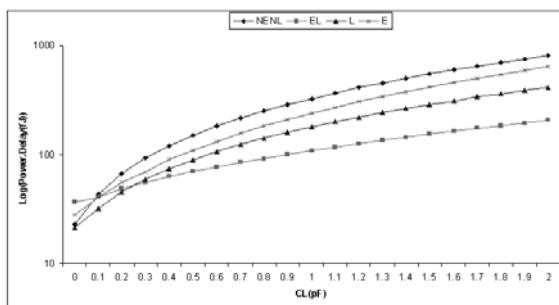


Fig. 12. Power-Delay vs CL for $L=1\text{mm}$ for general interconnect

has the best power-delay product since the complexity of the driver and receiver circuit is less than that of the driver and receiver of the EL scheme. But, as CL begins to increase, the savings from driver and the wire begin to dominate and the energy overhead of the receiver remains almost constant. Then the proposed EL technique has the best power-delay product.

6 Effect of Crosstalk and Process Variations

We simulated the effect of crosstalk and process variations for the four techniques that we are comparing. It is summarized in Table 1. To simulate the effect of crosstalk we assume that the crosscoupling capacitance value is as specified in the CMOS 0.13 μ m process for a metal 3 wire at minimum spacing. We assume a victim wire surrounded by two aggressors(3 bit wide interconnect). We assume the worst case scenario, in which the aggressors switch in the same direction and induce a noise over the victim wire, which is switching in the opposite direction. In this way, we are able to measure the effect on signal delay, of crosstalk induced slow down as well. We simulate this condition over the slow, fast and typical process corners. We denote this for a L=1mm interconnect line with a capacitive load for fanout (CL) assumed to be 100fF. It can be seen from the table that in the presence of crosstalk noise, the energy consumed and the delay over the interconnect increases for all the schemes. But, the techniques employing encoding are, on an average, less affected by the presence of crosstalk noise because of the reduction in the number of transitions. In the presence of crosstalk noise, the best power-delay product is of the proposed encoded low swing technique.

Table 1. Effect of crosstalk and process variations

Scheme	Process	Power(P)(μ W)	Delay(D)(ns)	PD(fJ)
NENL	slow	51.81	1.051	54.45
	typical	51.86	1.016	52.68
	fast	52.25	0.946	49.428
Proposed EL	slow	22.425	1.257	28.188
	typical	23.55	1.264	29.767
	fast	24.805	1.209	29.989
L	slow	25.15	1.434	36.065
	typical	26.285	1.405	36.93
	fast	29.51	1.376	40.605
E	slow	24.195	1.209	29.25
	typical	25	1.208	30.2
	fast	25.965	1.141	29.626

7 Signal to Noise Ratio Analysis

We use the worst case analysis method presented in [4] to measure the signal integrity of each circuit. The noise sources are classified into two categories: the proportional noise sources and the independent noise sources

$$V_N = K_N V_S + V_{IN} \quad (6)$$

$K_N V_S$ represents those noise sources that are proportional to the magnitude of the signal swing(V_S) such as crosstalk and signal induced power supply noise. V_{IN} includes those noise sources that are independent of V_S such as receiver input offset(due to process variation), receiver sensitivity, and signal-unrelated power supply noise.

Table 2 summarises the noise sources and their contributions. The crosstalk coupling coefficient is defined as $K_C = \frac{C_C}{C_B+C_C}$ where C_C represents the coupling capacitance and C_B represents the bottom capacitance. In the CMOS 0.13 μm process, the value of C_C is 109fF/ μm and C_B is 113fF/ μm . The crosstalk noise attenuation factor given by Atn_C [1] is defined to be 0.2 for a static driver and 1 for a dynamic driver. The signal induced power supply noise is estimated to

Table 2. Parameters in signal to noise ratio analysis

K_C	Crosstalk coupling coefficient=0.49 for 1mm wires with 3fF load and 0.18 μm spacing
Atn_C	Crosstalk noise attenuation=0.2 for static driver
K_{PS}	Power supply noise due to signal switching=0.05 for single ended signalling
Rx_O	Receiver input offset = 30mV for inverter
Rx_S	Receiver sensitivity = 30mV for inverter
PS	Unrelated power supply noise = 5% of V_{DD} 0.05*1.2 = 0.06
Atn_{PS}	Power supply noise attenuation= $\Delta V_{TH}/\Delta V_{DD}=0.5$
Tx_O	Transmitter offset=30mV

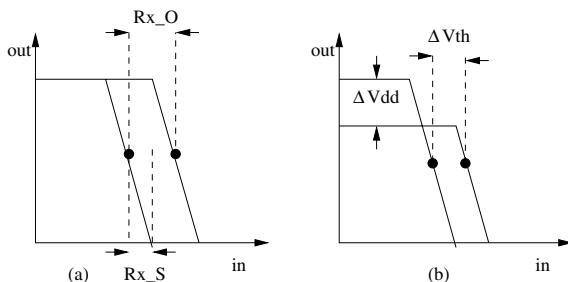


Fig. 13. Voltage transform curves

Table 3. Signal to noise ratio calculations

Scheme	K_{PS}	K_N	V_S	V_N	SNR
Proposed EL	0.025	0.123	0.8	0.2184	1.83
NENL	0.05	0.148	1.2	0.2976	2.016
L	0.05	0.148	0.8	0.2384	1.67
E	0.025	0.123	1.2	0.2676	2.242

be 5% of the signal-swing for single-ended signaling. Process variations such as device size mismatch, threshold voltage variation etc., will induce receiver input offset noise which is denoted by Rx_O . Rx_S indicates the receiver sensitivity. These are indicated in Fig. 13(a). Through simulations for the CMOS $0.13\mu m$ process, we find that Rx_O and Rx_S are 30 mV for an inverter.

The power supply noise attenuation Atn_{PS} is defined as $\frac{\Delta V^{th}}{\Delta V^{dd}}$ and is indicated in Fig. 13(b). Through simulations for the CMOS $0.13\mu m$ process we find that ΔV^{th} is 30mV, and so we calculate Atn_{PS} to be $\frac{0.03}{0.05 \times 1.2} = 0.5$. K_N and V_{IN} in Eqn.6 are defined by Eqn.7.

$$K_N = Atn_C K_C + K_{PS}, V_{IN} = Rx_O + Rx_S + Atn_{PS} PS + Tx_O \quad (7)$$

The noise margin is defined as $SNR = \frac{0.5V_S}{V_N}$ where V_S stands for the signal swing and V_N is the total noise induced voltage which is defined by Eqn.6. The power supply noise is maximum when all the signals switch. The techniques employing encoding (namely EL and E) prevents all the signals from switching simultaneously, infact only 50% of the signals are allowed to switch. So, in the worst case, the number of transitions is 50% compared to unencoded case. So the probability of the worst power supply noise is correspondingly reduced. This leads to a reduction in the factor K_{PS} . Using the values in Table 2, we compute the SNRs for the four schemes that we are comparing. This is shown in Table 3. The techniques employing full swing, namely NENL and E, have the best signal-to-noise ratios. This is due to the fact that the signal swing indicated by V_S in the SNR equation is higher (1.2V) than the signal swings (0.8V) of the techniques employing low-swing, namely EL and L. But it has to be noted that the power-delay product of the NENL and E techniques are much higher than the EL and L techniques. For comparable power-delay products, it can be seen from Table 3 that the worst-case SNR of the proposed EL technique is better than the L technique by 8.8%. This can be attributed to a reduction in the power supply noise due to signal switching. The worst case flipping of all bits is prevented by the encoding, leading to a direct reduction in power supply noise.

8 Conclusion

We introduced a novel encoded-low swing technique and an efficient circuit implementation of the same. This achieves the best power-delay product over the existing schemes when the capacitive load over the interconnect begins to increase above 200fF. Analyses of the simulation results show that the average

power-delay product of the proposed technique is superior by 45.7% with respect to techniques using only low swing, and by 75.8% with respect to techniques using only encoding for random data streams. In the presence of crosstalk noise, we show that the proposed technique has the best power-delay product even for small capacitive loads ($CL \leq 200fF$). The signal to noise ratio of the proposed technique is superior to existing low swing techniques by 8.8%. We perform the case study for both general and FPGA interconnects and prove the feasibility of the proposed technique for both. Since the power-delay product over FPGA interconnects is larger than over general interconnects the proposed technique is ideally suitable for FPGA interconnects and the energy savings are significant with increased loads. A disadvantage in the current circuit implementation is the need to have a reference low voltage power supply, but this can be easily overcome by using some of the techniques that exist in literature for generating a low swing signal without a reference low voltage power supply. In the future, the possibility of reducing the number of buffered switches on a FPGA as a consequence of the proposed technique needs to be investigated and quantified.

References

1. H. Zhang et al, "Low-Swing On-Chip Signaling Techniques: Effectiveness and Robustness", *IEEE Transactions on VLSI systems*, Vol.8, No.3, pp.264-272, June 2000.
2. K. Nakamura et. al. "A 50% Noise Reduction Interface Using Low-Weight Coding", *1996 Symposium on VLSI Circuits Digest of Technical Papers*, Pg.144-145, 1996.
3. Mircea R. Stan "Bus-Invert Coding for Low-Power I/O", *IEEE Transactions on VLSI systems*, Vol.3, No.1, pp.49-58, March 1995.
4. W. Dally and J. Poulton, *Digital Systems Engineering*, Cambridge, U.K., Cambridge Univ. Press, 1998.
5. Jonathan Rose, Vaughn Betz *Architecture and CAD for deep-submicron FPGAs*,
6. Varghese George et. al., "Design of a low energy FPGA", *International Symposium On Low Power Electronics and Design*, pp.188-193, 1999.
7. D.Liu et. al., "Power consumption estimation in CMOS VLSI chips", *IEEE Journal Solid-State Circuits*, Vol.29, pp. 663-670, June 1994.
8. E.Kusse, "Analysis and circuit design for low power programmable logic modules", *M.S. thesis, Univ. Calif., Berkeley*, 1997.
9. Musoll, E. et. al., "Working-zone encoding for reducing the energy in microprocessor address buses" *IEEE Transactions on VLSI Systems*, Vol. 6, Issue: 4, pp.568-572, Dec 1998.

Building Run-Time Reconfigurable Systems from Tiles

Gareth Lee and George Milne*

Department of Computer Science & Software Engineering,
The University of Western Australia.
[gel,george]@csse.uwa.edu.au

Abstract. This paper describes a component-based methodology tailored to the design of reconfigurable systems. Systems are constructed from *tiles*: localised, self contained blocks of reconfigurable logic which adhere to a specified interface. We present a state-based model for managing a hierarchical structure of tiles in a reconfigurable system and show how our approach allows automatic garbage collection techniques to be applied for reclaiming unused FPGA resources.

1 Introduction

Despite rapid growth in the commercial applications of field-programmable logic, few current systems utilise run-time reconfiguration. We believe this reflects the fact that designers of reconfigurable systems lack methodologies and tools to guide the design process. In a previous paper [1] we advocated the benefits of applying software engineering ideas to the design of reconfigurable computing devices. We propose a methodology based on composition of *tiles*: localised, self-contained blocks of logic, adhering to a specified interfaces [4] (see Section 2).

The tiles approach draws from established ideas in object oriented programming and software components and builds on previous work on reconfigurable cores [2]. Since each tile extends an *abstract tile* (which provides a formal abstract data type, much as an interface in java or an abstract class in C++) this opens the possibility of top-down design. But, in common with reconfigurable cores, our approach also allows tiles to be defined as a composition of simpler tiles, allowing the traditional bottom-up approach.

This paper focuses on how tiles can be used to construct systems which are both partially and dynamically reconfigurable. To avoid confusion we take *partial reconfiguration* to denote systems which allow adaptation of a subsection of the overall circuit, while the remaining circuit elements remain unchanged. Similarly we consider *dynamic reconfiguration* to be a case where a circuit is modified over time in a series of discrete steps. (For the synchronous logic considered here, this requires that the global clock be temporarily halted while reconfiguration occurs, but the methodology also applies to asynchronous logic.) Thus a system

* This research was funded by a grant from the Australian Research Council. We would also like to thank Xilinx Inc., San Jose, for the donation of devices and software.

may be dynamically reconfigurable but not partially reconfigurable, implying that the entire device must be reconfigured at each step. However a system which is partially reconfigurable must also be dynamically reconfigurable.

The reconfiguration benefits offered by our approach, stem from the strict adherence to an abstract tile (or interface type) for every tile. During dynamic reconfiguration we do not allow the interface type of a tile to change, only the internal realisation. This ensures that the new concrete tile shares a common interface with the old tile and will therefore be *plug-compatible*. It also ensures the side-effects of reconfiguration are limited, since reconfiguration can be localised to the region occupied by the tile (described in Section 3.1). The tiles methodology also enforces a formal life-cycle for each tile (Section 3.2).

The approach promises widespread applicability since many reconfigurable computing applications are constructed from repetitive sub-components: for example, digital filter pipelines, systolic arrays and cellular automata.

We demonstrate our ideas in the form of a reconfigurable regular expression matcher (Section 4). The choice of application stems from our previous work, however we see the scope of this approach as being much wider than this particular type of example. The application allows many patterns to be concurrently searched for, within a continuous stream of text. It allows patterns to be added or removed at any time during operation, resulting in partial reconfiguration of the FPGA. The system manages its internal resources by applying a copying garbage collection algorithm [3] to reclaim discarded space (see Section 5).

This paper makes a contribution to the field in its application of a fixed-interface tile approach for design and realisation of run-time reconfigurable systems. The state-model we present and message interchange approach is novel in this application, as is the use of automated garbage collection techniques for managing FPGA resources.

2 The Tiles Methodology

This section provides an overview of the salient points of our approach. We provide a more detailed treatment of tiles and a comparison with other work in this area in a previous publication [1].

2.1 Abstract Tiles

The basis for our methodology is the *abstract tile*, which is a rectangular region of logic with a defined interface to its neighbours. The abstract tile consists of a set of signal ports, declared much as in other HDLs [4], but each signal is also attributed to a specific edge of the rectangle: north, south, east or west. Thus the interface also contains spatial information about the directions in which signals flow in and out of a tile, which draws on previous work on structural descriptions of systems [5]. In addition to the interface information the abstract tile also has a specification which states what function the tile performs. The abstract tile does not provide a circuit which implements this function (hence the term abstract).

Multiple *concrete tiles* can be implemented which adhere to a single abstract tile. Each concrete tile is a system of logic elements which adheres to the interface defined by a single abstract tile and functions within the scope of the abstract tile's specification. The set of concrete implementations of a particular abstract tile form an equivalence class since they all perform the same function and all have the same I/O signals and the same spatial characteristics.

In our example (Section 4) we create an abstract tile which corresponds to a finite state machine for detecting regular expressions. We implement multiple concrete tiles which contain the circuits for various regular expression operators: for instance, one to match sequences such as "abc" and one to match one or more occurrences of a pattern "a+". Each of these tiles has the same set of input and output signals: we know that signal 'x' enters through the eastern edge.

Abstract tiles do not have the size of the circuit as an attribute since they do not presuppose an implementation. The circuit for detecting patterns of the form "a+" may be physically larger than that for detecting "abc", for example.

2.2 Primitive and Composite Tiles

Concrete tiles may be formed in two ways. They may be designed at the circuit level using existing HDLs, but in our case we build these *primitive tiles* out of established run-time parameterisable cores [2], hereafter referred to simply as *cores*. In previous work [1] we referred to primitive tiles as 'component tiles', but feel the new name better describes their role.

Cores provide a way of localising and controlling the layout of the design elements constituting static systems, but the designer still has to resort to *ad hoc* techniques when reconfiguring the elements; our approach provides a more methodical strategy for programming reconfiguration (described in Section 3).

Alternatively, *composite tiles* may be formed by composing one or more primitive tiles together. This establishes a hierarchical relationship between tiles and sub-tiles which we utilise heavily. Thus an entire system consists of a tree of tiles with each composite tile having one or more offspring and primitive tiles only occurring as leaf nodes (for example, see Figure 1).

The controlling mechanisms provided by this hierarchy lead directly to benefits in FPGA resource management. Each tile has a state variable attached indicating whether it is currently realised on the FPGA substrate and whether a tile is in use or has been declared redundant. We use these attributes to explore the benefits of automated garbage collection for managing FPGAs by periodically reclaiming CLBs occupied by discarded tiles (see Section 5).

2.3 Controlling Objects

In common with JBits cores [2] and approaches such as JHDL [6] each tile has two distinct aspects: a physical circuit configured into a small rectangular region of the FPGA and a *controlling object* existing within the memory of a supervising computer. This computer re-programs the FPGA, modifying its internal state each time reconfiguration is required. Our approach differs from others since the

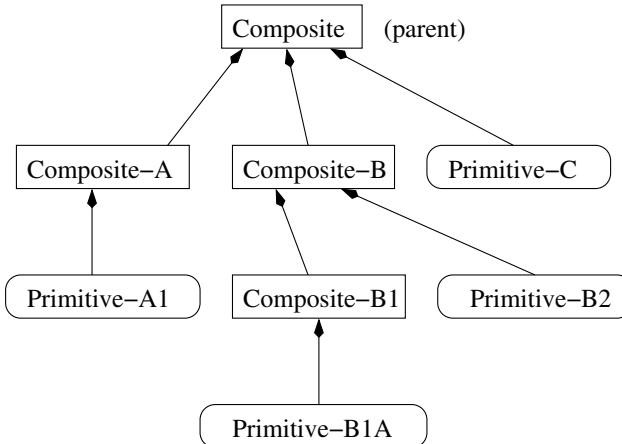


Fig. 1. An example of a tile hierarchy. Composite tiles are shown in rectangles and primitive tiles in boxes with rounded corners. We construct primitive tiles from JBits' ULPPrimitives cores, but this has been omitted for clarity.

controlling object not only acts as an *on-the-fly* generator for the physical circuit, but persists for the lifetime of the circuit, coordinating future reconfiguration. We have designed these controlling objects as a class library in Java.

Centralised control also allows us to ensure that the overall system configuration remains valid at all times, guaranteeing correct behaviour and avoiding deadlocks. In general, such problems can be solved by applying formal modelling techniques, during design, to the reconfiguration state-space.

3 Reconfiguration Approach

As mentioned in the previous section, a system of tiles can be thought of as a tree with composite tiles forming the branches and primitive tiles as the leaves. Each tile within the tree has a separate map of configuration attributes (and values) associated with it. Both types of tile can be reconfigured.

During reconfiguration tiles may be thought of as *active objects*, rather than passive circuits on the FPGA substrate, since each has a controlling object associated with it in the memory of the supervising processor. Therefore, in the following discussion the term *tile* refers both to the circuit which is being reconfigured and its controlling object.

3.1 Reconfiguring Tiles

Primitive tiles can be reconfigured when their configuration attributes are modified. For instance a tile might be created, as part of a digital filter, which multiplies a variable input by a constant value C . When the configuration at-

tribute C changes value it results in the primitive tile reconfiguring. This reconfiguration may result in minor changes to the internal connections within the tile (such as *constant folding*, when C changes from 6 to 7) or more radical reorganisation. For instance, when C changes to 0 the tile can discard all its internal circuitry and set its output to constant zero.

Each *composite tile* has a set of one or more sub-tiles, which along with any additional cores, define the tile. The sub-tiles and sub-cores are embedded within the composite tile's area on the FPGA substrate. Each of the offspring tiles has an immutable type (specified by an abstract tile) which is set when the composite tile is first defined. However the implementation of each sub-tile cannot be resolved until run-time since it may depend on external events.

Several distinct concrete tiles may be created adhering to the same abstract tile, forming an equivalence class since its members are interchangeable. The members of this set will share a common structure (or interface to their neighbours) but each will have a different behaviour, albeit within the scope of the specification of their abstract tile. A parent composite tile may only be reconfigured when one member of the class is swapped for another. Assuming sufficient space is allocated by the parent composite tile for the physically largest member of the equivalence class, the effect of implementation will be limited to the region of the sub-tile (*i.e.* CLBs within the sub-tile will need to be reconfigured to implement the new circuit). However the effect of swapping a tile may *spill over* slightly, since connections to neighbouring tiles must also be re-routed.

3.2 Tile State Model

We allow tiles to be dynamically created, modified and destroyed, much as an object in an object oriented program. Tiles must therefore follow a well defined life-cycle so that they can allocate and deallocate their resources in an ordered fashion. This is achieved by associating a state variable with each tile, which is held within its controlling object. The state variable can take one of six states as show in Figure 2, which also shows the valid state transitions tiles can follow.

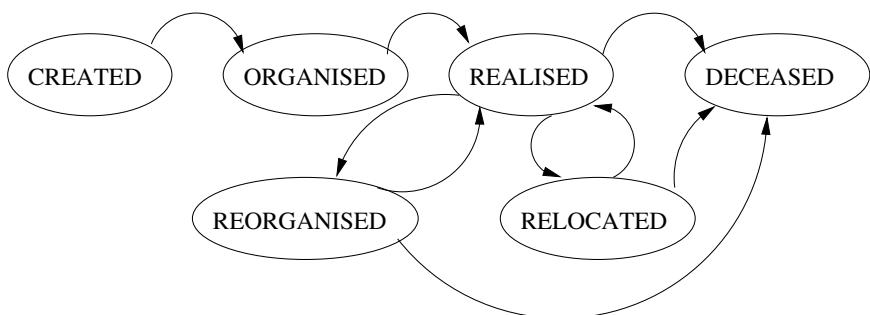


Fig. 2. The tile states and valid transitions. All tiles originate in the CREATED state and progress through configuration and optional reconfiguration to the DECEASED state, at which point their resources can be recycled.

Tiles coordinate their reconfiguration by exchanging messages between controlling objects (Section 3.3) instructing each to change state and by changing the configuration maps associated with their offspring (Section 3). To design a concrete tile using our approach the user has to program call-back handler functions in Java for some of the state transitions in Figure 2.

Tiles start in the CREATED state and must be configured before use. When a tile is instructed to move to the ORGANISED state by its parent, a tile reads its configuration parameters (if any) and creates its constituent parts. Composite tiles will organise themselves by creating and organising sub-tiles, whereas primitive tiles just create and configure their constituent cores. We refer to this organisational phase as *structural generation* [1].

Once all the tiles within the hierarchy have been organised a further parent message triggers transitions to the REALISED state. This instructs each controlling object to generate the appropriate circuit within the configuration bit-stream, which we refer to as *physical generation* [1].

Run-time reconfigurable systems will utilise three further states. Firstly, a tile can be instructed to move to the REORGANISED state when its configuration parameters are changed by its parent. A composite tile will reorganise its constituent parts, creating and deleting sub-tiles as appropriate. This may result in the circuit being regenerated and will move back to state REALISED once the bit-stream has been updated.

Secondly, a transition can occur to the RELOCATED state, when a parent tile instructs a sub-tile to physically relocate on the FPGA substrate. This can be used to create extra space for siblings when a parent tile is reorganising (or in garbage collection, Section 5.1). The state of relocating tiles must persist, therefore the controlling object detects all the memory elements within the tile and reads-back the values stored in these latches. Once the circuit corresponding to the tile is rebuilt at a new location on the FPGA substrate the latches will be preset to the stored values. (This poses a technical problem with Virtex FPGAs which have no direct I/O command to ‘write-in’ state to selected latches.)

Finally, superfluous tiles may be instructed to move to the DECEASED state so that they can disconnect all external connections. The region on the substrate corresponding to such a tile lays dormant. When a tile dies its controlling object can be garbage collected within the Java virtual machine. We describe a similar approach for reclaiming space on the FPGA substrate in Section 5.

3.3 Inter-tile Messaging

When tiles must be reconfigured their controlling objects coordinate by exchanging messages. Typically messages are sent from parent tiles to their offspring. Each message is an instruction to a sub-tile to change state, along with an optional map of configuration parameters. For instance, a parent tile might send a message to a REALISED sub-tile instructing it to enter the REORGANISED state, along with the changes to its configuration set that warrant the change. Alternatively, a message instructing a sub-tile to move to the DECEASED state does not require parameters (and is equivalent to the C++ delete operator).

Our current implementation treats message dispatch synchronously; the sender is blocked until the recipient has acted on the message. In future work we hope to investigate asynchronous dispatch of messages since this would allow the controlling objects to be executed on multiple supervisory processors. This would allow supervision to originate from multiple embedded processors such as the four PowerPC 405 processors included within Xilinx's VirtexPro FPGAs [7].

4 Regular Expression Application

A regular expression (regex) matching application was used as a vehicle to test this tile-based approach. Our interest in this application stems from the fact that it is representative of a wide class of reconfigurable systems [8]. Here, we extend an application presented previously [1], [9], to make it fully reconfigurable.

We assume the system is deployed in an environment where it receives a continuous stream of text, in which it is to detect a set of regular expression patterns. It generates an interrupt in real time when one or more patterns match and indicates which pattern(s) caused the interrupt. At any point the user may add additional patterns to the system or remove existing patterns. Any such command results in the clock being temporarily halted and the FPGA partially reconfigured. We therefore assume the data stream is buffered and that the FPGA, once restarted, can clear the backlog.

We previously presented a design process for finite state machines which match arbitrary regular expressions and how they can be implemented with tiles [1]. This system groups together a number of these regex sub-tiles into a 'Slice' composite tile which occupies a column within the FPGA. A Slice serves two purposes: (i) it groups the regex tiles together into a chunk, akin to a page in a virtual memory system, to facilitate garbage collection (described in Section 5) and; (ii) it combines the match outputs of an arbitrary number of regex tile via a chain of OR gates to detect whether any pattern is matched.

The top level composite tile, or 'Slice Manager', composes a user-specified number of Slices to build the system, as shown in Figure 3. Similarly, it ORs together the outputs of each Slice to generate a single aggregate match signal which acts as the interrupt source. When a match occurs this temporarily stops the data source sending more text and tests to see which Slice(s) are reporting a match. The state-persistence mechanism described previously in Section 3.2 provides a useful short-cut for reading-back the internal state of the regex tiles within the matching Slices to see which regular expressions are matching.

We have implemented this design on a Celoxica RC1000 board with a Virtex XCV1000 FPGA. The host computer acts as a supervisor, running our tiles library which uses Xilinx's JBits 2.8 API [2]. The application has not been optimised for performance but can be clocked up to 50 MHz. But since it uses four-phase handshaking to transfer text data through an 8-bit control port into the FPGA, this limits throughput to about 3000 chars/sec. However the regex tiles are fully pipelined and by using DMA to transfer data onto the board, much greater throughput should be possible.

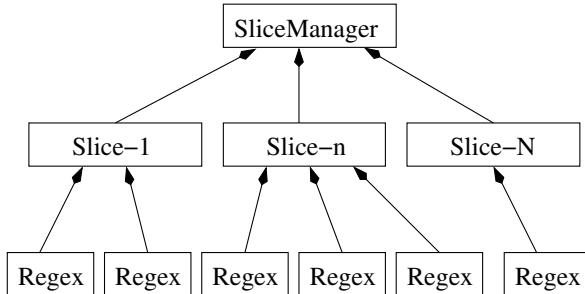


Fig. 3. The hierarchical structure of the regular expression matcher. The physical structure is shown in Figure 4. The Regex composite tiles are composed of, still simpler, primitive tiles.

Our current implementation does not consider how reconfiguration affects combinational logic delays. However the approach allows propagation delays to be precomputed for each tile and keeps inter-tile connections short, and therefore unlikely to impact on the global clock rate. This is a topic for further research.

5 Resource Management

The assumption underlying our approach is that a future system of field programmable logic will need to be able to adapt in complex ways to the environment in which it operates. This means that the system will need to be able to react to external events, such as someone plugging in a peripheral device, by dynamically creating internal circuit sub-systems to act as *drivers* [7].

One way this could be achieved would be with some form of *virtualisation* technology, providing virtual CLBs, similar to the virtual memory used by modern operating systems. Such a system would realise all the possible logic within its ‘virtual’ matrix of CLBs and then *page-in* the circuits on demand. There has been little progress in this direction; applying this concept to FPGAs, without diminishing the benefits of massively concurrent computation and communication, is a formidable technical problem.

Our approach assumes that exponential increases in FPGA size over time will ensure there is sufficient space available to implement all the sub-systems needed to perform a particular task. However, we assume that an application will need a mechanism for recycling its resources: old circuits will become redundant and must be deallocated, to allow CLBs to be recycled to create new sub-systems. In this section we explore whether tiles offer an appropriate level of granularity to support garbage collection.

5.1 Garbage Collection

As part of the regular expression matching system we have implemented a form of *incrementally-compacting garbage collection* (GC) [3]. The im-

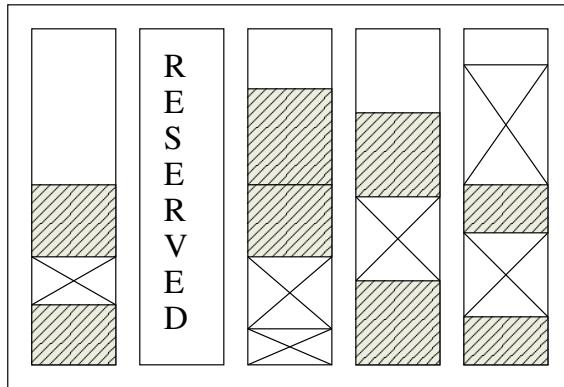


Fig. 4. The arrangement of tiles to permit GC. The outer rectangle corresponds to the SliceManager, whereas vertical columns represent Slices and each is filled with crosshatched regex tiles from the bottom. The white rectangle at the top of each column represents space which has not yet been used. Some tiles have been discarded and are marked with an X. Note the reserved Slice which is kept empty to permit GC.

plementation was simplified by the fact that each of the regex tiles is a single column wide and a variable number of rows high, depending of the complexity of the pattern. (This allows us to exploit the partial reconfiguration of individual columns provided by Virtex FPGAs.) Our design stacks up the expression tiles within columns of the FPGA as each new pattern is added. We store patterns within a user-specified number of columns but reserve a single column for GC. The arrangement of tiles within the FPGA is shown in Figure 4.

The tiles which correspond to discarded patterns are instructed by their parent Slice to enter state DECEASED. In response they disconnect themselves from their neighbours, but continue to occupy space within their column. When there is no longer space to add a new pattern the tile hierarchy is traversed to ascertain which column has the most ‘dead’ space. The active tiles within this column are copied across to the free column and the old Slice tile is instructed to enter the DECEASED state. The column is then overlaid by a new Slice tile, which is instructed to become ORGANISED then REALISED creating an empty column. This column then becomes the new reserved column to act as the destination for the next cycle of GC.

All GC techniques need to be able to traverse the available resources, usually the memory heap, following pointers to detect the memory regions that are used and those which, being unreachable, can be discarded. Our tile hierarchy, along with the state variable in each controlling object provides an ideal mechanism for following the same approach with FPGA resources.

Extending GC to arbitrary two dimensional arrangements of tiles would complicate the copying operation somewhat but doesn’t fundamentally change the approach. We intend to investigate this in future work.

6 Conclusions

This paper has argued the need for design of fine-grained reconfigurable systems using a more methodical approach than current reconfigurable cores. We believe that the tiles methodology sensibly builds on existing reconfigurable cores and bridges the gap between APIs, such as JBits, and system design methodologies such as object orientation. It also provides a useful starting point for investigating the automation of reconfiguration and resource management, allowing design and implementation of reconfigurable systems in a more abstract way. Construction of a real-time regular expression matcher has both proved the efficacy of this approach and provided an initial implementation of a garbage collector designed to manage FPGA resources.

References

1. Lee, G., Milne, G.: A methodology for design of run-time reconfigurable systems. In: Proceedings IEEE International Conference on Field-Programmable Technology, CUHK, Hong Kong, IEEE Computer Society Press (2002) 60–67
2. Guccione, S.A., Levi, D.: Run-time parameterizable cores. In Lysaght, P., Irvine, J., Hartenstein, R.W., eds.: Field-Programmable Logic and Applications, Springer-Verlag, Berlin (1999) 215–222 Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications, FPL 1999. Lecture Notes in Computer Science 1673.
3. Jones, R., Lins, R.: Garbage Collection – Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons (1996)
4. Milne, G.J.: CIRCAL and the representation of communication, concurrency, and time. ACM Transactions on Programming Languages and Systems **7** (1985) 270–298
5. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware design in Haskell. ACM SIGPLAN Notices **34** (1999) 174–184
6. Bellows, P., Hutchings, B.: JHDL – an HDL for reconfigurable systems. In: IEEE Symposium on FPGAs for Custom Computing Machines, IEEE Computer Society Press (1998) 175–184
7. Lysaght, P.: FPGAs as meta-platforms for embedded systems. In: Proceedings IEEE International Conference on Field-Programmable Technology, CUHK, Hong Kong, IEEE Computer Society Press (2002) 7–12
8. Lee, G.: Expression of information processing systems using diverse programming systems. Technical Report 2002/1, Reconfigurable Computing Research Group, The University of Western Australia, <http://www.csse.uwa.edu.au/~gel/reports/> (2002)
9. Gunther, B., Milne, G., Narasimhan, L.: Assessing document relevance with run-time reconfigurable machines. In Arnold, J., Pocek, K.L., eds.: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA (1996) 10–17

Exploiting Redundancy to Speedup Reconfiguration of an FPGA

Irwin Kennedy

Division of Informatics, University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, United Kingdom,
iok@dcs.ed.ac.uk

Abstract. Reconfigurable logic promises a flexible computing fabric well suited to the low cost, low power, high performance and fast time to market demanded of today's computing devices. This paper presents an analysis of what exactly occurs when a fine grain FPGA, specifically the Xilinx Virtex, is reconfigured, and proposes a tailororable approach to configuration architecture design trading off silicon area with reconfiguration time. It is shown that less than 3% of the bits contained in a typical Virtex reconfiguration bitstream are different to those already in the configuration memory, and a highly parallelisable compression technique is presented which achieves highly competitive results - 80% compression and better.

1 Introduction

Reconfigurable Field Programmable Logic (FPL) based Custom Computing Machines (CCM) provide the ability to alter their function by writing to the contents of their configuration memory at run-time. This paper explores ways of speeding up reconfiguration by analysing the changes that occur and proposing architectures and algorithms to leverage the observations.

One of the central aims of dynamic reconfiguration is to perform useful work by making the most efficient use of the silicon resources available. In applications where many thousands of specialisations per second are necessary to have the optimal algorithm implementation, the time to reconfigure the fabric would need to be much less than a milli-second, otherwise the benefit of specialisation is lost.

This paper examines the specific problem of speeding up partial dynamic reconfiguration of a fine grain FPGA. Section 2 introduces dynamic reconfiguration, gives relevant details of the Xilinx Virtex configuration architecture and presents an overview of previous work done in the area of speeding up reconfiguration. Section 3 analyses the resource redundancy in a circuit implemented on the Virtex. Section 4 presents an advanced configuration technique leveraging the redundancy in an FPGA to speed up partial reconfiguration. Section 5 describes three partial bitstream compression algorithms together with an example of a configuration architecture for using them.

2 Background

The Xilinx Virtex device [4] is chosen as the FPGA fabric for experimentation in this paper since it is a commercial device with a large established market. Another advantage of choosing the Virtex is the availability of the Java JBits API, enabling low level access and manipulation of bitstreams produced by the Xilinx tool flow.

The Virtex is composed of Configurable Logic Blocks (CLBs), Input Output Blocks (IOBs), block RAMs (BRAM), clock resources, routing resources and configuration circuitry. All of these resources are configured by a configuration bitstream that is read and written through an 8-pin configuration port.

The configuration memory [5] is arranged into a series of columns, each of which can be visualised as stretching from the top to the bottom of the device. A column is divided into 48 frames, and the frame forms the atomic unit of configuration, meaning it is the smallest piece of memory that can be read or written to. There are several different types of column in the Virtex device: a central clock column, two IOB columns and multiple CLB and Block RAM columns (depending on the part). For each frame in a CLB column the first 18 bits contribute to the control of the two IOBs at the top of the column, then there are 18 bits for each CLB in the column and finally another 18 bits for controlling the two IOBs at the bottom of the column.

The content of a frame is a seemingly unrelated subset of the configuration for the inter-connect, IOBs and CLBs. An example of the problems created by this for fast reconfiguration is that changing the configuration of the two IOBs at either end of a column of CLBs can require most of the frames in the column to be written to the device.

Configuration is done through a shift register called the Frame Data Register (FDR) into which the configuration data is loaded before being transferred in parallel to a configuration memory frame.

2.1 Existing Techniques for Speeding up Reconfiguration

The time taken to perform reconfiguration depends on a number of factors: the number of resources to be configured, off-chip configuration bandwidth, granularity (or atomic unit) of the configuration memory and the configuration memory organisation.

The importance of the first three factors to configuration time is obvious. The organisation of the configuration memory is important, since it can adversely affect the (naively) expected linear relationship between the number of resources being configured and the amount of data that must be loaded into the device. If configuration bits controlling unrelated resources are contained in the same memory locations, then there is a high likelihood that, with a small change to one area of the fabric, a disproportionately large number of memory locations will need to be written in order to bring about the change.

Configuration compression [1] exploits the similarities between frames but requires a large hardwired decompression unit which means the technique does

not scale well. The multi-context FPGA [2, 3] has multiple memory bits per configuration bit forming configuration planes. Although this provides extremely fast switching between preloaded contexts, the additional memory planes can require significant area and since a plane’s content is likely to change often, the off-chip configuration bandwidth is still a major bottleneck. Further, small changes are extremely wasteful since they require an entire context plane to implement.

3 Reconfiguration Bitstream Analysis

It is well known that due to the highly flexible nature of an FPGA’s interconnect, only a small fraction of the configuration bits loaded into a device for a particular circuit are important. This section of the paper explores the configuration changes necessary to switch between a pair of circuits.

The largest member of the Virtex device family has a configuration bit stream of over 1 million bits. Research topics of interest include finding how many of the configuration bits of a large device such as the Virtex are essential for the configuration of a circuit, and of those bits, how they break down across configuring routing resources, lookup-table (LUT) contents and the multiplexors providing I/O to the LUTs.

3.1 Technique

Analysis. The Xilinx JBits API provides a low level method of creating and manipulating Virtex bitstreams. It provides several layers of abstraction — from the provision of high-level utilities such as routers and tracers (JRoute and Route-Tracer), to the connection of individual wires and the setting of multiplexors (JBits.set()). It is possible to take circuits produced by an HDL synthesis flow and manipulate them using JBits.

To provide a means for analysing and observing reconfiguration in detail, a piece of software was written using the JBits API. The software takes as input two bitstreams generated by the standard Xilinx toolflow that have some CLB usage in common. It may be that one circuit uses all the CLBs used by the other circuit, or simply uses some of them, so the portion of the fabric where both circuits use the same CLBs is also specified as an input to the program. As output, the program produces a detailed list of the minimum number of changes necessary to reconfigure the fabric from implementing one circuit to the other.

The minimal set of changes necessary to reconfigure the fabric is produced by writing special wrapper functions for the low-level JBits calls and then using these wrappers to perform the reconfiguration. The wrappers record the setting of the resource being reconfigured in addition to modifying the bitstream. This means that when all necessary changes have been made, the final setting of each resource can be compared to its setting before any changes were made, producing the minimal set of changes.

The algorithm for implementing the reconfiguration is composed of three stages. The first stage involves reading the LUT and internal CLB multiplexors

configuration settings from the resultant circuit and writing these settings to the same CLBs in the starting circuit bitstream. The starting circuit bitstream is now a mix between the settings for CLB internals of the resultant circuit and the interconnect configuration of the starting circuit. The second stage consists of tracing the route of every possible source in the resultant circuit and extracting the settings of each resource in its path. The resource settings are then written to the starting circuit bitstream. After stages one and two, the starting circuit bitstream now contains the complete implementation of the resultant circuit with any non-overlapping resources for the first circuit left intact.

Unfortunately this bitstream will not necessarily configure the fabric to produce a working second circuit, this is because extraneous nets and partial nets belonging to the original circuit may overlap and interfere with the resultant circuit. For example, it is possible that a net will have more than one source. To remove this problem, all the sources in the new bitstream are traced and any sinks present which should not be are removed. The resulting netlists are now functionally correct, i.e., they connect a source to a list of sinks, but there maybe additional 'antenna' wires hanging off the netlist not performing any function. Although not altering the functional correctness of the netlist these antennae do potentially affect timing. For the purposes of this implementation, a simple timing analysis of the resultant circuit is performed and compared with the lean version. Those nets in the new circuit which are larger than the critical net in the lean version are trimmed appropriately.

The analysis tool enables a large number of different investigations to be carried out on the details of Virtex reconfiguration. The analyses carried out for this paper focus on revealing ways to reduce configuration time by providing a detailed view of what exactly occurs when reconfiguring.

Metric for Analysis. To put the number of bits that flip during reconfiguration in perspective, the actual number of bits controlling the configuration of a CLB is used as a metric. It is equal to the 18 bits in a frame allocated to a CLB multiplied by the 48 frames in a column, $18 \times 48 = 864$ bits.

3.2 Results

Two fundamental analyses of reconfiguration are presented in this section. The first is the percentage of bits in the configuration memory that flip during reconfiguration. The second is a breakdown of the percentage of bits that flip in each of the different resource types.

The experiments were conducted using a selection of circuits typically implemented on FPGAs: FIR and IIR filters, a DES encryption core, an FFT and a CORDIC. The circuits were placed and routed automatically, with area minimised and they ranged in size from 528 to 2320 slices.

It was found that the number of bits that change during reconfiguration is consistently between 8% and 10% of the total size of the configuration memory controlling that area of the fabric. It was also found that when the number of

bits that flip during reconfiguration is compared to the size of the bitstream loaded to perform the change, the percentage change is smaller again - less than 3%. The percentage number of bits that actually flip is consistently between 8% and 10% across the circuit pairs showing the generality of the result despite the large variation in circuit sizes.

Figure 1 shows that the majority of changes during reconfiguration occur in the multiplexors feeding the LUT inputs and the LUT contents themselves.

Sections 4 and 5 give two different approaches to leveraging the knowledge gained by the reconfiguration bitstream analysis. Section 4 gives details of a method called the overlay technique, which identifies and loads those changes that can be made before the fabric is taken off-line for the residual reconfiguration. Section 5 proposes a new configuration architecture design space, and explores some complementary reconfiguration compression algorithms that operate on the configuration changes required.

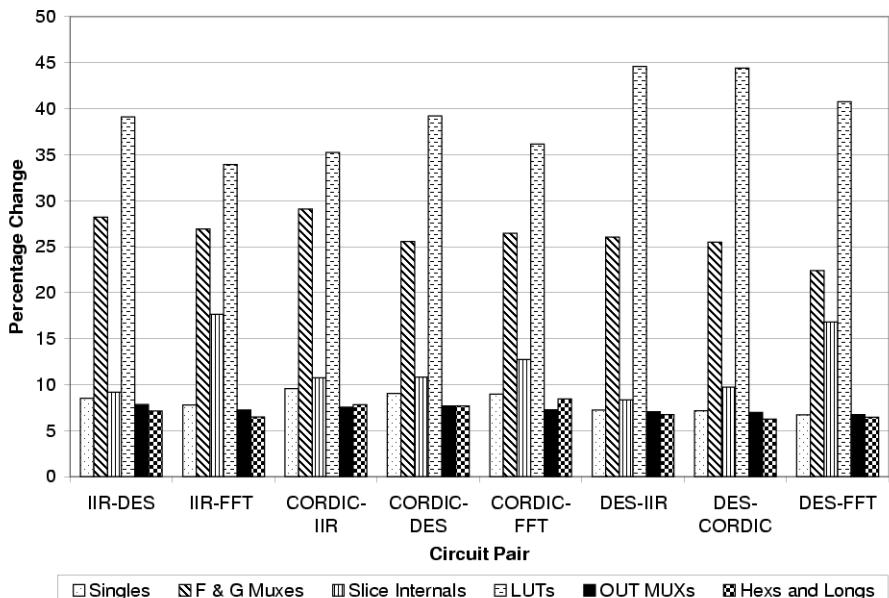


Fig. 1. The number of changed bits broken down by resource type and expressed as a percentage of the total number of change bits required to configure the area of fabric occupied.

4 Overlay Technique

From the analysis of the number of bits required for each resource type, it can be seen that a fair proportion of the bits that require changing are Programmable Interconnect Points (PIPs) controlling single connections. These are of interest because many of them could be set in advance of reconfiguration without affecting the operation of the existing circuit. This section presents and evaluates a technique for exploiting the potential of this redundancy.

A software tool was written, which takes in the bit streams describing two circuits, A and B, that time-share a section of the FPGA fabric and produces two new bit streams, called the advance bits and the residual bits. The advance bits describe a circuit equivalent to circuit A, except that in addition any resources used by B that can safely be set in advance are configured. The residual bit stream contains the remaining configuration data necessary to minimally switch from the advance bit configuration to a equivalent configuration for circuit B.

This overlay technique is suited to an embedded application where the computing requirements are easily predicted. In an application where circuit B is not necessarily required after circuit A, it may be desirable to separate the advance change bits from the definition of circuit A and only load the appropriate circuit's advance bits when the system's requirements can be better predicted. This is a simple extension of the algorithm implemented, producing three bitstreams instead of two - circuit A, advance bits for circuit B and the residual bits of circuit B.

When two circuits of different sizes are paired together, the area of the smaller circuit is chosen for the overlay experiment. Only PIPs are considered for overlaying in the experiments. Only CLB configuration data is manipulated, the circuits are not connected to IOBs, and any clocking or BRAM configuration is ignored. This is acceptable since the goal of this technique is the minimisation of the data required to reconfigure an area of logic, and logic is controlled by the CLB configuration data. It is expected that the technique can be extended to include IOB and clock configuration data.

4.1 Results

The overlay technique results in the identification that 10% of the change bits can be overlaid in advance of the fabric being taken offline for the second residual stage of reconfiguration. It is likely that with less densely packed circuits, and a more exhaustive set of techniques to identify resources that can be overlaid, e.g., LUT contents, CLB MUXes and CLB internal MUXes, the fraction of bits than can be overlaid at the advance stage could be increased.

5 Configuration Architecture Design Space

5.1 Overview

The reconfiguration bitstream analysis in Section 3 suggests that simply loading the changes required instead of the complete bitstream may improve reconfigura-

tion time. This Section investigates this within a new configuration architecture design space. A specific point within the new design space is selected and described before a number of algorithms are proposed to compress configuration changes. This approach can be used independently or enhanced through a combination with the overlay technique given in Section 4.

5.2 Architecture Description

RAM can be added to the configuration sub-system to the point where there are two (or more) RAM cells for every configuration bit (the multi-context configuration architecture), resulting in instantaneous whole-chip reconfiguration time in the order of a single clock cycle. This section of the paper aims to provide an illustrative example of how the analysis of what exactly occurs during reconfiguration presented in Section 3, identifies a new area of configuration architectures spanning the single context device and the multi-context device. The aim is not the definition of a new architecture, but more an exploratory feasibility study of a large family of architectures.

The specific architecture point chosen for study in this paper has a small RAM at the top of each column of configuration frames (or alternatively makes use of existing block RAM) and an associated configuration controller, as shown in Figure 2. The idea is to have the small configuration memory contain the changes that must be applied to the column, and a configuration controller applies the changes to the existing contents of the configuration memory. The configuration frames of each column can be read and written to by its configuration

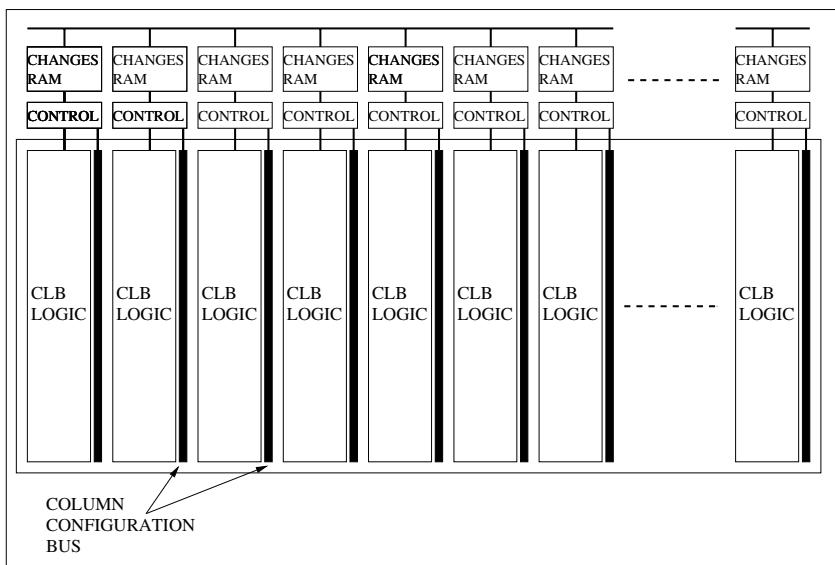


Fig. 2. Example of embedding extra RAM into the configuration subsystem.

controller, parallelising reconfiguration across columns. This architecture enables the sparse nature of the configuration bitstream changes to be compressed and stored on chip efficiently in advance of the reconfiguration stage. At the reconfiguration stage, the controller circuitry makes the changes by reading the existing configuration memory contents and applying the difference by inverting certain bits within the read back frame.

The configuration controller consists of an address generator unit, for reading from the small RAM, a frame address generator unit, two of registers for applying the difference function and the necessary control circuitry.

5.3 Configuration Change Compression Algorithms

This section proposes three possible compression techniques that leverage the observations from the earlier analysis and examines their effectiveness and cost in terms of silicon area.

Vanilla Compression Technique. The Vanilla compression scheme exhaustively searches and finds the optimum constant data chunk size for representing the changes and the corresponding optimum relative addressing scheme.

Banded Compression Technique. Change bits tend to be clustered into bands within the column because the change bits for every frame in the column are concentrated around the rows that are being changed. The banded technique expresses only the banded region of each frame. This reduces the space that must be covered by the addressing scheme of the change dataset.

Partitioned Compression Technique. Figure 1 reveals that expressing the LUT's value explicitly instead of attempting to compress it may be a better approach. The partitioned compression technique separates the LUT contents from the other configuration data, states its content explicitly and uses the banded compression algorithm on the remaining configuration bits.

Compression Results. Figure 3 shows the results for the three techniques presented as a percentage of the actual data configuring the portion of the FPGA. The partitioned technique provides the best results, reducing the amount of data that needs to be loaded into the proposed architecture to 45% of the bits configuring that part of the fabric. However, since the performance difference between the banded and partitioned techniques is small, the banded technique seems to be the best solution, as its implementation in terms of logic and memory resources is significantly simpler and is hence likely to require less silicon area despite its slightly bigger storage requirements.

When the banded techniques results are expressed as a percentage of the Xilinx bitstream that is loaded to configure the fabric the compression is around 80%, although this figure is highly dependent on the number of CLBs occupied in each column. This result competes well with the best reported configuration compression method for a complete bitstream [1].

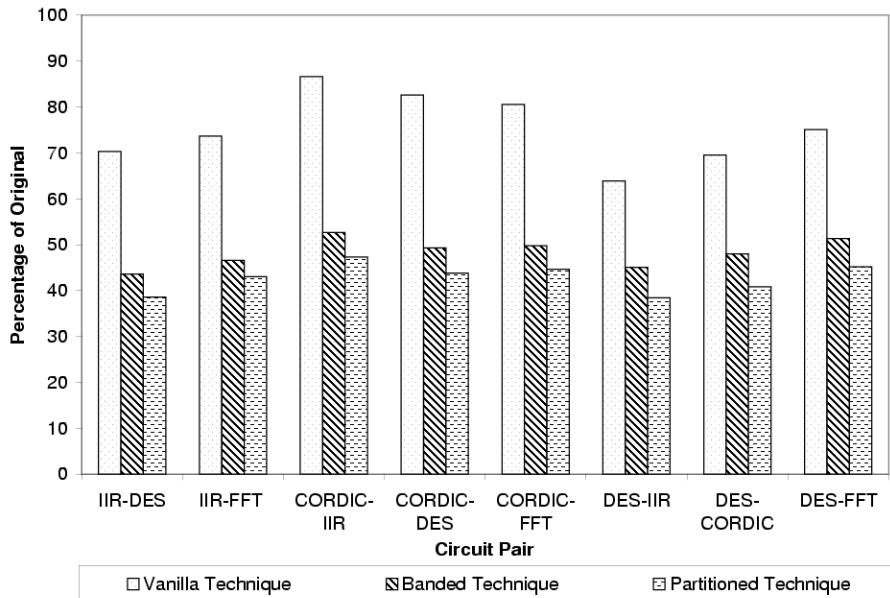


Fig. 3. The compressed changes dataset size expressed as a percentage of the number of bits required to configure the area of fabric occupied.

5.4 Architecture Evaluation and Discussion

A CLB has 864 configuration bits, so the configuration changes RAM is required to have 432 RAM cells per CLB to contain a complete configuration change for the column. At 5 transistors per cell, the changes RAM needs 2160 transistors which is about 1/3 of the estimated 6000 transistors in a CLB [6]. The control circuitry is negligible in size compared to the changes RAM. This is particularly true if the architecture is such that a frame may be read in a series of small chunks and hence doesn't require a large register to store its contents. Considering that IOBs, BRAM and other existing parts of the architecture mean that CLBs do not use all the silicon area it is estimated that the changes RAM would increase die size by less than 15%. This estimation is particularly conservative since it does not take into account the smaller area required by a single large RAM block per column compared to that of the highly distributed configuration RAM.

The time taken to reconfigure is proportional to the maximum number of CLBs requiring to be changed in any column. It is very dependent on the design of the control circuitry and the width of the changes RAM, but, assuming a column's control logic is capable of processing 1 change per clock cycle, then on average ($864/10=87$) cycles are required to reconfigure a CLB. So, as a concrete example, the XCV1000 part which has 72 CLBs per column would require a total of 72×87 (approx. 6300) cycles to entirely reconfigure. This is two orders of magnitude less than the time to reconfigure using the existing architecture.

The example architecture given is only one point in the design space of this configuration architecture domain and can be tailored to suit the application. For example, the number of change RAMs used could be reduced if parallel reconfiguration of the entire device is never necessary; say a design's requirements demonstrate no more than half the fabric's columns need to be reconfigured then the reconfiguration architecture would increase die size by less than 7.5%.

6 Conclusion

This paper's analysis of the actual changes made during reconfiguration shows the extent of redundancy present in a modern FPGAs bitstream to be 90%. Leveraging this insight, a configuration architecture design space was proposed that spans the area between the two main existing configuration architectures, and three complementary algorithms for compressing the reconfiguration changes were presented and tested. The compression technique proposed produces a 50% compression of the minimum bitstream required to configure a specific area of fabric, and an average of 80% compression of the bitstream used to configure the fabric in present Virtex devices. This result compares favourably with the existing algorithms for full bitstream compression, and brings the advantages of being less expensive to decompress and is highly parallelisable.

Acknowledgements

This work was sponsored by Lucent Bell Labs and the UK Engineering and Physical Sciences Research Council.

References

1. **Zhiyuan L. and Hauck S.**, Configuration Compression for Virtex FPGAs, *IEEE Symposium on FPGAs for Custom Computing Machines*, April, 2001.
2. **Motomura M., Aimoto yr., Shibayama A., Yabe Y. and Yamashina M.**, An embedded DRAM-FPGA chip with instantaneous logic reconfiguration, *Symposium on VLSI Circuits Digest of Technical Papers* pp. 55-56, June 1997.
3. **Trimberger S., Carberry D., Johnson A. and Wong J.**, A time multiplexed FPGA, *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997.
4. **Xilinx Corporation**, Virtex 2.5V Field-Programmable Gate Arrays, *DS003-1 (v.25)* April 2, 2001.
5. **Xilinx Corporation**, Virtex Series Configuration Architecture User Guide, *XAPP151 (v1.5)* September 27, 2000.
6. **Franklin N.**, Re: Silicon Area for Xilinx FPGAs, *comp.arch.fpga* 15:31:59 PST, 2 August 2002.

Run-Time Exchange of Mechatronic Controllers Using Partial Hardware Reconfiguration

Klaus Danne, Christophe Bobda, and Heiko Kalte

Heinz Nixdorf Institute, University of Paderborn,
Fürstenallee 11, 33102 Paderborn, Germany,
danne@upb.de, bobda@upb.de, kalte@hni.upb.de

Abstract. We present an efficient technique to implement multi-controller systems using partial reconfigurable hardware (FPGA). The control algorithm is implemented as a dedicated circuit. Partial runtime reconfiguration is used to increase the resource efficiency by keeping just the currently active controller modules on the FPGA while inactive controller modules are stored in an external memory.¹

1 Introduction

Control systems can be implemented in reconfigurable hardware as an efficient and high-performance alternative to control algorithms executed by processors [4, 7, 10]. The large design space offered by reconfigurable hardware allows an exploration of different area/time trade-offs and to customize the data-word width to the problem.

Complex mechatronic systems in a changing environment require adaptive control to perform well. The concepts vary from adaptive parameter control to the *multiple model approach* [8, 12, 15]. In the latter the plant is modeled as a process operating in a limited set of operating regimes. The control system consists of a set of controller modules, each optimized for a different operating regime. It automatically switches to the corresponding controller module during runtime. When using just one static FPGA configuration to implement the *multi-controller architecture*, most of the system resources would stay inactive. We use partial runtime reconfiguration to overcome this drawback and to increase the efficiency of the system. The following section introduces linear controller systems and our reconfigurable hardware solution for these controllers. In section 3 we introduce the enhanced class of multi-controller systems. Section 4 presents our prototyping system that allows the implementation of multi-controller systems based on reconfigurable hardware. We close with a conclusion and future work.

¹ This work was partly developed in the course of the Graduate College 776 -Automatic Configuration in Open Systems- and the Collaborative Research Center 614 - Self-Optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

2 Digital Linear Controller

The task of a controller is to influence the dynamic behavior of a system referred as *plant*. If the input values for the plant are calculated on basis of the plant's outputs, we refer to a control feedback (fig. 1).

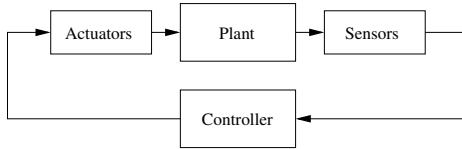


Fig. 1. Control Feedback Loop

A common basic approach is to model the plant as a linear time-invariant system. Based on this model and the requirements of the desired system behavior, a linear controller is systematically derived using formal design methods. The controller as a result of the synthesis considered above, is described as a linear time-invariant system and a time discretization is performed which results in eq. 1. The input vector of the controller is represented by \mathbf{u} (measurements from sensors of the plant), \mathbf{y} is the output vector of the controller (regulating variable to actuators of the plant) and \mathbf{x} is the inner state vector of the controller. The matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ and \mathbf{D} are used for the calculation of the outputs based on the inputs.

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{Ax}_k + \mathbf{Bu}_k \\ \mathbf{y}_k &= \mathbf{Cx}_k + \mathbf{Du}_k \end{aligned} \quad (1)$$

$$p = \dim(\mathbf{u}), \quad n = \dim(\mathbf{x}), \quad q = \dim(\mathbf{y})$$

The task of the digital system is to calculate eq. 1 during one sampling interval. That includes determining the new state \mathbf{x}_{k+1} and the output \mathbf{y}_k before the next sampling point $k + 1$.

2.1 Hardware Architecture of a Linear Control System

In previous work [4, 7] we developed a framework for the implementation of conventional digital linear controllers on reconfigurable hardware. It consists of transformation tools and a generic HDL (hardware description language) description of the controller architecture. Input to our design flow is the controller in form of eq. 1. Our framework generates fixed point arithmetic hardware which uses a fixed range of $[-1, +1]$. Thus a scaling of the equation system is necessary. The parameters of the matrices, the values of the input and output vectors, as well as the values of the state vector must not exceed this range during any time of operation. To perform the scaling transformation our tool requires the original range of the inputs and outputs determined by the physical sensors and

actuators and the original range of the state variables determined by a simulation or analytical methods [4, 7]. The scaled equation system is used as an input to our generic HDL controller architecture. The generated hardware structure is described below.

Both parts of eq. 1 share the same form and are independent of each other. So a generic *MEC* block (*Matrix Equation Calculator*) which processes eq. 2 is instantiated twice to process both parts of eq. 1 in parallel.

$$\mathbf{c} = \mathbf{Ma} + \mathbf{Nb} \quad (2)$$

Inside a *MEC* block the two terms \mathbf{Ma} and \mathbf{Nb} are also computed in parallel. For this purpose two *SMUL* blocks (*scalar-multiplier*) and an adder are instantiated. The first *SMUL* block multiplies one row of \mathbf{M} with vector \mathbf{a} at a time (the second *SMUL* block does the same for \mathbf{Nb} respectively). Therefore the rows of \mathbf{M} (and \mathbf{N}) are processed sequentially. A *SMUL* block multiplying two vectors of dimension n (a matrix row and a vector) consists of n multiplier instances and an adder tree to sum the results. The multipliers themselves are implemented as Booth multipliers which have a cycle delay that is equal to the word width of the operands. The parameters of the matrices are directly synthesized into the design. Some additional control logic is instantiated to control the sequential computing of the scalar products and to periodically start the computation at each sampling point.

The area/time trade-off of this architecture leads to $f_1(p, n)$ parallel instantiated multipliers and $f_2(n, q)$ sequential performed multiplications (eq. 3). $f_3(n, q, w)$ is the number of overall clock cycles per sampling point where w is the word width of the input values of \mathbf{u} . (see eq. 1 for p, n, q)

$$\begin{aligned} f_1 &= 2(p + n) \\ f_2 &= \max(n, q) \\ f_3 &= (w + 2) \times \max(n, q) + 2 \end{aligned} \quad (3)$$

In our implementation, the quadratic complexity of the problem (eq. 1) results into linear space complexity $\mathbf{O}(p + n)$ (parallel computing) and linear time complexity $\mathbf{O}(\max(n, q))$ (sequential computing). However, other mappings with different area/time trade-offs are possible. Moreover, the data width can be chosen at bit granularity according to the accuracy required by the application. Therefore, by using reconfigurable hardware the system can be implemented efficiently, i.e. using minimal hardware resources while meeting all constraints on computation time and accuracy.

3 Multi-controller Architecture

In the *multiple-model approach* the plant, e.g. a complex mechatronic system in an changing environment, is modeled as a physical process that is operating in a limited set of operating regimes. From time to time the plant changes the operating regime. With conventional methods it might be possible to design one

robust controller that controls the plant in all operating regimes, but it will not work optimal for the current operating regime. Parameter adaptive controllers can be used, but they may respond too slow to abrupt changes of the plant's dynamic behavior [13].

In the *multiple-model approach*, the plant is controlled by the architecture in fig. 2 [12]. It is composed of a set of *controller modules* (*CM*), each optimized for a special operating regime of the plant. The *supervisor* is able to switch between the controller modules to determine the active module. The decision to switch from one *CM* to the next is made on basis of measurements of physical values of the plant. The strategy of the *supervisor* can vary from simple functions of the measurements to agent-based techniques [15].

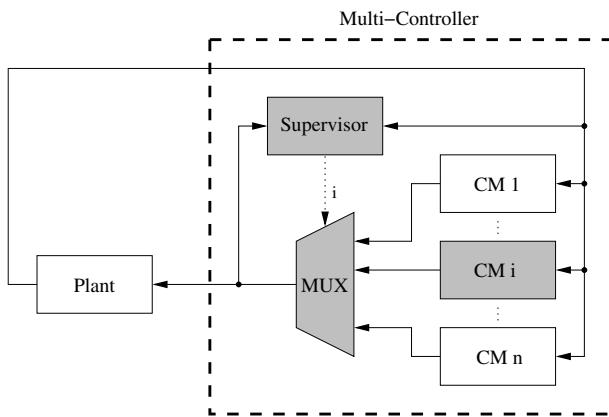


Fig. 2. Multi-Controller Architecture (The gray parts of the multi-controller show the active elements. The active controller module changes from time to time.)

4 Multi-controller on Reconfigurable Hardware

We assume that every *controller module* CM_i of fig. 2 can be represented by a linear controller. Therefore, we can use our framework to implement each *controller module* as a dedicated circuit. By using the standard approach, all modules would be instantiated in parallel. This leads to a design with a high area and power consumption, proportional to the number of controller modules. This number can be quite large, making this approach impossible due to the physical bounds of the FPGAs. Moreover, this design is very inefficient since only one controller module is active at a time (fig. 2). Considering just the resources for the controller modules, $n - 1$ of n resources stay inactive in contrast to efficient designs where almost all resources are active all the time. Our system proposed in the next section uses partial reconfiguration to overcome this drawback.

4.1 Architecture

To map the multi-controller architecture (fig. 2) to our prototyping system we divide the FPGA into three parts (fig. 3). In the middle the static module (*SM*) can be seen, which is not reconfigured and operates all the time. It contains the *supervisor*, a multiplexer and the communication interface to the plant. The left and the right parts of the FPGA are slots for reconfigurable modules (*RM-Slot-A*, *RM-Slot-B*). In these areas a controller module can be implemented. Like in the original architecture (fig. 2) the supervisor and the controller modules get their input from the output of the plant. The calculated outputs of the currently active controller module are propagated back to the inputs of the plant. The active controller module is set by the supervisor. The initial controller module is configured into the left slot (*RM-Slot-A*) of the FPGA and the supervisor activates this controller by setting the MUX to A. If the supervisor decides to switch to a new controller module CM_k , it request the configuration manager to reconfigure the right slot (*RM-Slot-B*). After the reconfiguration process is finished the MUX is set to B. Every time a switch event occurs the supervisor will toggle to the other target slot. An example sequence is illustrated in the timing diagramm in fig. 4.

In our prototyping environment the task of the host is to simulate the plant and to store the partial bit-streams of the controller modules (fig. 3). In a final product, the control system will consist of the FPGA, a flash memory to store the partial bit-streams, the configuration manager which is a simple state machine and the I/O converters to connect the system to the plant.

The reason the system has two reconfigurable slots (instead of just one) is the relatively slow reconfiguration time of FPGAs which is in the range of milliseconds. If the sampling frequency of the controller is in the range of kHz, it might not be possible to reconfigure a controller module within one sampling interval. With our method, the new CM can be reconfigured and synchronized

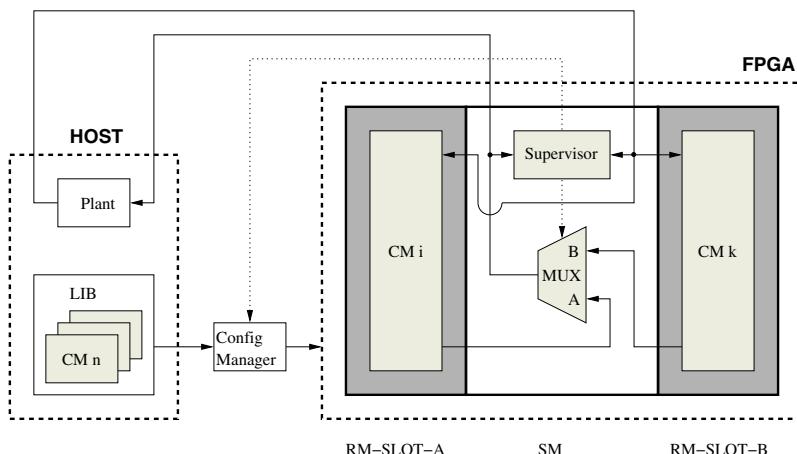


Fig. 3. Multi-Controller based on Reconfigurable Hardware

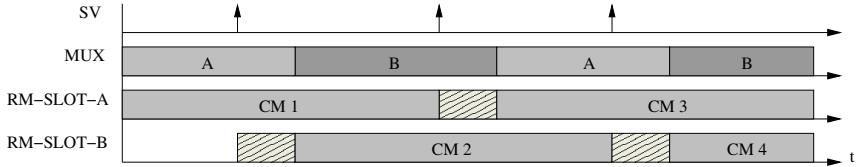


Fig. 4. State of the multi-controller system over time (The first row shows the reconfiguration requests generated by the *supervisor* and the second row shows the state of the MUX. The content of the two reconfigurable slots is shown in row three and four, where the fasciated area refers to the reconfiguration phase.)

while the other one controls the plant. This enables the switching between the two controllers within two sampling points, additionally smooth fading between the controller outputs becomes possible.

The FPGA area of our system is two times the area of a controller module A_{RM} plus the area of the static module A_{SM} (eq. 4). As a boundary condition, the computation time t_c of a CM must be less than the sampling period. In an optimized implementation of a CM the FPGA-area/computation-time trade-off is chosen in such a way, that the area is minimized while the computation time is close to the sampling period. Approximatly, we can assume a linear trade-off between the area and the computation time of a controller module since we can highly parallelize the control algorithm (eq. 5).²

$$A_{twoslotssystem} = 2A_{RM} + A_{SM} \quad (4)$$

$$A_{RM} \approx c_1 \frac{1}{t_c}, \quad t_c \leq T \quad (5)$$

A system which uses only one RM slot and the static module SM can be an alternative in some cases. Since the controller module slots require the mass portion of the FPGA area, using only one slot seems to be a great reduction of the overall system resource cost. However, if the application requires a new controller output \mathbf{y} within every sampling interval T , the new CM must be configured and compute the new output during one period. The reconfiguration time t_r plus the computing time t_c must be less than the period. The FPGA area of such a system is the area of one reconfigurable module slot A_{RM} plus the area of the static module A_{SM} (eq. 6). The reconfiguration time t_r increases approximatly linear with the module size (eq. 7).³

$$A_{oneslotssystem} = A_{RM} + A_{SM} \quad (6)$$

$$A_{RM} \approx c_1 \frac{1}{t_c}, \quad t_r + t_c \leq T, \quad t_r \approx c_2 A_{RM} \quad (7)$$

Finally, it depends on the reconfiguration speed and the sampling period, which system should be preferred. When t_r exceeds T , or when fading between the controller modules is required, the one slot solution cannot be used.

² c_1 is an application specific constant

³ c_2 is an FPGA device specific constant

4.2 Partial Reconfiguration Design Flow

Partial reconfiguration is not part of a standard design flow, but is the subject of current research. It is FPGA architecture dependent and many boundary conditions have to be considered. In this work we target the XILINX Virtex FPGAs [2]. These devices can be partially reconfigured column wise. During a complete reconfiguration a bit-stream with configuration data for all logic resources of the gate array is send to the device. In the partial reconfiguration mode the bit-stream contains just the configuration data for one or more columns of the gate array. The other columns keep their old configuration and their logic blocks stay active during the reconfiguration.

The design of a system using partial reconfiguration starts with the partitioning of the system into modules that should be reconfigured at runtime. Much theoretical research in the area of reconfigurable computing focuses on the systematic partitioning of the system and the optimal scheduling and placing of the reconfigurable modules, e.g. [3, 5, 6, 14]. Due to the complexity of the design flow, we started with a manual partitioning as described in section 4.1.

To implement the system the bit-streams for all modules have to be created during the design time. In the standard design flow the design entry is HDL and the synthesis tools are employed to generate the bit-stream. To generate partial bit-streams two methods exist. The first omits a synthesis by the standard tools and uses tools that allow access to the FPGA resources on a low abstraction level [1, 16]. The second uses the standard synthesis tools in a modified, restricted design flow [11]. The latter, which we used in our work, is described in the following.

The basic idea of the modified design flow for partial configuration is the constraint of the place and route process in that way, that all logic of a module is placed inside a restricted area. For this purpose, a net-list for every module is synthesized. For each net-list the place and route process is invoked, with the constraint to use only FPGA resources in a defined area. Some restrictions that have to be met to generate a reconfigurable module for XILINX Virtex FPGAs are listed below:

- The area of two modules that should work simultaneously must not overlap.
- The height of the module area is always the full height of the FPGA. (This is due to the column wise reconfiguration of Virtex FPGA)
- The position to place a module is fixed and determined during design time. Relative placement during runtime is not supported. (e.g. currently, a module bit-stream for the left FPGA half cannot be used for the right half)
- An active module occupies all resources in its area. That is, other modules cannot use logic blocks, routing resources, block ram, I/O pins or any other resource of the active module.
- Communication between adjacent modules is done via fixed well defined connections at the modules boarders.
- Communication between not adjacent modules can be realized by feed through wires of the between module.

- The modules must not share any other connection (except the clock nets, which are global resources). That means, that the modules must not share one reset signal or constant nets like VCC or GND.

4.3 Implementation and Results

For the implementation of the system we used our self developed flexible and modular rapid prototyping environment RAPTOR2000 [9]. In this work, the RAPTOR2000 motherboard hosts a module that carries a XILINX Virtex 800 FPGA. A PCI-bridge implements the connection to a host computer. A configuration manager implemented in a CPLD supports the fully and partially configuration of the FPGA. The bit-streams are taken from the host-memory and fed to the SelectMap interface of the FPGA allowing a reconfiguration rate of 50 MByte/s. To communicate between the host computer and the FPGA a graphical as well as an C-library interface exist. They allow to access the internal local bus which is connected to the FPGA. In the FPGA a module called *local-bus access* implements the bus-protocol and allows the host to access internal registers of the FPGA design. This logic as well as the *supervisor* are part of the static module.

Our first implementation differs from the proposed system in section 4.1. We use the discussed solution with one RM-slot instead of two RM-slots. While we still propose the three slot solution, many challenges and problems of partial reconfiguration can be explored using this first prototype. Our example controller (implementing the control of an inverse pedulum) is a module with three inputs of 16 bit width, and two inner states and three outputs of 32 bit width. The communication between the SM and the RM is done via special bus-macros. It consists of a 32-bit data bus from SM to RM, a 32-bit data bus from RM to SM and an 32-bit address bus from SM to RM. The module uses 1066 FPGA slices which is approximately 10 percent of the Virtex 800 FPGA logic resources. From this it follows that either smaller FPGAs can be used or more complex controllers can be implemented. The worst net delay is about 13.5 ns which results in a maximum clock frequency of more than 70 MHz. Since the computation time of the module is 56 clock cycles, it can operate with a sampling frequency above 1 MHz. There are just few applications requiring such high sampling rates. Nevertheless, this is the performance archived by our CM-architecture of section 2.1 for the dimensions mentioned above. Other area/time trade-offs can be implemented which reduce the performance to the timing requirements of the application and minimize the area of the CMs. The area we reserved for the RM covers about the half of the FPGA. The resulting bit-stream has a size of 274 kB which leads to a reconfiguration time of 5.5 ms using the XILINX SelectMAP interface at 50 MHz. However, the slice count assumes a much smaller area which might lead to an reconfiguration time of about one millisecond. The separated modules RM and SM as well as the bus-macros can be seen in the screenshot of the routed FPGA design in fig. 5.



Fig. 5. Screenshot of Routed FPGA Design (left: reconfigurable module slot, right: static module, middle: bus-macros for communication)

5 Conclusion

In this paper we presented an architecture for hardware based multi-controller systems. We motivated the implementation of controllers using reconfigurable logic as a good alternative to control algorithms executed on processors. After introducing the class of digital linear controllers we presented our framework to create them in reconfigurable logic. Later, we introduced control applications that follow the multiple-model approach. They target systems whose dynamic behavior is modeled as a process working in a limited set of operating regimes. While controllers derived by formal design methods result in low performance, a set of controllers optimized for different operating regimes is more suitable. In these multi-controller architecture, only one controller module is active at a time. In section 4 we enhanced our framework for linear controllers to multi-model controller systems. We showed, how partial dynamic reconfiguration of the FPGA led to an efficient design. In the architecture of our prototype, the host simulates the plant while the FPGA implements the controller. Therefore the FPGA is divided into three parts. The static part realizes communication between the host and the controller as well as the administration of the reconfiguration process. The other parts are reserved for controller modules and can be reconfigured. This architecture can guarantee that always one controller module is active, while the other slot is being reconfigured. The results of the implementation proved our concept and showed that reconfigurable hardware can be used to implement quite complex controller system with high performance. Also this is an example for close to reality employment of partial dynamic reconfiguration, increasing the efficiency of the system.

In our future work we will enhance the implementation to the two slot version and extend the supervisor allowing more intelligent self reconfiguration. In

addition we will enhance our framework for the controller modules to allow to chose different FPGA-area/computation-time trade-offs. Also our research will target the hardware implementation of non-linear controllers and floating point solutions.

References

1. *XILINX ISE 5 Software Manuals - FPGA Editor*, 2002.
2. *XILINX Virtex Data Sheet, DS003(1-4)*, 2002.
3. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast template placement for reconfigurable computing systems. *IEEE Design and Test of Computers*, Mar. 2000.
4. M. Bednara, K. Danne, M. Deppe, O. Oberschelp, F. Slomka, and J. Teich. Design and implementation of digital linear control systems on reconfigurable hardware. *EURASIP Journal on Applied Signal Processing*, 2003.
5. C. Bobda. IP based synthesis of reconfigurables systems. In *Tenth ACM International Symposium on Field Programmable Gate Arrays(FPGA 02)*, page 248, Monterey, California, 2002. ACM/SIGDA.
6. C. Bobda. Temporal partitioning and sequencing of dataflow graphs on reconfigurable systems. In *International IFIP TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002)*, pages 185–194, Montreal, Canada, 2002. IFIP.
7. K. Danne. Implementierung digitaler Regelungen in hardware. Bachelor's thesis, University of Paderborn, 2000.
8. Johanson and Murray-Smith. The operating regime approach to nonlinear modelling and control. *Multiple Model Approaches to Modelling and Control*, 42, 1997.
9. H. Kalte, M. Porrmann, and U. Rückert. A prototyping platform for dynamically reconfigurable system on chip designs. In *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*. Hindawi, 2002.
10. R. Kasper and T. Reinemann. Gate level implementation of high speed controllers and filters for mechatronic systems. *Mechatronic Workshop*, 2000.
11. D. Lim and M. Peattie. *Xilinx Application Note XAPP290: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations*, May 2002.
12. A. Morse. Control using logic-based switching. *Trends in Control, Springer, London*, 1995.
13. Narendra and Balakrishnan. Adaptive control using multiple models: Switching and tuning. *Yale Workshop on Adaptive and Learning Systems*, 1994.
14. J. Teich, S. Fekete, and J. Schepers. Optimization of dynamic hardware reconfigurations. *The J. of Supercomputing*, 19(1):57–75, May 2000.
15. A. van Breemen and T. de Vries. An agent-based framework for designing multi-controller systems. *Proc. of the Fifth International Conference on The Practical Applications of Intelligent Agents and Multi-Agent Technology*, pp. 219-235, Manchester, U.K, Apr. 2000.
16. www.xilinx.com/products/software/jbits.

Efficient Modular-Pipelined AES Implementation in Counter Mode on ALTERA FPGA

François Charot¹, Eslam Yahya², and Charles Wagner¹

¹IRISA/INRIA

Campus de Beaulieu

35042 Rennes Cedex, France

charot@irisa.fr, wagner@irisa.fr

²Information Technology Institute-ITI

Benha High Institute of Technology-BHIT

241-El Haram Street-Cairo/El Estad Street-Benha-Kaliobia, Egypt

esyahya@iti-idsc.net.eg

Abstract. This paper describes a high performance single-chip FPGA implementation of the new Advanced Encryption Standard (AES) algorithm dealing with 128-bit data/key blocks and operating in Counter (CTR) mode. Counter mode has a proven-tight security and it enables the simultaneous processing of multiple blocks without losing the feedback mode advantages. It also gives the advantage of allowing the use of similar hardware for both encryption and decryption parts. The proposed architecture is modular. The architecture basic module implements a single round of the algorithm with the required expansion hardware and control signals. It gives very high flexibility in choosing the degree of pipelining according to the throughput requirements and hardware limitations and this gives the ability to achieve the best compromised design due to these aspects. The FPGA implementation presented is that of a pipelined single chip Rijndael design which runs at a rate of 10.8 Gbits/sec for full pipelining on an ALTERA APEX-EP20KE platform.

1 Introduction

With the very large growth of network applications, the issue of security is becoming one of the most important aspects in network design. High security applications running on the Internet require hardware implementation of fast and secure protocols. In a protocol like IPsec, developed to secure the Internet traffic, there is a need for underlying ciphering algorithms. One of the best suitable is the Rijndael algorithm selected by NIST as the Advanced Encryption Standard (AES) in October 2000 and approved in the summer of 2001 as the Federal Information Processing Standard (FIPS) [5]. The suitability of this algorithm comes from its symmetry and high key agility as well.

This paper presents a fully-modular partially-pipelined AES architecture, dealing with 128-bit data/key blocks and operating in Counter (CTR) mode. CTR mode of operation is fully parallelizable, and has a proven-tight security. It also gives the advantage of allowing the use of similar hardware for both encryption and decryption. This architecture is suitable for secure network applications, especially when high

data rates are required. The proposed architecture exploits pipelining. Pipelining gives the ability to achieve higher throughput by processing multiple input blocks simultaneously.

The proposed architecture is modular. The architecture basic module implements a single round of the algorithm with the required expansion hardware and control signals. It gives very high flexibility in choosing the degree of pipelining according to the throughput requirements and hardware limitations, that gives the ability to achieve the best compromised design due to these two aspects. The key generation part of such an architecture is of major importance. Key agility, that is the possibility of calculating keys on the fly, is a necessity in most practical applications like IPsec, encrypted routers and secure ATM networks. In case of partial pipelining, key agility is a complicated part, especially from the synchronization point of view due to calculating multiple keys on the fly at the same time.

AES implementation needs storage RAMs, look-up table ROMs, registers, shift registers and simple boolean operations, which make the fine grain structure of Field Programmable Gate Arrays (FPGAs) quite suitable. Moreover FPGAs allow inherent parallelism and flexibility of the algorithm to be easily exploited with a fast development time. The proposed modular architecture has been implemented on an ALTERA APEX FPGA [1]. The first experiments, based on a full pipelining of the algorithm, allow a rate of 10.8 Gbits/sec to be achieved. Section 2 of this paper describes the Rijndael algorithm with a focus on the CTR mode of operation. The design of the pipelined Rijndael implementation is outlined in section 3. Performance results are given in section 4. Finally, concluding remarks are provided in section 5.

2 Rijndael Algorithm Overview

2.1 Basic Algorithm Features

The Rijndael algorithm is a block cipher using 128, 192 and 256-bit input/output blocks and keys [2]. The size of blocks and keys can be chosen independently. The encryption is done in a certain number of rounds, which may vary between 10, 12, and 14, and it depends on the block length and key length chosen.

In a 128-bit block encryption, plain text and cipher text are processed in blocks of 128 bits. The transformation considers the input data block as a four column rectangular array of 4-byte vectors called *State*. The Rijndael cipher algorithm consists of four basic operations:

- ByteSub: non-linear substitution based on Galois Field applied to each *State* byte.
- ShiftRow: cyclically shifting the last three rows of the *State* by different offsets.
- MixColumn: each *State* column is processed based on Galois Field polynomial multiplication.
- AddRoundKey: each round key byte is added (Xored) to the corresponding *State* byte.

The single round structure is repeated 10 times. An additional initial round is applied in which the original key is added to the input data. The first 9 rounds perform all the four transformations described above whereas the final round omits the MixColumn operation. The key entering the cipher is expanded so that a different

sub-key (round key) is created for each round of the algorithm. This round key generation is a process consisting of S-boxes, XORs and word rotation operations.

2.2 Modes of Operation

As described in [3], different modes of operation may be used in conjunction with any symmetric key block cipher algorithms. Modes of operation can be divided into two main categories: non-feedback modes, in which the main advantage is from architecture point of view, since any kind of parallelism is enabled and feedback modes, in which the main advantage is from security point of view but slow down the hardware implementation due to loop carried dependencies.

Among the different block cipher modes of operation proposed in the NIST recommendation detailed in [3], Counter (CTR) mode has a proven-tight security and it enables the simultaneous processing of multiple blocks without losing the feedback mode advantages. It also gives the advantage of allowing the use of similar hardware for encryption and decryption parts.

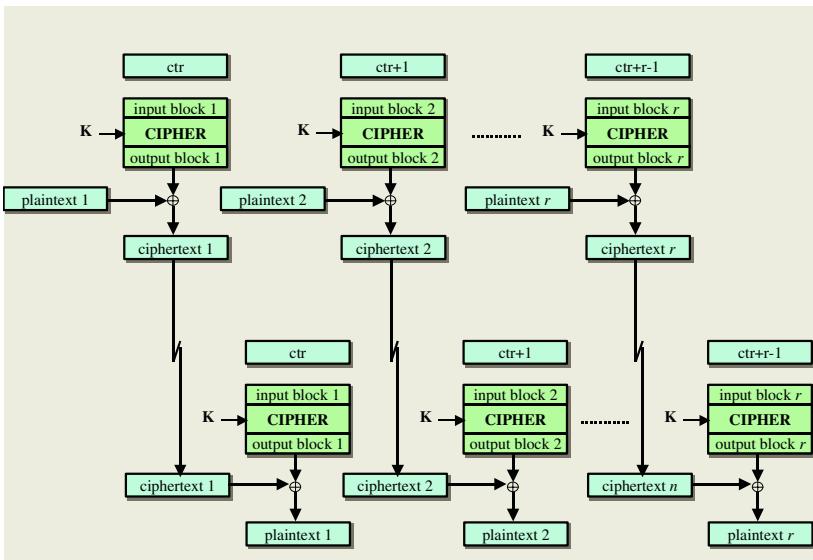


Fig. 1. Encryption and decryption process in CTR mode

In CTR mode, as illustrated in Figure 1, the cipher is applied to a set of input blocks, called counters, to produce a sequence of output blocks that are exclusive-ORed with the plaintext to produce the ciphertext. The sequence of counters must have the property that every block in the sequence is different from every other block.

3 Hardware Architecture

A block cipher consists generally of a single round module, which is applied to a data block multiple times, with a different sub-key applied in each round. The optimum hardware implementation for Rijndael is a single round with agile sub-key generation. Key agile implementation is preferable to stored key implementation due to the flexibility it allows and also to the reasonable size of sub-key generation. When high throughput is required, the single round and associated sub-key generation have to be pipelined. The pipelining greatly improves throughput. The throughput is proportional to the number of duplicated rounds at a price of an increase of silicon area.

The AES hardware architecture presented in this paper implements a 128-bit data/key Rijndael algorithm. In order to implement full encryption, 10 rounds of transformation have to be performed [2], [5]. Since high security applications and fast protocols implementations are aimed, the architecture design has been guided by the following decisions:

- The use of pipelining which allows the throughput constraints to be adapted according to the application requirements and which is mandatory when high throughput is required.
- Key agility, which is a demand for all practical applications.
- The use of CTR mode, which is one of the best known modes since it has significant efficiency advantages over the standard encryption modes without weakening the security.
- Modularity of the architecture which gives high flexibility and reconfigurability.

Through the next subsections, every aspect introduced above is discussed in details.

3.1 Pipelining

The simplest definition for pipelining is using multiple processing units running in parallel on multiple input blocks. A number of different architectures can be considered when designing encryption algorithms [4]. They are generally described as follows. Iterative looping, where only one round is designed. Loop unrolling involves the unrolling of multiple rounds. Pipelining consists in replicating the round and placing registers between each round to control the flow of data. When the round is complex, sub-pipelining can be used. Sub-pipelining decreases the pipeline delay between stages, but it increases the number of clock cycles required to perform the encryption.

Two pipelining approaches are considered here:

- Partial pipelining which consists in a partial replication of the round module, with a registering of the intermediate data between rounds and the need to iterate a given number of times on the pipeline structure in order to calculate ten rounds according to the number of replications.
- Full pipelining, a specific form of a partial pipeline, which consists in replicating the round module ten times, a ten-pipeline stage structure implements all rounds of the algorithm.

The higher is the degree of pipelining, the higher is the silicon area and the power consumption. So, very careful compromising between the required throughput and the

chosen degree of pipelining has to be done. For efficiency reasons (throughput versus hardware utilisation), a 5-stage partial pipelined module has been chosen. This pipelined processor requires two iterations for producing an output cipher text.

3.2 Key Agility

The Rijndael processor is fed by a 128-bit input key but every round of the algorithm needs its own round key. That means that the original key has to be expanded. There are two methods to do that:

- Calculation and storage, the key is expanded once and stored in a buffer, then used for all coming data blocks until a reset operation occurs.
- On the fly calculation, the key is expanded from step to step for every new coming data block, producing 10x128-bit key rounds.

The first method is common and it is used in most of the AES implementations proposed so far. It is simpler and faster but not practical. It supposes that all coming data will be encrypted using the same key. The second method allows a new key to be used with every new coming data block. This is required with most applications like for instance encrypted routers. But this method is slower and more hardware consuming since the total delay of the round is the sum of the round transformation delay and of the key expansion delay.

As reported in most papers, like in [6], the key expansion delay is much higher than the round transformation delay itself. That means that key agility has a price from the speed point of view, but it is a necessity in all practical applications.

3.3 Modes of Operation

The different standardized modes of operations are summarized in table 1. They are divided into two main categories: non-feedback modes and feedback modes.

Table 1. Modes of operation

	Non-feedback mode	Feedback mode
Security point of view	Same key + same data =	Same key + same data =
Architecture point of view	Pipelining is enabled	Pipelining is disabled

Table 1 shows that fast and secure implementation of AES is hard to achieve, but CTR mode solves the conflict. Its main advantages are the following [3], [7], [10]:

- Hardware efficiency since any kind of parallelism is enabled;
- Pre-processing since encryption can be done even before the input plaintext is known;
- Provable security since analysis proved that CTR mode has a tight security;

- Simplicity, important advantage of this mode with AES is that, typical hardware for encryption and decryption can be used, which prevent implementing different hardware for both, especially that the encryption is more complex.

3.4 Modularity

With a deep look to the Rijndael algorithm, it may be noticed that it is a modular one. The full algorithm can be implemented as repeated identical modules and the advantages of that way of thinking are:

- Efforts are done in designing a small module;
- Highly reconfigurable design where any modification in the system specifications can be achieved by only modifying the basic module;
- Flexible degree of pipelining, where it is very easy to instantiate modules exactly as necessary to achieve the required throughput.

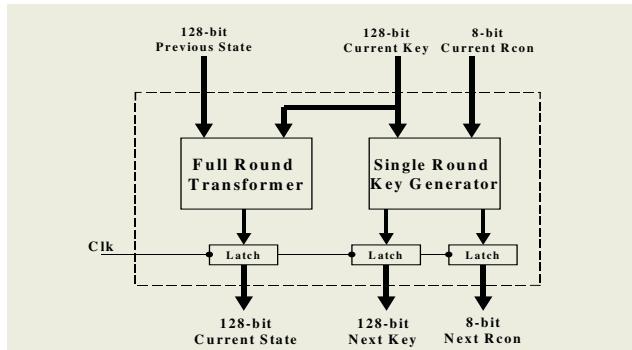


Fig. 2. The basic module block diagram

The implemented module, as illustrated in Figure 2, has only one disadvantage. Some unnecessary hardware will be added –for example the round constant $Rcon$ [2] is computed and not used as a constant– which means more hardware is required but the gain is very valuable. Round transformation operates in parallel with key generation, and during a given iteration the round key required by the next iteration is computed.

3.5 System Architecture

Two architectures are introduced. The first one corresponds to the initial design based on the choices previously explained, the second is characterized by some enhancements which greatly affect the final system throughput. In the next subsections we will discuss these two architectures, their differences and their performance results.

3.5.1 Initial System Architecture

Figure 3 introduces the full block diagram of the proposed AES processor. The main component in the system is the 128-bit pipelined Rijndael module. This module is composed of 5 instances of the basic module described in Figure 2. It implements the ten rounds. An additional Xor module is introduced before the 5-stage module instance to perform the initial key addition operation. The basic module implements the four transformations described in 2.1. The S-box transformation is implemented as a 256-entry 8-bit wide lookup table. This transformation is mapped into the dedicated ALTERA APEX Embedded System Block (ESB) resources. The other transformations can be reduced to a series of lookup table operations and bitwise XOR operations, which are easily implemented in ALTERA APEX logic elements (LE).

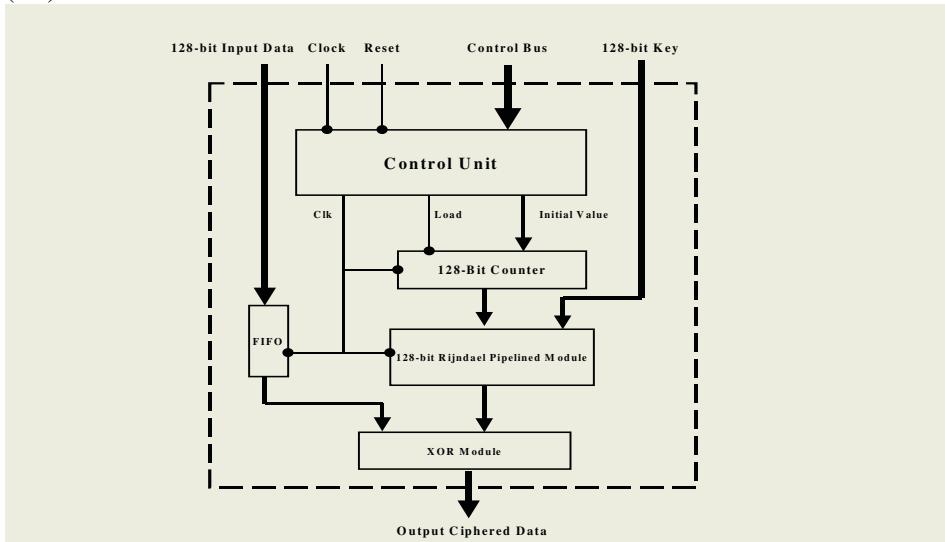


Fig. 3. AES system architecture

The control unit defines the initial value of the counter, *Reset*, *load* and *Clock* control signals. The Rijndael pipelined module is a 5-stage pipelined Rijndael processor. Registers and FIFO allow the synchronization requirements to be fully satisfied.

3.5.2 Optimised System Architecture

By analysing the architecture, some possible enhancements were studied. The floor plan analysis shows that all critical paths are located between I/O pins and internal logic. This has the effect of affecting the internal logic propagation delay since the later includes the input set-up time.

The solution consisting in inserting register set directly after/before I/O allows the critical path to be broken. Even if it introduces an additional pipelining stage, it greatly reduces the critical path delay.

Considering the concept above and by reviewing synthesis results of table 3, another enhancement can be achieved. Two clock speeds can be used, a first one for

the I/O and a second one for the core. In case of a 5-stage partial pipelining, good results can be achieved by forcing the core to use a clock with a twice frequency of the I/O clock. It is clear from Figure 4 that the core will be triggered two times every one I/O operation.

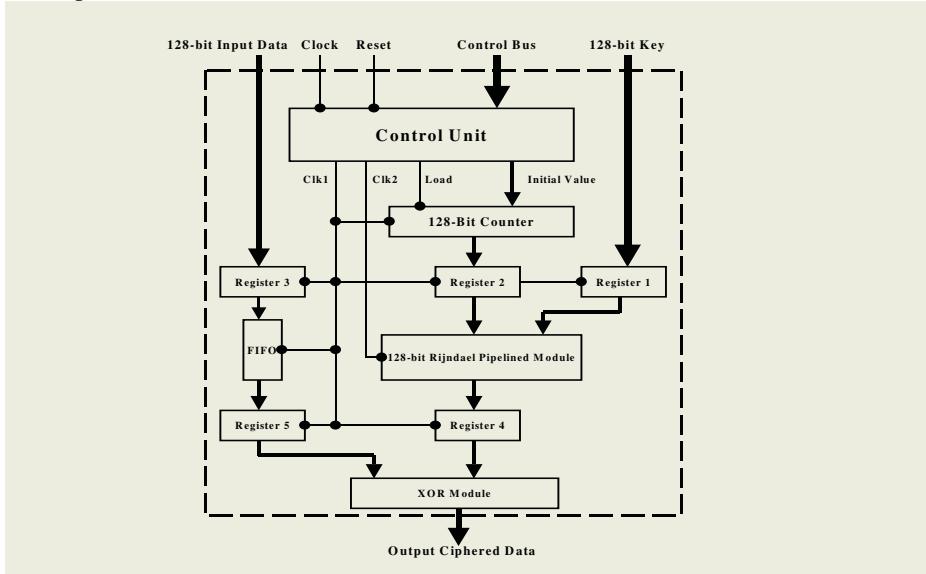


Fig. 4. Enhanced architecture

4 Performance Results

The Rijndael processor proposed in this paper is implemented using Leonardo Spectrum as the synthesis tool and Quartus II as the Placement and Routing tool, targeting ALTERA APEX FPGA [1]. The design is expressed in VHDL language. As illustrated in figure 2, the module is composed of two main components: the full round transformer and the round key generator. Both components run in parallel and feed the next stage. So the slowest clock of both components corresponds to the main system clock. In the next subsections, performance results for both initial and optimised architectures are presented.

4.1 Initial Architecture

Table 2 summarizes the synthesis results for the initial architecture. Since the critical path is as long as 25 ns, the system could operate under a clock speed of 40 MHz. When the cipher and the key are 128-bit, the throughput is 5.12 Gbits/sec for full pipelining. A rate of 2.56 Gbits/sec can be achieved in case of half pipelining.

Table 2. Initial architecture synthesis results

	Hardware	Critical Path Delay
Full Round Transformer	16 ESB, 356 LE, 128 FF	19.687 ns 50 MHz
Round Key Generator	4 ESB, 131 LE, 141 FF	25 ns 40 MHz

4.2 Optimised System Architecture

Table 3 summarizes the synthesis results for the optimised architecture. For a 5-stage partial pipelining, the achieved throughput is 7.1 Gbits/sec. It is even higher than the throughput achieved with full pipelining of the initial architecture. In this case the core clock frequency is 111 MHz and the I/O clock will be half of that frequency.

Table 3. Optimised architecture synthesis results

	Hardware	Critical Path Delay	
		Core	I/O
Full Round Transformer	16 ESP, 740 LE, 512 FF	8.934 ns 111.93 MHz	10.549 ns 94.79 MHz
Round Key Generator	4 ESP, 272 LE, 272 FF	7.225 ns 138.4 MHz	11.8 ns 84 .7 MHz

For full pipelining, the throughput is 10.8 Gbits/sec. This throughput is determined by the path delay of the I/O in the key generator part.

4.3 Performance Comparison

In order to evaluate our work we did a comparison with the fastest reported designs. In this comparison we divided the known designs into two categories key agile and non-key agile designs, because of the importance of the key agility property and its great effect on the throughput.

The ASIC chip proposed in [6] is the only one we know that reports a key agile design. It achieves a throughput of 1.82 Gbits/sec in case of full pipelining. The fastest known designs are reported in [8] (7 Gbits/sec), and in [9] (7.68 Gbits/sec), but none of them offers the key agility property.

From the point of view of silicon area, the proposed design is more memory block consuming than the one presented in [8]. The latter uses 82 Xilinx BRAMS, which corresponds to 164 ALTERA ESB whereas we used 200 ESB (for full pipelining). The difference comes from the part devoted to the key generation part of our design (key agility).

5 Conclusion

This paper describes a high performance FPGA implementation of the Rijndael algorithm. This 128-bit data/key encryption design performs at a data rate of 10.8 Gbits/sec for full pipelining and it can be considered as one of the fastest key agile AES design. It is 6 times faster than the fastest reported key agile chip. In comparison with the fastest reported AES chip, a rate of 7.1 Gbits/sec is achieved in case of half pipelining. The achieved throughput is nearly the same as in [8], [9] with half pipelining while they used full pipelining, and the design is 1.5 times faster for full pipelining. Modularity gives very high flexibility and great compromising between degree of pipelining and required throughput without any core modifications. Moreover, this design can be considered as the first one reporting AES chip running in CTR mode.

For future work, more detailed analysis and estimations especially on the basic module implementation should be done. Sub-pipelining can be used to enhance the core performance and hand review for the floor plan can reduce the I/O path delay. Combinational implementation of the S-box transformation is also under study.

Acknowledgments

This work has been partially funded by the French ministry of Defence under research contract of French procurement agency (DGA).

References

1. Altera APEX20K, APEX 20K Devices: System-on-a-Programmable-Chip Solutions, <http://www.altera.com>.
2. J. Daemen, V. Rijmen, The Rijndael Block Cipher: AES Proposal, First AES Candidate Conference (AES1), August 1998.
3. M. Dworkin, Recommendation for Block Cipher Modes of Operation, NIST special Publication 800-38A, December 2001.
4. A.J. Elbirt, W. Yip, B. Chetwynd, C. Paar, An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists, The Third AES Candidate Conference (AES3), April 2000.
5. FIPS Publication 197, Specification for the Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, U.S. DoC/NIST, November 2001.
6. H. Kuo, I. Verbauwhe, Architectural Optimisation for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm, CHES 2001, LNCS 2162, 2001.
7. H. Lipmaa, P. Rogaway, D. Wagner, Comments to NIST concerning AES Modes of Operations CTR-Mode Encryption, Symmetric Key Block Cipher Modes of Operation Workshop, October 2000.
8. M. McLoone, J.V. McCanny, High Performance Single-Chip FPGA Rijndael Algorithm Implementations, CHES 2001, LNCS 2162.
9. M. Alam, W. Badawy, G. Jullien A Novel Pipelined Threads Architecture for AES Encryption Algorithm, ASAP 2002, July 2002.
10. Report on the Symmetric Key Block Cipher Modes of Operations Workshop. (NIST), October 2000.

An FPGA-Based Performance Analysis of the Unrolling, Tiling, and Pipelining of the AES Algorithm

G.P. Saggese¹, A. Mazzeo¹, N. Mazzocca², and A.G.M. Strollo¹

¹ University of Naples Federico II, Via Claudio 21, 80125 Napoli, Italy

² Second University of Naples, Via Roma 29, 81031 Aversa, Italy

{saggese,mazzeo,n.mazzocca,astrollo}@unina.it

Abstract. In October 2000 the National Institute of Standards and Technology chose Rijndael algorithm as the new Advanced Encryption Standard (AES). AES finds wide deployment in a huge variety of products making efficient implementations a significant priority. In this paper we address the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit keys. We discuss the effectiveness of several design techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations (also in the case of feedback modes of operation) to explore the design space. Using these techniques, four architectures with different level of parallelism, trading off area for performance, are described and their implementations on a Virtex-E FPGA part are presented. The proposed implementations of AES achieve better performance as compared to other blocks in the literature and commercial IP core on the same device.

1 Introduction

Symmetric-key block ciphers are important in many cryptographic systems. Individually they provide confidentiality which means keeping information secret from all but those who are authorized to see it. As a basic building block, they allow construction of pseudo-random number generators, stream ciphers, message authentication code, and hash functions. These primitives have a crucial role in many real-world applications. In October 2000 the National Institute of Standards and Technology (NIST) chose Rijndael algorithm [1] as the new Advanced Encryption Standard (AES) [2]. This standard is become effective on May 2002. The use of AES and the AES replacement of DES and triple DES, is encouraged to provide the desired security of electronic data in commercial and private organizations.

Even if most new algorithm designs cite efficiency in software as a design objective, the software implementations of symmetric algorithms are often computationally expensive. Therefore in many applications the use of hardware-based solutions has become unavoidable in order to meet high performance requirements. Reconfigurable devices, like Field-Programmable Gate Arrays (FPGA),

are a promising alternative for the implementation of block ciphers, since they include many advantages of software implementations (like algorithm agility, algorithm modification capability [9], and cost efficiency) with physically security and high throughput of an Application-Specific Integrated Circuits (ASIC). In order to achieve maximum performance on FPGA, designs making use of architecture-specific features are required, which means that circuit implementations are not portable between different FPGA technologies.

In this paper we address the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit ($128b$) keys. We discuss the effectiveness of several techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations (also in the case of feedback modes of operation) to explore the design space, allowing to tradeoff area for performance. The proposed AES blocks achieve better performance (with respects to area and throughput metrics) than other implementations on the same Xilinx device, available in the technical literature and as commercial IP cores. The application of the above-mentioned techniques allow an effective design space exploration, since our least area (iterative) design requires 446 Virtex-E slices and 10 Select BlockRAM delivering 1 gigabit per second (Gbps) encryption rate, and our fastest AES block (fully unrolled and deeply pipelined) reaches a throughput of 20,3 Gbps.

The rest of the paper is organized as follows. In Section 2 we describe the AES algorithm and the block cipher modes of operation. Section 3 addresses main issues related to the design and the implementation of AES and discusses our solutions. Section 4 describes four architectures differing in unrolling and pipelining level and presents their floorplan aware implementations. Section 5 reports performance results and analyzes area vs throughput trade-offs, with respect to other academical and commercial implementations. Finally, Section 6 concludes the paper with some final remarks.

2 AES Encryption Algorithm and Modes of Operating

The AES algorithm selected by NIST as the successor of the DES encryption algorithm is Rijndael [1]. In AES standard [2] the length of the data block is always equal to $128b$ and so the parameter Nb , which represents the number of $32b$ words composing a data block, is 4. As far as different security requirements are concerned, the key length for AES can be chosen as either $128b$, $192b$, or $256b$ long, and the corresponding parameter Nk is 4, 6, or 8 respectively. The pseudo-algorithm for the AES encipherment and for the **KeyExpansion** procedure with a $128b$ or $192b$ key are reported in the left and right boxes of Fig. 1, respectively. The main computation of AES encryption is a loop repeated Nr times, where Nr is equal to either 10, 12, or 14, based on the key size. The cipher algorithm has the goal of obscuring (in a reversible manner) the plain text with a set of $Nb(Nr + 1)$ $32b$ words w (named *sub-keys*) derived by the expansion of the **key**. AES interprets the incoming data and the intermediate result (**state**) as a 4×4 array of $8b$ words (**byte**). The processing of a $128b$ plain text starts

```

Cipher (byte in[4*Nb],
       word w[Nb*(Nr+1)], byte out[4*Nb])
begin
    byte state[4,Nb] = in
    AddRoundKey(state, w[0, Nb-1])
    for round = 1 step 1 to Nr-1
        SubBytes(state)
        ShiftRows(state)
        MixColumns(state)
        AddRoundKey(state,
                    w[round*Nb, round*Nb+Nb-1])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state,
                w[Nr*Nb,Nr*(Nb+1)-1])
    out = state
end

```

Initial Round Round Final

```

KeyExpansion (byte key[4*Nb],
              word w[Nb*(Nr+1)])
begin
    word temp, i = 0
    while (i < Nk)
        w[i] = (key[4*i], key[4*i+1],
                 key[4*i+2], key[4*i+3])
        i = i+1
    end while
    while (i < Nb*(Nr+1))
        temp = w[i-1]
        if (i mod Nk = 0)
            temp = SubWord(RotWord(temp))
            xor Rcon[i/Nk]
        end if
        w[i] = w[i-Nk] xor temp
        i = i+1
    end while
end

```

Fig. 1. The AES Cipher algorithm in the case $Nb = 4$ and the KeyExpansion procedure particularized for $Nk = 4$ and 6 , in the left and right box, respectively.

copying it into the **state**. The **state** is mixed with the first part of the sub-keys by means of an **AddRoundKey** step (*InitialRound* in Fig. 1). Then the *Round* transformation, involving the four basic steps of Fig. 1, is applied $Nr - 1$ times. At last the *FinalRound* block, which differs slightly from the *Round* since it does not include **MixColumns**, is executed. The basic operations which compose AES are: 1) **SubBytes** – each byte of the **state** is transformed using an s-box function which is composed of two transformations over $GF(2^8)$, namely an inversion and an affine transformation (i.e. a matrix per vector multiplication); 2) **ShiftRows** – each row of the matrix **state** is circularly shifted with a row-dependent amount of positions; 3) **MixColumns** – is a multiplication over $GF(2^8)$ of each column for a constant, giving the corresponding column of the updated **state**; 4) **AddRoundKey** – is a bitwise modulo-2 addition, i.e. an addition over $GF(2^8)$, of a sub-key to the **state**. The AES algorithm takes the cipher key and performs a **KeyExpansion** procedure to generate the linear array of $Nb(Nr + 1)$ sub-keys, which are used in the $Nr + 1$ steps of the cipher algorithm. In the **KeyExpansion** procedure of Fig. 1 **RotWord** is a circular rotation of the bytes in a word and **SubWord** applies an s-box transformation to each byte of a $32b$ word. The constant **Rcon[h]** is $\{02\}^{h-1}, 00, 00, 00$ where the power is considered over $GF(2^8)$. Several modes of operation for use with an underlying symmetric key block cipher algorithm are recommended by NIST [4] and are widely employed. The main goal of these modes of operation is twofold [3]: 1) improving the security of block ciphers avoiding repeated inputs from being encrypted to the same value. To achieve this goal, the output of the block cipher is fed back, introducing a dependence of current output upon previous inputs. 2) Providing different security features (such as hashing, integrity check, etc.) using the block cipher as a building block. The main modes of operation are: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), and Counter (CTR). Specific information about the security

	ECB	CBC	CBC-MAC	CFB	OFB	OCB	CTR
Sender	Enc	Enc	Enc	Enc	Enc	Enc	Enc
Receiver	Dec	Dec	Enc	Enc	Enc	Dec	Enc
Feedback	No	Yes	Yes	Yes	Yes	No	No

Fig. 2. Block cipher modes of operation.

properties of these modes of operation, with respect to errors, manipulations, and substitutions, and the associated schematics, can be found in [3] [4]. The Fig. 2 reports, for each mode of operation, whether encrypter (Enc) or decrypter (Dec) is needed in transmitting and receiving phase, and the feedback loop in the processing flow of a data stream is present.

3 Main AES Design Choices

Unrolling, tiling and pipelining AES. – Many cipher algorithms are based on a basic looping structure (Feistel or other substitution-permutation network) whereby data are iteratively passed through a transformation, namely the “round” function. Three techniques are known in the technical literature to exploit the intrinsic parallelism of this kind of algorithm at different levels providing architecture with different performance: the unrolling, the tiling and the pipelining transformation. The *unrolling* [9] consists in allocating the body of different instances of a loop some number of times (called the unrolling factor U) and iterates the loop by step U . Referring to U , the number of instanced loop bodies, it is possible to distinguish an iterative (serial) architecture, a partially unrolled or a completely unrolled one. The *iterative* architecture consists in the instantiation of a single body of the loop. This block is fed back through a mux, in order to realize successive iterations of the loop. This approach can minimize the hardware requirement giving low throughput. In an *unrolled* architecture the algorithm is implemented by allocating U rounds as a single combinatorial logic block. This approach requires more area and does not necessarily increase the throughput but enables pipelining, as it is shown in the following of the paper. The *tiling* [5] transformation replicates the instances of a block in order to increase the functional parallelism. Finally, the *pipelining* increases the number of blocks of data that are being simultaneously operated upon, inserting registers to store the partial results.

In the recent past, some authors [5] asserted that, in order to increase the throughput, it is better tiling a serial AES block than unrolling it. They justified this remark noting that: 1) an unrolled architecture is faster than the iterative one in the encryption time of a block, but this difference is often negligible; 2) the time saving comes with a greatly increase of the area. We agree with this analysis (that can also be supported by the quantitative measurements presented in [9]), but we would emphasize that tiling has different flaws with respect to unrolling as a throughput increasing technique. First of all, the unrolling enables further increase of pipelining levels improving the throughput, and this can justify the

extra area. Second, when the design is tiled, additional logic (with corresponding area and time penalty) is usually involved: demultiplexing logic to feed each one of the blocks, and a block collecting the results. An unrolled and pipelined architecture does not require any other additional logic than pipeline registers. In this paper we propose (see Section 4) several AES implementations, differing in the number of rounds actually instanced and in the level of pipelining. These demonstrate that the throughput per area of an iterative AES block is lower than the one of an unrolled and pipelined version of AES, even without considering the extra logic to route the data in a tiled architecture.

Pipelining and Feedback Loops – As we already mentioned, in many real-world applications, cipher modes of operation are employed. The pipelining can be effectively used for applications requiring modes of operation not relying on feedback (such as ECB, OCB, and CTR) to enhance the performance. Some of these modes imply feedback (see last row of Fig. 2). The feedback introduces a loop carried dependence, i.e. the processing of a block can start only if the previous block has been processed, and this prohibits a pipelined implementation. For this reason, some cipher modes, such as like interleaved CFB or CFB-k [4], [3], have been proposed to allow security and high encryption rate when implemented in hardware. In the applications requiring feedback modes (such as CBC, CFB, and OFB), a method of increasing the throughput is to pipeline the cipher even in presence of feedback, and to interleave multiple data streams keeping full the pipeline. This technique [6] is called *innerroundpipelining* (or c-slow technique). Please note that it is possible to resort to the same technique also in an iterative architecture, with or without feedback cipher mode. As general rule in both cases, if the pipeline is composed by N stages, N different data flows should be available. The availability of many independent data flows is a common scenario in encrypted Internet router whereas multiple routes are active and have to be encrypted or authenticated. Since an (iterative or feedback) AES block can be pipelined, provided that a sufficient number of data flows is available, in the following of this paper we focus on the application of pipelining on the AES algorithm.

Decryption and Key-Length – In this paper we focus on the design and the implementation of an AES encrypter for two main reasons: 1) in many communication systems only the encipherment can suffice (see first and second row of Fig. 2); 2) the AES decryption process is roughly the same as the encryption one. In fact, the same loop structure of the encryption and similar steps (`InvByteSubs`, `InvShiftRows`, etc.) are used in the decryption and thus the same solutions to architectural and implementation issues we propose can be extended to an AES decryption core.

As far as the key-length is concerned, we consider the design of an AES encryption block for $128b$ keys. However in the same way, it is possible to design AES encryption cores with $192b$ and $256b$ keys, since they differ in the number of rounds and in a slightly different `KeyExpansion` algorithm.

On/Off-Line Subkey Generation – The sub-keys for encipherment/de-cipherment can be either stored in a local memory (*stored-key* approach) or

generated concurrently with the encryption/decryption process (*key-agile* approach). The stored-key approach requires a preprocessing phase every time the key is changed, and it needs a (quite large) memory block for the sub-keys. The key-agile approach allows the block cipher to work at full speed, even if the key is changed, and saves the extra memory for the sub-keys. It is worth noting that in an unrolled and pipelined version of a key-agile cipher, also the `KeyExpansion` has to be unrolled and further pipeline registers are needed. However in the stored-key approach, when multiple flows are processed in a pipelined AES block, the `KeyExpansion` and the sub-keys buffer have to be duplicated. The stored key approach is used by [7] and [9], while other AES implementations rely on key-agile approach [5], [8], [10]. In order to guarantee high-performance and maximum cipher flexibility we decided to resort to the key-agile approach.

Data-Path Placement – *Data-path placement* involves using the high-level structure of the computation and the data flow to better place the design. This allows many benefits, including shorter wires, more physically compact layout, and faster and easier place-&-route step. The shortening of the wires and their associated delays enables improved performance on FPGA, since a considerable contribution to the critical path is due to the net delay. Unfortunately a post synthesis hand layout on the FPGA of a given block demands for a high effort and time. Instead, we designed and implemented each basic block of AES cipher using appropriate VHDL attributes to address synthesis and placement, then we placed the overall architecture in accordance with the data flow. This approach is much more handy than a hand layout with a florplan editor. Some details about the implementation and the resulting placed data-paths are reported in Section 4 and in Fig. 4, respectively.

Target Device, Design Flow and Tools – The FPGA part we chose is a Xilinx Virtex-E 2000-8bg560. The Virtex device family appears to be a good representative for a modern FPGA, and is not fundamentally different from devices from other vendors. The Xilinx Virtex series of FPGAs, consists of Configurable Logic Blocks (CLBs) and interconnection circuitry tiled to form a chip. Each CLB consists of two slices, and each slice contains two 4 inputs Look-Up Tables (LUT), 2 flip-flops (FF), and associated carry chain logic. Each LUT can either be used as a 16x1 bit RAM, or as a 1-16 cycle delay shift register. The Xilinx Virtex-E 2000 presents 19200 slices and 19520 tristate buffers and incorporates also fully synchronous dual-ported 4096b block memories named Block SelectRAM. Block SelectRAM memories (BRAM) are organized in columns and each BRAM is four CLBs high. As far as design tools are concerned, we used Aldec Active VHDL 4.2 for simulation, Synplicity Synplify Pro 7.1, Synopsys FPGA Express, Xilinx XST for synthesizing VHDL, and Xilinx ISE 4.1 for place-&route and timing analysis. In our AES design experience Synplify appeared to achieve better synthesis results with respect to Xilinx XST and FPGA Express.

4 AES Blocks Design and Implementation

Iterative Architecture – The schematic of the proposed iterative AES block, the *Round* and the *KeyGen* blocks are reported in Fig. 3. The *Round* block can be configured to implement both the *Round* and the *FinalRound* functions of Fig. 1, while the *InitialRound* is instanced as a single block. The AES block (see Fig. 3) presents a latency equal to $N_R + 1$ clock ticks, but the processing of a new plain text can start and a cipher-text is output every N_R ticks. This is allowed by the overlapping of the first serial step of the processing (*InitialRound*) of a plain text block with the last *Round* of the previous plain text processing. We implemented the *InitialRound* (that is a $128b$ xor gate) and the following multiplexer together in the same set of 128 LUTs, since each LUT realizes any 4 inputs function. In other 128 LUTs the 2-to-1 multiplexer and the *AddRoundKey* can be accommodated. The matrix multiplication involved in the *MixColumns* block can be realized as suggested in [2] using the *xtime* function. In fact the *MixColumns* can be realized as a net of xor gate, whereas the maximum fan-in of a xor is 5 and this gate requires 2 LUTs. For implementing the *SubBytes* we used the Xilinx Virtex-E BRAMs. Each BRAM is configured as a dual-port synchronous $256 \times 8b$ words RAM, and it implements 2 s-boxes accessing two locations concurrently. Hence 8 BRAMs suffice for each *SubBytes* blocks. Since the BRAM are synchronous, they can also realize the *Register* before *SubBytes*. Finally the *ShiftRows* can be simply realized by hardwiring. As far as the *KeyGen* block is concerned, the *SubWord* is made up of 4 s-boxes and requires 2 BRAMs which implement also the $32b$ *Register* of Fig. 3. Four $32b$ registers balance the latency due to the $32b$ *Register*. The *Controller* has 4 main states and uses a counter to store the number of round functions applied to the current plain text. The *Controller* runs concurrently in pipelining with the data-path, in a such way that elaboration of data and sequencing of operations can be overlapped. The control signals are buffered into flip-flops that also break up the propagation of

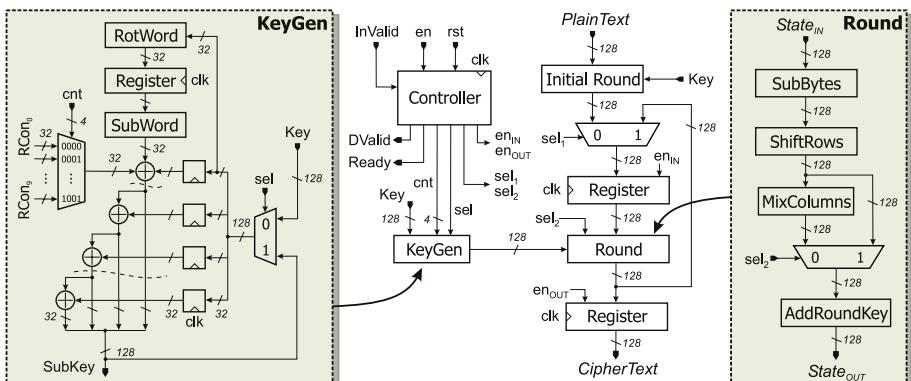


Fig. 3. The schematic of the iterative AES architecture.

these signals to the data-path, greatly limiting the contribution of the net delay on the critical path.

As far as the placement of the blocks is concerned, we decided to enclose the data-path of the iterative AES block in a box 4 CLBs high. Actually a *SubBytes* block requires 8 BRAMs and each BRAM is 4 CLBs high. 32 CLBs can accommodate 128 slices and 128 FF-slices and the data-path parallelism is exactly 128b. Therefore each block of Fig. 3 is placed in order to occupy a different number of 32 CLB columns, and the blocks are ordered in the same way as they process the data. The *Round* function requires a single column. The *MixColumns* is accommodated in 3 columns, since the *xtime* function is allocated in the first one but it does not fill all the available LUTs, while the remaining 2 columns are completely used for 5-inputs xor-s. The *KeyGen* employees 2 BRAMs and so it is enclosed in a box 8 CLBs high. For the *Controller* we did not impose any placement constraints. The resulting floorplan of the iterative AES block is reported in Fig. 4a. The implementation results are reported in Fig. 5. The same AES block without any optimization and placement constraints requires 1736 slices and presents a T_{CK} of 19,9 ns, and so our optimizations achieve great area saving (-74%) and throughput improvement (+55%).

5-Pipelined Iterative Architecture – Interleaving encryption flows allows to use the pipelining technique also for the iterative architecture. We analyzed the delay of each block in the AES core, in order to insert pipeline registers balancing the delay of the stages. We introduced in the schematic of Fig. 3 one 128b register between *SubBytes* and the wiring implementing *ShiftRows*, and two registers before the inputs of *AddRoundKey*, to obtain a 5 pipeline stages version of the iterative architecture presented in the previous paragraph. The introduction of the pipeline registers does not remarkably impact on the slices count, since many slices are partially occupied by the LUTs of the iterative non-pipelined AES block. As far as the sub-keys generator is concerned, several

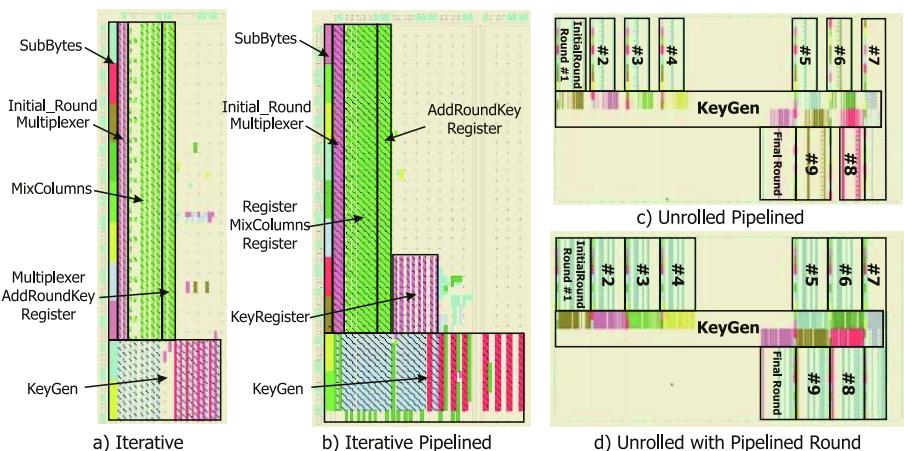


Fig. 4. The floorplans of all the proposed implementations of AES.

Design	Device	Pipeline [stages]	Clock [Mhz]	Th. [Gbps]	Slices (BRAM)	Mbps/Slices
Weaver et al. [5]	XVE600-8	1	60	0,69	460 (10)	1,5 (0,40)
Labbé et al. [7]	XCV1000-4	1	31	0,39	2151 (4)	0,18 (0,15)
Amphion CS5220 [10]	XVE-8	1	101	0,29	421 (4)	0,69 (0,31)
Amphion CS5230 [10]	XVE-8	1	91	1,06	573 (10)	1,9 (0,57)
Iterative	XVE2000-8	1	79	1	446 (10)	2,3 (0,58)
Weaver et al. [5]	XVE600-8	5	158	1,75	770 (10)	2,3 (0,85)
Labbé et al. [7]	XCV1000-4	5	30	1,91	8767 (4)	0,22 (0,21)
5-Pipe Iterative	XVE2000-8	5	142	1,82	648 (10)	2,8 (0,94)
Elbirt et al. [9]	XCV1000-4	4	40	0,98	5449 (0)	0,18 (0,18)
Elbirt et al. [9]	XCV1000-4	10	31	1,88	10923 (0)	0,17 (0,17)
1-Pipe Fully Unr.	XVE2000-8	10	70	8,9	2778 (100)	3,2 (0,57)
5-Pipe Fully Unr.	XVE2000-8	50	158	20,3	5810 (100)	3,5 (1,1)

Fig. 5. Performance results for the proposed, commercial and academical implementations of AES.

changes to the schematic of Fig. 3 are needed to realize a 5-stages pipelined key agile *KeyRegister*. Registers are inserted in the place of the dotted lines in Fig. 3. The registers after the multiplexer become shift-registers to keep aligned the data: the first 32b register on the top remains unchanged, the second and the third are delayed of 2 clock ticks, while the last of 3 clock ticks. Also the ‘1’ input of the feedback mux is delayed of other 2 clock ticks. A register (indicated as *KeyRegister* in Fig. 4b) on the output *SubKey* to split the propagation of the sub-keys towards the *Round*. We used LUTs configured as shift registers to realize efficiently these flip-flop chains. Finally the *Controller* has to be slightly modified in order to keep track of the current blocks being processed.

1-Pipelined and 5-Pipelined Fully Unrolled Architecture – The 1-pipelined Architecture and the 5-pipelined Architecture implement the completely unrolled AES loop and differ in the level of pipelining. The *1-pipelined Fully Unrolled Architecture*, which uses one pipeline stage per round, is obtained simply juxtaposing ten replica of the *Round* and *KeyGen* depicted in Fig. 3, and using a pipeline register for each round. The first round is an *InitialRound*, while the last round is a *FinalRound* block, and both are simpler than the intermediate *Round*. The *5-pipelined Fully Unrolled Architecture* makes use of 5 levels of pipelining per round, and is obtained from the 1-pipelined Architecture, increasing the pipelining level per round up to 5, following the same remarks described in the design of the 1-pipelined Iterative Architecture. In order to place 10 rounds, 10 BRAM columns are needed, but in a Virtex-E 2000 there are only 8 BRAM columns. Since each BRAM column has 20 BRAMs and each *Round* and *KeyGen* occupy 10 BRAMs, we decided to dispose the first 6 rounds on the upper half of the device, the round 7 split into two parts on the upper and lower half, while the remaining rounds on the lower part. The net delays in the round 7 are relevant, since the basic blocks are scattered on the device, so we inserted an additional pipe register to avoid that this could impact on the critical path. The resulting floorplans of these two AES implementations are reported in Fig. 4c,d and the performance results are given in the last two rows of the Fig. 5.

5 Performance Results and Comparison with Related Work

We adopted three different parameters to evaluate proposed implementations of AES: 1) the throughput T considered as the maximum encryption rate; 2) the cost A in terms of area as number of Virtex-E slices and BRAMs; 3) the throughput per area T/A , which measures the contribution of each slice to the throughput and hence the efficiency of the implementation. Performance results for each AES block of Section 4, and for other academical and commercial implementations are summarized in Fig. 5. The results are expressed in terms of the pipelining levels, the maximum clock frequency, the throughput, the area (as slices and as BRAMs) and finally T/A . It is worth noting that a dual-port $256 \times 8b$ BRAM can be replaced by a distributed memory composed of 256 LUTs (128 slices). Therefore T/A is also reported (in brackets) whereas A is given by the total number of slices using distributed memories and no BRAMs. The reported implementations lie only on 2 types of devices, namely XCV-4 and XVE-8, with differently available resources. These devices are based on the same building block, i.e. a slice containing 2 LUTs and 2 FFs. Our iterative architectures with 1 and 5 pipelining levels have the best T (ranging from 1 to 1,82 Gbps) and T/A among the other iterative implementations. Our fully unrolled architecture with 50 pipeline stages provides the highest encryption throughput (20,3 Gbps), the highest clock frequency (158 MHz) and the best T/A , among the reported implementations. These results demonstrate that the use of techniques such as unrolling and pipelining allow to explore the design space tailoring the performance and area requirements. These techniques are also able to deliver very high performance. Furthermore the accurate data-path placement is effective in reducing the area and improving the clock period of the AES blocks. The placed and routed AES blocks with the placement constraints present a clock period improvement ranging between the 36% and the 55%, with respect to the corresponding design without constraints. In [8] an iterative architecture implemented in a chip using a 0,18 micron CMOS technology with core supply at 1,8V is reported. It can deliver 1,6 gigabit per second encryption rate with 128b keys, while our best FPGA iterative implementation reaches 1 gigabit per second. So the floorplan aware design, the use of the architectural features of an FPGA family, and the deep pipelining enable a low-cost technology such as FPGA to reach performance comparable to an ASIC implementation, at least for algorithms which can be mapped in a highly regular structure.

6 Conclusions

This paper has addressed the design and the FPGA implementation of a fully key agile AES encryption core with 128-bit keys. We discussed the effectiveness of several techniques, such as accurate floorplanning, the unrolling, tiling and pipelining transformations to explore the design space. We dealt with the issues concerning the use of the pipelining in presence of feedback loops and analyzing

several solutions. Based on these solutions, four architectures with different level of parallelism have been designed and implemented on a Virtex-E FPGA part. Performance results show that the presented implementations achieve better performance with respect to other academical and commercial AES block.

References

1. J. Daemen and V. Rijmen, "AES Proposal: Rijndael, AES Algorithm Submission", September 1999, available at <http://www.nist.gov/CryptoToolkit>
2. National Institute of Standards and Technology (NIST), FIPS Publication 197, "Advanced Encryption Standard (AES)", November 2001.
3. A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996.
4. National Institute of Standards and Technology (NIST), Special Publication 800-38A, "Recommendation for Block Cipher Modes of Operation", December 2001.
5. N. Weaver, J. Wawrzynek, "Very High Performance, Compact AES Implementation in Xilinx FPGAs", September 2002, available at <http://www.cs.berkeley.edu/~nweaver/sfra/rijndael.pdf>
6. C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming", Third Caltech Conference On VLSI, March 1993.
7. A. Labb  , A. P  rez, "AES Implementations on FPGA: Time-Flexibility Tradeoff", Proceedings of FPL02, pp. 836-844.
8. P.R. Schaumont, H. Kuo, and I.M. Verbauwhede, "Unlocking the Design Secrets of a 2.29 Gb/s Rijndael Processor", Proceedings of the International Design Automation Conference 2002 (DAC02), pp. 634-639.
9. A.J. Elbirt, W. Yip, B. Chetwynd, and C. Paar, "An FPGA-Based Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists", IEEE Trans. on VLSI Systems, Vol.9, No.4, August 2001, pp. 545-557.
10. Amphion Semiconductor, "CS5210-40: High Performance AES Encryption Cores", Amphion(TM), Febrbruary 2003, available at <http://www.amphion.com/cs5210.html>

Two Approaches for a Single-Chip FPGA Implementation of an Encryptor/Decryptor AES Core

Nazar A. Saqib, Francisco Rodríguez-Henríquez, and Arturo Díaz-Pérez

Computer Science Section, Electrical Engineering Department
Centro de Investigación y de Estudios Avanzados del IPN
Av. Instituto Politécnico Nacional No. 2508, México D.F.
nabbas@computacion.cs.cinvestav.mx
{francisco, adiaz}@cs.cinvestav.mx

Abstract. In this paper we present a single-chip FPGA full encryptor/decryptor core design of the AES algorithm. Our design performs all of them, encryption, decryption and key scheduling processes. High performance timing figures are obtained through the use of a pipelined architecture. Moreover, several modifications to the conventional AES algorithm's formulations have been introduced, thus allowing us to obtain a significant reduction in the total number of computations and the path delay associated to them. Particularly, for the implementation of the most costly step of AES, multiplicative inverse in $GF(2^8)$, two approaches were considered. The first approach uses pre-computed values stored in a lookup table giving fast execution times of the algorithm at the price of memory requirements. Our second approach computes multiplicative inverse by using composite field techniques, yielding a reduction in the memory requirements at the cost of an increment in the execution time. The obtained results indicate that both designs are competitive with the fastest complete AES single-chip FGPA core implementations reported to date. Our first approach requires up to 11.8% less CLB slices, 21.5% less BRAMs and yields up to 18.5% higher throughput than the fastest comparable implementation reported in literature.

1 Introduction

Recently, Rijndael block cipher algorithm was chosen by NIST as the new Advanced Encryption Standard (AES) [2, 13]. Rijndael is a block cipher that can process blocks of 128, 192 and 256 bits and keys of same lengths, but for official AES version, the only legal block length is 128 bits.

FPGA AES implementations are attractive since costs of VLSI design and fabrication can be reduced. However, AES hardware implementation poses a challenge since encryption and decryption processes are not completely symmetrical [2, 7, 13]. Designing separated architectures for encryption and decryption processes would imply the allocation of a large amount of FPGA resources. Designs reported in [3, 4, 5] have considered only the encryption part of AES. A single-chip FPGA implementa-

tion of a full encryptor/decryptor AES core has been reported in [9]. Performance results for these designs are broadly variable; they range from 300 Mbit/s to 3.2 Gbit/s, approximately.

In this paper, we describe a fully pipelined AES implementation core for an FPGA device. It is a complete encryptor/decryptor core for which encryption, decryption and key scheduling work efficiently. We propose two approaches to implement multiplicative inverse for $GF(2^8)$ which is the most costly operation of AES. The first design uses pre-computed values through a lookup table requiring fast memory access to obtain a good throughput. The second approach eliminates memory requirements at the cost of more FPGA resources. Obtained results show that both designs are competitive with the previous full AES core reported in [9].

In Section 2, AES algorithm is briefly described. Several modifications to AES algorithm's expressions to gain performance are explained in Section 3. In Section 4, we discuss a three-stage computation to calculate multiplicative inverse in finite field $GF(2^8)$. In Section 5, a fully pipelined AES FPGA implementation is presented and performance results are provided. Finally, concluding remarks are included in section 6.

2 The AES algorithm

The AES cipher treats the input 128-bit block as a group of 16 bytes organized in a 4×4 matrix called *State* matrix. Fig. 1 depicts the AES cipher algorithm flow for encrypting one block of input data.

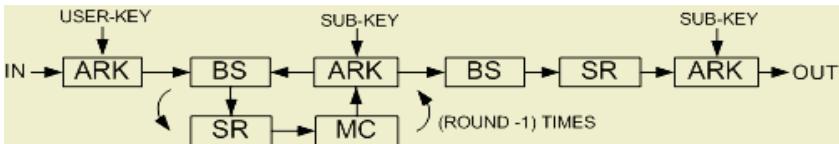


Fig. 1. Basic AES encryption flow.

The algorithm consists of an initial transformation, followed by a main loop where nine iterations called *rounds* are executed. Each round is composed of a sequence of four transformations: *Byte Substitution* (BS), *ShiftRows* (SR), *MixColumns* (MC) and *AddRoundKey* (ARK). For each round of the main loop, a round key is derived from the original key through a process called *Key Scheduling*. Finally, a last round consisting of three transformations, BS, SR and ARK, is executed. The AES decryption algorithm operates similarly by applying the inverse of all the transformations described above in reverse order. In the rest of this section we shall briefly describe the four AES round transformations BS, SR, MC and ARK.

In *ByteSubstitution* (BS), each input byte of the *State* matrix is independently replaced by another byte from a look-up table called *S-box*. The AES *S-box* is a 256-entry table composed of two transformations: First each input byte is replaced with its multiplicative inverse in $GF(2^8)$ with the element {00} being mapped onto itself;

followed by an affine transformation over $GF(2)$ [2, 13]. For decryption, inverse S-box is obtained by applying inverse affine transformation followed by multiplicative inversion in $GF(2^8)$ [13]. *ShiftRows* (SR) is a cyclic shift operation where each row is rotated cyclically to the left using 0,1,2 and 3-byte offset for encryption while for decryption, rotation is applied to the right. In *MixColumns*(MC) transformation, each column of the *State* matrix is multiplied by a constant fixed matrix as follows,

$$\begin{bmatrix} c'_{0,i} \\ c'_{1,i} \\ c'_{2,i} \\ c'_{3,i} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,i} \\ c_{1,i} \\ c_{2,i} \\ c_{3,i} \end{bmatrix} \quad i = 0,1,2,3 \quad (1)$$

Similarly, for decryption, we compute Inverse MixColumns, by multiplying each column of the *State* matrix by a constant fixed matrix as shown below

$$\begin{bmatrix} c'_{0,i} \\ c'_{1,i} \\ c'_{2,i} \\ c'_{3,i} \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} c_{0,i} \\ c_{1,i} \\ c_{2,i} \\ c_{3,i} \end{bmatrix} \quad i = 0,1,2,3 \quad (2)$$

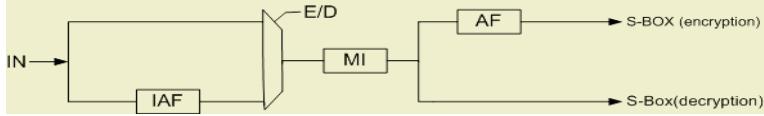
Finally, in *AddRoundKey* (ARK), output of MC is XOR-ed with the corresponding round sub-key derived from the user key. The ARK step is essentially same for AES encryption and decryption processes.

3 Novel Computational Expressions for Implementing AES on FPGA Devices

In this section, we introduce some novel techniques for implementing AES algorithm on FPGA devices. The three main AES algorithms, key schedule, encryption and decryption, are considered for optimization. Our optimization criteria are based on three main factors: To maximize path delay reductions, reutilization of pre-computed blocks and to exploit full resources of the target device. In addition, we have tried to use 4-input logic gates wherever is possible, since they can be efficiently implemented using FPGA CLBs.

3.1 AES Algorithm Optimizations

S-box and inverse S-box are required for computing the BS step of AES. Both may be computed by implementing affine (AF) and inverse affine (IAF) transformations together with a look-up table for Multiplicative Inverse (MI). In this way, the combination MI + AF provides S-box for encryption, while IAF + MI computes the Inverse S-box needed for decryption. To use only one MI module, a multiplexer is used to switch the data path for encryption/decryption, as shown in Fig. 2.

**Fig. 2.** S-box and inverse S-box implementation

SR and ISR Implementations do not require FPGA resources as they can be implemented by rewiring. For the sake of symmetry, ISR step is embedded into IAF while SR and AF steps are joined together. *MC and IMC transformations* are reviewed in deep. Encryption's MC can be efficiently computed by using only 3 steps [1]: a sum step, a doubling step and a final sum step. Further optimization consists on embedding ARK step to fully exploit 4-input FPGA slice resources. Let the elements of State matrix's column one be $a[0]$, $a[1]$, $a[2]$, $a[3]$, and let $k[0]$, $k[1]$, $k[2]$ and $k[3]$ represent first four bytes of a key block, then transformed matrix (MC + ARK) column $a'[0]$, $a'[1]$, $a'[2]$ and $a'[3]$ can be efficiently obtained as shown in Table 1

Table 1. The modified MC expression with ARK.

Step 1	Step 2	Step 3
$v = a[1] \oplus a[2] \oplus a[3]$	$xt_0 = xtime(a[0])$	$a'[0] = k[0] \oplus v \oplus xt_o \oplus xt_1$
$v = a[0] \oplus a[2] \oplus a[3]$	$xt_1 = xtime(a[1])$	$a'[1] = k[1] \oplus v \oplus xt_1 \oplus xt_2$
$v = a[0] \oplus a[1] \oplus a[3]$	$xt_2 = xtime(a[2])$	$a'[2] = k[2] \oplus v \oplus xt_2 \oplus xt_3$
$v = a[0] \oplus a[1] \oplus a[2]$	$xt_3 = xtime(a[3])$	$a'[3] = k[3] \oplus v \oplus xt_3 \oplus xt_0$

Here $xtime(v)$ represents finite field multiplication of $02 \times v$, where 02 stands for constant polynomial x in $GF(2^8)$. Note that all the computations shown in the same column of Table 1 can be performed in parallel.

Table 2. The modified IMC expression with IARK.

Step 1	Step 2	Step 3	Step 4	Step 5
$t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]$			$u = xtime(u)$	$t' = t \oplus u$
		$u = s'_0 \oplus s'_1 \oplus s'_2 \oplus s'_3$	Step 6	
$s_0 = xtime(a[0])$	$s'_0 = xtime(s_0)$	$v = s_0 \oplus s_1 \oplus s'_0 \oplus s'_2$	$a'[0] = a[0] \oplus t' \oplus v \oplus k[0]$	
$s_1 = xtime(a[1])$	$s'_1 = xtime(s_1)$	$v = s_1 \oplus s_2 \oplus s'_1 \oplus s'_3$	$a'[1] = a[1] \oplus t' \oplus v \oplus k[1]$	
$s_2 = xtime(a[2])$	$s'_2 = xtime(s_2)$	$v = s_2 \oplus s_3 \oplus s'_0 \oplus s'_2$	$a'[2] = a[2] \oplus t' \oplus v \oplus k[2]$	
$s_3 = xtime(a[3])$	$s'_3 = xtime(s_3)$	$v = s_3 \oplus s_0 \oplus s'_1 \oplus s'_3$	$a'[3] = a[3] \oplus t' \oplus v \oplus k[3]$	

The same strategy applied for MC would yield up to seven steps to compute IMC (four sum steps and three doubling steps). The difference is due to the fact that coefficients in equation (2) have a higher Hamming weight than the ones in equation (1). To overcome this drawback we use the strategy depicted in Table 2 where IMC manipulation is re-structured and seven steps are cut to five steps.

As explained above, for *ARK Implementation*, ARK step is embedded into MC step. For final round (Round 10), MC and IMC steps are not executed, therefore, a separate implementation of ARK is made.

The ideas discussed in this section can be implemented as shown in Fig. 3, where the block-diagram represents proposed architecture for implementing AES encryption/decryption processes on FPGA devices. Hence the critical path for encryption is MI•AF•SR•MC•ARK, while ISR•IAF•MI•IMC•IARK is the critical path for decryption as shown below.

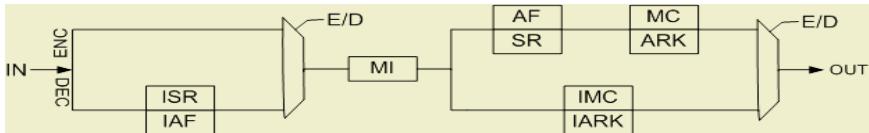


Fig. 3. AES algorithm encryptor/decryptor implementation.

3.2 Key Schedule Optimization

The original user key consists of 128 bits arranged as a 4×4 matrix of bytes. Let $w[0]$, $w[1]$, $w[2]$, and $w[3]$ be the four columns of the original key. Then, those four columns can be recursively expanded to obtain 40 more columns, as follows: Let the columns up to $w[i-1]$ have been defined then,

$$w[i] = \begin{cases} w[i-4] \oplus w[i-1] & \text{if } i \bmod 4 \neq 0 \\ w[i-4] \oplus T(w[i-1]) & \text{otherwise} \end{cases} \quad (3)$$

Where $T(w[i-1])$ is a non-linear transformation based on the application of the *S-box* to the four bytes of the column, an additional cyclic rotation of the bytes within the column and the addition of a round constant (*rcon*) for symmetry elimination [2]. Let $[k_0, k_4, k_8, k_{12}]$, $[k_1, k_5, k_9, k_{13}]$, $[k_2, k_6, k_{10}, k_{14}]$ and $[k_3, k_7, k_{11}, k_{15}]$ be the first four columns of the key. By combining and parallelizing expressions, first column of the new key $[k'_0, k'_4, k'_8, k'_{12}]$ can be calculated as shown in Table 3.

Table 3. Modified expressions for key schedule.

Step 1			
$k'_0 = k_0 \oplus Sbox(k_{13}) \oplus rcon$			
Step 2			
$k'_4 = k_4 \oplus k'_0$	$k'_8 = k_4 \oplus k_8 \oplus k'_0$	$k'_{12} = k_4 \oplus k_8 \oplus k_{12} \oplus k'_0$	

$Sbox(k_{13})$ refers to the data obtain by substituting k_{13} . The same process is used for calculating other 3 columns of the new key. In the same manner, next nine keys are obtained.

4 AES S-Box Design Based on Composite Field Techniques

The most costly operation in the BS transformation described in section 2.1, is the computation of the inverse multiplicative of a byte in the finite field $GF(2^8)$ defined by the AES cipher. In an effort to reduce the costs associated to this operation, several authors have designed AES *S-Box* based on the composite field technique reported first in [10, 11, 12]. That technique uses a three-stage strategy: Map the element $A \in GF(2^8)$ to a composite field F using a isomorphism function δ . Compute the multiplicative inverse over the field F and finally map the computation results back to the original field.

In [6] an efficient method to compute the inverse multiplicative based on Fermat's little theorem was outlined. That method is useful because it allows us to compute the multiplicative inverse over a composite field $GF((2^n)^m)$ as a combination of operations over the ground field $GF(2^n)$. It is based on the following theorem:

Theorem [7,12]: The multiplicative inverse of an element A of the composite field $GF((2^n)^m)$, $A \neq 0$, can be computed by

$$A^{-1} = (A^r)^{-1} A^{r-1} \bmod P(x), \text{ Where } A^r \in GF(2^n) \text{ and } r = \frac{2^{nm} - 1}{2^n - 1} \quad (4)$$

An important observation of the above theorem is that the element A^r belongs to the ground field $GF(2^n)$. This remarkable characteristic can be exploited to obtain an efficient implementation of the inverse multiplicative over the composite field. By selecting $m=4$ and $n=2$ in the above theorem we obtain $r=17$ and,

$$A^{-1} = (A^r)^{-1} \cdot A^{r-1} = (A^{17})^{-1} \cdot A^{16} \quad (5)$$

In case of AES, it is possible to construct a suitable composite field F , by using two degree-two extensions based on the following irreducible polynomials [10]:

$$\begin{aligned} F_1 &= GF(2^2): & P_0(x) &= x^2 + x + 1 \\ F_2 &= GF((2^2)^2): & P_1(y) &= y^2 + y + \phi \\ F_3 &= GF(((2^2)^2)^2): & P_2(z) &= z^2 + z + \lambda \end{aligned} \quad (6)$$

where $\phi = \{10\}_2$, $\lambda = \{1100\}_2$.

The inverse multiplicative over the composite field F_2 defined in the equation (6), can be found as follows. Let $A \in F_2 = GF((2^2)^2)$ be defined in polynomial basis as $A = A_H y + A_L$, and let the Galois field F_1 , F_2 , and F_3 be defined as shown in equation (5). Then it can be shown that

$$\begin{aligned} A^{16} &= A_H y + (A_H + A_L); \\ A^{17} &= A^{16} \cdot A = 0 \cdot y + (\lambda A_H^{16} A_H + A_L^{16} A_L) = \lambda A_H^2 + A_L^{16} A_L \end{aligned} \quad (7)$$

Fig. 4 depicts the corresponding block-diagram of the three-stage inverse multiplier represented by equations (5) and (7). The circuits shown in Fig. 5 and Fig. 6 present a gate-level implementation of the aforementioned strategy.

As we explained above, in order to obtain the multiplicative inverse of the element $A \in F = GF(2^8)$, we first map A to its equivalent representation (A_H, A_L) in the isomorphic field $F_2 = GF((2^2)^2)$ using the isomorphism δ (and its corresponding inverse δ^{-1}).

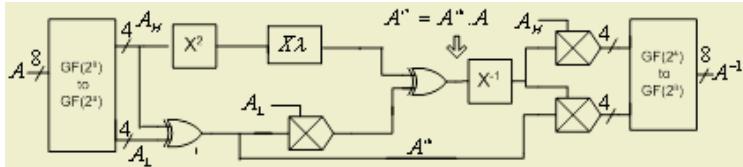


Fig. 4. Three-stage strategy to compute multiplicative inverse in composite fields.

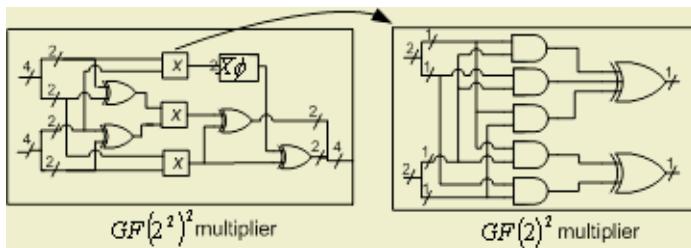


Fig. 5. $GF((2^2)^2)$ and $GF(2^2)$ multipliers

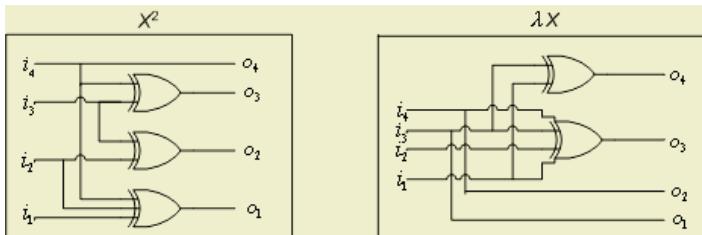


Fig. 6. Gate-level implementation for X^2 and λX

In order to map a given element A from the finite field F to its isomorphic composite field F_2 and vice versa, we only need to compute the matrix multiplication of the said element A , by the isomorphic functions shown in equation (8) given by [10]: Also by taking advantage of the fact that A^{17} is an element of F_2 , the final operation $(A^{17})^{-1} \cdot A^{16}$ of equation (7), can be easily computed with further gate reduction. Last stage of the algorithm consists of mapping computed value in the composite field, back to the field $F = GF(2^8)$.

$$\delta = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}; \delta^{-1} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad (8)$$

5 Performance Results

To achieve high throughput, we have designed a pipelined architecture for AES as shown in Fig. 7. Eleven AES rounds have been unrolled to develop a pipelined design. The design is symmetric in the sense that the same steps used for encryption are re-used for decryption. The corresponding round-keys for encryption or decryption are generated from the input key accordingly eleven stages of the pipeline. Each stage is clock triggered and data is transferred to next stage at rising-edge of the clock. The data blocks are accepted at each clock cycle and then after 11 cycles, output encrypted/decrypted blocks appear at the output at consecutive clock cycles.

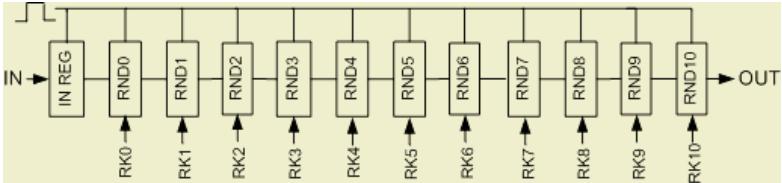


Fig. 7. Pipeline approach for 128 bit AES Encryption/decryption

AES memory requirements could be too high, especially for a pipeline design. To overcome this requirement, the design is implemented on Xilinx Virtex-E XCV2600. The Virtex and Virtex-E family of devices contains more than 280 BRAMs which are well suited for fast memory access [14]. A dual port BRAM can be configured into two single port BRAMs with independent data access. Xilinx Foundation Series F4.1i tool is used for design entry, verification and synthesis. Both approaches to implement MI, previously discussed, follow the same pipeline architecture. Table 4 details the total area required for each block as shown in Fig. 3.

The first design uses 80 BRAMs (43%) needed for MI implementation and occupies 386 I/O Blocks (48%) and 5677 slices (22.3%). The system runs at 30 MHz and data is processed at 3840 Mbits/s. The second design, using a three-stage MI computation, occupies 13416 CLB slices (52%) and 386 I/O Blocks(48%); no BRAMs are required here. The allowed system frequency is 24.5 MHz and the design achieves a throughput of 3136 Mbits/s.

Table 4. Summary of features for AES encryptor/decryptor cores.

	E/D $GF(2^8)$	E/D $GF(2^4)$
BLOCK	CLB SLICES	CLB SLICES
KeyBlock	1278	1278
MI (10 rounds)	(80 BRAMs)	6676
SR	-----	-----
AF (10 rounds)	800	800
IAF (10 rounds)	640	640
MC + ARK (combined) (9 rounds)	1368	1368
IMC + IARK (combined) (9 rounds)	2088	2088
ARK (1 st + last round)	128	128
Misc. (Timing + I/O registers etc.)	374	438
TOTAL	6676 + (80 BRAMs)	13416 (NO BRAMs)
Throughput (Mbits/s)	3840	3136
Throughput/Area (Mbits/s/Slices)	0.58	0.24

Some FPGA implementations [3, 4, 5, 9, 15] for AES do exist. But some of them deals only with encryption [3, 4, 5] while others like [15] show better results but can not be categorized as a single-chip implementation. A fair comparison of our design can be made with the design reported in [9] which is an encryptor/decryptor core realization. It was implemented on Virtex-E XCV3200 FPGA device, using 7576 CLB slices, 102 BRAMs and achieving a throughput of 3239 Mbits/s. That design uses the same set of BRAMs for encryption/decryption and consumes 256 cycles to prepare BRAMs for encryption or decryption. That design also occupies 20 BRAMs for key scheduling. Our first design compared with [9] has reduced the area requirements up to 11.8% while expected throughput has been increased more than 18.5%. For our second design, area requirements are high but throughput is competitive with [9]. An advantage of this design is the portability factor since it does not use BRAMs.

6 Conclusions

We have presented two AES encryptor/decryptor core designs following a pipelined architecture. Both designs take advantage of using only one S-box (to compute multiplicative inverse). The difference lies in the implementation of MI. In the first design, 80 BRAMs are utilized for MI pre-computed values. In the second design, a three-stage architecture has been adopted for MI where calculations are made in $GF(2^4)^2$ and $GF(2^2)^{2^2}$ instead of $GF(2^8)$. The goal was to reduce memory requirements as much as possible, and then this design does not require BRAMs. The encryptor/decryptor starts reporting results only after 11 cycles needed to latch the round keys. We have re-organized MC and IMC computations to reduce data-path and to

take advantage of four-input/one-output organization of CLBs. The ARK step has been embedded with MC and IMC step to enhance the timing performance.

Our designs have been implemented on Virtex-E family of devices (XCV2600) using Xilinx Foundation Series F4.1i. Our results have shown competitive behavior compared to the existing AES FPGA encryptor/decryptor known cores. In the first design, we have outperformed design reported in [9]. The second design can be implemented on any FPGA family of devices. Future work includes extending the design for variable key and block lengths for AES algorithm.

References

1. Guido Bertoni et al: Efficient Software Implementation of AES on 32-bits Platforms: CHES2002, LNCS 2523, Springer-Verlag, 2002.
2. Joan Daemen, Vincent Rijmen: The Design of Rijndael, AES-The Advanced Encryption Standard: Springer-Verlag Berlin Heidelberg, New York, 2002.
3. A. Dandalis, V.K. Prasanna, J.D.P. Rolim: A Comparative Study of Performance of AES Candidates Using FPGAs: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
4. J. Elbirt, W. Yip, B. Chetwynd and C. Paar: A FPGA implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists: The Third AES3 Candidate Conference, 13-14 April 2000, New York.
5. K. Gaj, P. Chodowiec: Comparison of the Hardware Performance of the AES Candidates using Reconfigurable Hardware: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
6. Brian Gladman: The AES Algorithm (AES) in C and C++: URL: http://fp.gladman.plus.com/cryptography_technology/rijndael/index.htm, April 2001.
7. Guajardo, C. Paar: Efficient Algorithms for Elliptic Curve Cryptosystems: CRYPTO '97, August 17-21, Santa Barbara, CA, USA.
8. T. Ichikawa, T. Kasuya, M. Matsui: Hardware Evaluation of the AES Finalists: The Third Advanced Encryption Standard (AES3) Candidate Conference, 13-14 April 2000, New York, USA.
9. Maire McLoone and J.V McCanny: High Performance FPGA Rijndael Algorithm Implementations: C. Koc, D. Naccache, and C.paar(Eds): CHES2001, LNCS 2162, pp. 65-76, Springer-Verlag, 2001.
10. S. Morioka and A. Satoh: An Optimized S-Box Circuit Architecture for Low Power AES Design: CHES2002, LNCS 2523, Springer-Verlag, 2002.
11. C. Paar: Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields: PhD thesis: Universitat GH Essen, VDI Verlag, 1994.
12. A. Rudra et al: Efficient Rijndael Encryption Implementation with Composed Field Arithmetic: CHES2001, LNCS 2162, pp. 171-184, Springer-Verlag, 2001.
13. W. Trappe and L. C. Washington: Introduction to Cryptography with Coding Theory: Prentice-Hall, Upper Saddle River, 2002.
14. Xilinx Virtex TM-E 1.8V Field Programmable Gate Arrays, URL: www.xilinx.com, November 2000.
15. URL: <http://ece.gmu.edu/crypto/rijndael.htm>

Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations

K.R. Shesha Shayee, Joonseok Park, and Pedro C. Diniz

University of Southern California / Information Sciences Institute
4676 Admiralty Way, Suite 1001
Marina del Rey, California 90292, USA,
`{shesha,joonseok,pedro}@isi.edu`

Abstract. Selecting which program transformations to apply when mapping computations to FPGA-based architectures leads to prohibitively long design exploration cycles. An alternative is to develop fast, yet accurate, performance and area models to understand the impact and interaction of the transformations. In this paper we present a combined analytical performance and area modeling for complete FPGA designs in the presence of loop transformations. Our approach takes into account the impact of input/output memory bandwidth and memory interface resources, often the limiting factor in the effective implementation of these computations. Our preliminary results reveal that our modeling is very accurate allowing a compiler tool to quickly explore a very large design space resulting in the selection of a feasible high-performance design.

1 Introduction

The application of loop-level transformations, important to expose vast amounts of fine-grain parallelism and to promote data reuse, substantially increases the complexity of mapping computations to FPGA-based architectures. Figure 1 illustrates a typical behavior for a design mapped to an FPGA exploring loop unrolling. At first as the unrolling factor increases the execution time decreases and the amount of consumed space resources increases. However, with additional unrolling, there is the need to exploit more memory resources and memory parallelism. Before the FPGA capacity limitation is reached (in this case for an unrolling of 16) another limit is reached for an unrolling factor of 8. At this point, the design requires more memory channels than the implementation can provide. As such, these resources must be time-multiplexed leading to a non-trivial increase in the execution time (illustrated by the heavy-shaded bar).

Understanding, and mitigating the impact of hardware limitations is important in deriving the parameters of the loop transformations in the quest for the best possible design. In the example illustrated below if a compilation tool is not aware of the fact that memory channels resources are limited, it incorrectly selects the design with an unrolling factor of 8. This selection is only better

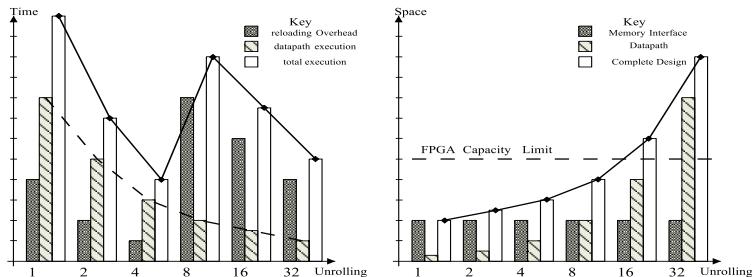


Fig. 1. Qualitative Execution and Space Plots in the Presence of Limited I/O Resources

than the design without any unrolling and far from the actual best design which corresponds to an unrolling amount of 4.

As this example illustrates, ignoring the real memory interface resource limitation can lead extremely poor design choices. Given the extremely large number of possible loop transformations and their interaction, a solution to the problem of finding the best performing (and feasible in terms of space) design is to develop performance and area estimation for the designs resulting from the application of a sequence of loop transformations. To this extend we focus on the development of a set of analytical models combined with behavioral estimation techniques for the performance and estimation of complete FPGA designs. In this work we model the application of a set of important loop transformations, *unrolling*, *tiling*, and *interchange*. We explicitly take into account the impact of the transformations on the limited resources of the design's memory interfaces.

This paper makes the following specific contributions:

- It presents an area and performance modeling approach that combines analytical and estimation techniques for complete FPGA designs.
- It describes the application of the proposed modeling to the mapping of computations to FPGAs in the presence of loop *unrolling*, *tiling*, *interchange* and *fission*.
- It validates the proposed modeling for a sample case study application on a real Xilinx Virtex™ FPGA device. This experience reveals our model to be very accurate even when dealing with the vagaries of synthesis tools.

Overall this paper argues that performance and area modeling for complete designs, either for FPGA or not, is an instrumental technique in handling the complexity of finding effective solutions in the current reconfigurable as well as future architectures.

This paper is organized as follows. In section 2 we describe the analysis and modeling approach in detail. In section 3 we present experimental results for a sample image processing kernel and – binary image correlation. In section 4 we describe related work and conclude in section 5.

2 Modeling

We now describe our analytical execution time and area modeling for complete designs as implemented in our target FPGA system.

2.1 Target Design Architecture and Execution Model

The designs considered in our modeling are composed of two primary components as illustrated in Figure 2(a). The first component is the *core datapath* that implements the core of the computation and is generated by the synthesis tools from a high-level description such as VHDL. This datapath is connected to an external memory via an *memory interface* component. This memory interface (see [1]) is responsible for generating the physical memory addresses of various data items as well as the electrical signaling with the external memory.

In many computations, such as image processing kernels, there is a substantial opportunity to reuse data in registers across iterations of a given loop. This is the case, for example, in window-based algorithm in which a computation is performed over shifted windows of the same image. The overlap between windows allows for a subset of the data to be reused in registers, therefore avoiding to load all of the data from memory. For computations with these features, our datapath implementations include an internal *tapped-delay line*.

Our model exploits the pipelined execution mode of memory accesses and the core datapath. To adequately support this execution mode for the class of regular image processing computations, our memory interface supports the concept of *streamed data channels* or simply *streams*. Associated with each of these *stream* the memory interface include resources to generate the corresponding sequence of memory addresses. Setting up and resetting these resources whenever the datapath requires data from channels not currently set-up (or set up at a different base address in memory) incurs an overhead. The important parameters for the performance modeling of the pipelined execution mode of the datapath are its *initiation interval* and *latency*. As to the memory interface, its important parameters are the *reloading overhead*, the *read* and *write* latencies and the *setup* for reading and writing in pipelined memory access mode.

Figure 2(b) illustrates the basic parameters of the execution model, which will be the basic parameters for the performance modeling described in the next section. We show a 2-channel implementation with 2 *buffers* in the datapath,

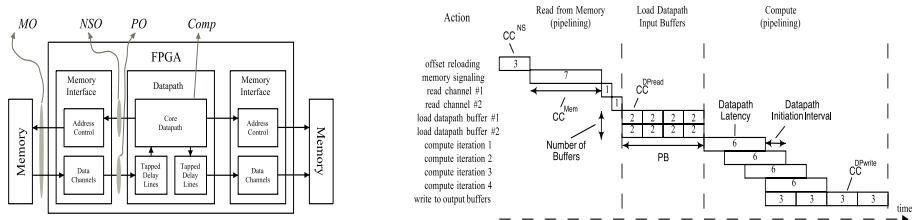


Fig. 2. (a) FPGA Design Architecture (b) Execution Mode

each with a depth of 4 elements and corresponding to 2 datapath input ports. The computation executes 4 loop iterations per block of data read from memory.

2.2 Performance Modeling

As with other compile-time modeling approaches we assume known compile-time loop bounds. In addition we assume that in the presence of control-flow the expressions must capture the longer execution path and that the datapath implementation will preemptively fetch all of the data items required in each of the possible control paths. While in general these assumptions would lead to excessively inflated performance results, for the target set of digital image applications, and due to their amenability to be modeled into perfectly loop nests without severe control flow, this potential phenomenon is rare, if at all observed. Under these assumptions we split the overall execution into 5 components summarized below and whose analytical expressions are presented below.

$$\begin{aligned}
 Comp &= LoopCnt * initInter \\
 NSO &= \sum_{0 \leq i \leq m} LoopCnt_i^{NS} * CC^{NS} \\
 PO &= \prod_{0 \leq i \leq m} LoopCnt_i^P * max(BD_j^B) * CC^{D\text{Pread}} \\
 MO &= \sum_{0 \leq i \leq m} LoopCnt_i^{Mem} * CC^{Mem} \\
 Exec &= (Comp + NSO + MO + PO) * Clk
 \end{aligned}$$

Computation Time (Comp) models the aggregate time the datapath spends actively computing results. We define it as the product of the overall number of iterations times the datapath initiation interval.

Non-Streaming Access Overhead (NSO) models the aggregate overhead in the reloading of the base and offset for non-streaming memory accesses and is defined by the NSO expression. In this expression $LoopCnt_i^{NS}$ captures the number of iterations the datapath performs a non-streaming memory accesses; and CC^{NS} is the individual cost of such accesses. The value m is the total number of array variables in the particular implementation with non-streaming accesses. For streaming memory access this metric is defined as 0.

Prologue Overhead (PO) models the overhead of filling buffers associated with a data stream before the datapath is ready to start its pipelined execution. The component PO defined above captures this cost of buffer preloading where $LoopCnt_i^P$ corresponds to the iteration count of the transformed loop nest that requires prologue loading. BD_i^B is the buffer depth of the B buffer (in the datapath) for variable i . The factor $\max(BD_i^B)$ defines the maximum length of time required to fill the longest buffer.

Memory access overhead (MO) models the latency on the memory interface for retrieving data from the memory. In the MO expression LC_i^{Mem} is the ratio of the number of memory accesses (or loop count accounting the memory access) to that of the granularity, for a variable i . Granularity is defined as the ratio of the data width of memory access to that of the data consumed in the datapath. Granularity is 1 if the data width of the memory access is the same as that of the data consumed in the datapath. CC^{Mem} is the number of cycles required for a memory data access.

Clock rate (Clk) We include this metric in the evaluation of the performance of a design as different designs will have can exhibit a wide disparity of clock rates due to radically different internal hardware implementations. Conducting relative comparison based on the clock cycles alone could lead to the wrong decision in selecting the best performing design.

Execution Time (Exec) simply defines the aggregate execution time and is simply the product of the number of execution cycles with the actual values of *Clk* as derived from the synthesis tools.

2.3 Area Modeling

This modeling is essential so that a compiler can use the area estimates derived from the synthesis tool and combine the predicted area for the memory interface when judging the area of a complete FPGA design.

To achieve this goal, we have derived a linear model using linear regression for a set of data points for various choices of number of memory interface channels. For 32-bit wide data channels the overall FPGA (Xilinx VirtexTM 1K device) area (in slices) for the memory interface can be approximated by the expression $Area_{32} = 137 * NumberChannels + 1514$. A similar empirical approach can be used for other channels widths and FPGA devices.

As to the modeling of the area for the datapath we use estimation results provided by synthesis tools such as the Mentor Graphics' MonetTM tool. The overall area estimation therefore combines the empirical model for the memory interface with the synthesis estimates.

2.4 Deriving the Model Parameters

In the current implementation we manually apply the various loop transformations, and derive the parameters of our modeling effort are derived using a crude emulation of the actual execution by actually running the computation in software and accumulating the number of occurrences of each metric. We are in the process of automating the application of the transformations and developing more sophisticated, and automated, approach to extract the values of the modeling parameters.

The modeling parameters that depend on the target FPGA device such as the maximum clock rate or the actual implementation of the memory interfaces (*e.g.*, the latency or the number of cycles for non-pipelined operations) are architecture dependent and known at compile time. The remaining model parameters are either derived from analysis of the source code, as is the case of the classification of which array data accesses are non-sequential, where the compiler can use data dependence analysis as described in [2].

3 Case Study: Binary Image Correlation

We now validate the proposed performance and area modeling for one image processing kernel — a binary image correlation (BIC) computation.

3.1 Original Computation and Role of Loop Transformations

Figure 3 depicts the pseudo-code for the example under study. This computation consists of a 4-loop nest with known loop bounds and implements binary image correlation using a `mask` variable (2D `mask` array) over an input image (2D `image` array). The computation scans the input image by sliding an $e \times e$ window over the input and accumulates the values of the corresponding image window for non-zero `mask` values in `th` (another 2D array variable).

The input image(`image[m+i][n+j]`) is accessed in a row-wise fashion. Unrolling the `j`-loop fully, $e - 1$ of the `e` values in a single row, can be reused in consecutive iterations of the `n`-loop if the `e` values are stored in a *tapped-delay line*. And, as the data access is row-wise, unrolling of the `i`-loop enhances the performance by increasing the reuse to 2 dimensions. As a consequence, after the first iteration (where $e \times e$ data are loaded), only `e` data values corresponding `e` data streams needs to be loaded in the sub-sequent iterations of the `n`-loop.

```

for m = 0 to m < (t-e)
  for n = 0 to n < (t-e)
    for i = 0 to i < e
      for j = 0 to j < e
        if(mask[i][j] != 0)
          th[m][n] += image[m+i][n+j];
    
```

(a) Original code.


```

for m = 0 to m < t
  for q = 0 to q < t by e
    for p = 0 to p < s by f
      for n = q to e-1
        for i = p to p+f-1 // unrolled loop
          if(mask[i][0] != 0)
            th[m][n] += image[m+i][n];
        ...
      if(mask[i][s-1] != 0)
        th[m][n] += image[m+i][n+s-1];
    
```

(b) Using $e \times f$ tiling.


```

for p = 0 to s/k
  for m = 0 to t
    for n = 0 to t
      for i = p*(s/k) to p*(s/k)+k // fully unrolled
        for j = 0 to s // fully unrolled
          if(mask[i][j] != 0)
            temp[p][m][n] += image[m+i][n+j];
    
```

// loop-fission and loop interchange

```

for m = 0 to t
  for n = 0 to t
    for p = 0 to s/k // fully unrolled
      th[m][n] += temp[p][m][n];
    
```

(c) Loop fission after interchange (loops `m` and `p`)

Fig. 3. BIC codes after applying loop transformations.

Full unrolling of the two inner most loops, as suggested above, may not always be possible as unrolling leads to the replication of the loop body operators generating very large designs. As such we can apply tiling of the innermost loops as illustrated in Figure 3(b). By tiling the `i` and `j`-loops by a factor of `e` and `f` respectively, we reduce the amount of hardware resources devoted to the datapath. The trade-off, however, is in terms of reduced reuse. The tapped-delay lines that, in case of unrolling, held the data of a row of `image`, now holds data of length `e`. But, of greater significance is the fact that the execution time increases as the memory interface resources, associated with address generation, requires reloading of address (and as a consequence increased data access) for the data streams associated with each tile. The performance of the overall design is thus substantially reduced.

In the presence of associative operations it is possible to use loop interchange with loop tiling to avoid the need of reloading the data in a tile. The implementation saves partial results of the computation in temporary datapath buffers thereby avoiding the need to reload data from external memory. This strategy comes at the cost of a, potentially large, multi-dimensional array for storing the partial results in the datapath. The size of this temporary array is in the order of the input image size, making it an infeasible solution for most cases.

To mitigate this issue of buffer space, we use yet another set of transformation: array privatization (to privatize temporal variable) and loop fission. The privatized temporal value is now stored in the external memory rather internal buffers (we pay the price for memory access). A second loop generates final results by adding up temporal values in the correct order.

The application of these 3 loop transformations to this case study illustrates the point of this paper. As these transformations affect the data reuse patterns and, in turn, the way the data needs to be streamed in and out of the corresponding datapath implementation, the overall performance in FPGA-based implementations is not only affected by the number of iterations executed but also, and infact more significantly, by the number of times the memory interface resources need to reset.

3.2 Experimental Results

We have manually derived behavioral specification for the BIC code applying the loop transformations described above for a variety of parameter choices. We then use Mentor Graphics' Monet high-level synthesis tool to derive the structural VHDL specification and obtain the estimated area and initiation interval and latency for their pipelined implementation. To derive a complete design we merge the structural VHDL with the structural code of the memory channel interfaces. Given a complete VHDL design we used Synplicity Synplify Pro 6.2 and Xilinx ISE 4.1i tool sets for logic synthesis and Place-and-Route (P&R) [3] targeting a Xilinx VirtexTM XCV 1000 BG560 device.

Performance Model Validation. We validate the performance modeling describe in section 2 for the VHDL code resulting from the application of the various loop transformations described above. In this modeling we use the numerical parameters as, $CC^{DPR} = 2$, $CC^{Mem} = 7$ and $CC^{NS} = 3$ and omit the symbolic loop expressions here for space considerations.

Figure 4 plots the measured vs. predicted performance for the various, loop unrolled (for the inner loop); tiled (the i and j loops and tiled-with-interchange. The vertical axis corresponds to the simulated clock cycle counts. We limit our experimental set to tiling versions of $1 \times f$ because other transformations do not deliver additional performance.

These results indicate that our performance modeling tracks the overall performance for the various implementations very well. The gap between the predicted performance and simulation results is due to our channel controller behavior. The channel controller used in our memory interface uses a round-robin

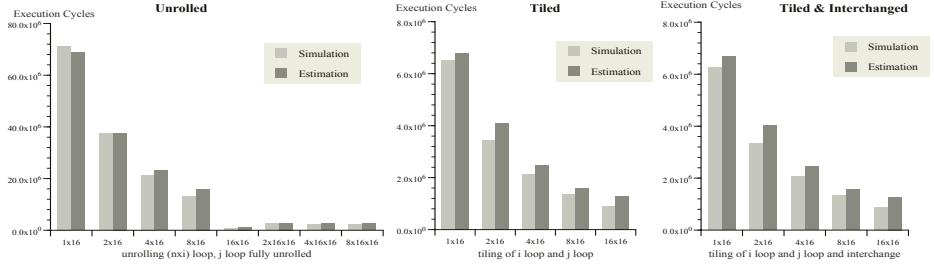


Fig. 4. Performance Estimation .vs. Simulation Results for BIC.

scheduling strategy which introduces a small number of additional cycles, as we verified through simulation.

Area and Implementation Results. We now validate the overall area estimation approach using the empirical model for the memory interface and the area estimation extracted from behavioral synthesis. In this validation we compare the results a compiler would obtain off-line (*i.e.*, without actually synthesizing any designs) with the real results synthesizing the actual complete designs.

Table 1 presents the synthesis results and complete design results for the various tiled and tiled&interchange designs. For the tiled&interchange design we present here only the design corresponding to the first loop that carries out the bulk of the computation. Area estimates produced by Monet behavioral synthesis tool are not compatible, in terms of units, with those of the P&R tools. However, these estimates to a large extent do capture the 'trend' of the results(area) produced by the synthesis and P&R tools as is evident from table 1. Table 1 compares the estimation numbers with those of the synthesis and P&R numbers

Table 1. Synthesis results for tiling (top) and tiling with interchanging and fission (bottom).

Tile	Datapath Only		Interface Estimates (Monet)	Full Design		
	Estimates (Monet)	P&R (Slices)		Estimates (Monet)	P&R (Slices(%))	Clk (MHz)
(1x1)	14621	4214	1788	16409	4558 (37)	35.3
(1x2)	15804	3808	2602	18406	6661 (54)	29.9
(1x4)	18191	5163	2610	20801	7070 (57)	21.7
(1x8)	22868	6369	3706	26574	9734 (79)	28.3
(1x16)	19704	5053	5898	25602	10417 (84)	24.2

Tile	Datapath Only		Interface Estimates (Monet)	Full Design		
	Estimates (Monet)	P&R (Slices)		Estimates (Monet)	P&R (Slices(%))	Clk (MHz)
(1x1)	1411.1	408	1788	3199.1	2326 (19)	43.4
(1x2)	2599.1	725	2602	5201.1	2984 (24)	36.1
(1x4)	4963.1	1333	2610	7573.1	4038 (33)	26.6
(1x8)	9694.9	2620	3706	13400.9	6119 (50)	16.7
(1x16)	19704.0	5053	5898	25602.0	10417 (84)	24.2

for the tiled implementations. Estimation numbers for datapath and interfaces are provided in columns 2 and 4 while columns 3 and 6 are the synthesis and P&R results for datapath only and full design (datapath+interface) respectively. Column 5 is the estimation number for the full design (col. 2 + col. 4).

As can be seen the combined estimation results for the complete designs track very well the real area results even for the very large designs that occupy more than 50% of the FPGA area. This is important as the application of loop transformations that expose vast amount of instruction level parallelism require the replication of functional operators and invariably lead to large designs. For some of the implementations the internal tapped-delay lines used to reduce the number of memory accesses is an important factor on the area as well.

We investigated the fact that the (1×8) tiled-version maps to more slices than the (1×16) version (see table 1 (top) col. 3). We concluded that although the computational for the (1×8) version are approximately half of those for the (1×16) version, the area consumed by the registers used to store intermediate results across the m-loop were responsible for this area anomaly.

3.3 Discussion

Experimental results reveal that our performance and area modeling correlates well with our simulation/synthesis results for various implementations of BIC. Our performance model is capable of identifying effects of various loop transformations, which ultimately lead to the best combination of loop transformations. In the BIC case study our model accurately captures the effect of tiling with different tiling factors, and we can find the best tiling shape/s. The results also reveal that our modeling successfully captures the performance behavior of the more sophisticated combination of loop interchange, fission and privatization.

Overall, we believe the modeling approach described in this paper to be applicable to a wider range of reconfigurable architectures as our model does not exploit any feature of the underlying architecture. In fact, our case study reveals that the major source of discrepancies was either due to characteristics of FPGA synthesis (*e.g.* the large impact of temporary buffers) or due to FPGA implementation execution features (*e.g.*, scheduling of memory accesses).

4 Related Work

Other researchers have also recognized the need for fast area and performance estimation to guide the application of compiler high-level loop transformations for deriving alternative designs. Derrien and S. Rajopadyhe[4] describe a processor array partitioning that takes into account the memory hierarchy and I/O bandwidth and apply tiling to maximize the performance of the mapping of a loop nest onto FPGA-based architectures. In this context they use an analytical performance model to determine the best tile size. So *et. al.* [5] have expanded the loop transformations to include unrolling and use behavioral synthesis estimates directly from commercially available synthesis tools in an integrated compilation and synthesis. The PICO project [6] takes functions defined as C loop nests

to a synchronous array of customizable VLIW processors. PICO uses estimates from its scheduling of the iterations of the nest to determine which loop transformation(s) lead to shorter scheduling time and therefore minimal completion time. The MILAN project [7] provides design space exploration and simulation environments for System-on-Chip(SoC) architecture. MILAN evaluates several possible partitions of the computation among the various system components (processor, memory, special purposed accelerators) using simulation techniques to derive estimates used in the evaluation of a given application mapping.

The work presented in this paper differs from these approaches in several respects. While other projects have focused on more general computation we have focused on the domain of image processing algorithms specified as tight loop nests. Second, rather than using profiling or simulation based estimates we have analytically modeled the performance and relied on the accuracy of behavioral synthesis estimation commercial tools for area modeling. In terms of loop transformations we have focused not only on loop unrolling and tiling but also on loop interchange in the presence of associative operators. Loop interchanging introduces the complication of having to save intermediate results for subsequent computations. Designs with large set of registers for temporary values lead to an explosion of area and substantial degradation of clock estimates.

5 Conclusion

In this paper we presented a performance and area modeling approach using analytical and empirical techniques for complete FPGA designs. Our modeling is geared towards computations expressed as loop nests in high-level programming languages such as C. Using the proposed modeling, compiler tools can evaluate the impact of multiple loop transformations on FPGA resources as well as that of the memory interface resources, often the limiting factor in the effective implementation of these computations. The preliminary results reveal that our approach delivers area and performance estimations that correlate very well with the corresponding metrics from the actual implementation. This experience suggests the proposed modeling approach to be an effective technique that allow compilers to quickly explore a wider range of loop transformations for selecting feasible and high-performance FPGA designs.

References

1. Park, J., Diniz, P.: Synthesis of Memory Access Controller for Streamed Data Applications for FPGA-based Computing Engines. In: Proc. of the 14th Intl. Symp. on System Synthesis (ISSS'2001), IEEE Computer Society Press (2001)
2. Diniz, P., Park, J.: Automatic synthesis of data storage and control structures for FPGA-based computing machines. In: In Proc. IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'00), IEEE Computer Society Press (2000)
3. (Virtex 2.5v FPGA product specification. ds003(v2.4)) Xilinx, Inc., 2000.
4. Derrien, S., Rajopadyhe, S.: Loop tiling for reconfigurable accelerators. In: Proc. of the Eleventh Int. Symp. on Field-Programmable Logic (FPL'2001). (2001)

5. So, B., Hall, M., Diniz, P.: A compiler approach to fast hardware design space exploration for fpga systems. In: Proc. of the 2001 ACM Conference on Programming Language Design and Implementation (PLDI'00), ACM Press (2001)
6. Kathail, V., Aditya, S., Schreiber, R., Rau, B., Cronquist, D., Sivaraman, M.: PICO: Automatically designing custom computers. In: IEEE Computer. (2002)
7. Bakshi, A., Prasanna, V., Ledeczi, A.: Milan: A model based integrated simulation framework for design of embedded systems. In: Proc. of the ACM Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001). (2001)

Branch Optimisation Techniques for Hardware Compilation

Henry Styles and Wayne Luk

Department of Computing, Imperial College, 180 Queen's Gate, London, England

Abstract. This paper explores using information about program branch probabilities to optimise reconfigurable designs. The basic premise is to promote utilization by dedicating more resources to branches which execute more frequently. A hardware compilation system has been developed for producing designs which are optimised for different branch probabilities. We propose an analytical queueing network performance model to determine the best design from observed branch probability information. The branch optimisation space is characterized in an experimental study for Xilinx Virtex FPGAs of two complex applications: video feature extraction and progressive refinement radiosity. For designs of equal performance, branch-optimised designs require 24% and 27.5% less area. For designs of equal area, branch optimised designs run upto 3 times faster. Our analytical performance model is shown to be highly accurate with relative error between 0.12 and 1.1×10^{-4} .

1 Introduction

For most computer programs, the execution frequency of each basic block is controlled by the runtime behavior of conditional branches. Optimal resource allocation between basic blocks requires that execution frequencies be known. Software profilers collect execution frequencies for a representative dataset to support static resource allocation. Microprocessors demonstrate that branch probability information can be used at runtime to aid dynamic resource allocation. In this paper we explore the analogous use of branch probability information to optimize resource allocation in hardware compilation. In particular, the novel aspects of our work include:

- a compiler that maps programs written in a subset of C to a set of hardware designs that are optimised for different branching probabilities;
- analytical methods, including a queueing network model, for elucidating the properties of the proposed compilation procedure;
- evaluation of our approach based on both analytical and experimental methods for two large applications: video feature extraction and radiosity.

The rest of the paper is organised as follows. Section 2 describes the two basic compilation phases: dependency analysis and circuit synthesis. Section 3 then presents the branch-optimised compilation path. Section 4 deals with the models for studying the analytical properties of this compilation procedure, and is followed by Section 5 which evaluates both analytically and experimentally the effectiveness of the proposed approach. Finally, Section 7 summarises our current and future research.

2 Compilation

Our compilation procedure consists of two phases: dependency analysis and circuit synthesis. The input language for the compiler is a streaming subset of the C language in which arbitrary pointers and loop carry dependencies are not supported. Each input program specifies the body of a single loop, with flow control specified by an *if..then..else* branch construct. These restrictions preclude certain types of program such as the Fibonacci generator, however an extensive set of applications can be automatically transformed [10] into this form. A simple example program, shown on the left of Fig. 1, will be used to illustrate our compilation procedure in the following sections.

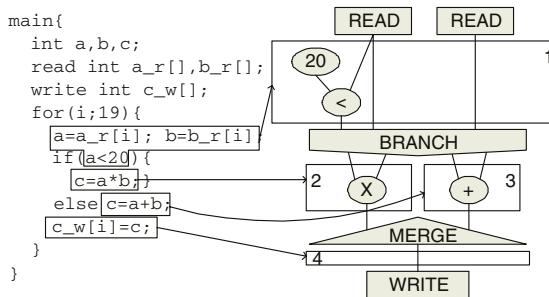


Fig. 1. A simple example program and its two-level data flow graph. The four basic blocks map to four numbered DAG subgraphs. The *if..then..else* maps to BRANCH and MERGE nodes. Array accesses map to READ and WRITE nodes.

The dependency analysis phase constructs a two-level data flow graph from the input program. The data flow graph for our simple example program is shown on the right of Fig. 1. It includes a numbered direct acyclic graph (DAG) for each basic block. Flow control between DAGs is represented by BRANCH and MERGE nodes with firing rule semantics as described in data flow computing literature [9]. Reads and writes to vector variables at the start and end of the data flow graph are mapped to READ and WRITE nodes.

The circuit synthesis phase transforms the dependency graph into a unidirectional pipeline captured in structural VHDL. It consists of module selection, scheduling, binding and instantiation of appropriate flow control circuits. The *initiation interval* of a library block is the number of cycles between each output. An XML library block database specifies the initiation interval, latency in cycles, and area of available library blocks. A static pipelined list scheduler [2] is provided for basic block scheduling.

Circuit synthesis is specialized to form two compilation paths: control study compilation path and branch-optimised compilation path. The control study compilation path is inspired by the StReAm [4] compiler. It creates pipelines which perform equally well under all branching conditions. Designs are parameterised with a global initiation interval parameter $b_{pipeline}$. The control study compilation path circuit with $b_{pipeline} = 1$ for our simple example program is shown on the left of Fig. 2.

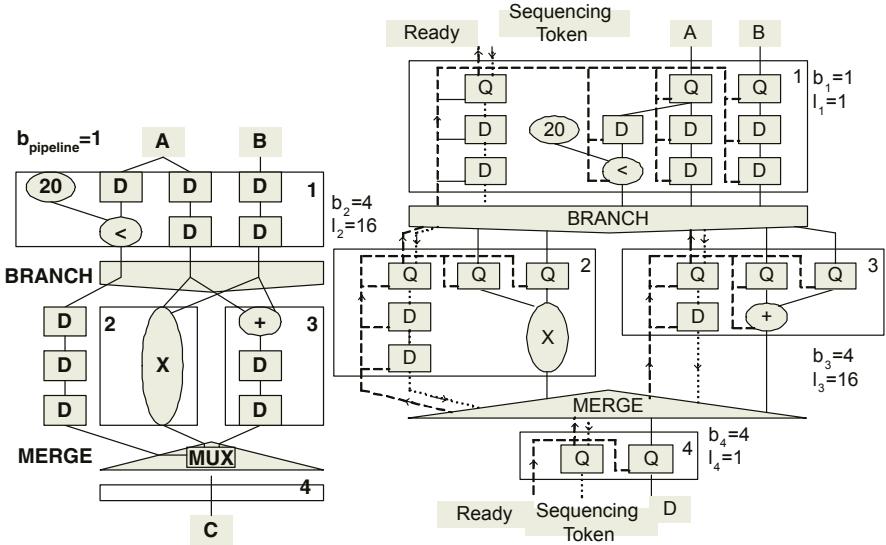


Fig. 2. Uniform (left) and multiple (right) rate circuits for the simple program of Fig. 1. For the control study compilation path, additional flip-flops (D) are inserted to synchronise data flow. For the branch-optimised compilation path, rate-smoothing queues (Q) and sequencing token (dotted line) are inserted to synchronise data flow. The add and multiply modules are pipelined and the flow can be halted by the *ready* control signal (dashed line) shown as an additional input to these components.

3 Branch-Optimised Compilation Path

In this section we introduce a new compilation scheme which promotes efficiency in the presence of branch probability information. A branch-optimised circuit for the simple example program is shown on the right of Fig. 2. The branch-optimised compilation scheme transforms the data flow graph into a set of hardware configurations in which different basic blocks run at different initiation intervals. From this set, a configuration can be chosen in which the resources assigned to different branches match the observed computational load. The branch-optimised compilation scheme creates designs with the following characteristics.

1. Basic blocks can run at independent rates, with different degrees of sequentialization. The rate of basic block i is controlled by the initiation interval parameter b_i .
2. Each basic block propagates a sequencing token downwards, shown as a dotted line in Fig. 2. Different paths through the pipeline run at different rates, and so computations may retire out of order. The sequencing token identifies the loop index associated with a set of results at the pipeline outputs, enabling the original ordering to be recovered. The width of the sequencing token is determined from the upper loop bound and maximum length path through the compiled design.
3. Basic blocks have rate smoothing FIFOs, labeled Q in Fig. 2, for the sequencing token and data inputs. The FIFO length for basic block i is given by parameter l_i .

4. Each basic block propagates a ready signal back up through the pipeline, shown as a dashed line in Fig. 2. The ready signal allows basic blocks to stall incoming computation when input queues are full. For each basic block, the incoming ready signal fans out to the clock-enable input of all registers in the datapath. In our current implementation we adopt a fully synchronous design style. However, a globally asynchronous locally synchronous (GALS) design style could potentially be adopted, in which each basic block operates in a separate clock domain and the ready signal is replaced with true asynchronous handshaking.
5. The BRANCH node routes sequencing token and data to the branch target specified by the branch condition. It receives ready signals from the two branch targets, and blocks computation if the branch target set by the branch condition is not ready.
6. The MERGE node forwards data and sequencing tokens from true and false branch targets. If sequencing tokens arrive from both branch targets simultaneously, the MERGE node blocks the branch targets alternately in a round robin fashion.

4 Analytical Modeling

In this section we describe analytical models of the area-throughput design space for the control study and branch-optimised compilation paths. These models are used to determine the best compilation path and parameterization from observed branch probability information. In the experimental study presented in Section 5, branch probability information is collected at compile time by profiling. In a future system, branch probability information could be collected and acted upon at runtime. Analytical techniques are of increasing importance, as severe time constraints on the optimisation process would almost certainly preclude more complex modelling.

We model the cycle count throughput of branch-optimised designs using a queueing network model. Branch-optimised designs introduce finite queue lengths, blocking, and the possibility of correlated arrival rates. Queueing networks which model these properties are generally solved by simulation [7]. We adopt a simple analytical model based on a $M/M/1/\infty/FCFS$ queueing network with saturating external arrivals to node one [5]. Given information about steady state branch probabilities, known variables in the model are:

1. The node initiation intervals vector $b \in \Re^N$. In the model, element b_i is the exponentially distributed mean initiation interval of node i and b_i is set as the basic block initiation interval for block i .
2. The “routing matrix” $Q \in \Re^{N \times N}$. In the model, element Q_{ij} is the steady state probability that a job, completing node i , routes to node j . A BRANCH after node x to select between branch targets y and z is modeled with $Q_{xy} + Q_{xz} = 1$. The summation of probabilities $q_i = \sum_{j=1}^N Q_{ij} \leq 1$ where $i = 1, 2, \dots, N$. If $q_i < 1$, then a job, on completing node i , exits the queuing network with probability $1 - q_i$. Q is filled with the known branch probability information.

To estimate performance, we determine the maximum sustainable external arrival rate to node one. In the model, external arrival rates are captured in $\gamma \in \Re^N$ where element γ_i is the Poisson process mean external arrival rate for node i . For compiled

designs, γ_i is of the form $\boldsymbol{\gamma} = [\gamma_1, 0, 0, \dots]$. The procedure to determine the maximum sustainable value of γ_1 is as follows.

1. Solve the traffic equations (eq.1) to determine the net arrival rate at each node in terms of the external arrival rate at node one. The mean net arrival at each node is an element in $\boldsymbol{\lambda} \in \Re^N$. An equation is formed for each element λ_i in terms of γ_1 .

$$\boldsymbol{\lambda}(I - Q) = \boldsymbol{\gamma} \quad (1)$$

2. Determine the maximum arrival rate at node one given that the utilization of each node is less than or equal to one. In the model, the utilization of each node is an element in $\boldsymbol{\rho} \in \Re^N$. We maximize γ_1 subject to the utilization constraint (eq.3).

$$\rho_i = \lambda_i b_i \quad (2)$$

$$\rho_i \leq 1 \quad i = 1, 2, \dots, N \quad (3)$$

Any design with $\rho_i = 1, i \neq 1$ for maximum λ_1 will exhibit steady state blocking.

The control study compilation path is parameterised with the global initiation interval $b_{pipeline}$. Designs sustain throughput $1/b_{pipeline}$ for all branching probabilities.

5 Case Studies

In this section we compare the performance of both compilation paths and evaluate the accuracy of analytical models for two case study applications. The input programs and their corresponding top-level data flow graphs for the case study applications are shown in Fig. 4 and Fig. 5. The test scenes are shown in Fig. 3.



Fig. 3. Left and center panes show Video Quality Expert Group test sequence 10 (VQEG10) before and after video feature extraction. Right pane shows the radiosity test scene.

Video Feature Extraction. The algorithm [11] consists of edge detection, thresholding and 3×3 sum-squared difference. There are four basic blocks and one branch.

Progressive Refinement Radiosity. Radiosity algorithms [8] simulate radiation of energy between surfaces. There are ten basic blocks guarded by three branches.

```

main{
    int gx,gy,g,ssd,t0 ..etc t8 ; //scalars
    read int f0_ulr[],f0_umr[],f0_ur[],f1_ulr[]; etc
    write int ssd_w[]; //vectors
    for(i;2^20){
        f0_u1=f0_ulr; f0_um=f0_umr; f0_ur=f0_ur;
        // sobel
        1 gx=abs((f0_ur-f0_u1)+((f0_mr<<1)-(f0_ml<<1))+(f0_lr-f0_ll));
        gy=abs((f0_ul+(f0_um<<1)+f0_ur)-(f0_ll+(f0_lm<<1)+f0_lr));
        g=gx+gy;
        //threshold
        if(f>220){
            // sum squared diff 9 MULTs 9 SUBs
            2 t0=(i0_mm-i1_ul); ...etc.. t8=(i0_mm-i1_um);
            ssd=(t0*t0)+(t1*t1)+...etc...(t8*t8));
            }else{ ssd=0; 3 }
        4 ssd_w[i]=ssd
    }
}

```

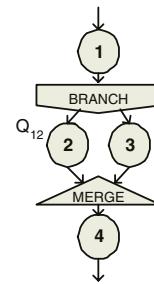


Fig. 4. Input program and the corresponding top-level data flow graph for video feature extraction.

```

main{
    //scalar declarations
    int orig_0,orig_1,orig2; etc
    //vector declarations.
    read int orig_0r[],orig_1r[],orig_2r[]; etc
    write int validhit_w[],u_w[],v_w[],t_w[]; etc
    for(i;2^22){
        orig_0=orig_0r[i]; orig_1=orig_1r[i]; etc.
        SUB(edge1, vert1, vert0); SUB(edge2, vert2, vert0);
        CROSS(pvec, dir, edge2);
        det = DOT(edge1, pvec);
        if (det >= EPSILON){
            1 SUB(tvec, orig, vert0);
            u = DOT(tvec, pvec);
            if ((u >= 0) && (u <= det)){
                2 CROSS(qvec, tvec, edge1);
                v = DOT(dir, qvec);
                if ((v >= 0) && ((u + v) <= det)){
                    3 t = DOT(edge2, qvec);
                    inv_det = 16777216 / det;
                    t = t*inv_det; u = u*inv_det; v = v*inv_det;
                    validhit=1;
                    4 }else{ 5 }
                6 }else{ 7 }
            8 }else{ 9 }
        10 validhit_w[i]=validhit; u_w[i]=u; v_w[i]=v; t_w[i]=t;
    }
}

```

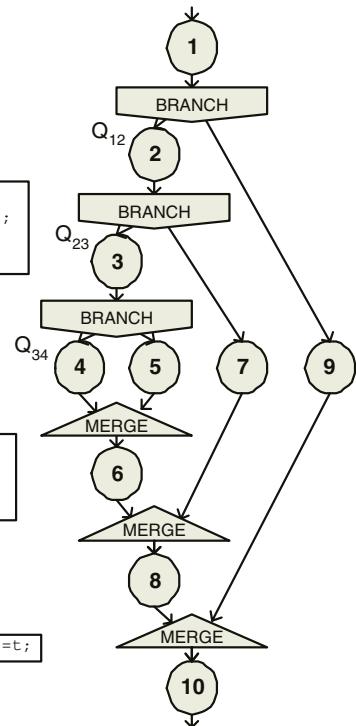


Fig. 5. Input program and the corresponding top-level data flow graph for radiosity Moller-Trumbore ray-triangle intersection.

6 Results

For the purposes of the experiments, all designs have a uniform word length of 32 bits. All results use the Xilinx XCV3200E-8 device. Arithmetic library blocks are generated using

Xilinx Core GENERATOR 5.1.02i, with $b_i = 1, 2, 4$, or 8 for multipliers and dividers. Other library blocks do not scale for initiation interval. VHDL output by the compiler is synthesised with Synplify Pro 7.1. Area and clock rate are collected from Xilinx 5.1i. Run-time basic block utilization and queue length behavior are observed by simulation using ModelSim SE Plus. A wrapper was constructed in Handel-C [1] to demonstrate designs on the RC1000-PP FPGA platform. The relative accuracy of the analytical model is calculated with the formula (*analytical* $\gamma_1 - \text{observed } \gamma_1$) / *observed* γ_1 .

Branch probability information was collected by profiling. The software implementation of the video feature extraction case study was profiled with the test sequence VQEG for 100 frames. Routing table entries relating to Fig. 4 are $Q_{12} = 0.0891$. The software implementation of the radiosity case study was profiled for 4 refinements, involving approximately 800K ray-triangle intersection tests. Routing table entries relating to Fig. 5 are $Q_{12} = 0.520$, $Q_{23} = 0.164$ and $Q_{34} = 0.367$.

The analytical and experimental results for both compilation paths and case studies are shown in Fig. 6 and Tables 1, 2, 3 and 4. Fig. 7 illustrates the effects of different probabilities on the performance of both compilation paths for the video feature extraction case study. The key results of the experimental study are as follows.

1. The branch-optimised compilation path automatically identifies the basic blocks that can benefit from branch probability information and produces designs with different parameterizations of b , the initiation interval vector. For the video feature extraction application, the compiler identifies basic block 2 and produces 10 different designs; for the progressive refinement radiosity application, the compiler identifies basic blocks 1, 2, 3 and 4 and produces 35 designs.

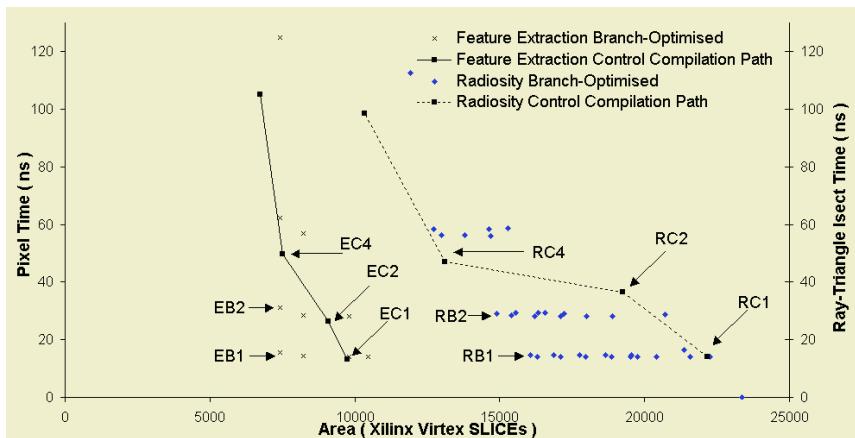


Fig. 6. Combined area-performance design space for video feature extraction (left) and radiosity (right) case study applications. The lines on the graph represent the control study compilation path designs, with b_{pipeline} varying from 1 to 8. The clusters of points are different branch-optimised designs with different parameterizations of b . EC1, EC2, EC4, EB1, EB2, RC1, RC2, RC4, RB1 and RB2 correspond to the optimal designs for each compilation path under different performance constraints as shown in Tables 1, 2, 3 and 4.

Table 1. Complete area-throughput design space with control study compilation path for video feature extraction case study, with input scenes shown in Fig. 3. Designs EC1, EC2 and EC4 are the smallest control study compilation path designs which meet performance constraints 64Mpixel/set, 32Mpixel/sec and 16Mpixel/sec. These designs are labeled in Fig. 6.

$b_{pipeline}$	Area (slice)	Clk (FFs)	Pixel Time (LUTs)	(ns)	(ns/pixel)	(Mpixels/sec)	
1	9753	11137	14623	13.32	13.316	75.10	EC1
2	9097	10006	11392	13.23	26.46	37.79	EC2
4	7514	7003	8764	12.47	49.86	20.06	EC4
8	6733	5405	7469	13.13	105.05	9.52	

Table 2. Selected area-throughput results for branch-optimised compilation path in the video feature extraction case study with input scene shown in Fig. 3. 10 designs are automatically generated. Designs EB1 and EB2 are the smallest branch-optimised designs which meet performance constraint 64Mpixel/sec and 32Mpixel/sec. Clock period for both designs is 13.96ns. ρ_1 can be calculated as $\gamma_1 \times b_1$. EB1 and EB2 are labeled in Fig. 6.

b	Area b_1 slice	Experimental Performance				Analytical Performance			
		Utilization ρ_2	Pixel Time γ_1	(ns/pix)	(Mpix/s)	Utilization ρ_2	Time γ_1	(ns/pix)	Error
1 8	7411 5562 8322	.635	.894	15.61	64.04	.7130 1	13.96	0.12	EB1
2 8	7411 5562 8322	.635	.447	31.22	32.03	.7130 0.5	27.92	0.12	EB2

Table 3. Complete area-throughput design space with control study compilation paths for radiosity case study with input scene shown in Fig. 3. Designs RC1, RC2 and RC4 are the smallest branch-optimised designs which meet performance constraint 70Mray-triangle intersections/sec, 33Mray-triangle intersections/sec and 17.5 Mray-triangle intersections/sec. RC1, RC2 and RC4 are labeled in Fig. 6.

$b_{pipeline}$	Area (slice)	Clk (FFs)	Intersection Time (LUTs)	(ns)	(ns/Isect)	(MIsects/sec)
1	22182	34338	34,823	14.12	14.12	70.84 RC1
2	19239	28638	25,238	18.27	36.55	27.36 RC2
4	13116	18852	16,070	11.78	47.13	21.22 RC4
8	10340	13657	11,809	12.32	98.53	10.15

2. For a given area, branch-optimised designs can often run significantly faster than non-branch-optimised designs. In Fig. 6, for instance, EB1 (7411 slices) is slightly smaller than EC4 (7514 slices), and at 15.61 ns/pixel is more than 3.2 times faster than EC4 at 49.86 ns/pixel. Similarly while RB1 and RB2 are respectively 22.6% and 13.5% larger than RC4, they run 322% and 162% faster than RC4.
3. For a given performance, branch-optimised designs often require smaller areas than non-branch-optimised designs. In Fig. 6, for instance, at 64 Mpixels/sec EB1 is 24% smaller than EC1 and at 32 Mpixels/sec EB2 is 18% smaller than EC2. Similarly at 70 Mray-triangle intersections per second, RB1 is 27.5% smaller than RC1 while at 35 Mray-triangle intersections per second, RB2 is 27.5% smaller than RC2.

Table 4. Selected area-throughput results for branch-optimised compilation path, radiosity case study with input scene in Fig. 3. 35 designs are automatically generated. Designs RB1 and RB2 are the smallest branch-optimised designs with approximate performance 70Mray-triangle intersections/sec and 35Mray-triangle intersection/sec. ρ_1 can be calculated as $\gamma_1 \times b_1$. Processing rate can be calculated as $1/\text{Itime(ns)}$. RB1 and RB2 are labeled in Fig. 6.

b	Area	Clk	Experimental				Analytical				
			Utilization	Itime	Utilization	Itime	Relative				
b_1	b_2	b_3	b_4	slice	ns	ρ_2	ρ_3	ρ_4	γ_1	ns	Error
1	2	8	8	16074	14.09	.99	.657	.241	.96	14.65	5.3×10^{-4}
2	4	8	8	14888	14.03	1	.328	.121	.48	29.16	8.1×10^{-5}
4	8	8	8	12723	14.08	1	.164	.060	.24	58.51	8.1×10^{-5}
8	8	8	8	11924	14.08	.52	.085	.031	.13	112.7	1.0×10^{-6}

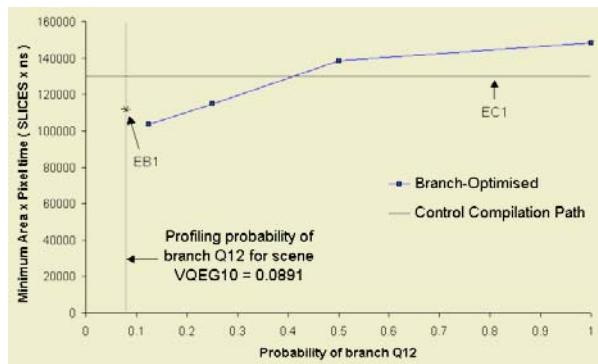


Fig. 7. Minimal area-time versus probability of branch Q_{12} for branch-optimised and control study compilation paths. Video feature extraction case study application is shown with performance constraint of 64Mpixels/sec. The observed probability for Q_{12} of 0.0891 is indicated with a vertical line through the graph. EC1 and EB1 correspond to the optimal designs for each compilation path as shown in Table 1 and Table 2. The trend line for branch-optimised compilation path with different probabilities is produced using our analytical model. The intersection of trend lines for branch-optimised compilation path and control study compilation path shows that branch-optimised compilation is favourable when $Q_{12} < 0.41$. As the probability Q_{12} decreases, branch-optimised compilation becomes increasingly attractive. EB1 performs worse than the analytical model trend line due to intermittent blocking.

4. The analytical performance model is shown to be accurate. For video feature extraction, the relative error varies between 0.12 and 2.4×10^{-5} ; for progressive refinement radiosity, the worst case relative error is smaller than 1.1×10^{-4} .
5. As the probability of a branch tends towards zero or one the branch becomes more biased and branch-optimised compilation becomes more attractive. Fig. 7 shows that for video feature extraction, branch-optimised compilation is favourable if branch probability Q_{12} is below a threshold of $Q_{12} < 0.41$. As Q_{12} tends towards zero, the performance gap between branch-optimised and non-branch-optimised designs increases.

7 Conclusion

This paper explores using branch probability information to optimise hardware compilation. We demonstrate that this technique can result in significant improvements in area and performance. Future work will focus on extending the analytical model and compilation system. In the long term we intend to develop a dynamically reconfigurable system in which branch optimisation techniques are applied at runtime.

References

1. Celoxica Limited, *Handel-C Language Reference Manual*, version 3.1, document number RM-1003-3.0, 2002.
2. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
3. T. Harriss, R. Walke, B. Kienhuis and E. Deprettere, “Compilation from Matlab to process networks”, *Design Automation for Embedded Systems*, Vol. 7, pp. 385–403, 2002.
4. O. Mencer, H Huebert, M. Morf and M.J. Flynn, “StReAm: Object-oriented programming of stream architectures using PAM-Blox”, in *Field-Programmable Logic: the Roadmap to Reconfigurable Systems*, LNCS 1896, Springer, 2000, pp. 595–604.
5. I. Mitran, *Probabalistic Modelling*, Cambridge University Press, 1998.
6. T. Moller and B. Trumbore, “Fast, minimum storage ray-triangle intersection”, *Journal of Graphics Tools*, 2(1), pp. 21–28, 1997.
7. R.O. Onvural, “Survey of closed queueing networks with blocking”, *ACM Computing Surveys*, 22(2), pp. 83–121, June 1990.
8. H. Styles and W. Luk, “Accelerating radiosity calculations using reconfigurable platforms”, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2002, pp. 279–281.
9. A.H. Veen, “Dataflow machine architecture”, *ACM Computing Surveys*, 18(4), pp. 365–396, 1986.
10. M. Weinhardt and W. Luk, “Pipeline vectorisation”, *IEEE Trans. on Comput.-Aided Design*, 20(2), pp. 234–248, February 2001.
11. H. Ziegler, B. So, M. Hall and P.C. Diniz, “Coarse-grain pipelining on multiple FPGA architectures”, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2002, pp. 77–86.

A Model for Hardware Realization of Kernel Loops

Jirong Liao, Weng-Fai Wong, and Tulika Mitra

Department of Computer Science, School of Computing,
National University of Singapore, 3 Science Drive 2, Singapore 117543,
`{liaojiro,wongwf,tulika}@comp.nus.edu.sg`

Abstract. Hardware realization of kernel loops holds the promise of accelerating the overall application performance and is therefore an important part of the synthesis process. In this paper, we consider two important loop optimization techniques, namely loop unrolling and software pipelining that can impact the performance and cost of the synthesized hardware. We propose a novel model that accounts for various characteristics of a loop, including dependencies, parallelism and resource requirement, as well as certain high level constraints of the implementation platform. Using this model, we are able to deduce the optimal unroll factor and technique for achieving the best performance given a fixed resource budget. The model was verified using a compiler-based FPGA synthesis framework on a number of kernel loops. We believe that our model is general and applicable to other synthesis frameworks, and will help reduce the time for design space exploration.

1 Introduction

A standard practice in synthesis of application specific hardware is to focus attention at kernel loops. In many applications, they account for the bulk of the execution time and are thus natural candidates for hardware acceleration. A key difficulty in synthesizing hardware for kernel loops is that there are many loop optimizations available and the complex interactions among these optimizations make it difficult to predict the cost-benefit of applying each. In particular, one cannot tell how much more or less resources a particular optimization will take or what its impact will be on performance. This means that one has to either settle for sub-optimal results or go through a costly process of trial-and-error in order to arrive at the correct combination of loop optimizations that fits the need of the user. Having a model of how a particular loop optimization will impact resource and performance is therefore necessary.

Two important loop optimizations applicable to kernel loops are *loop unrolling* and *software pipelining*. Loop unrolling is a technique to expand the loop such that a new iteration consists of 2 or more of the original iterations. This is performed by a compiler to expose more instruction level parallelism and reduce the overhead of updating index variables. The number of times the loop is expanded is called the *unroll factor*. If the loop iteration count is not a multiple

of unroll factor, then the remainder of the loop iterations needs to be executed at the end as it is.

Software pipelining [1] tries to achieve higher level of instruction level parallelism by moving operations across iteration boundaries. This optimization achieves overlap among the iterations by pipelining the execution of the iterations. The loop body is scheduled such that (a) all iterations have identical schedule and (b) each iteration is scheduled to start some fixed number of cycles later than the previous iteration. The delay between the start cycles of two successive iterations is called the *Initiation Interval (II)*. The modulo scheduling algorithm attempts to achieve the smallest value of II such that no intra- or inter-iteration dependencies and resource constraints are violated.

As multiple iterations are executed in parallel, both loop unrolling and software pipelining increase register pressure and resource requirement but in different ways. Furthermore, it is possible to use them in combination, i.e. it is possible to software pipeline unrolled loops. The complex interaction between the two optimizations makes it difficult to decide how they should be deployed in optimizing a loop given a particular resource constraint. Often the only way to tell is to exhaustively try various combinations of these two optimizations to obtain the optimal one.

In this paper, we propose a model for the performance and resource requirement for the hardware realization of unrolled and software pipelined loops. The novelty of our model lies in the use of the compiler to extract certain key parameters of the loop in question that characterize the code including the data dependences present for a given hardware. For example, the platform we use allows at most four parallel reads to memory and only if they do not hit the same memory bank. Such characteristics are hard to model. So instead we rely on the instruction scheduler of a compiler to capture these. From these parameters reported by the compiler, the model will inform the user if given a certain resource constraint, unrolling alone, or software pipelining used in combination with loop unrolling would deliver the better performance. It will also output the optimal unrolling factor that should be used. The contribution of this model is that without exhaustively trying a large number of possibilities, it can very quickly recommend a solution that we believe is optimal or very near it.

2 Related Work

Hardware realization of kernel loops has been actively studied by many research groups. However, the focus has been mainly on automatic synthesis of kernel loops from high level language constructs. The exploitation of compiler optimizations such as loop unrolling and modulo scheduling has largely remained unexplored. Even a few commercial synthesis tools that apply these compiler optimizations depend on user feedback to choose unroll factor or decide between unrolling and modulo scheduling. Our work bridges this gap in automatic hardware realization of kernel loops.

There are two main approaches towards hardware synthesis from high level constructs. One approach is to design new languages for hardware design which are at much higher level than traditional hardware description languages such as Verilog and VHDL. The claim is that the productivity gap will be reduced as software programmers can easily learn these new languages. An example is Handel-C [2] programming language which has C-like syntax with support for explicit hardware parallelism, communication, and hardware structures such as memory, bus etc.

The other approach attempts to map a subset of commonly used software programming languages such as C to hardware automatically. These efforts include SA-C [3], PipeRench C Compiler [4], Garp C compiler [5], work by Weinhardt et. al. [6] [7], Babb et. al. [8] and Snider et. al. [9]. The PACT project [10] at Northwestern University performs C to hardware synthesis by taking power/performance trade off into account. The PICO project [11, 12] performs static timing analysis to identify chain of operators to minimize number of cycles while maintaining cycle time constraints.

The only existing tool that allows application of high level compiler optimizations in hardware synthesis is Monet [13]. However, it requires user feedback in deciding unroll factor for example. Among research projects, Derien et. al [14] have developed an analytical model to choose a tiling strategy that will minimize loop execution time. The closest to our work is So et. al. [15]. They perform fast and automatic design space exploration to choose the right loop unrolling factor that satisfies the area constraints and maximizes performance. However, they do not use other compiler optimizations such as software pipeline which can potentially improve the performance significantly.

3 Our Model

In this section, we will present our proposed model. The novelty of the model lies in the use of key parameters supplied by the compiler in characterizing aspects of the kernel loop as well as the machine that are hard to model correctly.

3.1 Model for Performance

For the discussion below, we will assume a loop L that is executed N times. Let S_1 be the *schedule length* of the loop. In our model, S_1 is a quantity reported by the compiler as it performs instruction scheduling. As we are realizing the loop in hardware, we assumed infinite registers by skipping the traditional register allocation phase. In the quantity S_1 , various complex issues such as the machine's configuration, instruction type distribution, data dependencies etc. are encapsulated. The user, for example, can choose to use the machine configuration to constraint the amount of parallelism or number and types of functional units to be realized in hardware. We will also generalize S_1 to S_u which is the schedule length of the kernel when it is unrolled u times. The following formula gives the total number of cycles the unrolled kernel will take to execute N iterations.

$$C_{\text{unrolled}}(u) \approx \left\lfloor \frac{N}{u} \right\rfloor \times S_u + \left(N - \left\lfloor \frac{N}{u} \right\rfloor \times u \right) \times S_1 \quad (1)$$

After unrolling, the loop size is $\lfloor N/u \rfloor$ and the schedule length is S_u . Therefore the first term in Eq. 1 accounts for the total number of cycles executed by the unrolled loop. However, if N is not divisible by u , a compensation loop of size $N - \lfloor N/u \rfloor \times u$ and a schedule length of S_1 will be generated. In practice, we would not want to have to get all S_u 's from the compiler as that requires multiple runs. Rather, we estimate S_u given S_1 . In particular, we assumed that

$$S_u = S_{u-1} + c_S \quad (2)$$

where c_S is a constant. From the experience gained from our experimentation, we chose

$$c_S = \frac{(S_3 - S_1)}{2}$$

This is because we found that there may be a case where it so happens that empty resource slots available at the end of the instruction schedule can be filled up by a new instance of the loop.

To model software pipelining, we assumed the technique of *iterative modulo scheduling* given by Rau [1] that uses *predicated execution* and *rotating registers* [16]. It is characterized by two important parameters also obtained from the compiler, the initiation interval, II , and the epilog counter e . The initiation interval is the gap (in machine cycles) between two successive software pipelined iterations. In effect, after a successful modulo scheduling, each iteration of the software pipelined kernel loop takes exactly II cycles. The epilog count is the number of iterations in the epilog of the software pipelined loop. Again, in II and e , the complexity of machine configuration, resource requirements, and data dependencies are hidden away. Since we would like to combine software pipelining with unrolling, we will introduce II_u and e_u which are the II and e for a software pipelined loop that has been unrolled u times. We have the following formula for the total number of cycles a software pipelined loop that has been previous unrolled u times will take:

$$C_{\text{swp}}(u) \approx \left(\left\lfloor \frac{N}{u} \right\rfloor + e_u \right) \times (II_u + 1) + 3 + \left(N - \left\lfloor \frac{N}{u} \right\rfloor \times u \right) \times S_1 \quad (3)$$

A constant of 1 is added to II_u because at the end of each iteration, it is necessary to perform a shift of the content of the rotating registers so as to prepare for the next iteration. These shifts can be done in parallel in hardware and thus cost one cycle. The constant of 3 is needed because in our scheme, we needed one clock cycle at the beginning of the loop to set up the rotating registers, another clock cycle to initiate the loop and epilog counters, and one more at the end of the loop to copy out the content of the rotating registers.

S_u is obtained from Eq. 2. As is the case for S_u , we do not redo modulo scheduling over all possible u 's for II_u and e_u . Given a machine configuration, M , and a loop, L , the following holds:

$$II_u = II_{u-1} + c_{II} \quad (4)$$

$$e_u = \left\lceil \frac{S_u}{II_u} \right\rceil - 1 \quad (5)$$

where II_u is dependent on M and L . However, we also found that the simple recurrent relation for II_u do not necessarily end with the unroll size of 1. In particular, for software pipelining, if there is sufficient resources, then $II_i = II_{i-1}$ and the recurrent relations are not established until resource over-subscription comes into play. In our experiments, we used a machine that has only four memory port but otherwise has unlimited resources. The former condition is to reflect the limitation of the FPGA board that we are using. We used the following strategy: we perform software pipelining with II_1, II_2, \dots until $II_i \neq II_{i-1}$.

e_u can be derived from S_u and II_u through Eq. 5. This relationship is apparent once we see the idealized diagram for software pipelining shown in Fig. 1. In this example, $S_u = 4$, and $II_u = 1$, giving $e_u = 3$. Since $S_u > II_u$, $e_u \geq 1$.

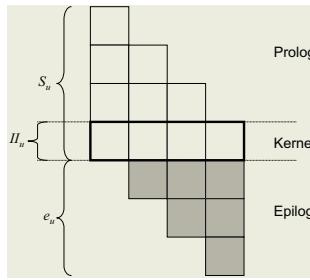


Fig. 1. Relationship between S_u , II_u and e_u .

Estimating FPGA Frequencies. The total running time of an implementation of a loop in a FPGA is given by the product of the number of cycles it takes to execute the code and the frequency of the FPGA which permits the safe operation of the realized design. It turns out that it is difficult to use static compiler information to obtain an accurate model of the final realizable frequency. In order to overcome this problem, we use the following strategy. We run place and route for three instances of the loop, namely the loop unrolled two, three, and four times. These three runs are also used in our resource estimation process described in the next section. Let the actual frequencies obtained from the three runs be $f_l(2)$, $f_l(3)$ and $f_l(4)$, respectively where l is either ‘unrolled’ or ‘swp’. We set the predicted frequency as follows:

$$F_l(u) = \begin{cases} \max(f_l(2), f_l(3), f_l(4)) & \text{if } u = 1 \\ f_l(u) & \text{if } u = 2, 3, \text{ or } 4 \\ \min(f_l(2), f_l(3), f_l(4)) & \text{if } u > 4 \end{cases} \quad (6)$$

Using these equations, we can finally approximate the time taken to execute the realized design to be

$$T_{\text{unrolled}}(u_1) = C_{\text{unrolled}}(u_1) \times F_{\text{unrolled}}(u_1) \quad \text{and} \quad T_{\text{swp}}(u_2) = C_{\text{swp}}(u_2) \times F_{\text{swp}}(u_2)$$

3.2 Model for Resource Usage

While we can easily count the various operators emitted by the compiler, optimizations further down the synthesis chain, in particular, the place and route pass, introduce non-trivial relationships between the high level hardware description our compiler output and the final resource usage. From experimental results, we found this to be especially true for the case of software pipelined loops. From the same three place and route runs used to obtain the frequencies, we also obtained the resource consumption information by means of linear regression. In particular, for a machine M and loop L , we model resource usage as:

$$R_{\text{unrolled}}(u) = m_{\text{unrolled}} \times u + c_{\text{unrolled}} \quad (7)$$

$$R_{\text{swp}}(u) = m_{\text{swp}} \times u + c_{\text{swp}} \quad (8)$$

where m_{unrolled} , c_{unrolled} , m_{swp} , and c_{swp} are constants obtained from the linear regression.

3.3 Putting It Together

The model is used as follows. The user will decide on a certain amount of resource, R_{user} , that he would like to use for realizing the loop in hardware. Using Equations 7 and 8, we obtained two maximal unroll factors u_1 and u_2 such that

$$R_{\text{unrolled}}(u_1) \leq R_{\text{user}} \quad \text{and} \quad R_{\text{swp}}(u_2) \leq R_{\text{user}}$$

Next we examine all unroll factors less than u_1 and u_2 to look for a $u'_1 \leq u_1$, and a $u'_2 \leq u_2$ such that $T_{\text{unrolled}}(u'_1)$ and $T_{\text{swp}}(u'_2)$ are the respective minimum. If $T_{\text{unrolled}}(u'_1) > T_{\text{swp}}(u'_2)$ then we will get better performance by using software pipelining with the loop unrolled u'_2 times and vice versa.

4 Compilation Framework

We used the Trimaran [17] compiler infrastructure to experiment with the model. The compiler targets for a parameterized Explicitly Parallel Instruction Computing (EPIC) architecture called HPL-PD [16]. We modified the compilation framework as follows:

- An EPIC machine with infinite resources except for four memory ports was defined. The four memory port was a constraint of the FPGA board which we used in our experiments. It has four banks of memory that can be simultaneously accessed with only one access to a bank at any time. Consequently, we also had to modify the instruction and modulo schedulers of Trimaran. We assumed that an entire array is stored in a single bank. Thus any two access to the same array has to be performed in different machine cycles.
- Trimaran uses some heuristics to guide unrolling. Furthermore, it does not always emit compensation loops during unrolling as these can be folded into the unrolled loop using predicated execution. For our purpose, we forced unrolling to be performed as per our requirements.

- Finally, we added a phase to generate Handel-C [18] code for Trimaran’s Elcor intermediate representation. Handel-C is a C-like behavioral hardware description language. The Handel-C compiler compiles our output into a EDIF [19] file for the FPGA vendor’s synthesis tools to process.

In the resultant design flow, we are able to utilize the advanced features used by Trimaran including predicated execution and rotating registers and translate them into Handel-C. From Handel-C’s EDIF output, we synthesis the bitmap for a Xilinx XCV1000 FPGA and execute it on a Celoxica RC1000 board.

5 Results

We used six kernel loops to verify our model:

- **Edge detection.** A 32×32 mask is computed over 128×128 image to detect edges.
- **Matrix multiplication.** Integer multiplication of 160×320 and 320×40 matrix.
- **Finite impulse response filter.** A 128-tap FIR filter on 256 integer data values.
- **Livermore Loop 1.** Hydro fragment loop of size 1001.
- **Jacobi.** 4-point stencil averaging computation over an array with loop size of 100.
- **Histogram.** Mapping from the old to the new grey levels with loop size of 1024.

The accuracy of our performance model is given in Table 1. The first set of columns present the result for loop unrolling and the second set of columns present the result for unrolling and software pipelining. “Est.” is the predicted execution time, i.e. $T_{\text{unrolled}}(u)$ and $T_{\text{swp}}(u)$. “Act.” is the actual execution time taken to execute the loop. This is obtained from multiplying the actual frequency obtained after place and route with the actual number of cycles executed. “Diff_T” represents the percentage difference between “Est.” and “Act.” while “Diff_C” represents the percentage difference in estimating $C_{\text{unrolled}}(u)$ and $C_{\text{swp}}(u)$. The average value for “Diff_C” for loop unrolling and loop unrolling with modulo scheduling are 2.84% and 2.19%, respectively. In addition, the values for S_u^p , II_u^p and e_u^p in Table 1 were computed using Equations 2, 4 and 5 while S_u^a , II_u^a and e_u^a were obtained from the actual compilation. The average relative error for “Diff_T” are 3.6% and 8.4% respectively for loop unrolling alone and software pipelining with unrolling. Given that the average relative difference between the actual execution time of the two strategies is 36%, we conclude that our performance estimation model is within the necessary margin and is accurate.

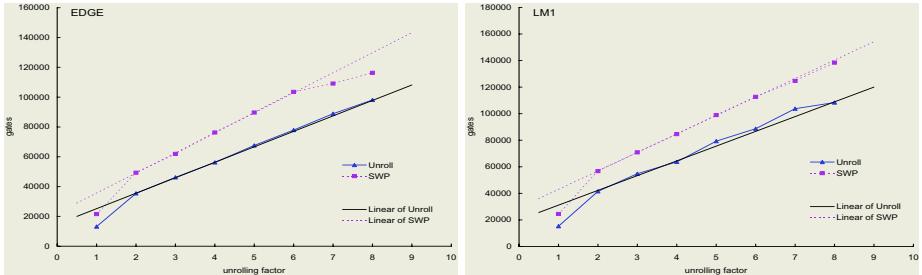
Fig. 2 shows the accuracy of our resource model. Due to space limitation, we will show the results for two benchmarks: Edge and LM1. The results for other benchmarks are similar. “Unroll” and “SWP” show the actual resource usage

Table 1. Accuracy of Performance Model.

Benchmark	(u)	Unrolling Only						Unrolling + SWP							
		S_u^p	S_u^a	Est. msec	Act. msec	Diff. T (%)	Diff. C (%)	II_u^p	II_u^a	e_u^p	e_u^a	Est. msec	Act. msec	Diff. T (%)	Diff. C (%)
Edge	1	4	4	0.421	0.391	7.46	-2.38	1	1	3	3	0.221	0.223	-1.09	-4.11
	2	5	4	0.282	0.296	-4.90	-4.90	2	2	2	2	0.177	0.189	-6.75	-6.75
	3	6	6	0.227	0.244	-7.01	-7.01	3	3	1	1	0.172	0.188	-8.52	-8.52
	4	7	7	0.184	0.201	-8.38	-8.38	4	4	1	1	0.145	0.161	-9.65	-9.65
	5	8	8	0.197	0.207	-5.00	-9.86	5	5	1	1	0.166	0.178	-7.05	-8.82
	6	9	9	0.187	0.201	-7.25	-10.36	6	6	1	1	0.166	0.198	-16.46	-7.22
	7	10	10	0.197	0.199	-0.77	-9.86	7	7	1	1	0.184	0.218	-15.37	-5.03
	8	11	11	0.155	0.158	-1.68	-12.22	8	8	1	1	0.15	0.226	-33.49	-4.24
MM	1	4	4	731.5	743.4	-1.60	-0.16	1	1	3	3	411.0	382.2	7.53	-0.47
	2	5	5	483.4	485.2	-0.38	-0.34	2	2	2	2	318.8	322.1	-1.03	-1.03
	3	6	6	368.0	370.3	-0.62	-0.63	3	3	1	1	291.2	295.2	-1.36	-1.36
	4	7	7	361.2	363.8	-0.72	-0.72	4	4	1	1	258.4	262.2	-1.47	-1.47
	5	8	8	330.3	308.6	7.00	-0.78	5	5	1	1	260.7	281.8	-7.50	-1.52
	6	9	9	312.8	289.8	7.97	-1.23	6	6	1	1	258.0	298.5	-13.54	-1.54
	7	10	10	303.2	275.0	10.24	-1.27	7	7	1	1	253.4	290.3	-12.70	-1.57
	8	11	11	283.8	283.5	0.12	-1.36	8	8	1	1	243.4	310.9	-21.70	-1.63
FIR	1	3	3	4.431	4.499	-1.51	-1.29	1	1	2	2	3.236	3.115	3.89	-1.89
	2	4	4	3.039	3.098	-1.93	-1.94	2	2	1	1	2.496	2.567	-2.75	-2.74
	3	5	5	2.492	2.559	-2.59	-2.58	3	3	1	1	2.391	2.476	-3.45	-3.45
	4	6	6	2.420	2.503	-3.30	-3.30	4	4	1	1	2.067	2.133	-3.09	-3.09
	5	7	7	2.319	2.390	-2.96	-3.41	5	5	1	1	2.219	2.330	-4.77	-3.09
	6	8	8	2.193	2.250	-2.51	-4.09	6	6	1	1	2.153	2.340	-7.99	-2.58
	7	9	9	2.118	2.283	-7.23	-4.21	7	7	1	1	2.127	2.312	-8.01	-2.00
	8	10	10	2.017	2.044	-1.36	-4.37	8	8	1	1	2.061	2.108	-2.26	-1.42
Lm1	1	8	8	0.754	0.779	-3.22	-0.06	2	2	3	3	0.303	0.286	6.05	-0.13
	2	9	9	0.441	0.441	-0.29	-0.29	3	3	2	2	0.209	0.209	-0.1	-0.70
	3	10	10	0.33	0.332	-0.6	-0.60	4	4	2	2	0.178	0.177	0.36	-1.07
	4	11	11	0.26	0.26	-0.22	-0.22	5	5	2	2	0.153	0.153	-0.2	-0.60
	5	12	12	0.239	0.229	4.17	-0.25	6	6	1	1	0.149	0.163	-8.33	-0.42
	6	13	13	0.218	0.208	4.52	-1.86	7	7	1	1	0.145	0.152	-4.74	-1.04
	7	14	14	0.198	0.19	4.52	-0.30	8	8	1	1	0.137	0.163	-15.97	-0.38
	8	15	15	0.187	0.19	-1.75	-0.32	9	9	1	1	0.134	0.164	-18.6	-0.32
Jacobi	1	10	10	5.712	5.367	6.42	-0.30	8	8	1	1	6.371	5.411	17.75	0.44
	2	13	13	3.713	3.736	-0.61	-3.21	8	8	1	2	3.29	3.249	1.29	1.29
	3	17	17	3.359	3.389	-0.87	-0.87	12	12	1	1	3.85	3.884	-0.89	-0.89
	4	20.5	21	3.095	3.125	-0.95	-3.31	16	16	1	1	4.695	4.642	1.13	1.13
	5	24	25	2.948	3.186	-7.47	-4.95	20	20	1	1	4.684	4.286	9.28	2.06
	6	27.5	29	2.972	3.139	-5.34	-5.70	24	24	1	1	4.821	4.931	-2.23	0.43
	7	31	33	2.842	3.137	-9.42	-6.78	28	28	1	1	4.758	5.405	-11.97	2.26
	8	34.5	37	2.854	3.177	-10.17	-7.16	32	32	1	1	4.863	5.912	-17.74	2.21
Histogram	1	5	5	0.313	0.312	0.36	-0.04	1	1	4	4	0.122	0.126	-3.29	0.44
	2	6	6	0.188	0.188	-0.1	-0.10	2	2	2	2	0.092	0.092	-0.19	1.29
	3	7	7	0.151	0.151	-0.17	-0.17	3	3	2	2	0.091	0.092	-0.36	-0.89
	4	8	8	0.126	0.126	-0.15	-0.15	4	4	1	1	0.102	0.102	-0.23	1.13
	5	9	9	0.116	0.121	-4.43	-1.23	5	5	1	1	0.1	0.111	-10.34	-1.24
	6	10	10	0.107	0.113	-5.09	-1.56	6	6	1	1	0.097	0.109	-11.18	-1.56
	7	11	11	0.102	0.105	-2.86	-0.31	7	7	1	1	0.095	0.135	-29.84	-0.31
	8	12	12	0.097	0.105	-7.57	-0.20	8	8	1	1	0.092	0.12	-23.07	2.21

due to unroll and unroll with software pipeline respectively. These points are obtained from the reports of the FPGA synthesis tool. The “Linear of Unroll” and “Linear of SWP” show the estimated resource usage using linear regression of $u = 2, 3$ and 4 . As can be seen from the figures, the estimated resource usage closely follows the actual resource usage.

It seem that in most cases, unrolling alone yields better performance under the same resource constraints. However, if we set $R_{\text{user}} = 100,000$, then for the

**Fig. 2.** Resource requirement for the benchmarks.

Lm1 benchmark, the unroll factor to be used for unrolling and software pipelining are 7 and 5, respectively. Using these unroll factors, our model predicts that we should use software pipelining instead of unrolling. The actual execution time given in Table 1 confirms that our prediction is correct.

Table 2 shows the various constants of Equations 7 and 8 obtained in our model. The results show that our model is fairly accurate and can significantly cut down the design space exploration time.

Table 2. Accuracy of Linear Regression.

Benchmark	Unrolling				Unrolling + SWP			
	m_{unrolled}	c_{unrolled}	Max Err.	Min Err.	m_{swp}	c_{swp}	Max Err.	Min Err.
Edge	10,371	14,826	1.67%	0.53%	13,471	22,104	11.67%	0.92%
MM (large)	10,468	15,333	4.55%	1.52%	13,365	24,686	20.86%	0.71%
FIR	9,496	13,115	4.18%	3.91%	11,701	20,457	9.87%	0.32%
Lm1	11,112	19,995	5.78%	0.44%	13,926	28,955	1.42%	0.08%
Jacobi	6,604	12,157	2.02%	0.63%	9,039	25,908	4.08%	0.37%
Histogram	3,973	5,676	13.50%	2.25%	3,714	1,6505	2.51%	0.01%

6 Conclusion

In this paper, we proposed a model that projects the data obtained from a small number of compilation and synthesis runs to obtain a global picture of the tradeoffs the designer faces in selecting between two loop optimizations, namely loop unrolling and software pipelining. The novelty of our approach is in the use of key parameters reported by the compiler to capture information about the machine configuration, data dependencies, and resource requirement patterns. This allowed us to obtain a very accurate model of the the cycle counts of the loops' execution. In the worst case, we are less than 5% off the actual cycle counts for larger loops.

The big challenge has been in modeling the two key parameters obtainable only after place and route, namely the circuit's realizable frequency and the resource consumption. For resource usage, we found good linear relations in the growth of resource consumption as unrolling increases especially within the realistic unroll factors that we studied.

Our approach is not very satisfying in modeling the frequency of software pipelined loops. In the worse case for software pipelined loop with high unroll numbers, we are can be off by 30%. Nonetheless, taken together as a whole, the average relative error in estimating $T_{\text{swp}}(u)$ is 8.4%. We would certainly like to improve this in future works.

Combining the resource model and the performance model, we have a methodology for deciding the optimal unroll factor as well as predict whether software pipelining will be beneficial given a certain resource constraint given by the user. We believe our model will reduce the time for design space exploration.

References

1. Rau, B.R.: Iterative Modulo Scheduling. *The International Journal of Parallel Processing* **24** (1996)
2. Page, I., Luk, W.: Compiling OCCAM into FPGAs. In: Proceedings of the International Symposium on Field Programmable Logic (FPL). (1991)
3. Rinker, R., et al.: An Automated Process for Compiling Dataflow Graphs into Reconfigurable Hardware. *IEEE Transactions on VLSI Systems* **9** (2001)
4. Goldstein, S.C., et al.: Piperench: A Reconfigurable Architecture and Compiler. *IEEE Computer* (2000)
5. Callahan, T., Hauser, J.R., Wawrzynek, J.: The Garp Architecture and C Compiler. *IEEE Computer* (2000)
6. Weinhardt, M.: Compilation and Pipeline Synthesis for Reconfigurable Architectures. In: Proceedings of the Reconfigurable Architecture Workshop (RAW). (1997)
7. Weinhardt, M., Luk, W.: Pipeline vectorization for reconfigurable systems. In: Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM). (1999)
8. Babb, J., et al.: Parallelizing Applications into Silicon. In: Proceedings of the IEEE Symposium on Field Programmable Custom Computing Machines (FCCM). (1999)
9. Snider, G., Shackleford, B., Carter, R.J.: Attacking the Semantic Gap between Application Programming Languages and Configurable Hardware. In: Proceedings of ACM FPGA. (2001)
10. Jones, A., et al.: PACT HDL: A C Compiler Targeting ASICs and FPGAs with Power and Performance Optimizations. In: Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). (2002)
11. Schreiber, R.: High-Level Synthesis of Nonprogrammable Hardware Accelerators. In: Proceedings of the IEEE International Conference on Application Specific Systems, Architectures, and Processors (ASAP). (2000)
12. Sivaraman, M., Aditya, S.: Cycle-time Aware Architecture Synthesis of Custom Hardware Accelerator. In: Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES). (2002)

13. Mentor Graphics Inc.: Mentor Graphics Monet User's Manual (release r42). (1999)
14. Derrien, S., Rajopadhye, S.: Loop Tiling for Reconfigurable Accelerators. In: Proceedings of the International Symposium on Field Programmable Logic (FPL). (2001)
15. So, B., Hall, M.W., Diniz, P.C.: A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI). (2002)
16. Kathail, V., Schlansker, M., Rau, B.: Hpl-pd architectural specifications: Version 1.1. Technical Report Technical Report HPL-93-80(R.1), Hewlett-Packard Laboratories (Revised 2000)
17. Trimaran Consortium: TRIMARAN: An Infrastructure for Research in Instruction Level Parallelism. (<http://www.trimaran.org>)
18. Celoxica Inc.: Handel-C. (<http://www.celoxica.com/tech/handel-c/>)
19. Electronic Industries Alliance:
Electronic Design Interface Format. (<http://www.edif.org>)

Programmable Asynchronous Pipeline Arrays

John Teifel and Rajit Manohar

Computer Systems Laboratory, Electrical and Computer Engineering,
Cornell University, Ithaca, NY 14853, U.S.A.,
 {teifel,rajit}@csl.cornell.edu, <http://vlsi.cornell.edu>

Abstract. We discuss high-performance programmable asynchronous pipeline arrays (PAPAs). These pipeline arrays are coarse-grain field programmable gate arrays (FPGAs) that realize high data throughput with fine-grain pipelined asynchronous circuits. We show how the PAPA architecture maintains most of the speed and energy benefits of a custom asynchronous design, while also providing post-fabrication logic reconfigurability. We report results for a prototype PAPA design in a $0.25\mu\text{m}$ CMOS process that has a peak pipeline throughput of 395MHz for asynchronous logic.

1 Introduction

We present programmable asynchronous pipeline arrays (PAPAs) as a high-performance FPGA architecture for implementing asynchronous circuits. Asynchronous design methodologies seek to address the design complexity, energy consumption, and timing issues affecting modern VLSI design [10]. Since most experimental high-performance asynchronous designs (e.g., [1, 13]) have been designed with labor-intensive custom layout, we propose the PAPA architecture as an alternative method for prototyping these asynchronous systems.

Previously proposed asynchronous FPGAs have shown that it is possible to port a clocked FGPA architecture to an asynchronous circuit implementation (e.g., [2, 14]). However, in an asynchronous system, logic computations are not artificially synchronized to a global clock signal and hence we can explore a larger programmable design space. In this paper we present one such exploration into the design of high-performance pipelines suitable for programmable asynchronous systems.

The PAPA architecture is inspired by high-performance, full-custom asynchronous designs [1, 13] that use very fine-grain pipelines. Each pipeline stage contains only a small amount of logic (e.g., a 1-bit full-adder) and combines computation with data latching, such that explicit output latches are absent from the pipeline. This pipeline style achieves high data throughput and can also be used to design energy-efficient systems [15]. As a result, we use fine-grain asynchronous pipelines as the basis for our high-performance FPGA architecture.

Existing work in programmable asynchronous circuits has concentrated on three design approaches: (1) mapping asynchronous logic to clocked FPGAs (e.g., [3, 5]), (2) asynchronous FPGA architectures for clocked logic [4, 16],

and (3) asynchronous FPGA architectures for asynchronous logic [2, 6, 8, 14]. The first approach suffers from an inherent performance penalty because of the circuit overhead in making a hazard-prone clocked FPGA operate in a hazard-free (the absence of glitches on wires) manner, which is necessary for correct asynchronous logic operation. Likewise, the second approach is not ideal because clocked logic does not behave like asynchronous logic and need not efficiently map to asynchronous circuits. The third approach runs asynchronous logic natively on asynchronous FPGA architectures. The work in this area has largely been modeled from existing clocked FPGA architectures, with the most recent running at an unencouraging 20MHz in $0.35\mu m$ CMOS [6].

In this paper we introduce the PAPA architecture as a new asynchronous FPGA that is designed to run asynchronous logic, yet differs from existing work because it is based on high-performance custom asynchronous circuits and is not a port of an existing clocked FPGA. The result is a programmable asynchronous architecture that is an order-of-magnitude improvement over [6]. Section 2 describes the asynchronous pipelines that our FPGA targets. In Section 3 we present the programmable asynchronous pipeline array architecture and in Section 4 describe its circuit implementation. Section 5 analyzes the performance of the PAPA architecture and Section 6 discusses logic synthesis results.

2 Asynchronous Pipelines

We design the logic that runs on PAPAs and other asynchronous systems as a collection of concurrent hardware processes that communicate with each other through message-passing channels [11]. Asynchronous pipelines can be constructed using such processes by connecting their channels in a FIFO configuration, where each pipeline stage consists of a single process. We refer to data items in a pipeline as *tokens* (i.e., messages passed on channels).

Since there is no clock in an asynchronous design, processes use handshake protocols to send and receive tokens on channels. All PAPA channels use three wires, two data wires and one acknowledge wire, to implement a four-phase handshake protocol. The data wires encode bits using a dual-rail code, such that setting “wire-0” transmits a “logic-0” and setting “wire-1” transmits a “logic-1”. The four-phase protocol operates as follows: the sender sets one of the data wires, the receiver latches the data and raises the acknowledge wire, the sender lowers both data wires, and finally the receiver lowers the acknowledge wire. The *cycle time* of a pipeline stage is the time required to complete one four-phase handshake.

In PAPA logic designs we enforce the following constraints on channels and processes: (1) no shared variables, (2) no shared channels, (3) no arbiters, and (4) the ability to add an arbitrary number of pipeline stages on a channel without changing the logical correctness of the original system. These system restrictions are reasonable for many high-performance asynchronous systems, including entire microprocessors [13], and in the rest of this paper we restrict our attention to asynchronous pipelines and circuits satisfying them.

A system that satisfies the aforementioned constraints is an example of a *slack-elastic* system [9] and has the nice property that a designer can locally add pipelining anywhere in the system without having to adjust the global pipeline structure. This property allows PAPA logic cells to be implemented with a variable number of pipeline stages and enables channels with long routes to be pipelined to improve performance. Any non-trivial clocked design will *not* be slack elastic, since changing local pipeline depths in a clocked system may require global retiming of the entire system. Adding high-speed retiming hardware support to a clocked FPGA incurs a significant register overhead [17], which the PAPA architecture can avoid because its logic cells are inherently pipelined and its channels are slack elastic.

Asynchronous (fine-grain) pipeline stages perform one or more of the following dataflow operations: (1) compute arbitrary logical functions, (2) store state, (3) conditionally receive tokens on input channels, (4) conditionally send tokens on output channels, and (5) copy tokens to multiple output channels. While strategies for implementing these pipeline operations in custom circuitry have been described in [7], the goal of the PAPA architecture is to implement these operations in a programmable manner.

Techniques for implementing operations 1 and 2 are well-known in both the clocked and asynchronous FPGA circuit literature (e.g., [2, 14]). PAPAs have a *Function* unit to compute arbitrary functions and use feedback loops to store state. However, because operations 3, 4, and 5 involve tokens they are inherently asynchronous pipeline structures. The PAPA architecture provides a *Merge* unit to conditionally receive tokens, a *Split* unit to conditionally send tokens, and an *Output-Copy* unit to copy tokens. Since a clocked FPGA circuit has no concept of a token, it uses multiplexers, demultiplexers, and wire fanout to implement structures similar to operations 3, 4, and 5, respectively. The main difference is that these clocked circuits are destructive (i.e., wire values not used are ignored and overwritten on the next cycle), whereas an asynchronous circuit is non-destructive (i.e., tokens remain on channels until they are used).

3 The PAPA Architecture

The PAPA architecture is a RAM-based, coarse-grain FPGA design and consists of *Logic Cells* surrounded by *Channel Routers*. Figure 1a shows the basic PAPA logic cell and channel router configuration that is used in this paper. Logic cells communicate through 1-bit wide, dual-rail encoded channels that have programmable connections configured by the channel routers.

Logic Cell. The pipeline structure of a PAPA logic cell is shown in Figure 1b. The *Input-Router* routes channels from the physical input ports (Nin , Ein , Sin , Win) to the three internal logical input channels (A , B , C). This router is implemented as a switch matrix and is unpipelined. If an internal input channel is not driven from a physical input port, a token with a “logic-1” value is internally sourced on the channel (not shown in the figure). The internal input channels are shared between four logical units, of which only one unit can be enabled.

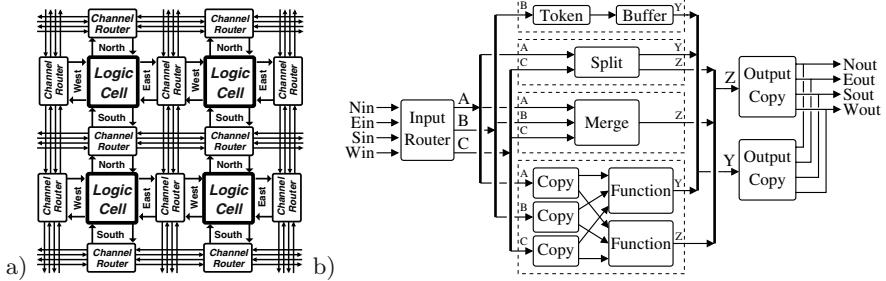


Fig. 1. PAPA architecture: (a) logic cell and channel router configuration, (b) pipeline structure of logic cell.

The logical units are as follows:

- *Function Unit* (2 pipeline stages): Two arbitrary functions of three variables. Receives tokens on channels (A, B, C) and sends function results on output channels (Y, Z) . (e.g., this unit efficiently implements a 1-bit full-adder).
- *Merge Unit* (1 pipeline stage): Two-way controlled merge. Receives a control token on channel C . If the control token equals “logic-0” it reads a data token from channel A , otherwise it reads a data token from channel B . Finally, the data token is sent on channel Z .
- *Split Unit* (1 pipeline stage): Two-way controlled split. Receives a control token on channel C and a data token on channel A . If the control token equals “logic-0” it sends the data token on channel Y , otherwise it sends the data token on channel Z .
- *Token Unit* (2 pipeline stages): Initializes with a token on its output. Upon system reset a token (with a programmable value) is sent on channel Y . Afterwards the unit acts as a normal pipeline (i.e., it receives a token on channel B and sends it on channel Y). Unit is used for state initialization.

The *Output-Copy* pipeline stage copies result tokens from channels Y and Z to one or more of the physical output ports ($Nout, Eout, Sout, Wout$) or sinks the result tokens before they reach any output port.

A PAPA logic cell uses 44 configuration bits to program its logic. The configuration bits are distributed as follows: 15 bits for the *Input-Router*, 4 bits for the logical unit enables, 16 bits for the *Function* unit, 1 bit for the *Token* unit, and 8 bits for the *Output-Copy* stages.

Unlike most existing FPGA architectures, PAPA logic cells do not have internal state feedback. Instead, state feedback logic is synthesized with an external feedback loop through an additional logic cell that is configured as a *Token* unit. This ensures that the state feedback loop is pipelined and operates at close to full throughput without adding additional area overhead to the logic cell to support an internal feedback path [7].

Channel Router. A PAPA channel router is an unpipelined switch matrix that *statically* routes channels between logic cells. PAPA channel routers route

all channels on point-to-point pathways and all routes are three wires wide (necessary to support the dual-rail channel protocol). Each channel router has 12 channel ports (6 input and 6 output) that can route up to six channels. Four of the ports are reserved for connecting channels to adjacent logic cells and the remaining ports are used to route channels to other channel routers. To keep the configuration overhead manageable, a PAPA channel router does not allow “backward” routes (i.e., changing a channel’s route direction by 180 degrees) and requires 26 configuration bits.

By examining numerous pipelined asynchronous logic examples, we empirically determined the PAPA logic cell and channel router interconnect topology (Fig.1a) as a good tradeoff between performance, routing capability, and cell area. We make no claims that it is the most optimal for this style of programmable asynchronous circuits and in fact it has several limitations. For example, it is not possible to directly route a channel diagonally on a 3x3 or larger PAPA grid using *only* channel routers (routing through one logic cell is required, which will improve performance for long routes). However, since most asynchronous logic processes communicate across short local channels we have not found this long-diagonal route limitation to be overly restrictive. More complicated channel routing configurations (such as those used in clocked FPGAs) could be adapted for the PAPA architecture, with the added cost of more configuration bits and cell area.

4 Pipelined Asynchronous Circuits

The asynchronous circuits we use are quasi-delay-insensitive (QDI). While they operate under the most conservative delay model that assumes gates and most wires have arbitrary delays [12], we believe QDI circuits to be the best asynchronous circuit style in terms of performance, energy, robustness, and area.

Although high-throughput, fine-grain QDI pipelined circuits have been used previously in several full-custom asynchronous designs [1, 13], the PAPA architecture is the first to adapt these circuits for programmable asynchronous logic. A detailed description on the design and behavior of this style of pipelined asynchronous circuits is in [7]. What follows is a summary of their salient features.

- **High throughput** – Minimum pipeline cycle times of \sim 10-16 FO4 (fanout-of-4) delays (competitive with clocked domino logic).
- **Low forward latency** – Delay of a token through a pipeline stage is \sim 2 FO4 delays (superior to clocked domino logic).
- **Data-dependent pipeline throughput** – Operating frequency depends on arrival rate of input tokens (varies from idle to full throughput).
- **Energy efficient** – Power savings from no extra output latch, no clock tree, and no dynamic power dissipation when the pipeline stage is idle.

Figure 2 shows the two pipeline circuit templates used in the PAPA architecture. L_0 and L_1 are the dual-rail inputs to the pipeline stage and R_0 and R_1 are the dual-rail outputs. We use inverted-sense acknowledge signals ($L_{\overline{ACK}}$, $R_{\overline{ACK}}$)

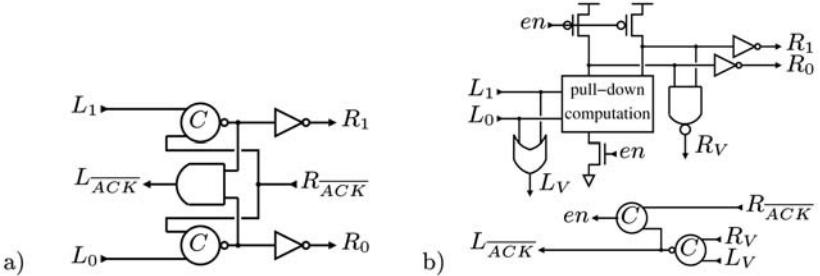


Fig. 2. Fine-grain pipelined asynchronous circuit templates: (a) weak-condition (dual-rail) pipeline stage, (b) precharge (dual-rail) pipeline stage.

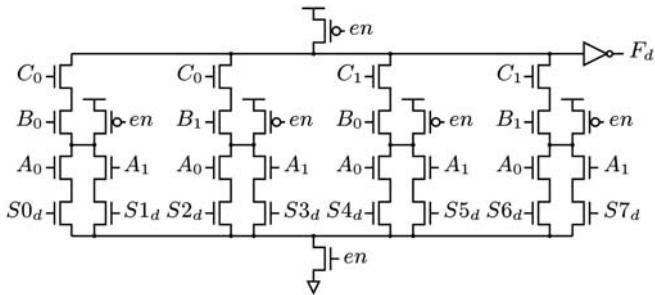


Fig. 3. One rail of a dual-rail precharge computation block for a 3-input function unit.

for circuit efficiency. The weak-condition pipeline stage (Fig.2a) is most useful for token buffering and token copying, while the precharge pipeline stage (Fig.2b) is optimized for performing logic computations (similar to dual-rail clocked domino circuits). Since the weak-condition and precharge pipeline stages both use the dual-rail handshake protocol, they can be freely mixed together in the same pipeline. Weak-condition pipeline stages are used in the *Token* unit, *Output-Copy*, and in the copy processes of the *Function* unit. The *Split* unit, *Merge* unit, and the evaluation part of the *Function* unit use precharge pipeline stages.

A partial circuit used in the evaluation part of the *Function* unit is shown in Figure 3. A , B , and C are the input channels and $S0_d \dots S7_d$ are the configuration bits that program the function result F_d , where d specifies the logic rail (e.g., $d=0$ computes F_0). As noted in [2], a function computation block of this style will suffer from charge sharing problems, which we solved using aggressive transistor folding and internal-node precharging techniques.

Physical Design. A prototype PAPA device has been designed and preliminarily layed out in TSMC's $0.25\mu\text{m}$ CMOS process (FO4 delay $\approx 120\text{ps}$) available via MOSIS. An arrayable PAPA cell that includes one logic cell and two channel routers is $144 \times 204 \mu\text{m}^2$ ($1200 \times 1700 \lambda^2$) in area, which is 50-100% larger than a conventional clocked FPGA cell but 50-33% the size of the pipelined clock FPGA in [17]. To minimize cell area and simplify programming, configuration bits are programmed using JTAG clocked circuitry. The area breakdown for the

architecture components is: function unit (14.4%), merge unit (2.5%), split unit (2.9%), token unit (2.6%), output copies (12.5%), configuration bits (37.7%), channel/input routers (18.2%), and miscellaneous (9.1%).

We have simulated our layout in SPICE (except for inter-cell wiring parasitics) and found the maximum inter-cell operating frequency for PAPA logic to be 395MHz. Internally the logical units can operate much faster, but are slowed by the channel routers. To observe this we configured the logical units to internally source “logic-1” tokens on their inputs and configured the *Output-Copy* stages to sink all result tokens (bypassing all routers). The results are: *Function* unit (498MHz, 26pJ/cycle), *Merge* unit (543MHz, 11pJ/cycle), *Split* unit (484MHz, 12pJ/cycle), and *Token* unit (887MHz, 7pJ/cycle). These measurements compare favorably to the pipelined clock FPGA in [17] that operates at 250MHz and consumes 15pJ/cycle of energy per logic cell. Our current work focuses on intelligently pipelining the channel routers to match the internal cycle times of the logical units and using improved circuit techniques to reduce the energy consumption of the PAPA logic cells.

5 Performance Analysis

The pipeline dynamics of asynchronous pipelines, due to their interdependent handshaking channels, are quite different from the dynamics of clocked pipelines. To operate at full throughput, a token in an asynchronous pipeline must be physically spaced across multiple pipeline stages, whereas in a clocked pipeline the optimum results when there is one token per stage [18]. The optimal number of pipeline stages, n_0 , per token in an asynchronous pipeline is attained when $n_0 = \tau_0/l_0$, where τ_0 is the cycle-time of a pipeline stage and l_0 is its forward latency. For circuits used in the PAPA design, n_0 ranges from 5 to 8 pipeline stages per token (for pipelines without switches). If a pipeline has fewer stages per token than n_0 , it will operate at a slower than maximal frequency but consume less energy [15]. On the other hand, if the pipeline has more stages per token than n_0 , it will both operate slower and consume more energy than the optimal case.

To observe the pipeline dynamics when there are programmable switches between pipeline stages, we modeled a PAPA pipeline with a linear pipeline of n weak-condition pipeline stages that contain a variable number of routing switches between each pipeline stage. This model uses layout from the *Token* unit, has $n_0=5$, and measures all results from full SPICE simulations (including inter-cell wiring parasitics). This model gives an upper bound on the performance of PAPA pipelines and shows the behavioral trends of inserting switches between fine-grain asynchronous pipeline stages.

Figure 4a shows the maximum operating frequency curves for our model pipeline when there are K routing switches between every pipeline stage ($K=0$ is the “custom” case when there are no switches between stages). We observe that as K increases, n_0 decreases from 5 stages to 4 stages and the frequency curves shift downward because the switches uniformly increase the cycle time of

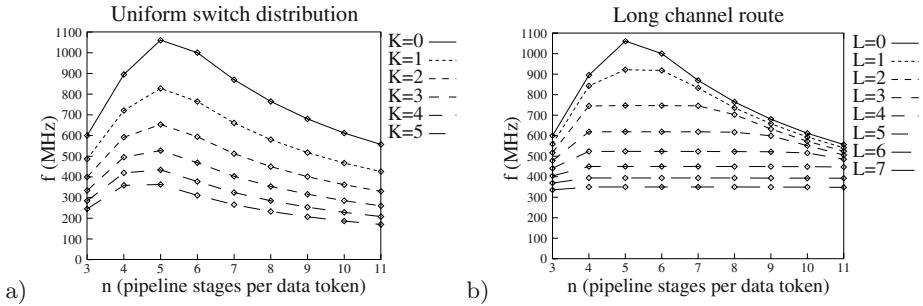


Fig. 4. Maximum operating frequency curves for one token in a linear pipeline of n weak-condition pipeline stages, when (a) there are K routing switches between every pipeline stage and (b) one pipeline stage has a long route through L switches.

every pipeline stage. Figure 4b shows the effect of one pipeline stage having a long route through L switches (when the other pipeline stages have no switches). In this case, the frequency curves flatten as L increases because the cycle time of the pipeline is mainly determined by the cycle time of the stage containing the long route (i.e., the long route behaves as a pipeline bottleneck).

In addition to decreasing their operating frequency, the energy consumption of asynchronous pipelined circuits also increases when routing switches are added between pipeline stages. To observe the energy effect of adding switches to asynchronous pipelines we use the $E\tau^2$ energy-time metric [13, 15]. E is the energy consumed in the pipeline per cycle and τ is the cycle time ($1/f$). Since E is proportional to V^2 and τ is proportional to $1/V$, to first order this metric is independent of voltage and provides an energy-efficiency measure to compare both low-power designs (low voltage) and high-performance designs (normal voltage). Figure 5 shows energy-efficiency curves for our model pipeline under the two switch scenarios examined earlier (lower values imply more energy efficiency).

The maximum operating frequency and energy-efficiency curves for a PAPA pipeline will look like a mixture of the two switch scenarios we investigated, since

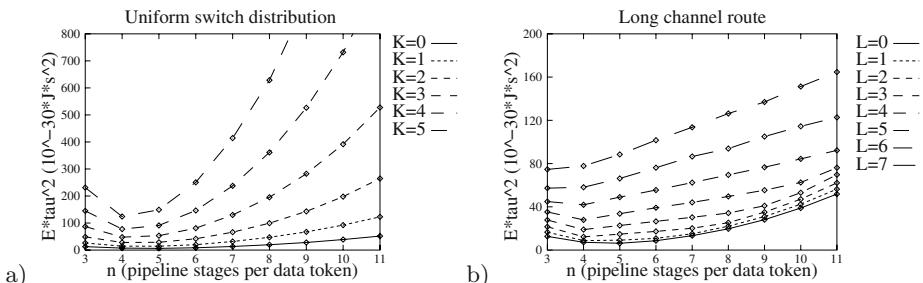


Fig. 5. Energy-efficiency curves for one token in a linear pipeline of n weak-condition pipeline stages, when (a) there are K routing switches between every pipeline stage and (b) one pipeline stage has a long route through L switches.

some pipeline stages will have no switches between them (channels inside of the logic cell) and some will have two or more (channels going through the input and channel routers). We have found that in synthesized PAPA logic there is at most six switches between logic cells, and on average two to four (including input routers). While the plots in this section show that (as expected) adding routing switches to full-custom, high-throughput pipelined circuits decreases both their speed and energy efficiency, they also show that there is still much performance remaining ($\approx 50\%$) to make them attractive for high-speed programmable asynchronous logic.

6 Logic Synthesis Results

High-level logic synthesis for PAPA designs borrows heavily from the formal synthesis methods we use to design full-custom asynchronous circuits [10]. We begin with a sequential description of the logic that is written in the CHP (Communicating Hardware Processes) hardware description language and apply (already existing) semantics-preserving program transformations to get a set of fine-grain concurrent CHP processes. Each of the resulting processes can be implemented in a single PAPA logic cell. The processes are then physically mapped onto PAPA logic cells. While this procedure is currently only semi-automated, it is not as tedious a task as for gate-level FPGAs. Finally, channels connecting logic cells are automatically routed and a configuration file generated.

We report SPICE simulations for several synthesized logic examples:

- N-bit ripple-carry adder (N logic cells) – Throughput of 292MHz, with a data input-to-output latency of 1.91ns, and a carry input-to-output propagation latency of 1.04ns per bit (the router was directed to minimize carry latency).
- Pipelined Booth encoded multiplier 1-bit cell (12 logic cells) – Throughput of 222MHz (original full-custom version ran at 190MHz in $0.8\mu m$ [1]).
- Register bit (5 logic cells) – Throughput of 272MHz, can read and/or write on same cycle.

7 Summary

We introduced a new high-performance asynchronous FPGA architecture. The architecture uses fine-grain asynchronous pipelines to implement a coarse-grain FPGA and is suitable for prototyping pipelined asynchronous logic. Our preliminary circuit simulations demonstrate that PAPA logic systems are a promising alternative to full-custom asynchronous designs.

Acknowledgments

This research was supported in part by the Multidisciplinary University Research Initiative (MURI) under the Office of Naval Research Contract N00014-00-1-0564, and in part by an NSF CAREER award under contract CCR 9984299. John Teifel was supported in part by an NSF Graduate Research Fellowship.

References

1. Cummings, U.V., Lines, A.M., Martin, A.J.: An Asynchronous Pipeline Lattice-structure Filter. *Proc. Int'l Symp. on Asynchronous Circuits and Systems* (1994)
2. Hauck, S., Burns, S., Borriello, G., Ebeling, C.: An FPGA for implementing asynchronous circuits. *IEEE Design & Test of Computers* **11**(3) (1994) 60-69
3. Ho, Q.T., et al.: Implementing asynchronous circuits on LUT based FPGAs. *Proc. 12th Int'l Conf. on Field Programmable Logic and Applications* (2002)
4. How, D.L.: A Self Clocked FPGA for General Purpose Logic Emulation. *Proc. of the IEEE 1996 Custom Integrated Circuits Conf.* (1996)
5. Keller, E.: Building Asynchronous Circuits with JBits. *Proc. 11th Int'l Conf. on Field Programmable Logic and Applications* (2001)
6. Konishi, R., et al.: PCA-1: A fully asynchronous self-reconfigurable LSI. *Proc. 7th Int'l Symp. on Asynchronous Circuits and Systems* (2001)
7. Lines, A.M.: *Pipelined Asynchronous Circuits*. M.S. Thesis, California Institute of Technology (1996)
8. Maheswaran, K.: *Implementing Self-Timed Circuits in Field Programmable Gate Arrays*. M.S. Thesis, U.C. Davis (1995)
9. Manohar, R., Martin, A.J.: Slack Elasticity in Concurrent Computing. *Proc. of the 4th Int'l Conf. on the Mathematics of Program Construction* (1998)
10. Manohar, R.: A Case for Asynchronous Computer Architecture. *Proc. of the ISCA Workshop on Complexity-Effective Design* (2000).
11. Martin, A.J.: Compiling Communicating Processes into Delay-insensitive VLSI circuits. *Distributed Computing*, **1**(4) (1986)
12. Martin, A.J.: The Limitations to Delay-Insensitivity in Asynchronous Circuits. *Sixth MIT Conf. on Advanced Research in VLSI* (1990)
13. Martin, A.J., Lines, A., Manohar, R., et al.: The Design of an Asynchronous MIPS R3000. *Proc. of the 17th Conf. on Advanced Research in VLSI* (1997)
14. Payne, R.: Asynchronous FPGA architectures. *IEE Computers and Digital Techniques* **143**(5) (1996) 282-286
15. Teifel, J., Fang, D., Biermann, D., Kelly, C., Manohar, R.: Energy-Efficient Pipelines. *Proc. 8th Int'l Symp. on Asynchronous Circuits and Systems* (2002)
16. Traver, C., Reese, R.B., Thornton, M.A.: Cell Designs for Self-timed FPGAs. *Proc. of the 2001 ASIC/SOC Conf.* (2001)
17. Tsu, W., et al.: HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array. *Proc. 7th Int'l Symp. on Field-Programmable Gate Arrays* (1999)
18. Williams, T.E.: *Self-Timed Rings and their Application to Division*. Ph.D. thesis, Stanford University (1991)

Globally Asynchronous Locally Synchronous FPGA Architectures

Andrew Royal and Peter Y.K. Cheung

Department of Electrical & Electronic Engineering, Imperial College, London, UK,
{a.royal,p.cheung}@imperial.ac.uk

Abstract. Globally Asynchronous Locally Synchronous (GALS) Systems have provoked renewed interest over recent years as they have the potential to combine the benefits of asynchronous and synchronous design paradigms. It has been applied to ASICs, but not yet applied to FPGAs. In this paper we propose applying GALS techniques to FPGAs in order to overcome the limitation on timing imposed by slow routing.

1 Introduction

Most Field Programmable Gate Arrays (FPGAs) are designed with one or more global clocks. FPGAs with multiple clock domains must provide some mechanism for synchronising data passing between them, which will increase latency and be prone to metastability.

In addition a signal routed over a great distance on an FPGA must pass through many long wires, transistor switches and buffers. Consequently these long connections usually prove to exhibit the longest delays in the whole device. If transmission is to occur over one clock cycle or even a fraction of a clock cycle, this routing delay will limit the clock frequency.

One solution is to pipeline the routing, as used in [1]. A potential problem of this is that the number of clock cycles allocated to the routing must be determined at the routing stage and may impact upon any cycle allocation assumed at the circuit design stage. Also, concurrent data travelling along different routing paths need to contain exactly the same number of pipeline stages or data will arrive on different cycles.

We could also use asynchronous circuits. Rather than assume that data takes a clock period to pass through a pipeline stage, asynchronous circuits use hand-shake protocols to indicate when each stage has data to pass to the next stage. This allows each stage to be independently timed. Were we to use asynchronous routing for FPGAs, the speed of the rest of the circuit would no longer be limited by the speed of the routing. As they have no clocks, there is no need for multiple clock domains and no problem with synchronisation.

Asynchronous FPGAs have already been proposed for prototyping asynchronous circuits. However, the drawbacks of asynchronous circuits deter designers from using them in preference to synchronous arrays. A compromise is to use a Globally Asynchronous Locally Synchronous (GALS) system. In such a

system synchronous modules with locally generated clocks are used, with asynchronous connections between them. Hence we retain the advantages of synchronous circuits, but can also exploit the advantages of asynchronous routing. This technique has not previously been applied to FPGAs.

In this paper we propose adding asynchronous routing to a synchronous FPGA. In section 2 a brief overview of asynchronous communication, asynchronous FPGAs and Globally Asynchronous Locally Synchronous Systems is given. We go on to describe an architecture for applying this work to synchronous FPGAs in section 3 and assess its viability in section 4. Finally, in section 5 we describe our future work into improving this architecture.

2 Background and Related Work

2.1 Asynchronous Systems

In this paper, we use the term “Asynchronous” to refer to circuits designed without clocks, also known as Self-Timed circuits, where the clock is replaced by handshaking signals. In a synchronous system, all blocks are assumed to have finished computation when a clock edge arrives. The blocks in Self-timed systems independently indicate completion by sending out a request and only proceed when that request has been acknowledged. Blocks only operate as needed, there is little redundant processing. Also, a self-timed system is very composable as blocks can be individually optimised and timing of one block does not affect another. We wish to exploit this feature for our FPGA architecture.

Data is transmitted using a bundled data protocol [2]. This means that data signals are grouped into a bundle and the validity of the whole bundle is indicated by a single request/acknowledge handshake pair. The bundling constraint states that a request must arrive after the corresponding data, so data can be latched safely, so the delay of request lines often needs tuning.

Asynchronous designs require some circuit components which are rarely used in synchronous design. Ebergen [3] showed many of the circuit elements required to build delay insensitive circuits. We require some of these components for building our architecture. A C-element is a component which fires a transition on its output when each of its inputs have made transitions in the same direction, it is effectively an AND for events. An isochronic fork is simply a signal which branches out to two destinations, where the delay on the wires is negligible compared to the delays of the gates they are driving and so can be assumed to arrive at the same instant. A mutual exclusion element (often ME element or MUTEX) is used to arbitrate between requests. ME elements may go metastable if the requests arrive close to each other, but the Seitz arbiter [4] is a mutual exclusion element designed such that any metastability is internal and not propagated to its outputs. Finally, Micropipelines [2] are an asynchronous equivalent of synchronous pipelines, where the registers are controlled by request/acknowledge signals rather than a clock.

2.2 Asynchronous FPGAs

There have been several attempts to implement asynchronous circuits on FPGAs designed for synchronous circuits [5], [6]. The main problems with this are that synchronous FPGAs are not designed to be hazard free and do not provide many of the components commonly used in asynchronous design.

There have also been FPGAs designed specifically for implementing asynchronous circuits. MONTAGE [7] is an extension of the synchronous TRIPTYCH architecture [8]. Fast feedback in functional units allows asynchronous-specific components to be built and specific arbiter blocks are also provided. Routing is organised to allow isochronic forks. PGA-STC [6] is similar to MONTAGE, but also includes a reconfigurable delay line which can be used in the implementation of a bundled data protocol. The delay uses a ring coupled oscillator which is unfortunately very large and power consuming. STACC [9], [10] is loosely based on Sutherland's micropipeline design [2]. The data array can be like that of any synchronous FPGA, but the clock is replaced by control signals from a timing array which consists of a micropipeline-like structure.

Asynchronous FPGAs are not widely used. They are fraught with problems with hazards, critical races and metastability. Asynchronous circuits are hard to design and tools have only recently begun to reach maturity. Asynchronous buses are difficult to construct [11]. We also find that the additional completion detection circuitry required takes considerable area and power and slows the circuit down. Hence we propose a Globally Asynchronous Locally Synchronous FPGA as a compromise between synchronous and asynchronous styles.

2.3 Globally Asynchronous Locally Synchronous (GALS) Systems

Globally Asynchronous Locally Synchronous (GALS) Systems combine the benefits of synchronous and asynchronous systems. Modules can be designed like modules in a globally synchronous design, using the same tools and methodologies. Each block is independently clocked, which helps to alleviate clock skew. Connections between the synchronous blocks are asynchronous.

Early work on GALS systems ([12] and [13]) introduced clock stretching or pausing. When data enters a synchronous system from an asynchronous environment, registers at the input are prone to metastability. To avoid this, the arrival of data is indicated by an asynchronous handshaking protocol. When data arrives, the locally generated clock is paused: in practice the rising edge of the clock is delayed. Once data has safely arrived, the clock can be released so data is latched with zero probability of metastability on the datapath. [14] used ME elements to arbitrate between the clock and incoming requests, which helped to eliminate metastability. [15] introduced asynchronous wrappers, standard components which can be placed around synchronous modules to provide the handshake signals and make them GALS modules.

The local clock generator is constructed from an inverter and a delay line, similar to an inverter ring oscillator. The problem with using inverters alone as a delay line is that it is difficult to accurately tune the clock period as process

variations and temperature affect the delay. Hence accurate delay lines have been developed which are capable of maintaining a stable clock frequency [16], [17]. These use a global reference clock for calibration. The former can use either standard cells or full custom blocks for the tunable delay and was shown to exhibit less than 1% jitter around the chosen frequency.

To make the clock pausable, an ME element is added to the ring as shown in figure 1(a). This arbitrates between the rising edge of the clock and an incoming request. Hence the clock is prevented from rising as the input registers are being enabled by the request and metastability is prevented. For each bundle of data a port controller, request and ME element is required. Only when all of the ME elements have been locked out by the clock is the rising clock edge permitted to occur.

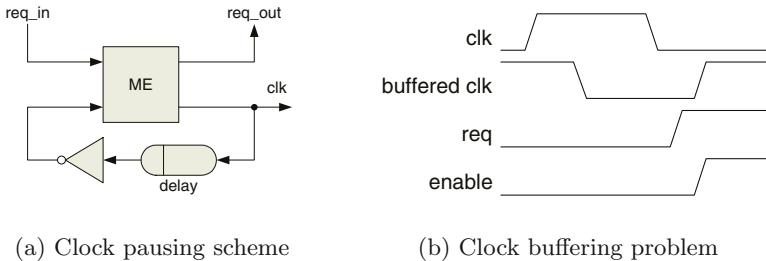


Fig. 1. Pausable clock

Port controllers are required to generate and accept handshaking signals at the inputs and outputs of modules. These port controllers are asynchronous state machines, which are similar to on inputs rather than a clock.

A problem with clock pausing is that the clock is delayed as it is distributed across the clock domain, but the clock must be paused at its source. When the clock releases the ME elements there may still be a clock edge in the buffer tree. Hence it is possible that registers will be enabled as the clock rises, as shown in figure 1(b). But while the source clock is high, ME elements will remain locked so for this phase of the cycle no requests are permitted. For this reason, we must ensure that the delay of the clock buffer is shorter than the duration of the high phase of the clock. Limiting this delay limits the size of the clock tree, hence defining the size of GALS blocks.

3 System Architecture

We propose converting a conventional, synchronous FPGA into a GALS system. To do this we partition the FPGA into smaller blocks of FPGA cells. Within one of these blocks, the local connections are synchronous to a local clock for

that block and hence the block resembles the original FPGA. However, longer communication channels between blocks become asynchronous.

Figure 2 shows the proposed architecture in place around a block of FPGA cells. Note in particular the dividing line between the synchronous and asynchronous domains. All of the FPGA cells are in an isolated block above the line in the synchronous domain. Internally, the FPGA block could resemble any synchronous FPGA as it is hidden from the rest of the system. Below the line there is an asynchronous wrapper: this interfaces between the synchronous and asynchronous domains. Outside the asynchronous wrapper blocks are connected together using asynchronous routing. All of these blocks are explained in detail in the following section.

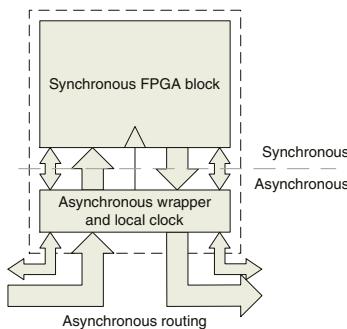


Fig. 2. FPGA block with asynchronous routing

3.1 Additional Synchronous FPGA Fabric

Figure 3 shows the boundary of a synchronous FPGA block. The FPGA block could contain any type of FPGA cell and its associated routing. There could be a number of different levels of routing within the block, for example nearest neighbour routing or fast interconnect spanning the block, as there are within a conventional FPGA. At the boundary, we can use the same routing schemes to connect to the asynchronous interface.

In this instance only one input port and one output port is shown for clarity, though it would be possible to design a system in which each FPGA block has several input and output ports to allow communication with a number of different FPGA blocks. As well as the data itself, the asynchronous ports needs to communicate with the synchronous to indicate when data is valid. To accomplish this, a wire for each port spans the routing leaving the block. For nearest neighbour or other unidirectional connections, one of the inputs to each block is allowed to connect to the data valid wire for each input port. Similarly, one of the output wires from each block may connect to the data valid wire of the output port. All other inputs and outputs from the FPGA cells can be used for data transfer.

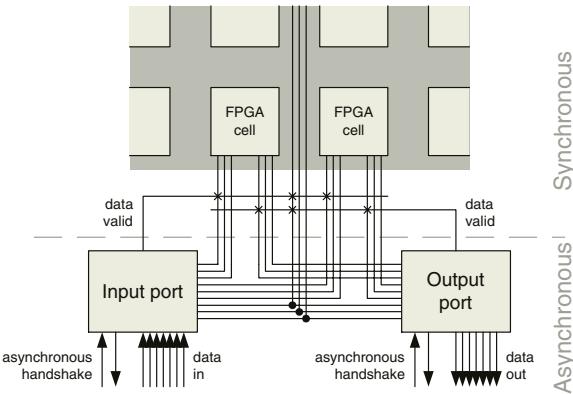


Fig. 3. Interconnect between FPGA cells and GALS ports

We work under the assumption that should we use part of an FPGA cell to create a data valid signal, that signal could be mapped to any of the inputs or any output as appropriate. In the case of a look-up table (LUT) based FPGA this can easily be done. Hence by only allowing one wire per cell to map to an input data valid signal and one for the corresponding output signal, we save on switching with little impact on flexibility.

Similarly, we also allow longer routing channels to be connected to the data valid signals. Again, in order to cut down on the number of switches used we only allow a fraction of the wires in the channel to be connected to the data valid signals. However, as long routing wires can usually be configured with data flow in either direction, we allow the chosen wires to be connected to both the input and output data valid signals. Here, we can expect some loss in flexibility as each routing wire could be connected to a completely different area of the FPGA block. But as each port can only have a single data valid signal, we should not need to allow many connections to the routing. In many cases it will be necessary to route to an FPGA cell and perform some logic function to merge several data valid signals into a single signal, which could be done in a boundary FPGA cell with a nearest-neighbour connection to the port's signals.

In figure 3, data is shown to enter the ports. This is merely a grouping of input wires and output wires, there is no need for any processing or even latching as that is handled in the synchronous part of the FPGA block. However, handshake signals accompany each “bundle” of data and so inside the synchronous FPGA block the data valid signal corresponding to the handshake must control the same data.

To complete this part of the system, we need to place a few requirements on any circuit mapped to the FPGA block. As discussed above, data valid signals need to be generated or processed. Data must leave the module with corresponding data valid signal and be latched when incoming data is valid. Any data signals also need to be routed to the boundary where they will leave the FPGA block accompanied by the handshake. For our implementation of the output port, we

require that the data valid signal be “differential”, i.e. it indicates valid data by changing its value and that no valid data is present by remaining at the same value.

3.2 Asynchronous Wrapper

The asynchronous wrapper shown in figure 2 is formed of 2 components: a local clock generator and port controllers. These components were described in more detail in section 2.3. For our implementation we use a four phase bundled data protocol. The interface between the synchronous and asynchronous domains is facilitated by making the synchronous signals differential, so an event is created whenever a signal changes. Our port controllers have been designed under the assumption that the synchronous block produces these differential signals and so they are a requirement of the circuit mapped to the FPGA block.

Note that we require a separate clock tree for each locally synchronous block. Clearly using the global trees featured in current FPGAs would be wasteful, hence it is preferable to use a dedicated local clock buffer. As mentioned in section 2.3, to prevent the size and delay of the clock buffers from becoming too large a limit of the size of the FPGA blocks within each wrapper is imposed.

3.3 Asynchronous Routing

The four phase bundled data transmission protocol continues into our routing. Not only must the data be routed, but also the corresponding handshakes. Furthermore, the bundling constraint must be maintained by delaying the request lines sufficiently that they always arrive after the corresponding data. When data arrives at a register, that register must be disabled so it will not go metastable if the rising edge of the clock arrives at the same instant. Once the register has won control of the ME element ahead of the clock, the register can be enabled and the ME element released.

Transfer of data is facilitated by inserting micropipelines into the routing. We exclusively use unidirectional wires to make point-to-point connections rather than using buses. The configuration in the routing is greatly simplified and in particular if micropipeline stages are used they need only operate in a single direction. The overall routing scheme is shown in figure 4. We have no long lines as long, slow lines are what we are trying to avoid. Instead, all long connections are made through a series of block-to-block connections. Each wire entering the FPGA block can either be routed to an input port or bypass the block completely. A connection between any two modules must pass through at least one micropipeline, which helps reduce the time each module remains paused and eliminate deadlock. Connections between rows must pass through at least two micropipeline stages, which adds a little latency.

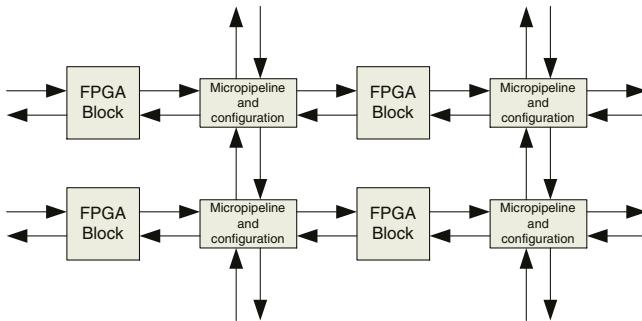


Fig. 4. Routing Scheme

4 Analysis of Architecture

This system confers several advantages. Our aims of metastability free independent clock domains and synchronous blocks whose timings are independent of the routing are met. The asynchrony of the routing allows data tokens to take an arbitrary number of clock cycles to reach their destinations. Indeed, as the source and destination are not in the same clock domain the number of clock cycles which pass in each domain as the data is transmitted can be expected to be different. Additionally, the FPGA system becomes more composable, i.e. the asynchrony of blocks makes it easy to design the component blocks independently without too much concern on how they will operate together as a system. It is no longer timing but the handshaking protocol which ensures data integrity. Independent design also allows the blocks to be independently optimised. As well as preventing metastability, clock pausing can be used to force a module to wait for data to arrive before proceeding, without wasting energy on redundant clocking.

However, the scheme does have several disadvantages. Firstly, we have now forced a separation of local routing signals, which are contained within the synchronous FPGA block, and global routing signals. This reduces the flexibility of the FPGA as a router is no longer able to use long routing tracks for local routing or join shorter routing tracks to make longer connections.

Secondly, we force additional constraints on placement. Any circuit which is too big to fit into a single FPGA block will necessarily need to be partitioned across several different blocks which will be in different clock domains. In some cases this may be inappropriate. For example feedback loops should ideally be contained on a single block due to the latency incurred. Due to the differing data rates, partitioning which results in data being transferred between blocks every cycle should be avoided.

To interface with the port controllers the synchronous block may need some additional circuitry. Some form of synchronous handshake is required to indicate to the output controller when data is valid and to accept data from the input

port when incoming valid data is present. In our implementation we require differential data valid signals. If the design is contained within a single block, the external ports may already include these signals. However, this need not necessarily be the case and if a design needs to be partitioned the required signals will almost certainly not be present at the block boundary and therefore need to be inserted. This may prove to be problematic as the timing of such signals requires not only the circuit information but also some knowledge of the pattern of the data.

Finally, as the size and complexity of the system implemented grows, it may become necessary to partition across many blocks. In this case, data being sent from a number of blocks to a single destination may be required to arrive at the same time. To make allowances for this, rendezvous elements are needed at each input port. These elements must be fully configurable to allow several possible combinations of data channels or to allow a bypass when rendezvous is not required. This will however move the system away from one in which the only configurable components are within the synchronous blocks.

5 Conclusion and Future Work

We have presented an extension to existing FPGA architecture with the potential to prevent long routing delays from dominating FPGA performance. However, the solution is not without its drawbacks.

To address some of these problems, some alternative architecture may be required. It has already been mentioned that configurable rendezvous elements are required at the input ports which adds configuration required in the routing. It may also be possible to join local clock trees to effectively combine two smaller blocks into a larger block under a single clock domain, though great care must be taken to maintain the balance of the tree. If this is possible, it may also be possible to join all clock trees and allow a globally synchronous mode to be retained alongside the GALS mode. Request lines need some tuning to force requests to arrive after the data and meet the bundling constraint, but alternatively we can use a dual rail protocol to provide delay insensitive routing. Though we currently use a four phase protocol, a two phase protocol may have some advantages.

We have yet to fully verify the scheme and prove its effectiveness. The difficulty lies in simulation as we need to concurrently simulate synchronous, asynchronous and FPGA components. Furthermore we have yet to extensively investigate how circuits may map to the system.

References

1. Mirsky, E., DeHon, A.: MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In: Proceedings of the IEEE Symposium in Field-Programmable Custom Computing Machines. (1996) 157–166

2. Sutherland, I.E.: Micropipelines. *Communications of the ACM* (1987) 720–738
3. Ebergen, C.: A Formal Approach to Designing Delay-Insensitive Circuits. *Distributed Computing* 5 (1988) 107–119
4. Seitz, C.L.: System timing. In Mead, C.A., Conway, L.A., eds.: *Introduction to VLSI Systems*. Addison-Wesley (1980)
5. Brunvand, E.: Using FPGAs to Implement Self-Timed Systems. *J. VLSI Signal Process.* 6 (1993) 173–190
6. Maheswaran, K.: Implementing Self-Timed Circuits in Field Programmable Gate Arrays. Master's thesis, University Of California Davis (1995)
7. Hauck, S., Burns, S., Borriello, G., Ebeling, C.: An (fpga) for Implementing Asynchronous Circuits. In: *IEEE Design & Test of Computers*. Volume 11. (1994) 60–69
8. Borriello, G., Ebeling, C., Hauck, S., Burns, S.: The Triptych FPGA Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 3 (1995) 491–501
9. Payne, R.E.: Self-Timed FPGA Systems. In: *Proceedings of the 5th International Workshop on Field Programmable Logic and Applications*. (1995)
10. Payne, R.: Self-Timed Field Programmable Gate Array Architectures. PhD thesis, University of Edinburgh (1997)
11. Molina, P.: The Design of a Delay-Insensitive Bus Architecture using Handshake Circuits. PhD thesis, Imperial College (1997)
12. Chapiro, D.M.: Globally Asynchronous Locally Synchronous Systems. PhD thesis, Stanford University (1984)
13. Pěchouček, M.: Anomalous response times of input synchronisers. *IEEE Transactions on Computers* C-25 (1976) 133–139
14. Yun, K.Y., Donohue, R.P.: Pausible clocking: A first step toward heterogeneous systems. In: *Proceedings of the International Conference on VLSI in Computers and Processors*. (1996) 118–123
15. Bormann, D.S., Cheung, P.Y.K.: Asynchronous wrapper for heterogeneous systems. In: *Proceedings of the International Conference on Computer Design (ICCD)*. (1997) 307–314
16. Moore, S.W., Taylor, G.S., Cunningham, P.A., Mullins, R.D., Robinson, P.: Self-calibrating clocks for globally asynchronous locally synchronous systems. In: *Proceedings of the International Conference on Computer Design (ICCD)*. (2000) 37–78
17. Olsson, T., Nilsson, P., Meincke, T., Hemam, A., Tokelson, M.: A digitally controlled low-power clock multiplier for globally asynchronous locally synchronous designs. In: *The IEEE International Symposium on Circuits and Systems (ISCAS)*. Volume 3. (2000) 13–16

Case Study of a Functional Genomics Application for an FPGA-Based Coprocessor^{*}

Tom Van Court¹, Martin C. Herbordt¹, and Richard J. Barton²

¹ Department of Electrical and Computer Engineering,
Boston University, Boston, MA 02215,
herbordt|tvancour@bu.edu

² Department of Electrical and Computer Engineering,
University of Houston, Houston, TX 77204,
rbarton@uh.edu

Abstract. Although microarrays are already having a tremendous impact on biomedical science, they still present great computational challenges. We examine a particular problem involving the computation of linear regressions on a large number of vector combinations in a high-dimensional parameter space, a problem that was found to be virtually intractable on a PC cluster. We observe that characteristics of this problem map particularly well to FPGAs and confirm this with an implementation that results in a 1000-fold speed-up over a serial implementation. Other contributions involve the data routing structure, the analysis of bit-width allocation, and the handling of missing data. Since this problem is representative of many in functional genomics, part of the overall significance of this work is that it points to a potential new area of applicability for FPGA coprocessors.

1 Introduction

Microarrays measure simultaneously the expression products of thousands of genes in a tissue sample and so are being used to investigate a number of critical biology questions. Among these are (paraphrasing from pages 19-20 of [4]): Given the effect of 5000 drugs on various cancer cell lines, which gene is most predictive of the responsiveness of the cell line to a particular chemotherapeutic agent? or Is there a group of genes that can serve to distinguish the outcomes of patients with disease ijk who are otherwise indistinguishable? or Which of all known genes have a pattern of expression similar to those genes regulated by factor xyz ?

As exciting as this usage is, microarray analysis is extremely challenging. One issue is that the data are noisy and the noise is often difficult to characterize. Another issue is that the number of measured quantities is invariably much larger than the number of samples. This results in an underconstrained system

* This work was supported in part by the National Science Foundation through CAREER award #9702483 and by a grant from the Compaq Computer Corporation.

not amenable to traditional statistical analysis such as finding correlations. As a result of these and other difficulties, techniques are used (often derived from machine learning) that provide a focus of attention, or a visualization, from which biological significance can be inferred. Among these are various forms of clustering, inference nets, and decision trees. Their computational complexity ranges from the trivial to the intractable. However, given the cost of obtaining microarray data, the fact that further biological interpretation is usually still required, and the value of many of the most rudimentary computations, most analysis applications are tailored to run fairly quickly on ordinary PCs.

It is undoubtedly the case, however, that biologists would like to ask far more complex questions and that increased computational capability would help to answer them. With applications such as those listed above, however, even a slightly harder question can result in an increase by orders of magnitude in computation. This is true of the problem we investigate here.

Kim [3] would like to find a set of genes whose expression can be used to determine whether liver tissue samples are metastatic or non-metastatic. For biological reasons, it is likely that three genes is an appropriate number to make this determination. Kim further proposes that use of linear regression would be appropriate to evaluate the gene subsets over the available samples. Since there are thousands of potential genes, 10^{10} to 10^{12} data subsets need to be processed. Although simple to implement, he reported that this computation was intractable even on his small cluster of PCs.

We decided to investigate the prospects of supporting this computation on an FPGA-based coprocessor. There are many reasons:

- It is an excellent match computationally. The data-set size, the amount of computation per datum, the nature of the individual computations, and the data-type size all are favorable to FPGAs.
- This computation is representative of a large number of similar computations in microarray analysis as well as in broader bioinformatics and computational biology (BCB). Therefore, demonstrating the efficacy for this problem would have much wider significance. Note that Yamaguchi et al. [11] have shown similar success (to what we show here) for a different application in bioinformatics, but one that has substantially different computational characteristics.
- There is a large number—perhaps thousands—of potential users. Therefore a low-cost distributed solution is much more attractive than a centralized resource such as a large-scale cluster. This is especially true since (as we will show) it would take a very large cluster to match performance.
- The state of algorithmics in microarray analysis (and some other areas of bioinformatics) is one of flux. Therefore a solution based on a generic PC and coprocessor may be more attractive than hardwired alternatives such as ASICs. The same would be true with respect to turn-key software/hardware systems, such as those provided by a number of vendors, assuming that they provided solutions to these problems at all. Also, both of these “hardwired” alternatives are extremely expensive.

What we have found is that we can obtain a speed-up of a factor of more than 1500 over an optimized serial version running on a 1.7GHz Pentium IV PC. These results have so far been achieved in simulation using post place-and-route timing for the Xilinx XC2VP100-7.

Our most basic contribution is the speed-up for this particular problem: a set of 10,000 genes can be examined in 10 minutes instead of 19 days. Other contributions have to do with the actual implementation, including a novel data routing structure, the analysis of the bit-width allocation, and our handling of missing data. Finally, as this problem has similar characteristics to many others in microarray analysis, we show that there is potential for broader applicability of FPGA coprocessors in this very important domain.

The rest of this paper is organized as follows. In the next section we present the problem formally and describe the serial implementation. There follows a description of the FPGA design and implementation including an analysis of data path widths. We conclude with a discussion.

2 Application Detail and Serial Implementation

The data to be analyzed are derived from n microarrays; each consists of a binary diagnosis and an expression value for each of the genes being analyzed. Expression values are tabulated in a matrix with row vectors corresponding to microarrays and column vectors corresponding to particular genes. The outcomes are tabulated in a column vector \mathbf{Y} . The technique used is to compute the linear regressions for all 3-way combinations of genes. Pearson's R^2 defines the goodness of fit for each regression. Examining a standard statistics reference [8], we find that the estimators $\hat{\beta}_0, \dots, \hat{\beta}_n$ comprising the column vector $\hat{\beta}$ can be computed as follows

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$$

The first column of \mathbf{X} consists of n 1s and each remaining column consists of the n expression values for one gene of the subset being considered in this particular combination. However, since much of computational complexity results from the inversion, it is important to reduce its rank. This is done by centering the data, which results in a $\hat{\beta} = \hat{\beta}_1, \dots, \hat{\beta}_n$ of

$$\hat{\beta} = [(\mathbf{X}^+)^T \mathbf{X}^+]^{-1} (\mathbf{X}^+)^T \mathbf{Y}^+$$

and

$$R^2 = \frac{\hat{\beta}(\mathbf{X}^+)^T \mathbf{Y}^+}{(\mathbf{Y}^+)^T \mathbf{Y}^+}$$

where \mathbf{X}^+ is the $n \times 3$ matrix containing column vectors with elements $X_{ij} - \bar{\mathbf{X}}_i$ and \mathbf{Y}^+ is the column vector containing the values of $Y_j - \bar{\mathbf{Y}}$. Note that in centered mode we do not need to compute $\hat{\beta}_0$ and thus do not need the column of ones in \mathbf{X} . We therefore operate on 3×3 matrices rather than 4×4 .

The covariance matrix

$$c_{ij} = (\mathbf{X}_i - \bar{\mathbf{X}}_i)^T (\mathbf{X}_j - \bar{\mathbf{X}}_j),$$

can also be written as

$$nc_{ij} = n\Sigma_k (X_{ik} X_{jk}) - (\Sigma_k X_{ik})(\Sigma_k X_{jk}).$$

The factor of n cancels in later operations, and eliminates the need to perform the division implied by \bar{X}_i . The entire computation for each gene combination thus partitions into two parts: the first consists of the 9 dot products and 4 summations while the second consists of computing the covariance matrix, inverting the matrix, and using those results to obtain β and R^2 .

In our tests we used data derived from a human breast tumor study by Perou, et al. [7]. The data are typical of those generated in microarray studies and consist of expressions of 9218 genes (including controls) from 84 microarray samples. Raw expression data are ratios. As is standard practice, we took the log, normalized, and rounded the data to integer values in the range -4 to +4. Using four bits is on the high end of information per sample; often only two bits are used. The result vector is binary: 0/1 for diseased/healthy.

An important consideration in microarray analysis is dealing with missing data. That is, the microarray value for a gene/sample expression is sometimes unreadable; in that case no value at all is reported. In the Perou data set, 46% of genes contain missing data. If not handled properly, missing data can dominate the regressions and render results meaningless. We take a simple approach: for a given combination of three genes, for each sample with missing data, we eliminate that sample from consideration for all genes. The combination is rejected completely if too many healthy samples are dropped.

Statistical literature [5] refers to this as a form of complete-case analysis within any one combination of genes, and available-case analysis in selecting combinations of genes. A χ^2 test of Perou's data shows that the frequency of missing values is not decisively different among healthy samples than among diseased samples. Thus, the missing data matches the Missing At Random (MAR) assumption and complete-case analysis appears justified.

When implementing the algorithm, one notes that each dot-product is used a large number of times. It seems to makes sense to precompute *all* of the $n \times n$ dot products, then use them in the $\binom{n}{3}$ inversions later. This eliminates roughly a factor of n dot products. Unfortunately, this does not account for missing data: each dot-product can have drop-outs not just from data missing from the two vectors being multiplied, but also from the third vector of the set. Since this third vector changes for every set, it follows that all dot products must be recomputed for every iteration.

We created two versions of the serial code. One was a mirror of the FPGA implementation and was used to verify results, especially with respect to maintaining precision. The other was used to generate timing and so was highly optimized for serial execution on a modern processor.

The R^2 for each set of genes was computed in 10.1us on a 1.7GHz Pentium IV PC. Because of the tremendous data locality, there was no drop-off in perfor-

mance with respect to the number of gene combinations evaluated. This means that evaluating 3-way combinations of 1,000 genes takes about half an hour, while 3-way combinations of 10,000 genes takes more than 19 days, and 20,000 genes takes more than five months. Clearly L1 cache and available ILP are being used to a very high degree, the latter not surprisingly due to the numerous multiply-accumulate (MAC) computations and the hardwired invert.

Still, these results confirm our initial assumption: that this computation, while perhaps not “heroic” in the grand-challenge sense, is still outside the realm of usage in exploratory data analysis. For this and similar computations to be readily usable as part of an analysis toolkit, days need to be reduced to minutes.

3 FPGA Methods, Implementation, and Results

3.1 Description

Hardware is assumed to be an FPGA on a commercial PCI board plugged into a PC. As the amount of reuse per datum is very high, details about the particular board and interface do not have a significant impact on our results: only a KB/second input rate needs to be supported.

The rest of this discussion describes simulations, synthesis, and place-and-route in the Xilinx ISE 5.2.02i environment [10] for Virtex-II Pro XC2VP100 gate array [9] and anticipates implementation on a generic coprocessor board when one becomes available later this year. The Virtex-II Pro product family is especially interesting here because of its large gate count, large on-chip memory, and dedicated multipliers. Implementations on other devices in this family follow from the one described here using analogous optimizations.

The circuitry consists of three parts: (i) vector storage and distribution (VSD) and the two computational segments described in Section 2: (ii) dot-products and summations (DPS), and (iii) covariance matrix, inverse, and regression (CIR). We now describe these, starting with the computations. In the following subsections we talk about optimizations and safety checks and then about integration and speed-matching. At that point the responsibilities of the vector storage and distribution unit are clear and it is then described.

Each DPS accepts four data vectors, three X values and one Y . The Y represents the diagnosis, 0 or 1 for cancerous or healthy samples respectively. The X values represent expression levels, encoded as four bit values. The encoding represents a symmetric range from $-N$ to $+N$. Earlier, we noted that the application works well when expression data is quantized to a range of -4 to 4 (encoded as 0 to 8). A special value (15, binary 1111) is a Not-a-Number (NaN) code that represents missing or invalid input data. The DPS unit, illustrated in Figure 1, consists of: counters to tally valid data sets and Y values, accumulators to total the X vectors and XY dot products, and MAC sections to total the $X_i X_j$ dot products and X_i^2 .

Note the handling of missing data. In Figure 1, the boxes labeled = NaN? detect missing data and propagate that fact to the MAC units where the accumulation of the invalid summands is blocked. If, for example X_{1i} is a missing

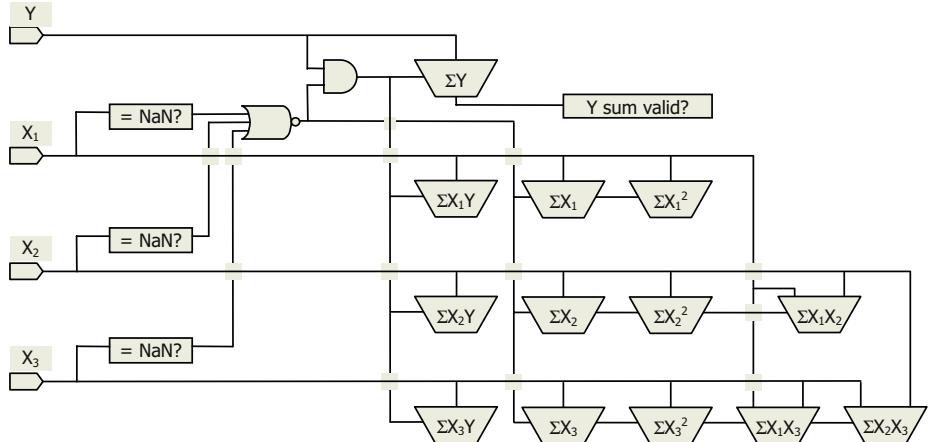


Fig. 1. Dot products and summations.

value, then the summands X_{1i}^2 , $X_{1i}X_{2i}$, $X_{1i}X_{3i}$ and $X_{1i}Y_i$ are obviously meaningless. Following [5], X_{2i} , X_{2i}^2 , X_{3i} , X_{3i}^2 , $X_{2i}X_{3i}$, $X_{2i}Y_i$, $X_{3i}Y_i$, and Y_i are also omitted from their respective sums and from the vector length count.

Once the vector is processed, DPS results are latched for input to the CIR. DPS results consist of outputs from all accumulators in Figure 1, a valid data counter, and the validity indicator (overflow and minimum-Y tests—details below). The result is presented broadside to the CIR. That section consists entirely of unclocked combinational logic, including adders, subtracters, and multipliers. It first computes the 3×3 covariance matrix from the dot products. That feeds the closed form inversion of the covariance matrix.

3.2 Optimizations and Safety Checks

One fundamental optimization is to minimize the datapath width at all points. Worst-case calculations of the minimum width are simple and safe, but result in a far more conservative implementation than necessary. We address this by: (i) *a priori* determining the maximum datapath widths and (ii) confirming that overflow has not occurred. There are two aspects to datapath width determination: the bits that can be dropped at the high end (because of worst cases that never occur in practice) and bits that can be dropped at the low end (because the loss of precision is insignificant).

The *a priori* path width determination is done in two ways: empirical and theoretical. On the theoretical side, the need for less significant bits is estimated using interval arithmetic [1]. Precision is verified empirically by sampling test data off-line. Also on the theoretical side, worst-case analysis sets an upper bound on the number of high-order bits required. It is interesting that most authors (e.g. [6]) perform worst-case bit allocation in whole bit increments, even when their bitwidth allocation is otherwise sensitive to application data values.

For example, consider the accumulator needed to add up 84 terms in a dot-product of vectors with 9 element values, 0 to 8, as described above. Naively, that would require $7 + 4 + 4 = 15$ bits. In fact, the accumulator need only hold $\lceil \log_2(84 \times 8 \times 8) \rceil = 13$ bits to handle the worst-case for this application.

Empirical estimates of high order bit requirements also come from measurements of calculations on sample data. The calculation is performed off-line, for a meaningful subset of the application cases, and the number of bits used at each step is measured. This gives statistics of actual bit usage. High-order bits are allocated to cover the samples observed, plus some margin based on observed standard deviations.

The implementation, then, handles all data values that can reasonably be expected. To be thorough, however, the FPGA logic checks for overflow at each step. The DPS and CIR stages both present validity indicators, stating whether an erroneous calculation was detected at any point.

As noted earlier, some regressions may be invalid because missing data values leave too few $Y = 1$ (healthy) samples. Figure 1 illustrates the “ Y sum valid?” check, a configurable test requiring some minimum number of healthy samples in a regression. If inadequate data with $Y = 1$ appears in a regression, that also marks the whole calculation as invalid. A similar test of the number of $Y = 0$ samples is not necessary for this data set.

In the CIR, one optimization is the use of closed form inversion. Although this method has many problems when applied to large matrices, it works well with these matrix, vector, and data sizes. Besides the obvious speed advantage, hardwiring allows us to keep cofactors and determinant separate for independent use in computing correlation coefficients and regression.

Another optimization is that all multiplications in the CIR use the FPGA’s block multipliers. The DPS, with smaller operands, uses multipliers built from logic. Further optimization may be possible in the CIR by exchanging some block multipliers for multipliers built from logic for sufficiently small operands.

Pipelining the CIR (not currently done) may also be advantageous. The CIR unit is implemented as three combinational logic elements in series: the correlation matrix, its inverse, and the regression results. These represent natural boundaries for pipelining.

It may also be possible to speed up the DPS by processing vectors in parallel instead of serially as is currently done. However, timing and resource allocation tradeoffs make it unlikely that this will improve performance significantly.

3.3 Timing and Speed-Matching Optimization

Timing is computed statically, using post place-and-route (PAR) results. PAR is completely automatic, with no manual guidance, timing constraints, or floor-planning. Timing estimates are based on synthesis of 9 CIR units, each supplied by 10 DPS units (90 DPSs total; see Figure 2 for one CIR/10 DPS unit) and VSD as described below. The target is a XC2VP100 gate array with speed grade -7. The entire design, minus ‘glue’ needed to attach the application logic to its host environment, occupies 73% of the logic slices and 94% of the block multipliers.

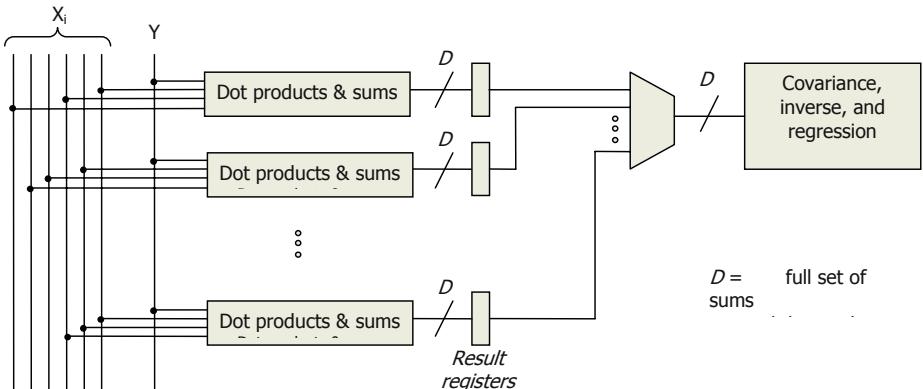


Fig. 2. Computation balancing.

Post-PAR timing analysis indicates a DPS clock of 5.35ns while propagation delay through the CIR to its result register is 47.36ns. Using conservative approximations, this implies a 5.5ns DPS clock rate and very roughly a 10:1 ratio of CIR delay to DPS rate. The system clock is set to the DPS rate. A separate counter divides the system rate down so that CIR results are latched and new results presented to the CIR at 1/10 the DPS clock rate.

Given data vectors of length ~ 80 , the DPS requires about 500ns to compute a result for one set of vectors. The CIR versus DPS imbalance, about 50ns vs. 500ns, is conspicuous. The other conspicuous imbalance is in the resources used by each section. Each CIR requires 48 of the target FPGA's 444 block multipliers. The DPS, however, uses only 4×4 -bit products, which can be built more effectively from ordinary logic. A DPS uses smaller, less specialized logic resources, under 1% of the chip total. Considering both execution time and logic resources used, to achieve maximum logic activity, each CIR serves 10 DPS units. Figure 2 shows how this is done.

The DPSs all start and end their computations at the same time, latch their results, and begin processing the next set of vectors. After latching the DPS results, separate logic presents results from each DPS to its CIR sequentially. A counter (not shown in Figure 2) holds input stable for the CIR propagation delay, then latches the CIR result and reads the next saved DPS result.

These design values (vector length, CIR propagation delay, etc.) and available resources are all parameters specific to the problem and technology at hand. Different parameters would yield different specific results, but the analysis would follow the same general pattern over a wide range.

3.4 Vector Store and Distribution

Recall that each DPS unit processes 3 vectors and that there are 90 DPS units. Therefore, the VSD must simultaneously distribute 270 vector streams. A general (but impractical) solution would be to store all $v = 10,000$ vectors on chip in a

270-port memory. Size itself is not the problem: the entire data set fits in 1MB. Our solution leverages the v^2 reuse of each vector in a hierarchical structure.

We observe that 9 vectors form $\binom{v}{3} = 84$ combinations and so are nearly sufficient to keep the 90 DPS units busy. This leaves six DPSs idle, the price paid for efficient memory access. A simple network (shown on the left in Figure 2 and the right in Figure 3) distributes vector data to the DPSs.

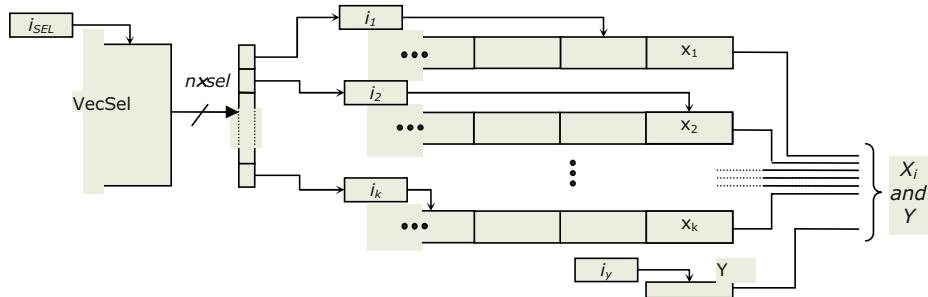


Fig. 3. Vector store and distribution to the DPS units.

Each vector store unit (labeled $x_1 \dots x_k$ in Figure 3) feeds one of the distribution network's nine X inputs and is indexed by an independent address register (labeled $i_1 \dots i_k$). The sequence of vectors is stored in VecSel. The chip's 16Kb RAM blocks can each hold 50 vectors, so vector storage uses only a small amount of the available RAM. Vector store units and the VecSel RAM are loaded from off-chip. Double buffering ensures that loading does not interfere with data access for computation. Note that many more values are read from the vector stores than are loaded into them, since each vector is reused in many size-9 combinations, so the DPS should never be idle while the VSD is reloaded.

Operation is as follows. A set of 9 vectors is chosen at the beginning of a vector computation, one from each vector store unit, as indicated by the address registers. Successive vector elements are read by incrementing all of the data address registers in unison, through the length of the vector. At the end of a vector, the address registers are reloaded from the VecSel and the next set of data vectors is ready for transfer.

This VSD design supports the solution of the following combinatorial problem: choosing sets of size 9 (or k) from a set of 9218 (or v) elements such that every size-3 subset of those v appears in one size- k set. This is exactly the Steiner System $S(3, k, v)$. Generating algorithms exist for these size- k sets [2], but are not amenable to FPGA implementation. Off-line computation generates that set of size- k sets, and loads the vector stores and VecSel accordingly. The VecSel RAM and vector stores must be reloaded $10^5 - 10^6$ times to cover all size-3 sets; however, this is a trivial requirement for a run of several minutes.

3.5 Throughput

As described, the basic component of our design consists of a pipeline with a single CIR (~ 50 ns) and single DPS (~ 500 ns for a length-80 vector). This gives a speed-up of a factor of 20 over the serial PC version. The chosen Virtex II Pro easily fits 9 of these units (the critical resource being the multipliers) increasing the speed-up to $180 \times$. Each CIR serves 10 DPSs reducing the cycle time of the original unit to 50 ns and increasing the total speed-up to a factor of $1800 \times$. Combinatorial inefficiency in the VSD reduces this factor to $1600 \times$.

4 Discussion and Extensions

The 1000-fold+ speed-up derived from our FPGA implementation achieves our goal of reducing the duration of this computation from days to minutes. This is done while keeping the system cost (hardware and IT support) low. It is therefore quite plausible that this and related techniques could indeed become part of a computational toolbox for functional genomics, broadening the types of inferences possible from microarray data.

A necessary extension to this work is achieving some generality in implementation: it is certainly expensive to buy a large cluster and hire a parallel applications programmer; it is perhaps even more problematic to find good FPGA designers. We are currently working in this direction.

References

1. Hansen, E., et al.: Topics in Interval Analysis. Clarendon Press, Oxford, U.K. (1969)
2. Hartman, A.: The fundamental construction for 3-designs. Discrete Mathematics **124** 107-132 (1994)
3. Kim, S.: Finding Genes for Cancer Classification: Many Genes and Small Number of Samples. 2nd Ann. Houston Forum on Cancer Genomics and Informatics (2001)
4. Kohane, I.S., Kho, A.T., Butte, A.J.: Microarrays for an Integrative Genomics. MIT Press, Cambridge, MA (2003)
5. Little, R.J.A., Rubin, D.B.: Statistical Analysis with Missing Data John Wiley and Sons, Hoboken, NJ (2002)
6. Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., Sherwood, T.: Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. IEEE Trans. on CAD of Integrated Circuits and Systems **20** (2001) 1355-1370
7. Perou, C.M., et al.: Molecular Portraits of Human Breast Tumors. Nature **406** (2000) 747-752
8. Ryan, T.P.: Modern Regression Methods. John Wiley and Sons, Inc., New York (1997)
9. Xilinx, Inc.: Virtex-II Pro Platform FPGA User Guide. (2002)
10. Xilinx, Inc.: Integrated Software Environment. (2002)
11. Yamaguchi, Y., Miyajima, Y., Maruyama, T., Konagaya, A.: High Speed Homology Search Using Run-Time Reconfiguration. In Proceedings of the 12th International Conference on Field Programmable Logic and Applications (2002) 281-291

A Smith-Waterman Systolic Cell

C.W. Yu, K.H. Kwong, K.H. Lee, and P.H.W. Leong

Department of Computer Science and Engineering,
The Chinese University of Hong Kong,
Shatin, New Territories, Hong Kong SAR,
`{cwyu1, khkwong, khlee, phwl}@cse.cuhk.edu.hk`

Abstract. With an aim to understand the information encoded by DNA sequences, databases containing large amount of DNA sequence information are frequently compared and searched for matching or near-matching patterns. This kind of similarity calculation is known as sequence alignment. To date, the most popular algorithms for this operation are heuristic approaches such as BLAST and FASTA which give high speed but low sensitivity, i.e. significant matches may be missed by the searches. Another algorithm, the Smith-Waterman algorithm, is a more computationally expensive algorithm but achieves higher sensitivity. In this paper, an improved systolic processing element cell for implementing the Smith-Waterman on a Xilinx Virtex FPGA is presented.

1 Introduction

Bioinformatics is becoming an increasingly important field of research. With the ability to rapidly sequence DNA information, biologists have the tools to, among other things, study the structure and function of DNA; study evolutionary trends; and correlate DNA information with disease. For example, two genes were identified to be involved in the origins of breast cancer in 1994 [1]. Such research is only possible through the help of high speed sequence comparison.

All the cells of an organism consist of some kind of genetic information. They are carried by a chemical known as the deoxyribonucleic acid (DNA) in the nucleus of the cell. DNA is a very large molecule and nucleotide is the basic unit of this type of molecule. There are 4 kinds of nucleotides and each have different bases, namely adenine, cytosine, guanine and thymine. Their abbreviated forms are “A”, “C”, “G” and “T” respectively. In this paper, the sequence is referred to as a string, and the bases form the alphabet for the string.

It is possible to deduce the original sequencing in DNA which codes for a particular amino acid. By finding the similarity between a number of “amino-acid producing” DNA sequences and a genuine DNA sequence of an individual, one can identify the protein encoded by the DNA sequence of the individual. In addition, if biologists succeed in finding the similarity between DNA sequences of two different species, they can understand the evolutionary trend between them. Another important usage is that the relation between disease and inheritance can also be studied. This is done by aligning specific DNA sequences of individuals

with disease to those of normal people. If correlations can be found which can be used to identify those susceptible to certain diseases, new drugs may be made or better techniques invented to treat the disease. There are many other applications of bioinformatics and this field is expanding at an extremely fast rate.

A human genome contains approximately 3 billion DNA base pairs. In order to discover which amino acids are produced by each part of a DNA sequence, it is necessary to find the similarity between two sequences. This is done by finding the minimum string edit distance between the two sequences and the process is known as sequence alignment.

There are many algorithms for doing sequence alignment. The most commonly used ones are FASTA [2] and BLAST [3]. BLAST and FASTA are fast algorithms which prune the search involved in a sequence alignment using heuristic methods. The Smith-Waterman algorithm [4] is an optimal method for homology searches and sequence alignment in genetic databases and makes all pairwise comparisons between the two strings. It achieves high sensitivity as all the matched and near-matched pairs are detected, however, the computation time required strongly limits its use.

Sencel Bioinformatics [5] compared the sensitivity and selectivity of various searching methods. The sensitivity was measured by the coverage, which is the fraction of correctly identified homologues (true positives). The coverage indicates what fraction of structurally similar proteins one may expect to identify based on sequence alone. Their experiments show that for a coverage around 0.18, the errors per query of BLAST and FASTA are about two times that of the Smith-Waterman Algorithm.

Many previous ASIC and FPGA implementations of the Smith-Waterman algorithm have been proposed and some are reviewed in Section 4. To date, the highest performance chip [6] and system level [7] performance figures have been achieved using a runtime reconfigurable implementation which directly writes one of the strings into the FPGA's bitstream.

In this work, an FPGA-based implementation of the Smith-Waterman algorithm is presented. The main contribution of this work is a new 3 Xilinx Virtex slice Smith-Waterman cell which is able to achieve the same density and performance as an earlier reported cell [6], without the need to perform runtime reconfiguration. This has advantages in that the design is less FPGA device specific and thus can be used for non-Xilinx FPGA devices as well as ASICs. Whereas the runtime reconfigurable design requires JBits, a Xilinx specific API for runtime reconfiguration, the design presented in this paper was written in standard VHDL. Moreover, in the proposed design, both strings being compared can be changed rapidly as compared to a runtime reconfigurable system in which the bitstream must be generated and downloaded, which is typically a very slow process since a large bitstream must be manipulated and downloaded via a slow interface. This reconfiguration process may become a bottleneck, particularly for small databases. Furthermore, other applications may require both strings

to change quickly. The design was implemented and verified using Pilchard [8], a memory-slot based reconfigurable computing environment.

2 The Smith-Waterman Algorithm

The Smith-Waterman Algorithm is a dynamic programming technique which utilizes a 2D table. As an example of its application, suppose that one wishes to compare sequence S (“ACG”) with sequence T (“ATC”). The intermediate values a , b and c (shown in Fig. 1(b)) are then used to compute d according to the following formula:

$$d = \min \begin{cases} a & \text{if } S_i = T_j \\ a + \text{sub} & \text{if } S_i \neq T_j \\ b + \text{ins} \\ c + \text{del} \end{cases} \quad (1)$$

If the strings being compared are the same, the value a is used for d . Otherwise, the minimum of a plus some substitution penalty sub , b plus some insertion penalty ins and c plus some deletion penalty del is used for d . Data dependencies mean that entries d in the table can only be calculated if the corresponding a , b , c values are already known and so the computation of the table spreads out from the origin as illustrated in Fig. 2. As an example, the first entry that can be computed is that for “AA” in Fig. 1(a). Since $S_i = T - i = 'A'$, according to Equation 1, $d = a$ and so the entry is set to 0. In order to complete the table, the template of Fig. 1(b) is moved around the table constrained by the dependencies indicated by Fig. 2.

The substitution, insertion and deletion penalties can be adjusted for different comparison requirements. If the presence of redundant characters is relatively less acceptable than just a difference in characters, the insertion and deletion penalties can be set to a higher value than the substitution penalty. In the

		A	C	G				
	0	1	2	3				
A	1	?	?	?	S_i			
T	2	?	?	?	T_i	a	b	
C	3	?	?	?		c	d	

(a) Initial table.
(b) Equation 1 values.
(c) Final table.

Fig. 1. Figure showing the progress of the Smith-Waterman algorithm, when the string “ACG” is compared with “ATC”.

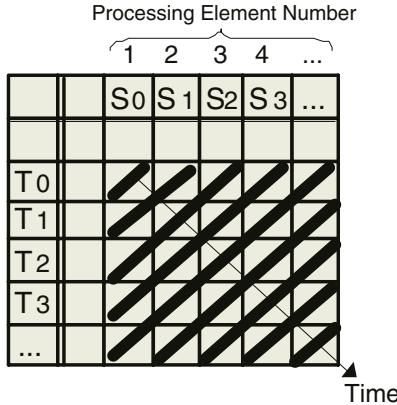


Fig. 2. Data dependencies in the alignment table. Thick lines show entries which can be computed in parallel and the time axis is arranged diagonally.

alignment system presented, the insertion and deletion penalties were fixed at 1 and the substitution penalty set to 2, as is typical in many applications.

If S and T are m and n in length respectively, then the time complexity of a serial implementation of the Smith-Waterman algorithm is $O(mn)$. After all the squares have been processed, the result of Fig. 1(c) is obtained. In a parallel implementation, the positive slope diagonal entries of Fig. 2 can be computed simultaneously. The final edit distance between the two strings appears in the bottom right table entry.

3 FPGA Implementation

In 1985, Lipton and Lopresti observed that the values of b and c in Equation 1 are restricted to $a \pm 1$ and the equation can be simplified to obtain [9]:

$$d = \begin{cases} a & \text{if } ((b \text{ or } c) = a - 1) \text{ or } (S_i = T_j) \\ a + 2 & \text{if } ((b \text{ and } c) = a + 1) \text{ and } (S_i \neq T_j) \end{cases} \quad (2)$$

Using Equation 2, it can be seen that the data elements b, c and d only have two possible values. Therefore, the number of data bits used for the representation of b, c and d can be reduced to 1 bit. Furthermore, two bits can be used to represent the four possible values of the alphabet.

The processing element (PE) shown in Fig. 3 was used to implement Equation 2. A number of PEs are then connected in a linear systolic array to process diagonal elements in the table in parallel. As shown in Fig. 2, PEs are arranged horizontally and are responsible for its corresponding column. In the description that follows, the sequence that changes infrequently is S and the sequences from the database are T . In each PE, two latches are used to store a character S_i .

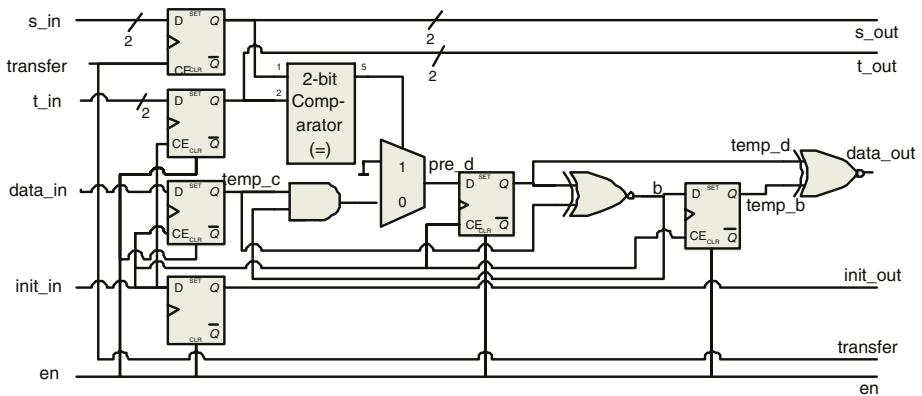


Fig. 3. The Smith-Waterman processing element (PE). Boxes represent D-type flip-flops.

These characters are shifted and distributed to every processing element before the actual comparison process begins. The base pairs of T are passed through the array during the comparison process, during which the d of Equation 2 is also computed.

In order to calculate d , inputs a, b and c should be available. In the actual implementation, the new values b, c and d are calculated during the comparison of the characters as follows:

1. data_in is the new value of c and it is stored in a flip-flop. At the same time, this new c value and the previous d value (from a flip-flop) determines the new b value ($b = \text{temp}_c \text{ XNOR } \text{temp}_d$)
2. The new b value is stored in a flip-flop. At the same time, the output of a 2-to-1 MUX is then selected depending on whether $S_i = T_i$. The output of the MUX (a '0' value or (b AND temp_c)) becomes the new d value. This new d value is stored in a flip-flop.
3. Values of b and d determine the data output of the PE ($\text{data_out} = \text{temp}_b \text{ XNOR } \text{temp}_d$). The data output from this PE is connected to the next PE as its data input (its new c value)

When the transfer signal is high, the sequence S is shifted through the PEs. When the en signal is high, all the flip-flops (except the two which store the string S) are reset to their initial values. The init signal is high when new signals from the preceding PE are input and the new value of d calculated. When the init signal is low, the data in all the flip-flops are unchanged.

Each PE used 8 flip-flops as storage elements and 4 LUTs to implement the combinational logic. Thus the design occupied 4 Xilinx Virtex slices. Guccione and Keller [6] used runtime reconfiguration to write one of the sequences into the bitstream, saving 2 flip-flops and implementing a PE in 3 slices. In the proposed approach, two otherwise unused LUTs were configured as shift register elements

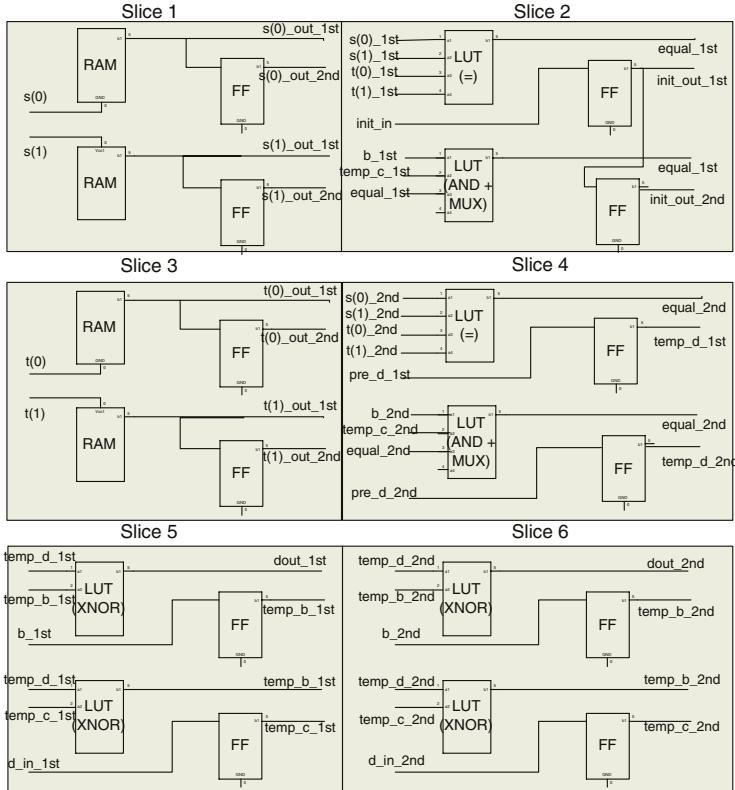


Fig. 4. Two processing elements mapped to 6 Virtex slices.

using the Xilinx distributed RAM feature [10]. Thus the design occupies 3 Xilinx Virtex slices per PE, without requiring runtime reconfiguration to change S . In the actual implementation, 2 PEs were implemented in 6 slices since sharing of resources between adjacent PEs was necessary in the actual implementation.

Fig. 4 shows the mapping of the PEs to slices. All the signals ending with “_1st” were used in PE Number 1, and signals ending with “_2nd” were used for PE2. The purpose of each signal can be understood by referring back to Fig. 3. It was necessary to connect the output of the RAM-based flip-flops directly to a flip-flop (FF in the diagram) in the same logic cell (LC) since internal LC connections do not permit them to be used independently (i.e. it was not possible to avoid connecting the output of the RAM and the input of the FF). Thus, Slice 1 was configured as a 2 stage shift register for consecutive values of S_i and Slice 3 was used for two consecutive values of T_i .

Fig. 5 shows the overall system data path. Since the T sequences are shifted continuously, the system used a FIFO constructed from Block RAM to buffer the sequence data supplied by a host computer. This improves throughput of the system since a large number of string comparisons can be completed before

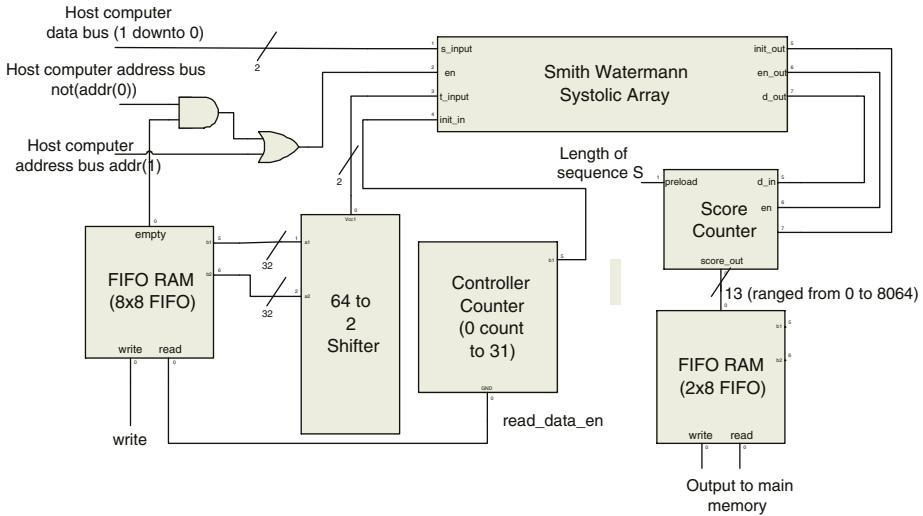


Fig. 5. System data path.

all of their scores are read from the controller, reducing the amount of idle time in the systolic array. The input and output data width of the FIFO RAM were both 64 bits. The wide input data width helped to improve IO bandwidth from the host computer to the FIFO RAM. A 64-to-2 shifter and a controller counter were used for reducing the output data width of the FIFO RAM from 64 bits to 2 bits, so as to allow data to be fed into the systolic array.

The Score Counter computes the edit distance by accumulating results calculated in the last PE of the systolic array. The output of the last PE is actually the d value in the squares of the rightmost column of the matrix, and differences in values of consecutive squares in the rightmost column must be 1. The d_{out} of the last PE is '0' when $d = b - 1$, and the output '1' when $d = b + 1$. Therefore, a Shift Counter was initialized to the length of the sequence S . It was decremented if the output value is '0', otherwise it was incremented. After the entire string T is passed through the systolic array, the counter contains the final string comparison score.

4 Results

The design was synthesized from VHDL using the Xilinx Foundation 5.1i software tools and implemented on Pilchard, a reconfigurable computing platform [8] (Fig. 6). The Pilchard platform uses a Xilinx Virtex XCV1000E-6 FPGA (which has 12288 slices) and uses a SDRAM memory bus interface instead of the conventional PCI bus to reduce latency.



Fig. 6. Photograph of the Pilchard board.

A total of 4,032 PEs were places on an XCV1000E-6 device (this number was chosen for floorplanning reasons). As reported by the Xilinx timing analyzer, the maximum frequency was 202 MHz.

A number of commercial and research implementations of the Smith Waterman algorithm have been reported and their performance are summarized in Table 1. Examples are Splash [11], Splash 2 [12], SAMBA [13], Paracel [14], Celera [15], JBits from Xilinx [6], and the HokieGene Bioinformatics Project [7]. The performance measure of cell updates per second (CUPS) is widely used in the literature and hence adopted for our results.

Splash contains 746 PEs in a Xilinx XC3090 FPGA performing the Smith-Waterman Algorithm. Splash 2's hardware was different from Splash, which used XC4010 FPGAs with a total of 248 PEs. SAMBA [13] incorporated 16 Xilinx XC3090 FPGAs with 128 PEs altogether dedicated to the comparison of biological sequences.

ASIC and software implementations have also been reported. Paracel, Inc. used a custom ASIC approach to do the sequence alignment. Their system used 144 identical custom ASIC devices, each containing approximately 192 processing elements. Celera Genomics Inc. reported a software based system using an 800 node Compaq Alpha cluster.

Both the JBits and the HokieGene Bioinformatics Project were the latest reported sequence alignment systems using the Smith-Waterman Algorithm and use the same PE design. JBits reported performance for two different FPGA chips, the XCV1000-6 and the XC2V6000-5. The HokieGene Bioinformatics Project used an XCV6000-4. As can be seen from the table, the performance of the proposed design is similar to the JBits design on the same size FPGA (a XCV1000-6), and the JBits and HokieGene implementations on an XCV6000 gain performance by fitting more PEs on a chip, and our performance on the same chip would be similar.

The implementation was successfully verified using the Pilchard platform whcih provides a 133 MHz, 64-bit wide memory mapped bus to the FPGA. The processing elements and all other logic of the implementation operate from the same 133 MHz clock. The interface logic occupied 3% of the Virtex device. The working design was used mainly for verification performance and had a disappointing performance of approximately 136 B CUPS, limited by the simple polling based host interface used. A high speed interface which performs more buffering and is able to cause the memory system to perform block transfers between the host and Pilchard is under development.

Table 1. Performance and hardware size comparison of previous implementations (processor core not including system overheads). Device performance is measured in cell updates per second (CUPS).

System	Number of Chips	PEs per chip	System Performance (CUPS)	Device Performance (CUPS)	Run-time reconfiguration requirement
Splash(XC3090)	32	8	370 M	11 M	No
Splash 2(XC4010)	16	14	43 B	2,687 M	No
SAMBA(XC3090)	32	4	1,280 M	80 M	No
Paracel(ASIC)	144	192	276 B	1,900 M	N/A
<i>Celera (software implementation)</i>	800	1	250 B	312 M	N/A
JBits (XCV1000-6)	1	4,000	757 B	757 B	Yes
JBits (XC2V6000-5)	1	11,000	3,225 B	3,225 B	Yes
HokieGene (XC2V6000-4)	1	7000	1,260 B	1,260 B	Yes
This implementation (XCV1000-6)	1	4,032	742 B	742 B	No
This implementation (XCV1000E-6)	1	4,032	814 B	814 B	No

5 Conclusion

A technique, commonly used in VLSI layout, in which two processing elements are merged into a compact cell was used to develop a Smith-Waterman systolic processing element design which computes the edit distance between two strings. This cell occupies 3 Xilinx Virtex slices and allows both strings to be loaded into the system without runtime reconfiguration. Using this cell, 4032 PEs can fit on a Xilinx XCV1000E-6, operate at 202 MHz and achieve a device performance of 814 B CUPS.

References

1. Y. Miki, et. al. A Strong Candidate for the Breast and Ovarian Cancer Susceptibility Gene, BRCA1. *Science*, 266:66–71, 1994.
2. European Bioinformatics Institute Home Page, FASTA searching program, 2003. <http://www.ebi.ac.uk/fasta33/>.
3. National Center for Biotechnology Information. NCBI BLAST home page, 2003. <http://www.ncbi.nlm.nih.gov/blast>.

4. T. F. Smith and M. S. Watermann. Identification of common molecular subsequence. *Journal of Molecular Biology*, 147:196–197, 1981.
5. Sencel's search software, 2003. <http://www.sencel.com>.
6. Steven A. Guccione and Eric Keller. Gene matching using JBits, 2002. <http://www.ccm.ece.vt.edu/hokiegene/papers/GeneMatching.pdf>.
7. K. Puttegowda, W. Worek, N. Pappas, A. Danapani and P. Athanas. A run-time reconfigurable system for gene-sequence searching. In *Proceedings of the International VLSI Design Conference*, page (to appear), Jan 2003.
8. P. Leong , M. Leong , O. Cheung , T. Tung , C. Kwok , M. Wong and K. H. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, page (to appear), April 2001.
9. Richard J.Lipton and Daniel Lopresti. A systolic array for rapid string comparison. In *Proceedings of the Chapel Hill Conference on VLSI*, pages 363–376, 1985.
10. Xilinx. *The programmable logic data book*, 2003.
11. D. T. Hoang. A systolic array for the sequence alignment problem. *Brown University, Providence, RI, Technical Report*, pages CS–92–22, 1992.
12. D. T. Hoang. Searching genetic databases on splash 2. In *Proceedings 1993 IEEE Workshop on Field-Programmable Custom Computing Machines*, pages 185–192, 1993.
13. Dominique Lavenier. SAMBA: Systolic Accelerators for Molecular Biological Applications, March 1996.
14. Paracel, inc, 2003. <http://www.paracel.com>.
15. Celera genomics, inc, 2003. <http://www.celera.com>.

Software Decelerators

Eric Keller, Gordon Brebner, and Phil James-Roxby

Xilinx Research Labs, Xilinx Inc., U.S.A.,
`{Eric.Keller,Gordon.Brebner,Phil.James-Roxby}@xilinx.com`

Abstract. This paper introduces the notion of a *software decelerator*, to be used in logic-centric system architectures. Functions are offloaded from logic to a processor, accepting a speed penalty in order to derive overall system benefits in terms of improved resource use (e.g. reduced area or lower power consumption) and/or a more efficient design process. The background rationale for such a strategy is the increasing availability of embedded processors ‘for free’ in Platform FPGAs. A detailed case study of the concept is presented, involving the provision of a high-level technology-independent design methodology based upon a finite state machine model. This illustrates easier design and saving of logic resource, with timing performance still meeting necessary requirements.

1 Introduction

The research literature in field-programmable logic contains many examples of ‘hardware accelerators’. In short, these concern a processor-centric system model: algorithms are executed on a processor, with certain key functions being performed by an associated programmable logic array, the intention being to achieve greater overall performance. The exact arrangements may vary from the tightly-integrated case of a processor with an augmented instruction set (e.g. [3]) to the more loosely-coupled case of a processor interacting with a co-processing logic device (e.g. [6]).

This paper is concerned with a *logic-centric* system model, of the sort described for example in earlier papers by Brebner [2]. Here, the main computational focus is on logic circuitry, with other components — in particular processors — viewed as additional system components rather than central system components. Aside from computational differences, there are implied architectural differences, notably concerning serial data access and shared buses, features both tailored to processor behavior. The logic-centric model is well-suited to systems that react to, and are driven by, inputs received over time. Thus, in contrast to the usual processor-centric model, the environment, not the computer, is the controlling force. We feel that this view will be of increasing relevance to real-life systems in the future.

In the logic-centric model, it is fairly natural to invert accepted wisdom about system organization. In this paper, we consider the benefits of ‘software decelerators’, inverting the notion of hardware accelerators. The basic idea is that algorithms are executed in programmable logic, with certain functions being

performed by an associated processor. In general, this direction of migration is not likely to lead to speed increases — indeed quite the opposite — which is why we use the word ‘decelerator’; however, overall system speed requirements will still be met. It is intended that there are other motivations that lead to an overall increase in the quality of the design process and/or the resulting systems. We discuss various possible motivations, and then describe one detailed case study experiment where the main motivation was to provide a high-level, technology-independent design methodology based upon finite state machines.

In this case study, the software decelerator technique concerns finite state machines being implemented on the embedded processor of a Platform FPGA. With this approach, the cost of implementing a machine in terms of logic resource usage is greatly reduced, since the processor is always present on the Platform FPGA, whether it is used or not. The emphasis differs from that of some conventional state machine design methodologies, such as Esterel Studio, the principal aims being to consume as few logic resources as possible, optimize the interfacing between logic and processor, and run code directly from the processor’s built-in cache.

The paper is organized as follows. Section 2 considers the technological background that motivates software decelerators as a viable concept in system design. Then, Section 3 describes the case study, and Section 4 discusses initial experimental results. Finally, Section 5 contains some conclusions and directions for future work.

2 Technological Background

2.1 Emergence of Platform FPGAs

Early Field Programmable Gate Array (FPGA) architectures consisted of an array of similar programmable logic elements interfaced to interconnection elements. With the emergence of the Platform FPGA, architectures have evolved to a pre-defined mix of very different elements — which in time will largely supersede monolithic logic arrays. Today, for example, the Xilinx Virtex -II Pro Platform FPGA contains configurable logic blocks, I/O blocks supporting many different I/O standards, distributed Block RAM memories, internal access to the configuration memory, digital clock managers, gigabit transceivers, dedicated multiplier blocks and embedded PowerPC 405 processors. Platform FPGAs present both unique design challenges and unique design opportunities. Importantly, the mix of resources is pre-determined by the FPGA vendor rather than by the designer, and so a particular set of resources will be present whether or not the designer makes use of them, a situation unlike that in ASIC design.

Therefore, in designs that aim to maximize the use of the resources of a certain device size, the designer may often make decisions which on the surface appear non-intuitive. For example, in designs which use little BlockRAM memory but use many configurable logic blocks (CLBs), a designer may choose to implement certain logic functions by lookup tables in BlockRAM rather than

in CLBs. This is despite the fact that this approach wastes much of the Block-RAM data width, and does not take advantage of their dual-port nature. Such an approach would be unthinkable in ASIC design, but makes a lot of sense in Platform FPGA design. This use of pre-existing resources in unusual ways is likely to become more and more prevalent as the mixture of resources becomes richer and richer. The overall message is that one has to be very fluid in terms of how and where computation, storage and communication are carried out in systems.

The logic-centric system paradigm is one approach to assist in managing the potential design space. This focuses on inputs, outputs and programmable logic circuitry as the core system architecture, with all other components (memory, processors, etc.) being seen as *assists* to the circuitry. Of course, it is also possible to view the system in other ways. For example, in a more conventional processor-centric manner, the Virtex-II Pro can also be seen as a ‘motherboard on a chip’, and indeed it has been demonstrated as such, with the Linux operating system running on the PowerPC processor.

2.2 Motivation for Software Decelerators

The concept of using software decelerators derives directly from the discussion of using Platform FPGA resources in unusual ways, and focuses on processors in particular for non-standard treatment. In fact, what is sought here is not necessarily a completely different and unusual harnessing of processors, rather some balance between the long and rich legacies of processor development and software engineering, and the new context of the logic-centric system.

The more general backdrop to this reconsideration of the role of processors is an examination of the role of any kind of *universal machine* within a logic-centric system. Other examples, simpler than a traditional microprocessor, would be programmable state machines or microcontrollers. In all such cases, there is a basic trade-off between speed — one normally expects a universal machine implementation of a function to be slower than a bespoke implementation — and other issues as diverse as chip area requirements and ease and speed of implementing new functions.

So far, in the development of Platform FPGAs that include processors, there has largely been a drive to maximize processor clock rates, reflecting a view either that the processor is the central system component or perhaps that the processor plays a key time-critical supporting role in the system. This drive parallels the continuing race to increase clock rates of microprocessor chips (although, very recently, there has been acknowledgement that raw speed is not everything, power consumption being of importance in an increasing proportion of systems). It is our belief that, as far as Platform FPGAs are concerned, the clock rate of the processor may not be the dominant concern for many future systems. This follows from a prediction that the processor may either be called upon relatively infrequently to carry out work in the logic-centric system, or may be called upon to perform work of a non time-critical nature. One recent example of such

a system is the high throughput, low latency mixed-version IP router developed for the Virtex-II Pro [2].

As soon as the clock rate requirement is relaxed, it becomes possible to focus the treatment of a processor on other goals, like saving logic resource by employing the processor plus its supporting cache memory. In this paper, the goal is to combine this particular trade-off with the provision of a particular high-level design flow that hides the nature of the implementation from the user. A more straightforward and obvious application of software deceleration is just to make the overall system implementation process easier by allowing a designer access to the wide range of software tools (and human expertise) to implement functions on the processor, rather than implementing logic circuitry.

To ensure that software decelerators provide the anticipated benefits, and do not impose resource demands on an overall logic-centric system design, nor impose unnatural design methodologies on the user, there are some particular attributes that are desirable:

- The overall area consumed by a software decelerator implementation should not be greater than its logic circuitry counterpart unless there are other strong benefits.
- The interfacing between logic circuitry and processor should consume minimal programmable logic resources, and should be designed to shield the processor from the logic and vice-versa.
- The method of capturing designer intent should be independent of the actual implementation mechanism chosen for the Platform FPGA unless there is a particular benefit in permitting the use of certain familiar tools.
- The designer should be able to get accurate timing information, and resource usage information in general, for the overall logic-centric system, taking into account the behavior of the various non-logic system components that are being harnessed.

The case study that is presented in the subsequent sections illustrates that it is feasible to meet these goals, in the framework of software decelerator use.

3 Case Study: FSM-Based Design Methodology

Finite state machines (FSMs) are an important component of many digital systems, and can be implemented well on FPGAs. Particular FSMs may contain a large number of states, and may involve much computation to determine the next state and the state outputs based on varying inputs. However, they may actually have relatively relaxed timing constraints compared to the rest of a system, an attribute that points to possible software decelerator implementation.

The case study problem tackled was to implement FSMs as an example of a software decelerator. Referring back to three desirable attributes defined in Section 2.2, the interfacing hardware should consume minimal resources and act as a shield between the processor and the rest of the system — the rest of the

system should not know it is working with a processor. Capture should be implementation independent, that is, the designer should not be aware that an embedded processor is being targeted. Finally, accurate timing and resource usage information should be obtainable so that the designer is able to get hardware-like metrics. The net effect of using a software decelerator in a system where the processor would otherwise be idle is that logic resources are freed, without penalizing the designer in terms of design style or quality of timing information.

There has been a fairly substantial body of prior work on implementing finite state machines in software. Perhaps the most notable effort is the Berkeley POLIS system [1]. POLIS is a complete co-design solution, which uses the codesign finite state machine (CFSM) as the central representation of the required system behavior. One significant difference between a CFSM and a classical FSM is that a CFSM allows that the reaction time to events will be non-zero, unlike the FSM which assumes a synchronous communication model.

POLIS allows the designer to implement CFSMs in either software or hardware, and since this is a co-design solution — a single CFSM can be partitioned into multiple CFSM sub-networks, and have different target implementations. The hardware CFSM sub-networks are constructed using standard logic synthesis techniques, and in this case a CFSM can execute a transition in a single clock cycle.

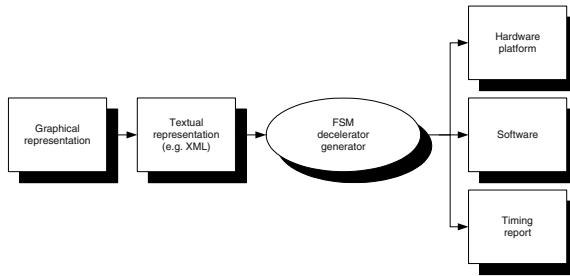
A CFSM sub-network chosen for software implementation is transformed to a program and a simple custom real time operating system (RTOS). The program is generated from a control/data flow graph, and is coded in C. The designer can use a timing estimator to find quickly the speed of the software, or instead produce an instrumented version of the code and run this on an actual processor. The instrumented version counts the actual cycles used, giving a more accurate way of extracting timing information. The custom RTOS consists of a scheduler for organizing the execution of the procedures, using policies such as rate-monotonic or deadline-monotonic scheduling. The RTOS also includes I/O drivers. The case study here is rather different in nature from POLIS, as it has very different aims, including minimizing logic-processor interfacing and code size.

3.1 System Description

The case study involved producing a tool which takes a textual representation of a finite state machine and produces: a hardware platform that can be interfaced to existing logic circuitry; software to run on the embedded processor; and a timing report. The tool flow is shown in Figure 1.

3.2 Design Entry

A number of methods are available for capturing FSMs. Tools such as Esterel Studio or Xilinx's StateCAD allow a designer to capture graphically the FSM as a state transition diagram. Designers annotate the state transition diagram with conditions for taking branches, and define the calculations for the state outputs.

**Fig. 1.** Tool flow

Alternatively, FSMs can be described in a conventional HDL. In order to support the maximum number of possible design methods, an XML grammar was defined to capture the functionality of an FSM. In a file containing a description following this grammar, the interface of the machine is specified first. For example:

```

<variables>
  <variable name="rst" dir="in" width="1" registered="true" />
  <variable name="clk" dir="in" width="1" registered="true" />
  <variable name="phy_ad" dir="in" width="5" registered="true" />
  <variable name="mdio_tristate" dir="out" width="1" registered="true" />
</variables>
  
```

After this, the description specifies the states. An initial tag specifies the global conditions for the state machine, such as reset input, clock input, reset state, and synchronous or asynchronous reset. A description of the individual states follows this. A state has equations and transitions associated with it. Each equation assigns some value to an output. Input, constants and basic operators (add, sub, and, or, etc.) are used to form the right-hand side of the equation. Transitions include the next state and the condition when the transition occurs. Equations can also be associated with transitions. The time when the equation is executed depends on where the equation is located — in the state or in the transition.

```

<state name="stateADD">
  <equations> <equation lhs="out0" rhs="in1 + in2" /> </equations>
  <transitions>
    <transition condition="else" next="state1">
      <equations> <equation lhs="ready" rhs="1" /> </equations>
    </transition>
  </transitions>
</state>
  
```

3.3 Logic-Processor Interfacing

In a conventional SoC design containing processors, the processor is normally connected to the rest of the logic via a system bus or other on-chip network [5].

The processor is a master on the network, initiating and responding to transfers. Since multiple masters may be present in a bus-based system, an arbiter is needed, and masters must first request the bus from it. In the case of an SoC implementation using Virtex-II Pro Platform FPGAs, the logic required to generate the arbiter, the bus itself and the bus interfaces on each of the slaves is implemented in the fabric of the FPGA.

Examining the interface of the PowerPC to the logic fabric in the Virtex-II Pro, there is some flexibility in choosing a method of transferring information between the processor and the logic, and vice-versa. In essence though, for all methods, communication is done over a bus at the processor/logic boundary. Three data buses are natively interfaced to the PowerPC: the On-Chip Memory (OCM) bus, and the Device Control Register (DCR) and Processor Local Bus (PLB) buses from the CoreConnect family of SoC interconnect.

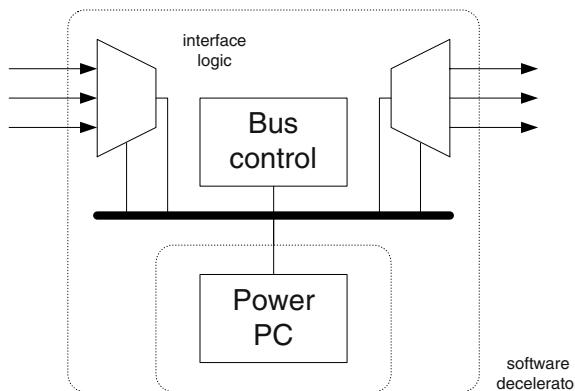


Fig. 2. Software decelerator architecture

The logic design task in the tool development was to take the XML description of the FSM, and produce the interfacing harness as illustrated in Figure 2. The role of this harness is to shield the software decelerator, so in effect the FSM code running on the processor looks like an FSM implemented in logic — that is, the rest of the system should not see any processor specific signals. Moreover, the philosophy is to allow logic to communicate with software using the minimum amount of interfacing.

To interface with the embedded PowerPC processor, it is necessary to use one of the three native buses mentioned above. Simplifications to the interface logic can be made by relying on the fact that only one master (the processor) is present and that it only talks to one slave (the logic circuitry). In each case, the slave can assume it is being addressed all the time, and can simply write outputs and read inputs directly to and from the data bus. For the DCR and the PLB buses, the master interface in the processor assumes the existence of an acknowledge signal, but the logic to do this is very simple.

State machines are normally driven by a clock that dictates when the machine should move between states. For software decelerators in general, there will not be a relationship between the clocks of the rest of the system and the processor. For the state machine and general decelerators, there are two methods for dealing with clocks.

The first method is to use the clock for the state machine directly, as if it were a normal input. Since the processor operates at a much higher frequency than the rest of the system, it is possible for the processor to poll the clock input, and begin processing when it detects a rising edge. The limitation with this method for finite state machines is that the worse case state execution dictates whether the system will meet the timing requirements or not. Figure 3 shows a simplified timing diagram using this method.

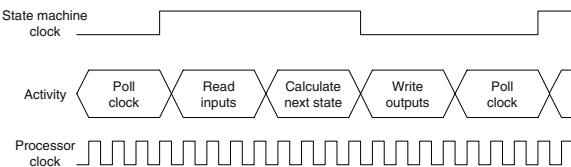


Fig. 3. Simplified timing diagram

A second method is to generate a clock pulse from the processor itself using a memory-mapped one-shot circuit. In this case, states would be allowed to take a different number of processor cycles to complete — the clock pulses would simply appear after a different number of processor cycles, but the external circuits would know when their inputs and outputs have been clocked.

3.4 Timing Generation

The ability to get accurate timing information is crucial to the success of the software decelerator technique. The designer needs to be confident that the overall system, including the decelerator, meets overall timing constraints. To do this, a measure of the delay through the software decelerator is required, in exactly the same way as in hardware design, where the delay through logical elements is required.

Li and Malik present a good discussion of the state of the art of determining the worst-case execution time for software [4]. Similar techniques would be needed for general software decelerator use, where arbitrary code structures are permissible. However, in the case study, the structure of the generated software is strictly under the control of the tool. Therefore, given that the execution times for each instruction type are documented, the tool can count the actual number of processor cycles. Since the software runs out of cache and the time to perform bus transfers to the rest of the system is known, this cycle count is very accu-

rate. This is different to the approach used by POLIS, which generates execution characteristics by instrumenting and running code.

3.5 Software Design

Section 2.2 stated that interfacing should consume minimal resources, to make the software decelerator a value proposition to the designer. Similarly, other support for the processor (e.g. memory and clock control) should consume minimal resources. In the case of the PowerPC, it is possible to reduce external memory requirement to zero by using the instruction and data caches as main memory. This means that the whole executable needs to fit inside the 16Kb instruction cache. In the F SM case under consideration, where no lavish software support is required, extremely complex state machines would still fit inside the 16Kb limit.

It was decided to use assembly code directly for the software implementation. This had two specific advantages. The first advantage is that using assembler simplifies the problem of extracting timing information as described in Section 3.4. The other is that the PowerPC instructions *mfdcr* and *mtdcr*, which move data from and to the DCR bus respectively, can be used if the DCR bus is chosen as the interface. These instructions move data between a specified general-purpose register and the DCR, and thus are difficult to deal with from compiled high-level languages.

Every state uses the same template to create output assembly code. The most complex task is the translation to assembly code of the equations to determine the next state and the state outputs. Inputs are only read if they are used in these equations. The conditions for translations use the same method to calculate the value of the condition. Special care was taken to maximize the usage of registers and only use the cache memory if needed. This can lead to more efficient code — a useful feature since the state machine is already operating at a much lower frequency than the FPGA fabric.

4 Experimental Results

The tool was used on three state machines from the networking domain, with very different performance and I/O requirements. In each case, the clock is supplied by the system's environment. The first state machine (*rs232echo*) was an RS232 protocol handling machine, which echoed received inputs onto outputs, and the second one (*miim*) handled the Media Independent Interface (MII) of an Ethernet MAC which runs at 2.5MHz. The third state machine (*tx_host_io*) handles the host interface to a 10G Ethernet MAC. This machine clearly has a need for high performance, and is included here as an illustration of a case where a software decelerator is not likely to be chosen as the implementation technique. The first table shows the results for a direct logic circuit implementation from synthesized VHDL.

Machine	Input width	Output width	Number of states	Registers	LUTs	Required frequency
rs232echo	3	1	12	92	111	115 kHz
miim	33	20	33	26	61	2.5 MHz
tx_host_io	90	94	5	142	320	156 MHz

To determine the possible real estate savings through using a software decelerator, the new tool was run targeting each of the three available buses. The resource usage and the relative savings in terms of LUTs used compared with the direct logic implementation for each of the buses is shown in the next table.

Machine	OCM			DCR			PLB		
	Registers	LUTs	Ratio	Registers	LUTs	Ratio	Registers	LUTs	Ratio
rs232echo	1	4	3.6%	2	6	5.4%	4	8	7.2%
miim	20	38	62.3%	21	40	65.6%	23	42	68.9%
tx_host_io	94	75	23.4%	95	77	24.1%	97	79	24.7%

The OCM-based implementations are the smallest of the three for each example. Also, the OCM is as fast as the PLB bus for processor/logic interaction — in both cases they take four processor cycles, as opposed to the DCR which takes nine processor cycles. These figures assume the system bus operates at half the frequency of the PowerPC core, that is the PowerPC operates at 350MHz, and the bus clock runs at 175 MHz. In order to determine timing figures, each of these machines was implemented using the new tool, targeting the OCM bus. The final table shows the results for each of the example machines.

Machine	Worst-case performance (cycles)	Worst-case performance (MHz)	% of time in I/O	Code size (kbytes)	Code size as % of cache
rs232echo	40	8.75	30.95%	1416	8.6%
miim	74	4.730	25.22%	2968	18.1%
tx_host_io	135	2.593	33.99%	1952	11.9%

The rs232echo would work at all required baud rates, and the performance of the miim state machine is well inside the required 2.5MHz limit. These examples illustrate a software decelerator delivering the required performance, rather than an unnecessarily high performance. The tx_host_io state machine however, and as expected, does not operate at anything like the required frequency, and is an example of a case where high performance is very much the key focus. In each case, the code only occupies a fraction of the cache. Thus, it would be possible to run multiple state machines, that is multiple software decelerators, on the same processor, as long as any cumulative timing requirements are still be met.

It can be seen that I/O occupies a large proportion of time in each machine, and is clearly a limiting factor in the machine speeds that can be achieved. In the case of state machines, it may be possible to exploit the rich routing resources of the FPGA, and introduce parallel transfers by packing the inputs required for each state into a single word. This would require adjustments to the input

multiplexor shown in Figure 2. The parallelized input signals could then be unpacked inside the processor by shift and masks or alternatively, instructions could be applied directly to the packed signals.

5 Conclusions and Future Work

This research has introduced software decelerators as a mechanism for harnessing the resources of an embedded processor in a Platform FPGA, making the point that maximizing raw processor speed is not likely to be an issue in many logic-centric systems. An application of this philosophy has been demonstrated through the FSM-based design methodology. This experiment shows encouraging initial results in terms of overall resource usage and ease of design. The authors are now evaluating the methodology on larger Xilinx customer designs containing multiple state machines. Future work foci will include: further study of the implications of adopting a logic-centric system model; automatic selection and synthesis of apt logic-processor interfaces; characteristics of soft and hard embedded processors; FSM-based architectural components; and the provision of domain-specific high-level design entry and tools.

References

1. F. Baloron, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, M. Chioldo, H. Hsieh, L. Lavagno, A. L. Sangiovanni-Vincentelli, and K. Suzuki. *Hardware-Software co-design of embedded systems: the POLIS approach*. Kluwer, 1997.
2. Gordon Brebner. Single-chip Gigabit mixed-version IP router on Virtex-II Pro. *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM02)*, pp.35–44, April 2002.
3. M. Dales. The Proteus processor - a conventional CPU with reconfigurable functionality. *9th Int. Workshop on Field Programmable Logic (FPL99)*, pp.431–437, September 1999.
4. Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *ACM/IEEE Design Automation Conference (DAC95)*, pp.456–461, June 1995.
5. M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the system-on-a-chip interconnect woes through communication-based design. *ACM/IEEE Design Automation Conference (DAC01)*, pp.667–672, June 2001.
6. S. Singh and R. Slous. Accelerating Adobe Photoshop with reconfigurable logic. *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM98)*, pp.15–17, April 1998.

A Unified Codesign Run-Time Environment for the UltraSONIC Reconfigurable Computer

Theerayod Wiangtong¹, Peter Y.K. Cheung¹, and Wayne Luk²

¹ Department of Electrical & Electronic Engineering, Imperial College, London, UK,
{tw1, p.cheung}@imperial.ac.uk

² Department of Computing, Imperial College, London, UK,
w1@imperial.ac.uk

Abstract. This paper presents a codesign environment for the Ultra-SONIC reconfigurable computing platform which is designed specifically for real-time video applications. A codesign environment with automatic partitioning and scheduling between a host microprocessor and a number of reconfigurable processors is described. A unified runtime environment for both hardware and software tasks under the control of a task manager is proposed. The practicality of our system is demonstrated with an FFT application.

1 Introduction

Reconfigurable hardware has received increasing attention from the research community in the last decade. FPGA-based designs become popular because of their reconfigurable capability and short design-time which the old design style, like ASICs, cannot offer. Instead of using FPGAs simply as ASICs replacements, combining reconfigurable hardware with conventional microprocessors in a codesign system provides an even more flexible and powerful approach for implementing computation intensive applications, and this type of codesign system is our attention in this paper.

The major concerns in the design process for such codesign system are the synchronization and the integration of the hardware and the software design [1]. Examples are the partitioning between hardware and software, the scheduling of tasks and the communication between hardware and software tasks in order to achieve the shortest overall runtime. Decision on partitioning plays a major role in the overall system performance and cost. Scheduling is important to complete all the tasks in a real-time environment. These decisions are based heavily on design experience and are difficult to automate. Many design tools leave this burden to the designer by providing semi-auto interactive design environments. Fully automatic design approach for codesign system is generally considered to be impossible at present. In addition, traditional implementation employs a hardware model that is very different from that used in software. These distinctive views of hardware and software tasks can cause problem in the design process. For example, swapping tasks between hardware and software can result in a totally new structure in the control circuit.

This paper reports a new method of constructing and handling tasks in a codesign system. We structure both hardware and software tasks in an interchangeable way without sacrificing the benefit of concurrency found in conventional hardware implementations. At the same time, the hardware task model exploits the advantages of modularity, scalability, cohesion and structured approach offered by software tasks. We further present a codesign flow containing the partitioning and scheduling algorithms to automate the decision process of where and when tasks are implemented and run. Our codesign system using this unified hardware/software task model is applied to a reconfigurable computer system known as UltraSONIC [14]. Finally we demonstrate the practicality of our system by implementing an FFT computational engine. The novel contributions of this paper are: 1) a unified way of structuring and modelling hardware and software tasks in a codesign system; 2) proposing a codesign flow for a system with mixed programmable hardware and microprocessor resource; 3) propose a run-time task manager design to exploit the unified model of tasks; 4) the demonstration of implementing a real application by using our model in the UltraSONIC reconfigurable platform.

The rest of this paper is organized as follows. Section 2 presents some of work related to codesign development systems. In section 3, we explain how the hardware and software tasks are modelled in our codesign system. Section 4 describes the codesign flow and environments of the UltraSONIC system used as the realistic target. A case study of implementing the FFT algorithm on the UltraSONIC system is discussed in section 5. Section 6 concludes this paper.

2 Related Work

There are many approaches to hardware/software codesign. Most focus on some particular stages in the design process such as system specification and modelling, partitioning and scheduling, compilation, system co-verification, cosimulation, and code generator for hardware and software interfacing. For example, Ptolemy [2] concentrates on hardware-software co-simulation and system modelling. Chinook [5] focuses on the synthesis of hardware and software interface. MUSIC [6], a multi-language design tools between SDL and MATLAB, is applied to mechatronic system. For embedded systems, CASTLE [7] and COSYMA [8] design environments are developed.

Codesign system specifically targeted for reconfigurable systems are PAM-Blox [4] and DEFACTO [3]. PAM-Blox is a design environment focusing on hardware synthesis to support PCI Pamette board that consists of five XC4000 Xilinx FPGAs. This design framework does not provide a complete codesign process. DEFACTO is an end-to-end design environment for developing applications mapped to configurable platform consists of FPGAs and a general-purpose microprocessor. DEFACTO concentrates on raising the level of abstraction to a higher level, and develop the parallelizing compiler technique to achieve optimizations on loop transformations and memory accesses. Although both PAM-Blox and DEFACTO are developed specifically for reconfigurable platforms, they

take no account of the existence of tasks in microprocessor. Consequentially, neither system is suitable for hardware/software codesign.

In this work, tasks can interchangeably be implemented in software or reconfigurable hardware resources. We suggest a novel way for task modelling and task management which will be described in details in the next section. We also present a novel idea of building infrastructure for dataflow-based applications implementing on this type of codesign system consisting of a single software resource (in the form of a microprocessor) and multiple reconfigurable hardware. Our approach is inherently modular and is suitable for implementing runtime reconfigurable designs.

3 System Architecture

Fig. 1 shows the hardware/software system model adopted in this work. We assume the use of a single processor (software) resource SW capable of multi-tasking, and a number of concurrent hardware processing elements PE0 to PEn, which are implemented on FPGAs. We employ a loosely coupled model with each processing element (PE) having its own single local memory. All system constraints such as shared resource conflicts, reconfiguration times (of the FPGAs) and communication times are all taken into account.

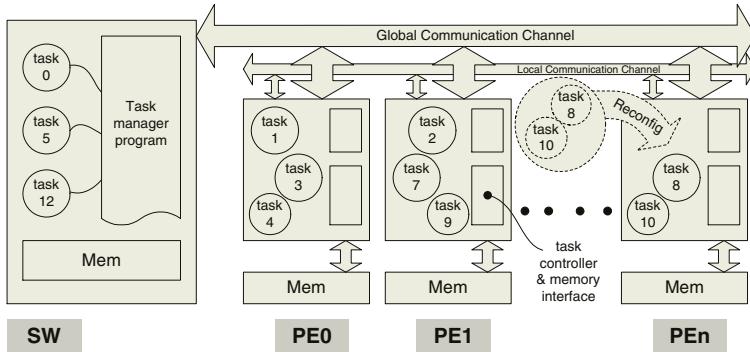


Fig. 1. Codesign System Architecture

The assumptions used in our model are: 1) tasks implemented in each hardware PE are coarse grain tasks which may consist of one or more functional tasks (blocks, loops). 2) Each PE has one local memory, only one task can access the local memory at any given time. Therefore multiple tasks residing in a given PE must execute sequentially; however, tasks residing across different PEs can execute concurrently. 3) Tasks for a PE may be dynamically swapped in and out through dynamic reconfiguration. 4) A global communication channel is available for the processor and the PEs to communicate with each other. 5) Local

communication channels are available for neighboring PEs to communicate with each other in a pipeline ring.

Because of the reconfigurable capability of the hardware, we can build the hardware tasks very much like software tasks. In this way, the management of task scheduling, task swapping and task allocation can be done in a unified manner, no matter whether the task in question is implemented in hardware or in software. Concurrency is not affected as long as we map concurrent tasks onto separate PEs. Although conceptually different PEs are separate from each other, multiple PEs may be implemented on a single FPGA device.

3.1 Hardware Task Model

UltraSONIC is a reconfigurable computer system designed specifically for real-time video processing applications. In such an application domain, it is reasonable to assume that applications are dominated by dataflow behavior with few control flow constructs[12]. Algorithms can be broken down into tasks in coarse (or functional) granularity and are represented as a directed acyclic graph (DAG). Nodes in the graph represent tasks and edges represent data dependency between tasks. Each task is characterized by its execution time, resource occupied, and its communication cost with other tasks.

The tasks we implement on our system are assumed to conform the following restrictions:

- Tasks in the DAG are processed (once for each task) from top to bottom according to their precedence levels and priorities.
- Communication between tasks is always through local single-port memory.
- Task execution is done in three consecutive steps: read input data, process the data, and write the results. This is done repeatedly until input data stored in memory are all processed. Thus the communication time between memory and task while executing is considered to be a part of the task execution time [11].
- Exactly one task in a given PE is active at any one time. This is a direct consequence of the single port memory restriction. However, multiple tasks may run concurrently provided that they are mapped to different PEs. This is an improvement over the model proposed by others in [9].
- A task starts executing as soon as all the necessary incoming data from its sources have arrived. It starts writing outgoing data to destinations immediately after processing is completed [10].

4 The Design Environment

Fig. 2 depicts the codesign environment in our system. The system to be implemented is assumed to be described in some suitable high level language, which is then mapped to a DAG. Tasks are represented by nodes and communications by edges. The nodes are then partitioned into hardware or software tasks, and are

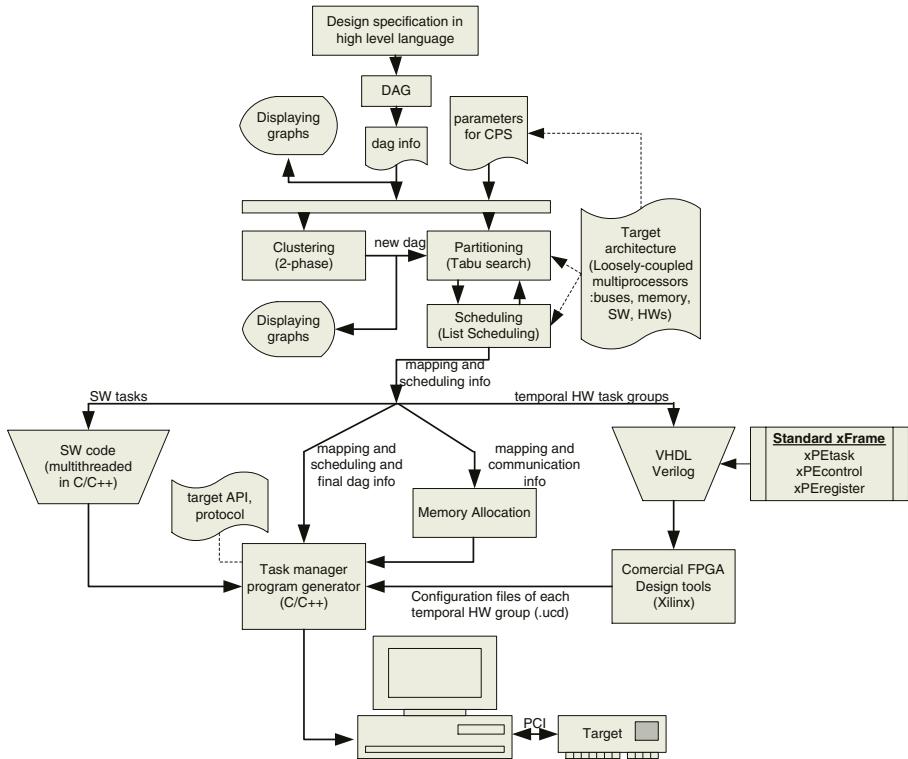


Fig. 2. Codesign environment

scheduled to execute in order to obtain the minimum makespan (total processing time). The partitioning and scheduling software used is adapted from tabu search and list scheduling algorithms reported earlier by the authors [13]. The algorithms are, however, modified to be compatible with this architectural model. A two-phase clustering algorithm is used as a pre-processing step to modify the granularity of the tasks in the DAG in order to improve the critical path delay, enable more task parallelism and provide results the achieve 13%-17% shorter makespan [15].

This group of algorithms, containing clustering, partitioning and scheduling, is collectively called the *CPS* algorithm for short. The *CPS* algorithm reads textual input files including DAG information and control information for clustering, partitioning and scheduling process. During this input stage, the user can optionally specify the type of tasks as *software-only* task, *hardware-only* task, or *dummy* task. A software-only task is a task that the user intentionally implements in software without exception, and similarly for a hardware-only task. Dummy tasks are either source or sink for inputting and outputting data respectively, and are not involved in any computation. In our system, we assume that input and output data are initially provided and written to the microprocessor

memory. Unspecified tasks are then free to be partitioned, scheduled and bound to either hardware or software resources.

The result of the partitioning and scheduling process are the physical and temporal binding for each task. Note that in case of hardware tasks, they may be divided into many temporal groups that can either be statically mapped to the hardware resource, or dynamically configured during runtime.

We currently assume that software tasks are manually written in C/C++, while hardware tasks are designed manually in a HDL (such as Verilog) using a library-based approach. Once all the hardware tasks for a given PE are available, they are wrapped in a standard frame with a pre-designed circuit (xPEtask, xPEregister and xPEcontrol) which is task independent. Commercially available synthesis and place-and-route tools are then used to produce the final configuration files for each hardware. Each task in this implementation method requires some hardware overhead to implement the task frame wrapper circuit. Therefore our system favours partitioning algorithms that generate coarse grain tasks.

The results from the partitioning and scheduling process, the memory allocator, the task control protocol, the API functions, the configuration files of hardware tasks, are used to automatically generate the codes for a *task manager program* that controls all operations in this system, such as dynamic configuration, task execution and data transfer. The resulting task manager is inherently multi-threaded to ensure that tasks are run concurrently.

4.1 The Task Manager Program

The center of our implementation is the task manager (TM) program running on the microprocessor (software) resource to manage both hardware and software tasks. This program controls the sequencing of all the tasks, the transfer of data and the synchronization between them, and the dynamic reconfiguration of the FPGA in the PEs when required. Fig. 3 shows the conceptual control view of the TM and its operation. The TM communicates with a local task controller on each PE in order to assert control. A message board is used in each PE to receive commands from the TM or to flag finishing status to the TM.

In order to properly synchronize the execution of the tasks and the communication between tasks, our task manager employs a message-passing, event-triggered protocol when running a process. However, unlike a reactive codesign system [16], we do not use external real-time events as triggers. Instead, we use the termination of each task or data transfer as event-triggers, and signaling of such events is done through dedicated registers. For example, in Fig. 3, messages indicating execution completion from tasks 1 are posted to registers inside PE0. The task manager program polls these registers, finds the message, then proceeds to the next scheduled task, in this case task 3. By using this method, tasks on each PE is run independently because the program operates asynchronously at the system level.

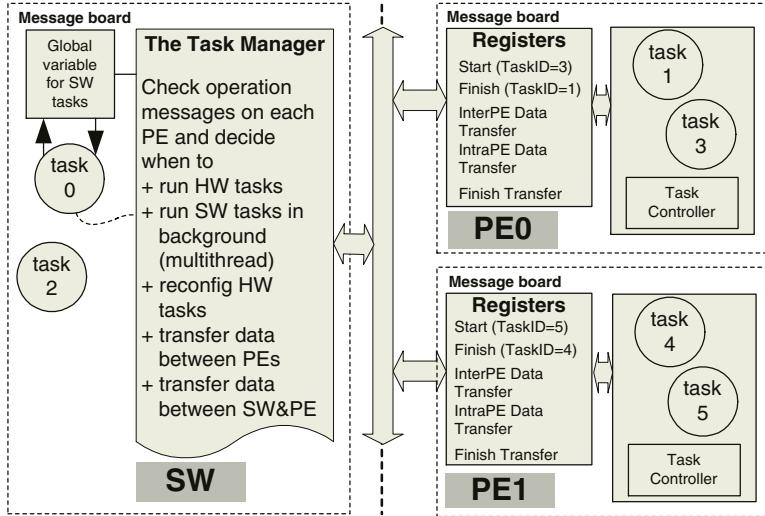


Fig. 3. The Task Manager control view

4.2 The UltraSONIC Reconfigurable Platform

The codesign described above is targetted for the UltraSONIC System [14]. UltraSONIC (see Fig. 4(a)) is a reconfigurable computing system designed to cope with the computational power and the high data throughput demanded by real-time video applications. The system consists of Plug-In Processing Elements (PIPEs) interconnected by local and global buses. The architecture exploits the spatial and the temporal parallelism in video processing algorithms. It also facilitates design reuse and supports the software plug-in methodology.

Fig. 4(b) shows how our codesign model is implemented in the UltraSONIC PIPE. The xPEcontrol implements the message passing protocol to control the operation of all the hardware tasks (xPEtask) resident in this PIPE. The message board is implemented in xPEregister. The total hardware overhead of using the Task Manager is modest. It consumes around 10% of the reconfigurable resource on each PIPE (which is implemented on Xilinx's XCV1000E).

5 An Example: FFT Implementation

In order to demonstrate the working of our system, we chose to implement the well known FFT algorithm. Although we can implement the algorithm for an arbitrary data length, we use an 8-point FFT implementation to illustrate the results of our codesign system. The DAG of an 8-point FFT can be straightforwardly extracted as shown in Fig. 5(a). Nodes 0, 1, 2 are used for arranging inputs data and are implemented as software-only tasks. Tasks 3 to 14 are butterfly computation nodes. The number shown inside the parenthesis (the second

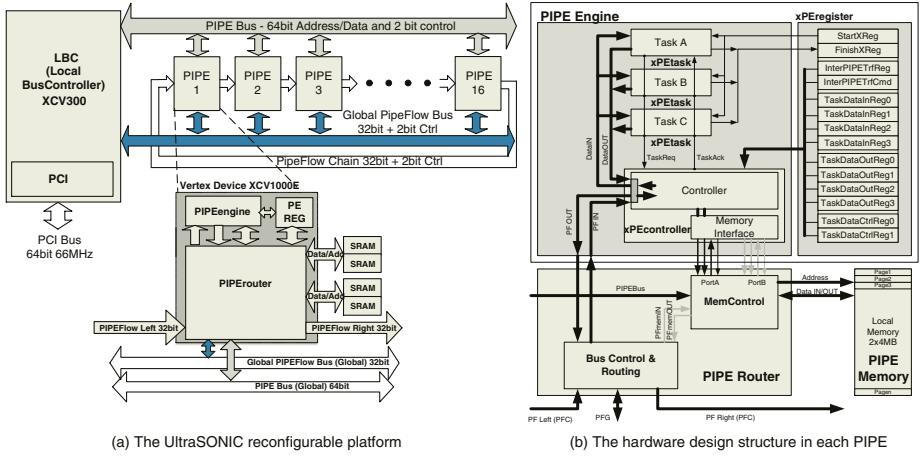


Fig. 4. The hardware implementation in UltraSONIC architecture

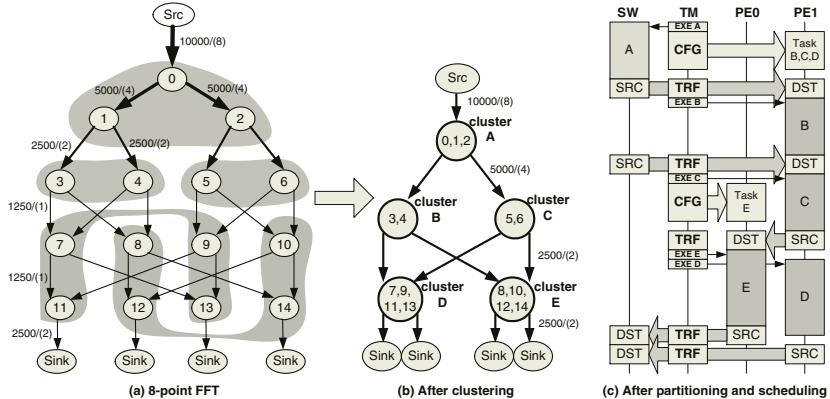


Fig. 5. The DAG of 8-point FFT algorithm

number) on each edge is the number of data values needed for each iteration of the task. Each task is executed repeatedly until all data (shown as the first number on the edge) are processed. Initially in this DAG, the software execution times are obtained by profiling tasks on PC, while the hardware times and areas are obtained by using Xilinx development tools.

This DAG information is supplied to the CPS algorithm in our design environment (see Fig. 2) to perform automatic clustering, partitioning and scheduling. Parameters such as reconfiguration time, bus speed, FPGA size, are all based on the UltraSONIC system. The clustering algorithm produces a new DAG that contains tasks in higher granularity as shown in Fig. 5(b). These new tasks are then iteratively partitioned and scheduled. The computational part of

each of the hardware tasks are designed manually in Verilog and the software tasks are written in C. Our tools then combine the results from the partitioning and scheduling algorithms, wrap the hardware task designs automatically with the standard task frame, and generate the task manager program.

Fig. 5(c) depicts the run-time profile of this implementation. Each column represents activities on the available resources which are software tasks (SW), and two hardware processing elements (PE0 and PE1). TM is the task manager program which is also running on the software resource. The execution of this algorithm proceeds from top to bottom. It shows all the runtime activities including configuration (CFG), transferring data (TRF), events that trigger task executions (EXE), the source of data (SRC), receiving data (DST) and the executions of the tasks (A to E).

The FFT algorithm for different data window sizes are also tested on the UltraSONIC system and are shown to work correctly. The method, although requires some manual design steps, is very quick. Implementing the FFT algorithm only took a few hours from specification to completion.

6 Conclusions

This paper presents a semi-automatic codesign environment for a system consisting of single software and multiple reconfigurable hardware. It proposes the use of a task manager to combine the runtime support for hardware and software in order to improve modularity and scalability of the design. Partitioning and scheduling are done automatically. Codes for software tasks are run in software concurrently (using multi-threaded programming) with the task manager program which is based on message-passing and event-triggered protocol. Implementation of the FFT algorithm on UltraSONIC demonstrates the practicality of our approach.

Future work includes testing our codesign system with more complex applications, tools to map behavioral or structural descriptions to DAG automatically and to improve the task management environment so that external asynchronous real-time events can also be handled.

Acknowledgement

The authors would like to acknowledge the continuing support of John Stone, Simon Haynes, Henry Epsom and the rest of the UltraSONIC team at Sony Broadcast Professional Research Laboratory in UK.

References

1. Ernst, R., "Codesign of embedded systems: status and trends", *IEEE Design & Test of Computers*, 1998.
2. Manikutty, G.; Hanson, H., "Hardware/Software Partitioning of Synchronous Dataflow Graphs in the ACS domain of Ptolemy", University of Texas, Literature Survey, Final Report May 12 1999.

3. Hall, M.; Diniz, P.; Bondalapati, K.; Ziegler, H.; et al., "DEFACTO:A Design Environment for Adaptive Computing Technology", *Proceedings of the 6th Reconfigurable Architectures Workshop*, 1999.
4. Mencer, O.; Morf, M.; Flynn, M.J., "PAM-Blox: high performance FPGA design for adaptive computing", *FPGAs for Custom Computing Machines*, 1998.
5. Chou, P.H.; Ortega, R.B.; Borriello, G., "The Chinook hardware/software co-synthesis system", *System Synthesis*, 1995.
6. Coste, P.; Hessel, F.; Le Marrec, P.; Sugar, Z.; et al., "Multilanguage design of heterogeneous systems", *Hardware/Software Codesign*, 1999.
7. Wilberg, J.; Kuth, A.; Camposano, R.; Rosenstiel, W.; et al., "Design Exploration in CASTLE", *Workshop on High Level Synthesis Algorithms Tools and Design (HILES)*, 1995.
8. Ernst, R., "Hardware/Software Co-Design of Embedded Systems", *Asia Pacific Conference on Computer Hardware Description Languages*, 1997.
9. Srinivasan, V.; Govindarajan, S.; Vemuri, R., "Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 140 -158, 2001.
10. Hou, J.; Wolf, W., "Process partitioning for distributed embedded systems", *Hardware/Software Co-Design*, 1996.
11. Pop, T.; Eles, P.; Peng, Z., "Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems", *Hardware/Software Codesign*, 2002.
12. Chattha, K.S.; Vemuri, R., "Hardware-software partitioning and pipelined scheduling of transformative applications", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, pp. 193-208, 2002.
13. Wiangtong, T.; Cheung, P.Y.K.; Luk, W., "Comparing Three Heuristic Search Methods for Functional Partitioning in HW-SW Codesign", *International Journal on Design Automation for Embedded Systems*, vol. 6, pp. 425-449, July 2002.
14. Haynes, S.D.; others, a., "UltraSONIC: A Reconfigurable Architecture for Video Image Processing", *Field-Programmable Logic and Applications (FPL)*, 2002.
15. Wiangtong, T.; Cheung, P.Y.K.; Luk, W., "Cluster-Driven Hardware/Software Partitioning and Scheduling Approach For a Reconfigurable Computer System", submit to *Field-Programmable Logic and Applications (FPL)*, 2003.
16. De Micheli, G., "Computer-aided hardware-software codesign", *IEEE Micro*, Vol 14, pp. 10-16, 1994.

Extra-dimensional Island-Style FPGAs

Herman Schmit

Dept. of Electrical and Computer Engineering, Carnegie Mellon University,
Pittsburgh, PA, 15213 USA,
herman@ece.cmu.edu

Abstract. This paper proposes modifications to standard island-style FPGAs that provide interconnect capable of scaling at the same rate as typical netlists, unlike traditionally tiled FPGAs. The proposal uses a logical third and fourth dimensions to create increasing wire density for increasing logic capacity. The additional dimensions are mapped to standard two-dimensional silicon. This innovation will increase the longevity of a given cell architecture, and reduce the cost of hardware, CAD tool and Intellectual Property (IP) redesign. In addition, extra-dimensional FPGA architectures provide a conceptual unification of standard FPGAs and time-multiplexed FPGAs.

1 Introduction

Island-style FPGAs consist of mesh interconnection of logic blocks, connection blocks and switchboxes. Commercial FPGAs from vendors such as Xilinx are implemented in an island-style. One advantage of island-style FPGAs is that they are completely tileable. A single optimized hardware tile design can be used in multiple products all with different capacity, pin count, package, etc., allowing FPGA vendors to build a whole product line out of a single relatively small design. The tile also simplifies the CAD tool development effort. The phases of technology mapping, placement, routing and configuration generation are all simplified by the regularity and homogeneity of the island-style FPGA. Finally, these tiled architectures are ammenable to a re-use based design methodology. A design created for embedding in larger designs, often called a core, can be placed to any location within the array, while preserving routing and timing characteristics.

The inherent problem with tiled architectures is that they do not scale to provide the interconnect typical of digital circuits. The relationship of design interconnect and design size is called Rent's rule. This relationship is described in [12], where it is shown that a partition containing n elements has Cn^p pins either entering or leaving the partition. The quantity p , which is known as the Rent exponent, has been empirically determined to be in the range from 0.57 to 0.75 in a variety of logic designs. In a tiled architecture, the number of logic elements in any square bounding box is proportional to the number of tiles contained in the square, and the number of wires crossing the square is proportional to the perimeter of the square. Therefore, in a traditional tiled architecture, the interconnect is a function of the square root of the logic capacity, and the “supplied”

Rent exponent is 1/2. Supply therefore can never keep up with interconnect demand, and at some point a tiled architecture will fail to have enough interconnect.

Commercial vendors have recognized this phenomenon. Succeeding generations of FPGAs always have more interconnect per logic block. This has the effect of keeping up with the Rent exponent by increasing the constant interconnect associated with each tile, i.e. increasing the C term in Rent's formula. There are several problems with this solution. First, it requires that the tile must be periodically redesigned, and all the CAD tools developed for that tile must be updated, tested and optimized. Second, all the cores designed for an early architecture cannot be trivially mapped to the new architecture.

What would be preferable would be to somehow scale the amount of interconnect between cells, while still providing the benefits of a tiled architecture, such as reduced hardware redesign cost and CAD tool development. This is impossible in conventional island-style FPGAs because they are two-dimensional, which means that the Rent exponent of supplied interconnect is fixed at one half. This paper proposes a new island-style architecture, based on three- and four-dimensional tiling of blocks, that can support Rent exponents greater than 0.5 across an entire family of FPGAs.

Three-dimensional FPGAs have been proposed frequently in the literature [1, 2, 8, 13, 14], but commercial three-dimensional fabrication techniques, where the transistors are present in multiple planes, do not exist. An optoelectrical coupling for three-dimensional FPGAs has also been proposed in [6], but there are no instances of such couplings in large commercial designs as yet. Four-dimensional FPGAs face even greater challenges to actual implementation, at least in this universe. Therefore, this paper proposes ways to implement three- and four-dimensional FPGAs in planar silicon technology. Surprisingly, these two-dimensional interconnect structures resemble double and quad interconnect lines provided in commercial island-style FPGAs, albeit with different scaling behavior.

2 Architecture

Extra-dimensional FPGAs can provide interconnect that matches the interconnect required by commercial designs and that scales with design size. This section describes how these extra dimensions provide scalable interconnect. Possible architectures for multi-dimensional FPGAs are discussed in the context of implementation on two-dimensional silicon. Three- and four-dimensional FPGA architectures are proposed, which will be the subject of placement and routing experiments performed in subsequent sections.

As discussed previously, a 2-D tiled FPGA provides a Rent exponent of 1/2 because of the relationship of the area and perimeter of a square. In a three-dimensional FPGA, assuming that all dimensions grow at an equal rate, the relationship between volume and surface area is governed by an exponent of 2/3. Therefore in a perfect three-dimensional FPGA, the supplied interconnect has a

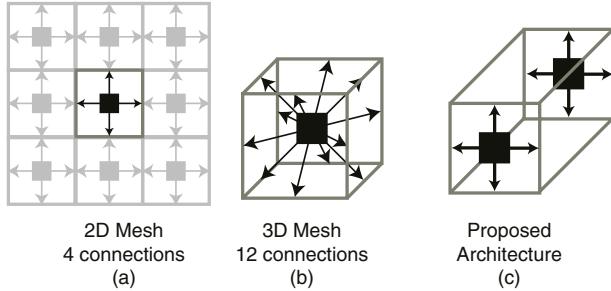


Fig. 1. Two and three dimensional interconnect points: as shown in (a) a CLB in a 2D mesh must connect to 4 wires. A CLB in a 3D mesh must (b) connect to twelve points. The proposed architecture has 2D CLBs interconnected to the squares in a 3D mesh.

Rent exponent of $2/3$. By extension, the Rent exponent for a four-dimensional architecture is $3/4$.

There are many ways to construct an extra-dimensional FPGA. A conventional island-style FPGA can be envisioned as an array of CLBs, each of which is surrounded by a square of interconnect resources. The CLB connects to each of the edges of the square that surrounds it, as shown in Figure 1(a). By extension, a 3-D FPGA would have a three dimensional interconnection mesh, with CLBs located in the center of the cubes that make up this mesh. Each CLB would have to connect to all twelve edges of the cube, as illustrated in Figure 1(b). This entails a three-fold increase in the number of places that CLB IOs connect, which either means greater delay, greater area, and perhaps worse placement and routing. In addition, this “cube” interconnect is very difficult to lay out in two-dimensional silicon. In a four-dimensional FPGA, the CLB would have to connect to thirty-two of the wires interconnecting a hypercube in the mesh, and is even harder to lay out in two dimensions.

An alternative proposal is to allow CLBs to exist only on planes formed by the x and y dimensions. These CLBs only connect to wires on the same xy plane. A multi-dimensional FPGA is logically constructed by interconnecting planes of two-dimensional FPGAs, as illustrated in Figure 1(c). This keeps the the number of IOs per CLB equal to conventional FPGAs, while still providing the benefits of an extra-dimensional FPGA.

In our proposed architecture, two adjacent xy planes of CLBs are interconnected by the corresponding switch boxes. For example, in a logically three-dimensional FPGA, the switch box at (x, y) in plane z is connected to the switch box at (x, y) in plane $z + 1$ and plane $z - 1$. Such a three-dimensional switchbox architecture was used in [1].

A problem with these extra-dimensional switch boxes is the increased number of transistors necessary to interconnect any wire coming from three or four dimensions. When four wires meet within a switch box, coming from all four directions in a two-dimensional plane, six transistors are necessary to provide any interconnection. With three dimensions, six wires come from every direction,

requiring 15 transistors to provide the same flexibility of interconnect. With four dimensions, 28 transistors are necessary at each of these switch points. We will assume that extra-dimensional switch boxes include these extra number of transistors. We will later show that this overhead is mitigated by the reduction in channel width due to extra-dimensional interconnect. It is worth noting that there are only 58 possible configurations of switch point in a 3D switch box,¹ and 248 possible configurations of a 4D switch point. Therefore the amount of configuration for these switch points could potentially be significantly reduced to six or eight bits. It is also possible that all 15 or 28 transistors are not necessary to provide adequate interconnect flexibility. Future work will explore optimizations of the switch box architectures for extra-dimensional FPGAs.

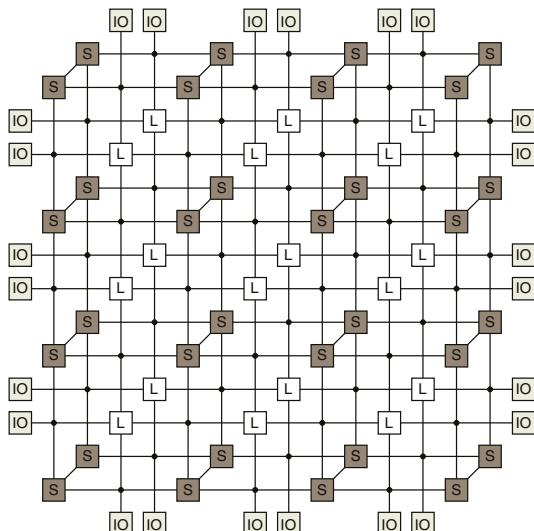


Fig. 2. Three dimensional FPGA in two dimensions: This particular FPGA is a 3×3 $\times 2$ array of CLBs. There are IOs around the periphery. Switch Blocks are labelled “S” and CLBs are labelled “L”. Connections between xy planes take place in the diagonal channels between switch boxes. All lines in this figure are channels containing multiple wires.

Figure 2 and Figure 3 show feasible layouts for three- and four-dimensional FPGAs in two-dimensional silicon. The four-dimensional layout is particularly interesting, because it allows for close interconnections between neighboring CLBs, and because it is symmetric in both x and y dimensions. The most interesting aspect of the four-dimensional FPGA is how the interconnect resembles the interconnect structure of commercial island-style FPGAs. Particularly, the interconnections between different xy planes resemble the double and quad lines

¹ This is determined by $1 + \binom{6}{2} + \binom{6}{3} + \binom{6}{4} + \binom{6}{5} + \binom{6}{6} = 58$

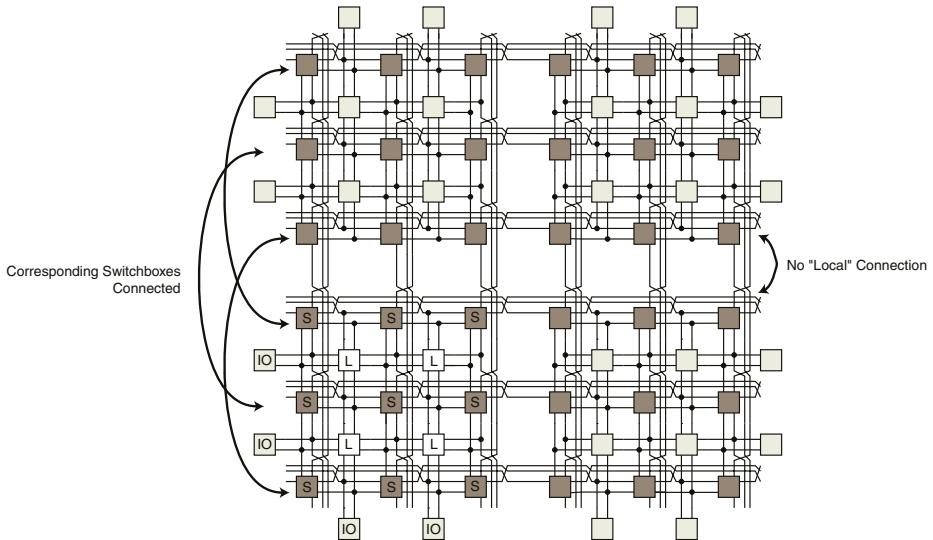


Fig. 3. Four dimensional FPGA: This FPGA has dimensions of $2 \times 2 \times 2 \times 2$ CLBs. The connections between dimensions take place on the long lines that go from one switch box to the corresponding box in another dimension.

in Xilinx 4000 and Virtex FPGAs. These lines skip across a number of CLBs, in order to provide faster interconnect over long distance.

The primary difference between the traditional island-style with double and quad lines and the proposed 4D FPGA is that the length of the long wires increases when the size of the xy plane increases. This is the key to providing scalable interconnect. The length of wires in the two-dimensional implementation of the 4D FPGA increases as the logical size of the device increases. This does mean that the delay across those long wires is a function of the device size, which presents an added complication to timing-oriented design tools. Fortunately, interconnection delay in commercial FPGAs is still dominated by the number of switches in any path and not the length of the net.²

The assumption that the physical channel width is proportional to the number of wires passing through that channel leads to the conclusion that the four-dimensional FPGA in Figure 3 would be much larger than the two-dimensional FPGA with the same number of CLBs. In real FPGA layouts however, the largest contributor to channel width is the number of programmable interconnect points in the channel, and the width of the switch box. Because these points require one to six transistors, they are much larger than the minimum metal pitch. Most wires in the channel of the four-dimensional FPGA do not contact any interconnect points however. These wires can be tightly routed at a higher level of metal, and may only minimally contribute to total channel width.

² The increasing “logical length” of these lines may also be counter-acted to some extent by the scaling of technology.

A second substantial difference between the proposed 4D FPGA and the traditional island-style FPGA is that in the commercial FPGA, the local connections are never interrupted. In the 4D FPGA, there is no local interconnection from the boundary of one xy plane to the corresponding boundary of the adjacent xy plane. The reason for this is to provide for tiling of IP blocks. Suppose we want to use a four-dimensional core in a four-dimensional FPGA. To guarantee that this core will fit in this FPGA, all four dimensions of the core must be smaller than the four dimensions of the FPGA. If there are local connections between xy planes, that cannot be guaranteed. A core that was built on a device with $x = 3$, and which used local connections between two xy planes would only fit onto other devices with $x = 3$.

The next section will describe an experiment that compares the scaling of interconnect in two- and four-dimensional FPGA. First, Rent's rule is demonstrated by measuring how the channel width of a two-dimensional FPGA increases as the size of the design increases. When the same suite of netlists is placed and routed on an 4D FPGA, the channel width remains nearly constant.

3 Experimental Evaluation

In order to demonstrate the effectiveness of extra-dimensional FPGAs, a large set of netlists have been placed and routed on both two- and four-dimensional FPGAs. The minimum channel width required in order to route these designs is compared. In the two-dimensional FPGA, the channel width grows with respect to the logical size of the design. The four-dimensional FPGA scales to provide exactly the required amount of interconnect, and channel width remains nearly constant regardless of the design size.

A significant obstacle to performing this experiment was to acquire a sufficiently large set of netlists in order to make conclusions based on measured data. Publicly available circuit benchmark suites are small, both in the number of gates in the designs, and the number of designs in the suites. In this paper we use synthetic netlists generated by the CIRC and GEN tools [10, 11]. CIRC measures graph characteristics, and has been run on large sets of commercial circuits. GEN constructs random netlists that have the same characteristics as the measured designs. We used the characteristics measured by CIRC on finite state machines to generate a suite of 820 netlists ranging from 20 to 585 CLBs.

The placement and routing software we will use is an extended version of the VPR tool [3]. The modifications to this software were relatively simple. The placement algorithm is extended to use a multi-dimensional placement cost. The cost function is a straight-forward four-dimensional extension of the two-dimensional cost, which is based on the RISA cost function [4]. The cost of routing in the two additional dimensions is scaled by a small factor to encourage nets to be routed in a single xy plane.

The logic block architecture used for this experiment is the standard architecture used in VPR. This architecture consist of a four-input lookup table (LUT) and a bypassable register. The switch box architecture is the Xilinx-type (also

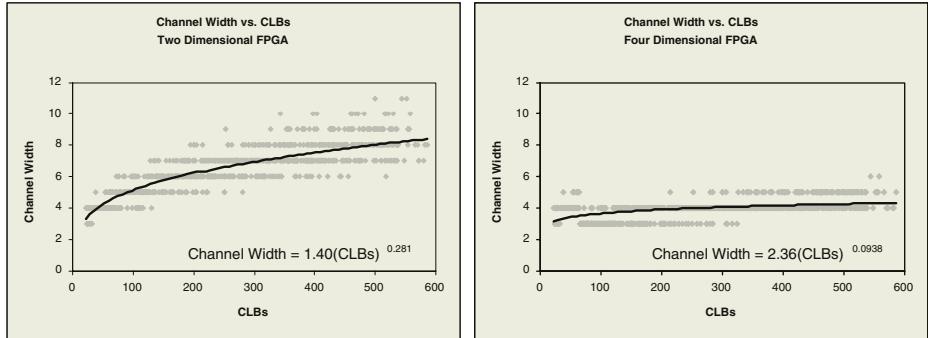


Fig. 4. Two- and Four-Dimensional FPGA Channel Width

known as subset or disjoint), where the n -th channel from one direction connects to the n -th channel in every other direction.

VPR works by first placing the design, and then attempting to find the minimal channel width that will allow routing of the design. It performs a binary search in order to find this minimal channel width. The results of routing the netlist suite on the two-dimensional and four-dimensional architecture are shown in Figure 4, which plots the maximum channel width versus the netlist size in CLBs. The best-fit power graph has been fit to this data, showing an exponent of 0.28 for the two-dimensional FPGA. As shown in [7] and [9], this indicates the Rent exponent of these designs is approximately $0.28 + 0.5 = 0.78$, which is large, but possibly reasonable for the design of an FPGA architecture.

Even a four dimensional FPGA cannot provide interconnect with a Rent exponent of 0.78. In our experiment, the extra dimensions of the FPGA scale at one-eighth the rate of the x and y dimensions. If the x and y dimensions are less than eight, there is no other extra dimensions to the FPGA (the FPGA is simply two-dimensional). With x or y in the range of 8 to 16, there are two planes of CLBs in each of the extra dimensions.

The channel width for the four-dimensional FPGA, as shown in Figure 4 is comparatively flat across the suite of netlists. Figure 5 demonstrates the effectiveness of the technique by showing total wire length, in channel segments, for each design. When fit to a power curve, the two dimensional FPGA exhibits a much larger growth rate than the four dimensional FPGA. Superlinear growth of wire length was predicted by [7] as a result of Rent's rule. The four dimensional FPGA evidences a nearly linear relationship between wire length and CLBs, indicating that the supply of interconnect is keeping pace with the demand.

As mentioned in previously, the four dimensional switch box might have more than four times more bits per channel compared to the two dimensional switch box. Table 1 shows the number of configuration bits necessary to program the interconnect for a two- and four-dimensional FPGA with 576 CLBs. Using our data from the previous experiments, we have assumed a channel width of twelve for the two-dimensional FPGA, and six for the four-dimensional FPGA. Con-

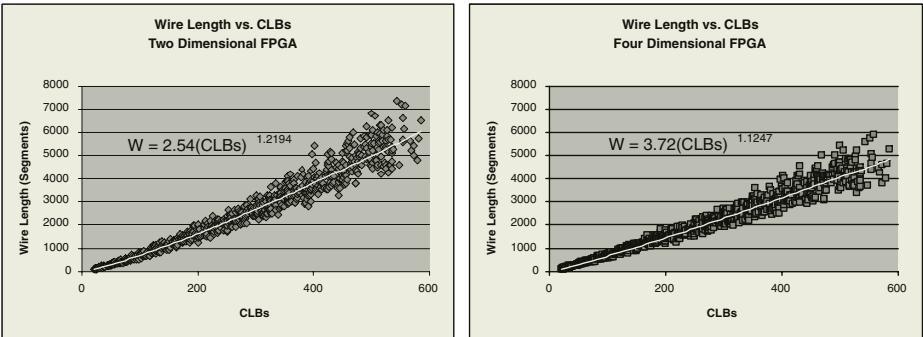


Fig. 5. Wirelengths for Two- and Four-dimensional FPGAs

Table 1. Computation of configuration bits required for large two- and four-dimensional FPGAs. The larger number of bits per switch point the four-dimensional FPGA is countered by the reduced channel width. If the switch point configuration is encoded with sixteen bits, there is no configuration overhead.

FPGA Dimensions	Channel Width	Switch Point Bits	Total Switchbox Bits	Total Channel Bits	Total Config Bits
24 x 24	12	6	41,472	27,648	69,120
2 x 12 x 12 x 2	6	28	96,768	13,824	110,592
2 x 12 x 12 x 2	6	16	55,296	13,824	69,120

sidering both switch box and connection box bits, the four-dimensional FPGA requires 60% more configuration bits. If we can control the four-dimensional switch point with just sixteen bits, then the reduction in channel width completely compensates for the more complex switch point.

Four-dimensional FPGAs provide scalable interconnect across a wide suite of netlists. Required channel width can remain constant across a large family of devices, allowing the hardware and CAD tool development efforts to be amortized over many devices, and reducing the waste of silicon in smaller devices.

4 Time Multiplexing and Forward-Compatibility

Since their proposal [5, 15, 17], time-multiplexed FPGAs have been thought of as fundamentally different creatures from standard FPGAs. The Xilinx and Sanders FPGAs operate by separating every logical cycle into multiple micro-cycles. Results from a micro-cycle are passed to subsequent micro-cycles through registers.

Time-multiplexed FPGAs could also be constructed using a three-dimensional FPGA architecture like that shown in Figure 2. Micro-registers would be inserted on the inter-plane connections that exist between switchboxes. The number of registers between planes would correspond to the channel width, which allows it to be designed to provide scalable interconnect.

By viewing time as a third dimension, the scheduling task can be accomplished within the scope of a three dimensional placement algorithm. All combinatorial paths must go from a previous configuration, or plane, to a future one, and not vice-versa. Therefore real logical registers must be placed in a configuration that is evaluated after its entire fanin cone. This is conceptually much simpler than the separated scheduling and place-and-route phases implemented in [16].

5 Conclusions

This paper discussed the benefits of three- and four-dimensional FPGAs, and their implementation in conventional two-dimensional silicon. Primary among the benefits is the ability to provide interconnect that scales at the same rate as typical netlists. Four dimensional FPGAs resemble the double and quad lines in commercial FPGAs, although the key to providing scalable interconnect is to increase the length of those lines as the device grows.

References

1. M. J. Alexander, J. P. Cohoon, J. L. Colflesh, J. Karro, and G. Robins. Three-dimensional field-programmable gate arrays. In *Proceedings of the IEEE International ASIC Conference*, pages 253–256, September 1995.
2. M. J. Alexander, J. P. Cohoon, J. Karro, J. L. Colflesh, E. L. Peters, and G. Robins. Placement and routing for three-dimensional FPGAs. In *Fourth Canadian Workshop on Field-Programmable Devices*, pages 11–18, Toronto, Canada, May 1996.
3. V. Betz and J. Rose. Effect of the prefabricated routing track distribution on FPGA area efficiency. *IEEE Transactions on VLSI Systems*, 6(3):445–456, September 1998.
4. C. E. Cheng. RISA: Accurate and efficient placement routability modeling. In *Proceedings of IEEE/ACM International Conference on CAD (ICCAD)*, pages 690–695, November 1996.
5. A. DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, Napa, CA, April 1994.
6. J. Depreitere, H. Neefs, H. Van Marck, J. Van Campenhout, R. Baets, B. Dhoedt, H. Thienpont, and I. Veretennicoff. An optoelectronic 3-D field programmable gate array. In R. Hartenstein and M. Z. Servit, editors, *Field-Programmable Logic: Architectures, Synthesis and Applications. 4th International Workshop on Field-Programmable Logic and Applications*, pages 352–360, Prague, Czech Republic, September 1994. Springer-Verlag.
7. W. E. Donath. Placement and average interconnection lengths of computer logic. *IEEE Transactions on Circuits and Systems*, pages 272–277, April 1979.
8. Hongbing Fan, Jiping Liu, and Yu-Liang Wu. General models for optimum arbitrary-dimension fpga switch box designs. In *Proceedings of IEEE/ACM International Conference on CAD (ICCAD)*, pages 93–98, November 2000.
9. A. El Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Transactions on Circuits and Systems*, 28(2):127–138, February 1981.

10. M. Hutton, J.P. Grossman, J. Rose, and D. Corneil. Characterization and parameterized random generation of digital circuits. In *Proceedings of the 33rd ACM/SIGDA Design Automation Conference (DAC)*, pages 94–99, Las Vegas, NV, June 1996.
11. M. Hutton, J. Rose, and D. Corneil. Generation of synthetic sequential benchmark circuits. In *5th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, (FPGA 97)*, February 1997.
12. B. S. Landman and R. L. Russo. On pin versus block relationship for partitions of logic circuits. *IEEE Transactions on Computers*, C-20:1469–1479, 1971.
13. M. Leeser, W. M. Meleis, M. M. Vai, W. Xu S. Chiricescu, and P. M. Zavracky. Rothko: A three-dimensional FPGA. *IEEE Design and Test of Computers*, 15(1):16–23, January 1998.
14. W. M. Meleis, M. Leeser, P. Zavracky, and M. M. Vai. Architectural design of a three dimensional FPGA. In *Proceedings of the 17th Conference on Advanced Research in VLSI (ARVLSI)*, pages 256–268, September 1997.
15. S. Scalera and J. R. Vazquez. The design and implementation of a context switching FPGA. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 78–85, Napa, CA, April 1998.
16. S. Trimberger. Scheduling designs into a time-multiplexed FPGA. In *6th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, (FPGA 98)*, pages 153–160, February 1998.
17. S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A time-multiplexed FPGA. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 22–28, Napa, CA, April 1997.

Using Multiplexers for Control and Data in D-Fabrix

Tony Stansfield

Elixent Limited, Castlemead, Lower Castle Street, Bristol, United Kingdom
tony.stansfield@elixent.com

Abstract. This paper describes the use of dynamically controlled multiplexers in the Elixent D-Fabrix Reconfigurable Algorithm Processor (RAP) for both datapath functions and to implement simple logic functions for control circuits.

1 Introduction

Reconfigurable computing applications can be viewed as consisting of a mixture of control and datapath blocks. These two types of blocks have different characteristics, in particular the datapath blocks process word-based data and the control blocks manipulate bits. Applications vary in the relative amounts of control and datapath that they contain. At one extreme are DSP functions such as the discrete cosine transform (DCT) that can be expressed as almost 100% datapath, and at the other are control intensive tasks such as searching, sorting, and boolean satisfiability.

The majority of implementation approaches for reconfigurable computing use the same hardware for both control and datapath. Typically, FPGAs provide the hardware, and their bit-based lookup tables (LUTs) are used to construct both the bit-oriented control logic and the word-based datapaths. The use of LUTs to construct a datapath is an inefficient use of the underlying silicon resources, as it fails to exploit the repetitive nature of the datapath – all bits in the word are processed in the same way and could therefore share the same control state.

An alternative implementation approach is to use a Reconfigurable Computing device (such as Elixents D-Fabrix Reconfigurable Algorithm Processor) that consists of an array of word-based processing elements – in the case of D-Fabrix all the processing units are designed to handle 4-bit quantities and can be easily combined for larger words. Here there is the opposite problem to the FPGA case – the basic units are designed for the implementation of datapaths and lose efficiency when used to process 1-bit control signals – and if this efficiency loss becomes too great then the overall solution would be no better than the FPGA-based approach.

This paper describes the dedicated multiplexers in D-Fabrix, and how they can be used to assist with the implementation of control structures. The next section describes the kinds of control structures that may be encountered in applications and ways of implementing them as gates. Later sections then describe the D-Fabrix array, and ways to implement these structures on it. This is followed by an assessment of the ability of software to efficiently target the available hardware resources, and a comparison of the current work with previous work on heterogeneous architectures.

2 Types of Control Structures

Control can be divided into three parts:

- Control of the flow of data, as expressed in the C/Java conditional assignment operator: $a = b ? c : d;$
- Control of the flow of program control, as typified by programming language constructs such as `if(expr)` and `while(expr)`, and:
- Calculation of the expressions with Boolean results that are inputs to these control circuits (the `b` and `expr` in the above examples)

The control of the flow of data can be implemented with multiplexers, choosing between 2 (or more) word-wide data inputs.

Page and Luk [1] have described circuits that can be used to implement the flow of program control. Every program statement that affects control flow results in the creation of a simple sequential circuit that waits for a “start” token from a preceding statement and passes the token to an appropriate successor. For instance, in the case of an `if...then...else...` statement the token is passed to one branch or the other depending on the result of evaluating the condition. The overall circuit created by combining the individual circuits from the individual statements is effectively a distributed state machine with a one-hot encoding. The token is represented as a single bit, propagated from stage to stage on a single wire. The logic gates that control the path of the token are all either AND gates, OR gates or inverters, and (with the exception of the wide OR gates used to merge tokens after branches and PAR statements) are all 1 or 2 input gates.

This distributed state machine is one source of the control inputs of the data multiplexers. The other possible source is an expression that produces a single bit result – such as a comparison ($A == B$, $A > B$ etc.).

3 D-Fabrix Architecture

The D-Fabrix architecture is an evolution of the Chess architecture [2] developed by Hewlett-Packard Laboratories. It shares with Chess the following features:

- The basic processing elements are 4-bit ALUs, each with:
 - Two 4-bit data inputs, A and B
 - One 4-bit data output, F
 - 1-bit Cin (input) and Cout (output) terminals to create carry chains linking ALUs to process wider words
 - A 4-bit instruction input, I. The ALU operation can be set either statically (via an internal configuration register) or dynamically (through the instruction input).
 - An optional output register on the 4-bit data output.
- The routing network propagates data as 4-bit quantities:
 - All routing buses are 4-bit buses
 - All routing switches connect two 4-bit buses
 - 1-bit data (carry and control) is carried in the LSB of a 4-bit word.

- There are direct 1-bit connections between Cin and Cout of neighboring ALUs, separate from the main routing network.
- ALUs and switchboxes are arranged in a checkerboard pattern.
- ALUs and Switchboxes are paired, and share some common signals within the pair
- Switchboxes contain additional 4-bit registers that can be used to pipeline applications.

The operations performed by the ALU are of four basic types:

- Arithmetic operations (using Cin and Cout as a carry chain):
 $\text{ADD: } F = A + B + \text{Cin}$
 $\text{SUB: } F = A - B + \text{Cin}$
- Bitwise logical operations, such as:
 $\text{AND: } F = A \& B$
 $\text{OR: } F = A \mid B$
 $\text{XOR: } F = A \wedge B$
 $\text{NOT: } F = \sim A$
- Multiplexer operations (with Cout = Cin in all cases):
 $\text{MUX: } F = \text{Cin} ? A : B$
 $\text{INVMUX: } F = \text{Cin} ? \sim A : \sim B$
- Comparison and test operations:
 $\text{NEQ: } \text{Cout} = (A == B) ? \text{Cin} : 1$
 When Cin = 0, Cout is 0 if A is equal to B and 1 otherwise.
 $\text{ORAND: } \text{Cout} = (A \& B == 0) ? \text{Cin} : 1$
 When Cin = 1, Cout is 1 only if the AND of A and B is not 0. This is equivalent to an AND of A and B, followed by an OR reduction.
 Less-than and Greater-than functions are available via the carry output of an appropriate subtraction.

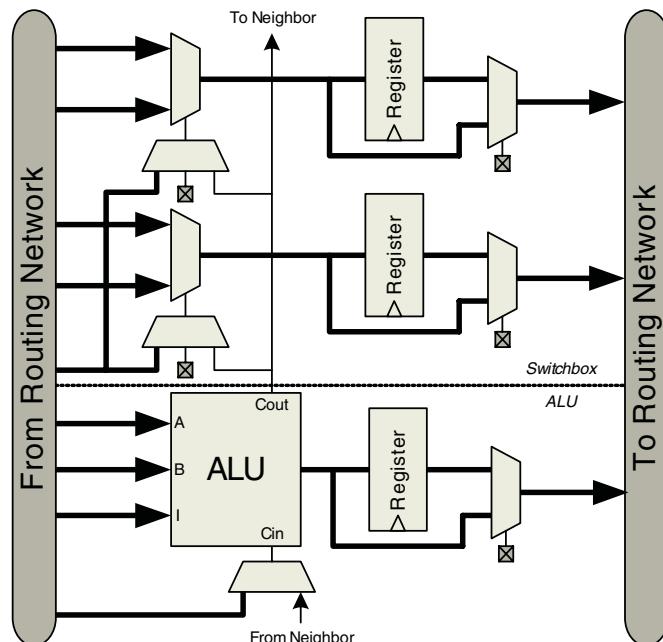


Fig. 1. Simplified diagram of logic in D-Fabrix ALU and Switchbox pair

There is however one significant difference between D-Fabrix and Chess, which is illustrated in figure 1. The multiplexers that select signals from the routing network to pass to the pipeline registers were statically configured in Chess, but D-Fabrix allows them to be dynamically controlled, using either the ALU Cout or a signal from the routing network as the multiplexer control signal. The sources of dynamic control for these multiplexers (normally referred to as “Switchbox Multiplexers” due to their being located in the switchbox) are broadly equivalent to those of the ALU Cin – either Cout from a nearby ALU or a signal from the global routing network.

These dynamic switchbox multiplexers are in addition to the MUX instruction of the ALU. It is possible to use both the switchbox multiplexers and the ALU as a multiplexer simultaneously, all controlled by the same signal, so that a single ALU and switchbox pair can implement a multiplexer for up to 12-bit data inputs.

This change to dynamic switchbox multiplexers has resulted in a slight overall increase in the area of the combined ALU and switchbox. This increase is due to the addition of the circuits that select the control inputs to the switchbox multiplexers and their configuration memories – the switchbox multiplexers themselves were already present, but with only static control. The area increase due to this change is no more than 10%. Compared to an ALU a switchbox multiplexer is one-third of the size, has half the propagation delay and 40% of the power dissipation. (Area comparisons are based on the Chess and D-Fabrix layouts, speed and power comparisons on spice simulation of the D-Fabrix circuit). Consequently, there are significant gains in area, speed and power dissipation to be made if an application can use these dedicated multiplexers instead of ALUs.

4 Multiplexer-Based Control

Multiplexers can be used to implement simple logic functions of two 1-bit inputs. Table 1 lists all 16 Boolean functions of up to 2 inputs, and possible implementations constructed from a multiplexer controlled by one of the function inputs, and with the multiplexer data inputs being connected to either a constant or the other function input.

Table 1. Boolean Functions of up to 2 Inputs

Function	Implementation	Function	Implementation
0	A ? 0 : 0	1	A ? 1 : 1
A & B	A ? B : 0	A B	A ? 1 : B
A & !B	B ? 0 : A	A !B	B ? A : 1
!A & B	A ? 0 : B	!A B	A ? B : 1
!A & !B	(NOR)	!A !B	(NAND)
A	A ? 1 : 0	B	B ? 1 : 0
!A	A ? 0 : 1	!B	B ? 0 : 1
A^B	(XOR)	!A^B	(NXOR)

Of these 16 functions, four are trivial (0, 1, A and B), and can be implemented without a multiplexer, eight have multiplexer-based implementations and four do not.

These four are NAND, NOR, XOR and NXOR. A simple logic optimization pass can commonly turn NAND and NOR into other AND and OR-type functions, so it is only XOR and NXOR that are difficult to implement in a compact form with multiplexers.

Historically, Actels antifuse-based FPGA families [9] have used multiplexers as the basis of their logic elements. Actel used multiplexers with 1-bit data and control inputs, but it is also possible to use larger multiplexers in the same way, with some of the available data bits being unused if necessary. The D-Fabrix switchbox multiplexers can therefore be used to construct 2-input logic gates, with one of the inputs connecting to the control input of the multiplexer and the other to one of the data inputs. Only one of the four available data bits is used, but since multiplexers are much smaller than ALUs it is more area efficient to use the multiplexers for this purpose than to use an ALU.

The control flow circuits referred to in section 2 are all constructed from AND and OR gates, some with inverters on their inputs. It can be seen from table 1 that such functions all have multiplexer-based implementations. Thus it appears that both the data flow and the program control flow aspects of control can be converted into arrangements of multiplexers controlled by the outputs of expressions with Boolean results. Figure 2 illustrates the use of the switchbox multiplexers for controlling the flow of (a) data and (b) application control. Figure 2(b) is an implementation of the circuit shown in figure 4 of [1]. Both 2(a) and 2(b) consist of two parts – a group of ALUs that compute a Boolean function of their word-wide inputs, and a group of multiplexers whose control input is driven by Cout from one of the ALUs. All test operations computed by the ALUs produce their Boolean result on Cout, and (as shown in figure 1) D-Fabrix has direct support for Cout to multiplexer control connections.

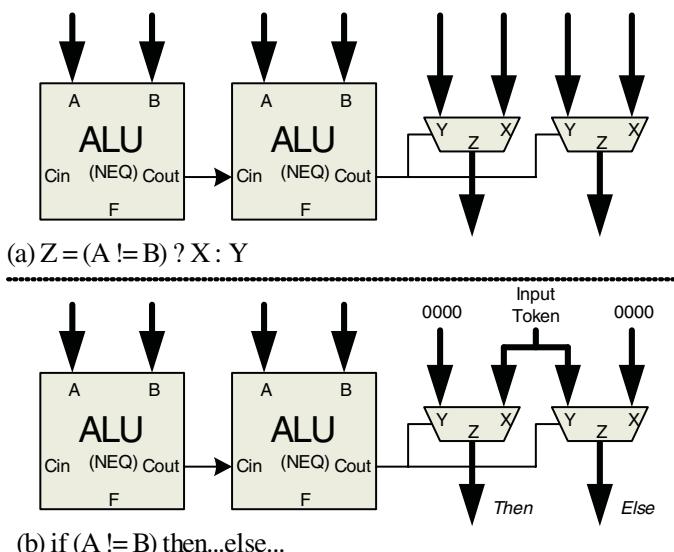


Fig. 2. Use of multiplexers to route (a) data and (b) application control

Furthermore, many of the control expressions themselves contain subexpressions that can be implemented with 2-input gates. For example, consider the expression:

$$Z = (A_1 \neq B_1) \& (A_2 \neq B_2)$$

The two comparisons are both functions of word-based data, but the AND that combines them is a function of single-bit inputs, and can therefore be implemented with a multiplexer. Figure 3 shows such an implementation of this expression.

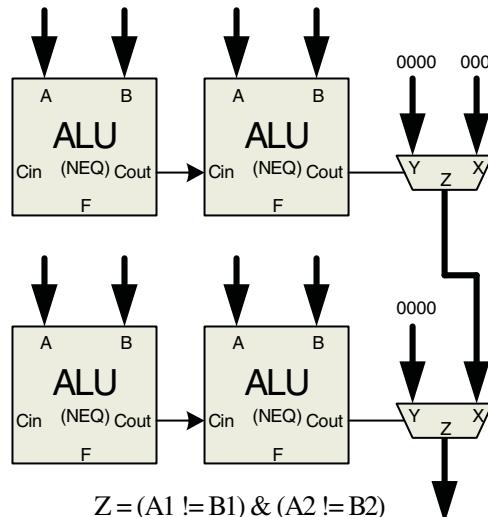


Fig. 3. Use of Multiplexers to combine results of Boolean expressions

5 Other Uses of Multiplexers

The switchbox multiplexers have other uses beyond the ones mentioned above. Some of these uses are described below.

5.1 1-Bit to 4-Bit Gateway

Figure 1 shows direct connections from the 4-bit routing network to the multiplexers that select among 1-bit signals to provide both the ALU Cin and the switchbox multiplexer control signals. These direct connections allow the least-significant bit of a 4-bit bus to be used as the source of these signals. The switchbox multiplexers can be used to provide the opposite 1-bit (i.e. Cout) to 4-bit connection. If the multiplexer has as its data inputs the binary constants 0000 and 0001 then its function is:

$$Z = \text{Cout} ? 0001 : 0000 = 000<\text{Cout}>$$

i.e. the output is a 4-bit version of the multiplexer control input. Such a use of the multiplexer is illustrated in the upper half of figure 3. It is possible to produce other functions of Cout simply by using different input constants for the multiplexers. For example:

```
Z = Cout ? 0000 : 0001 = 000<~Cout>
Z = Cout ? 1111 : 0000 = <Cout><Cout><Cout><Cout>
```

i.e. a 1-bit inverse of Cout, or a 4-bit version.

5.2 Register Control

The switchbox multiplexers are located at the inputs of the switchbox registers. This makes them easy to use to add control options to the registers, such as reset and enable, as illustrated in figure 4.

This usage is more flexible than having a dedicated reset connection to the register, as it is possible to change the reset value of the register by changing the constant supplied to the multiplexer input.

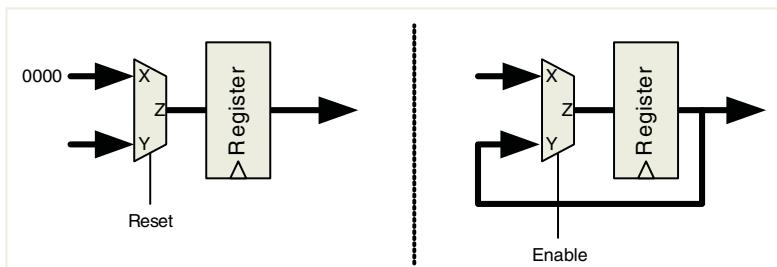


Fig. 4. Use of Multiplexers for register reset and enable

6 Software Support

Preceding sections have described the D-Fabrix hardware architecture. In this section we consider the ability of software to make use of the hardware.

Table 2 shows the relative proportions of the different types of ALU operations generated by the Elixent verilog compiler on 5 different stages of a JPEG encoder. JPEG is used as an example here because:

- a) It is a widely used (and understood) example, and
- b) It contains some sections that are almost all datapath (such as the DCT), and others that have a significant amount of control logic.

It therefore provides a reasonable test of an architectures ability to handle both datapath-intensive and control-intensive designs.

Table 2 shows that the compiler consistently generates a high proportion of multiplex operations. Between 40% and 60% of operations are multiplexers. The variation in the proportion of multiplexers is lower than that for the other types of instruction – the maximum percentage is 1.5 times the minimum for multiplexers, more than twice the minimum for arithmetic operators and greater still for test and bitwise logic operations.

Table 2. 4-bit Operation Usage in JPEG Encoder

Block	Arithmetic	Multiplex	Test	Bitwise Logic
Fmerge	35.7%	42.9%	14.3%	7.1%
FDCT	35.1%	59.3%	4.0%	1.5%
Quantize	51.1%	42.1%	6.8%	0%
Zigzag	19.0%	42.9%	31.0%	7.1%
Ent&Pack	21.9%	57.2%	8.2%	12.7%

This result is not specific to JPEG. When measured across a larger set of benchmarks the fraction of operators that can be implemented with a multiplexer typically lies in the 40% to 70% range. The extremes of this range are represented at the high end by applications with a large amount of bit manipulation (such as DES, and an arithmetic codec for JPEG2000), and at the low end by multiplication intensive pure datapath functions (such as a CDMA matched filter). This is in accordance with expectation based on the description of multiplexer usage given above.

Given that multiplexers account for 40% – 70% of an application, it is clear that the switchbox multiplexers have a net positive impact on array density. If all the multiplex operations had to be implemented with ALUs then an array would need between 1.6 and 3 times the number of ALUs for the same application. Compared with this the 10% area increase required to make the multiplexers dynamic rather than static is a significantly smaller cost. There are also gains in speed and power dissipation as a result of the use of multiplexers instead of ALUs for large parts of an application.

The array is constructed with 2 multiplexers per ALU, corresponding to 66% of the processing resources in the array. This ratio means that for almost all applications the number of ALUs is the limiting factor, not the number of multiplexers. If the ratio was increased to 3 multiplexers per ALU the majority of the extra multiplexers would be unused in almost all applications.

7 Comparison to Other Work

The D-Fabrix architecture as described above can be categorized as a “Spatially Uniform Heterogeneous Array” – heterogeneous in that it contains two distinct types of processing element (ALUs and multiplexers), but spatially uniform in that the array is a regular arrangement of these two types of resource (it does not have distinct large groups of ALUs, and of multiplexers). Previously published work on heterogeneous arrays falls into two main classes:

- Studies of the impact of using two or more different sizes of LUT [3][4]. The use of embedded RAMs as supplementary logic when not used as memory [5] can be regarded as a special case of this category.
- Studies of arrays containing both LUTs and product term arrays [6][7][8].

Neither of these categories is directly relevant to the current work:

- D-Fabrix has two types of logic element, but they differ in the functionality that they provide, not in the number of inputs.
- Product-term arrays generate multiple functions of the same inputs, whereas both the ALUs and Multiplexers in D-Fabrix generate a single function of independent inputs.

The key difference between D-Fabrix and these other examples is that the two types of processing element are chosen so that the functionality provided by one (the multiplexer) is a subset of that of the other (the ALU). This greatly simplifies the main problem identified in the earlier work – the increased complexity of synthesis and/or technology mapping due to the need to decide which parts of the application to map to the different types of processing element. Since the multiplexers provide a subset of the ALU functionality, the synthesis process need only target the ALUs, and placement can then determine whether a given multiplexer in the netlist maps to an ALU or a switchbox multiplexer. The only change made to the synthesis process is a modification of some logic optimization rules so that simple gates are converted to their multiplexer-based equivalents.

It is worth noting that although the D-Fabrix architecture described in this paper is significantly different to the arrays containing LUTs of two different sizes considered by He and Rose [3], the basic conclusion is the same – a heterogeneous array offers better density than a comparable homogeneous array.

The reasoning behind these similar conclusions is also the same. In [3] an array of mostly 4-input LUTs (4-LUTs) with some 2-LUTs is shown to be more efficient than a homogeneous array of 4-LUTs. The explanation given for this is that applications often contain functions with fanout greater than 1 but only a small number of inputs, and these are the functions that map well onto the 2-LUTs. In effect there is a significant fraction of the application for which the 4-LUT provides more logic than necessary, and the removal of this overhead results in a more compact solution. In the current case, an array of multiplexers and ALUs is denser than an array of ALUs alone, again because there is a significant fraction of the target applications for which the ALU provides more functionality than required, and so a simplified functional unit targeted at the needs of this fraction results in an increase in density.

8 Summary

This paper has described how the addition of dynamic multiplexers to an ALU array provides an overall improvement in performance and density. These multiplexers are used for both data manipulation and to implement simple 2-input gates to process control signals, and account for around half of the total logic required. Since multiplexers are smaller and faster than ALUs there is a net reduction in both area and delay, and software is easily able to make good use of this mixture of ALUs and multiplexers.

References

- [1] I. Page and W. Luk, Compiling occam into FPGAs, in FPGAs, W. Moore and W. Luk (editors), Abingdon EE&CS Books, 1991, pp. 271-283.
- [2] Marshall et. al., A Reconfigurable Arithmetic Array for Multimedia Applications. In Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA 99), pp 135-143, February 1999.
- [3] J. He and J. Rose, Advantages of Heterogeneous Logic Block Architectures, CICC 1993, pp7.4.1 – 7.4.5
- [4] J. Cong and S. Xu, Delay-Optimal Technology Mapping for FPGAs with Heterogeneous LUTs, Proc. of 35th Design Automation Conf., San Francisco, California, pp. 704-707, June 1998
- [5] J. Cong and S. Xu, Delay-Oriented Technology Mapping for Heterogeneous FPGAs with Bounded Resources, Proc. ACM/IEEE International Conference on Computer Aided Design, San Jose, California, pp. 40-45, November 1998.
- [6] A. Kaviani and S. Brown, Hybrid FPGA architecture, In Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA 96), pp 3-9, February 1996.
- [7] F. Heile and A. Leaver, Hybrid Product Term and LUT Based Architectures Using Embedded Memory Blocks, In Proc. of the International Symposium on Field Programmable Gate Arrays (FPGA 99), pp 13-16, February 1999.
- [8] E. Lin and S. Wilton, Macrocell Architectures for Product Term Embedded Memory Arrays, in Field-Programmable Logic and Applications (FPL 2001), pp 48-58
- [9] Actel ACT1 datasheet at <http://www.actel.com/docs/datasheets/db97s01d07.pdf>

Heterogeneous Logic Block Architectures for Via-Patterned Programmable Fabrics

Aneesh Koorapaty¹, Lawrence Pileggi¹, and Herman Schmit¹

Carnegie Mellon University, Pittsburgh PA 15213, USA,
`aneeshk@ece.cmu.edu`, `pileggi@ece.cmu.edu`, `herman@ece.cmu.edu`

Abstract. ASIC designs are becoming increasingly unaffordable due to rapidly increasing mask costs, greater manufacturing complexity, and the need for several re-spins to meet design constraints. Although FPGAs solve the NRE cost problem, they often fail to achieve the required performance and density. A Via-Patterned Gate Array (VPGA) that combines the regularity and design cost amortization benefits of FPGAs with silicon area and power consumption comparable to ASICs, was presented in [1]. The VPGA fabric consists of a regular interconnect architecture laid on top of an array of patternable logic blocks (PLBs). Customization of the logic and interconnect is done by the placement or removal of vias at a subset of the potential via locations. In this paper, we propose four heterogeneous PLBs for via-patterned fabrics and explore their performance, density and fabric utilization characteristics across several applications. Although this analysis is done in the context of the VPGA fabric, the proposed heterogeneous PLBs and the experimental methodology can be employed for any embedded programmable fabric.

1 Introduction

Application Specific Integrated Circuits (ASICs) consist of pre-designed logic cells and up to seven layers of metal wiring. The high degree of flexibility in placement and routing of the cells necessitates unique, customized masks for all fabrication layers. With shrinking feature sizes and increasing design complexity, mask costs and design costs for re-spins are becoming prohibitively expensive. As a result, programmable fabrics like Field Programmable Gate Arrays (FPGAs) are becoming increasingly attractive. Unlike ASICs, FPGAs amortize design costs across several applications and enhance manufacturability via greater layout regularity. However, FPGAs are at least three times slower and require at least ten times as much die area as ASICs. The area overhead is due to the island style topology in which the interconnect is placed adjacent to the logic array. The performance penalty is due to the RC delays of the SRAM controlled pass transistors in the interconnect switchboxes, and the SRAM based LUTs in the PLBs.

A Via-Patterned Gate Array [1] is a novel, regular fabric that bridges the cost-performance gap between ASICs and FPGAs. Like an FPGA, a VPGA

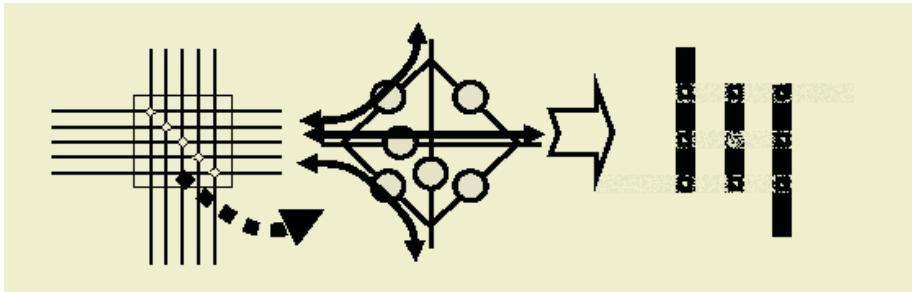


Fig. 1. VPGA Switchpoint Design

consists of an array of PLBs, and a fixed interconnect architecture. However, there are two key differences: First, the routing architecture is laid on top of, instead of adjacent to the PLB array, resulting in a significant reduction in die area. Second, the customization of the logic and interconnect is done by the placement or removal of vias at the potential via locations, as opposed to configuring SRAM bits.

Figure 1 illustrates a via-patterned switchpoint which replaces between 36 to 120 transistors for an FPGA switchpoint with 8 potential mask configurable vias. Grouping many of these via-patterned switchpoints results in a via-patterned switchbox. This switchbox is similar to an FPGA switchbox in that each track requires a switchpoint to connect to the same track in a neighboring switchbox. The difference, however, is that there are no pass transistors to define the connectivity of the switchpoints. The switchbox shown in Figure 2 is a 3x3 arrangement of metal lines with vias inserted at the possible connectivity points.

The key objective of the VPGA fabric is to combine the cost amortization benefits of FPGAs with a level of performance close to ASICs, for a *domain of application*.

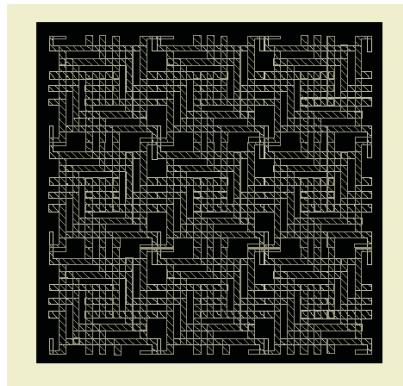


Fig. 2. VPGA Switchbox Layout

plications. The choice of PLB architecture is crucial for achieving this objective. In [4] the authors investigated the optimal LUT size for homogeneous LUT-based PLBs for the VPGA fabric. However, [2] and [3] showed that heterogeneous PLBs with a combination of logic gates, LUTs, and MUXes offer significantly better performance and density than homogeneous LUT-based PLBs.

In this paper, we investigate four heterogeneous PLB architectures for via-patterned programmable fabrics. Using a fabric-specific synthesis engine [2], we map a set of standard benchmarks to each of the PLBs, and compare their performance, density and fabric utilization characteristics across applications. Based on this analysis, we select a high performance heterogeneous logic block architecture for the VPGA fabric. Although this analysis is conducted in the context of the VPGA architecture, the proposed heterogeneous PLBs and the experimental methodology can be employed for any embedded programmable fabric.

The remainder of this paper is organized as follows. Section 2 illustrates the proposed heterogeneous architectures, followed by a description of the experimental methodology in Section 3. Section 4 compares the performance, density and fabric utilization characteristics of each of the PLBs across a host of applications. Based on these results, we select a PLB architecture for the VPGA fabric. Section 5 concludes the paper.

2 Heterogeneous Programmable Logic Blocks

There are two key design considerations for heterogeneous PLBs. The first is to determine which logic gates are suitable and which combinations of LUT sizes to employ. The second is to explore various configurations of these logic elements.

2.1 Logic Gates for Heterogeneous PLBs

Nand Gates with Programmable Inversion. It can be shown that a 2-input Nand gate with programmable inversion on the inputs and outputs (ND2WI) can implement all 2-input functions except exclusive or (XOR) and exclusive nor (XNOR). Similarly, it can be shown that a 3-input Nand gate with programmable inversion (ND3WI) can implement 46 of the 256 3-input functions. In [3], twenty MCNC benchmarks were mapped to 3-LUTs and 4-LUTs with Flowmap. For most of the benchmarks, between 40-80% of the functions in the 3-LUT mapped netlists could be implemented by a ND3WI gate. The ND3WI gate also achieved significant coverage of functions in the 4-LUT mapped netlists. These results show that ND3WI gates are good candidates for heterogeneous PLBs.

3-input Semi-LUTs. [3] proposed a logic structure called an S3 gate (3-input Semi-LUT) consisting of two ND2WI gates driving a 2:1 MUX. Since each of the ND2WI gates can implement 14 of the 16 2-input functions, this structure can implement at least 196 of the 256 3-input functions. For the same set of benchmarks as in the ND3WI analysis, [3] showed that the S3 gate could implement

over 90% of the functions in the 3-LUT mapped netlists for well over half the benchmarks. Furthermore, the S3 gate consistently achieved a coverage between 20% to 40% for 4-LUT mapped netlists as well. Hence, we also consider the S3 gate for our heterogeneous PLBs.

2.2 Proposed Heterogeneous PLB Architectures

In previous work, [2] showed that a heterogeneous PLB consisting of two 3-LUTs and one 2-LUT is 30% more area-efficient than a homogeneous 4-LUT based PLB. Also, [5] showed that a combination of 4 and 2-input LUTs result in significant density benefits. Based on these results and the above analysis of logic gates, we propose four heterogeneous PLBs for the VPGA fabric. These PLBs are illustrated in Figure 3. Although not shown in Figure 3 all of these PLBs also contain I/O buffers and a D flip flop. The I/O buffers ensure that all primary inputs and outputs are available in any polarity.

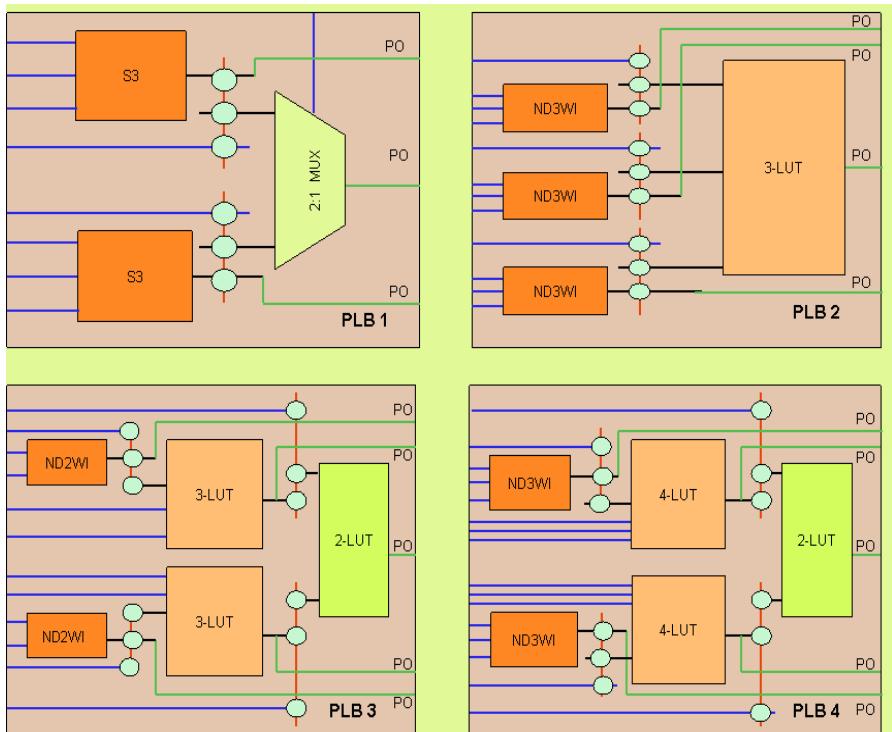


Fig. 3. Proposed Heterogeneous Logic Block Architectures

PLB 1: The first PLB consists of two S3 gates and a 2:1 MUX. This cell has nine unique primary inputs and three primary outputs. As shown in Figure 3

each of the MUX inputs can connect either to a primary input or the output of an S3 gate by placing or removing the appropriate vias on the corresponding jumper wires. Since each of the S3 gates can implement all 2-input functions the entire PLB can be utilized as a 3-LUT. Alternatively, since each S3 gate can implement several 3-input functions, this PLB can simultaneously implement two 3-input functions and a 2-input function (in the MUX). Since a large % of functions in typical 3-LUT mapped netlists can be implemented by a single S3 gate, PLB 1 is more likely to be utilized in the latter configuration.

PLB 2: The second PLB architecture consists of one 3-LUT and three ND3WI gates. Each of the LUT inputs can be connected either to a primary input, or one of the ND3WI gates by patterning the vias on the jumper wires. This PLB has twelve unique inputs and four primary outputs, enabling it to simultaneously implement four 3-input functions.

PLB 3: [2] showed that a heterogeneous PLB consisting of two 3-LUTs and one 2-LUT is 30% more area-efficient than a homogeneous 4-LUT based PLB. Based on this result, we propose a PLB consisting of one 2-LUT, two 3-LUTs, and two ND2WI gates. As shown in Figure 3, the 2-LUT can connect either to primary inputs or the outputs of the 3-LUTs. Furthermore, the 3-LUTs can be driven either by three primary inputs, or two primary inputs and a ND2WI gate. With five primary outputs, this PLB can implement up to three 2-input functions, and two three input functions simultaneously.

PLB 4: [5] showed that an architecture with one 2-LUT, and two 4-LUTs requires 22% fewer bits and 10% fewer pins than a homogeneous 4-LUT based PLB. Based on this result, and the high functional coverage of ND3WI gates, we propose a PLB with one 2-LUT, two 4-LUTs, and two ND3WI gates. As shown in Figure 3, each of the logic elements in this PLB can be driven by primary inputs of the cell, or by other PLB logic elements by patterning the vias on the appropriate jumpers. With five primary outputs, this PLB can implement up to two 4-input functions, two 3-input functions, and one 2-input function simultaneously.

3 Experimental Methodology

To compare performance and density, we first map and pack a set of standard benchmarks to each of the proposed PLBs with a fabric-specific approach [2] that captures and exploits the benefits of heterogeneity. Next, we place and route the netlists with VPR [7]. To emulate the VPGA routing architecture, we make some modifications to the VPR PLB and delay models. In this section, we discuss the area, delay, and VPR models for our heterogeneous PLBs.

3.1 Area and Delay Models

In the VPGA fabric, the LUTs in the logic blocks are also via-patterned. As shown in Figure 4, the LUT is constructed as a K-1 level tree with a complementary pull up and pull down network.

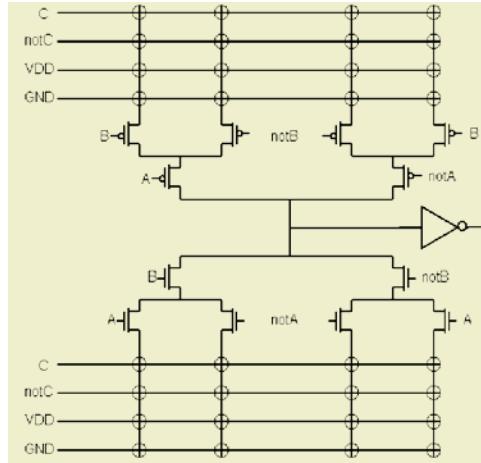


Fig. 4. Via-patterned 3-LUT

Each of the leaf nodes in the LUT can directly connect to VDD, GND, and another input or its complement. We estimate the layout area of the 3-LUT shown in Figure 4 as the area required for three 2:1 MUXEs. The layout area of the MUX, and other logic gates in the heterogeneous PLBs, is estimated from the technology specifications for the corresponding minimum sized gates in our commercial $0.13\ \mu$ process. The total PLB area is the sum of the areas for all its logic elements including I/O buffers at each of the PLB inputs, and a single D flip flop.

To determine the delay of the logic elements, we ran HSPICE simulations with an output load comparable to an output buffer. To remain consistent with our area model and control the complexity of the experiment, we kept all transistors minimum sized. For simplicity, the LUTs were configured as NANDs for these simulations. Table 1 summarizes the area and delay values for the logic elements.

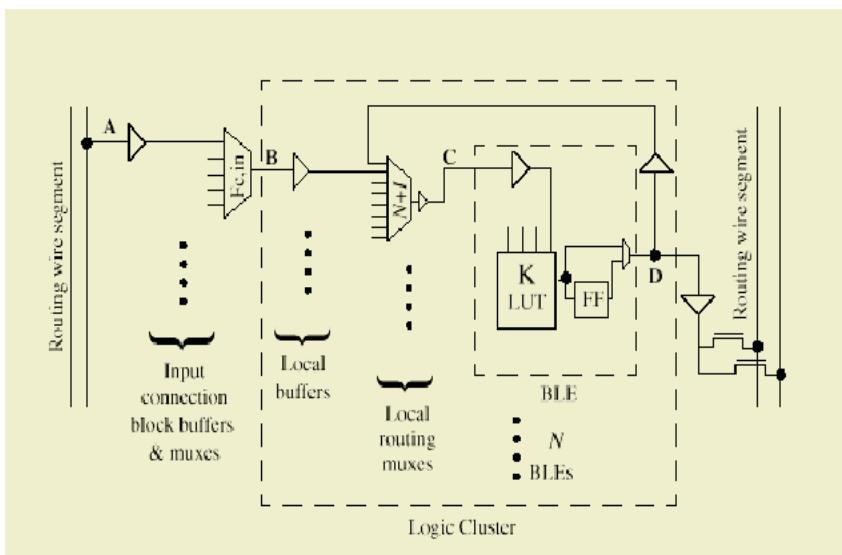
3.2 VPR Logic Block Model

As illustrated in Figure 5, VPR assumes that the logic blocks consist of a cluster of N basic logic elements (BLE), each containing a K-input LUT and a register. Furthermore, VPR assumes a fully connected local routing architecture [6] in which each BLE input can connect to a primary input, or the output of any

Table 1. Area and Delay Values

Logic Element	Area (μ sq.)	Delay (ps)
Inverter	6.052	15
ND2WI	24.208	45
ND3WI	32.277	55
S3	64.554	65
2:1 MUX	16.138	20
2-LUT	16.138	50
3-LUT	48.414	70
4-LUT	112.966	90

of the N BLEs via a multiplexer. To evaluate the proposed PLBs with VPR we model each of the PLB logic elements as a K-input BLE, with K set to the number of inputs for the largest logic element. As the VPGA fabric is via-patterned, the connection block and local routing MUXEs in Figure 5 are replaced by vias. Hence, the corresponding delay values are set to zero. Similarly, the resistance and capacitance values for the switchboxes are modified to reflect the via-patterned routing, and the combinational delays for the logic elements are set to the values shown in Table 1.

**Fig. 5.** VPR Logic Block and Delay Model

4 Experimental Results

With the assumptions outlined in the previous section, we place and route the packed netlists with VPR. Then, we compare the total layout area and critical path delay for the benchmark set on each of the architectures. Since the global routing is on top of the CLBs, the total layout area is computed as: $\max(\text{Routing area}, \text{PLB Area}) * \# \text{PLBs}$. Furthermore the routing area is estimated as $[(N + (2 * \sqrt{N})) * 0.4]^2$, where:

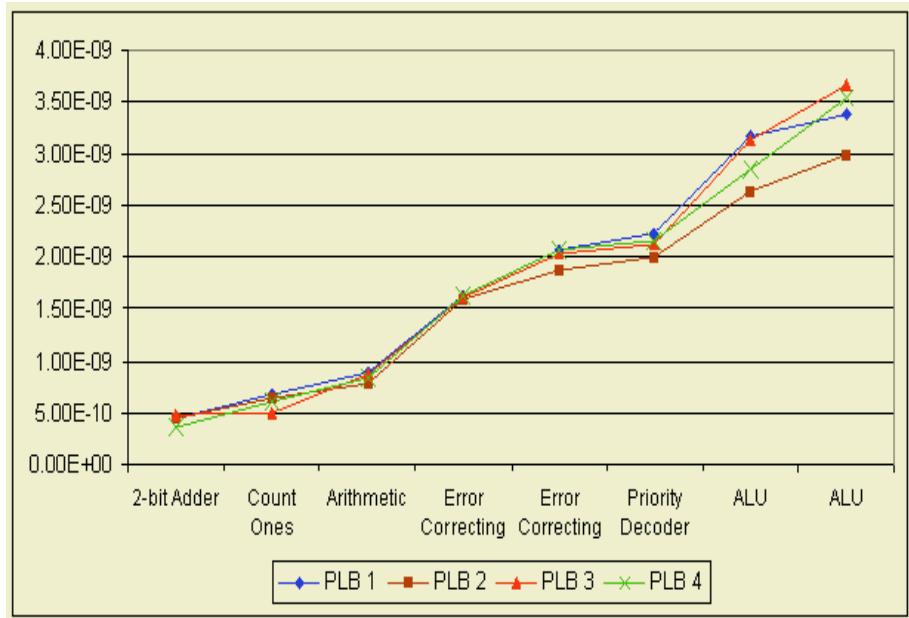
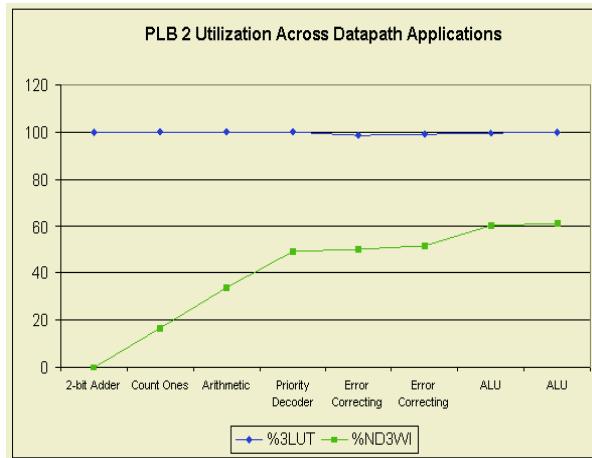
$$\begin{aligned} N &= \text{Channel width required by VPR to successfully route the packed netlist} \\ 0.4 &= \text{Pitch width} + \text{Minimum spacing for our } 0.13\mu \text{ process} \end{aligned}$$

Since one of the key objectives of the VPGA fabric is to achieve high performance for a *domain of applications*, we present the results in an application-specific manner. Table 2 presents the area, delay, and area-delay product for each of the PLBs across the two application domains. Results for each benchmark are averages over five VPR runs with different placement seeds. The values in Table 2 represent the sum across all the benchmarks in the corresponding application domain.

Table 2. Area, Delay, and Area-Delay Product Results

PLB	Logic Circuits			Datapath Circuits		
	Total Area	Delay	Area*Delay	Total Area	Delay	Area*Delay
	(μ sq.)	(ns)	(*E-03)	(μ sq.)	(ns)	(*E-03)
1	83498	12.26	1.02	62909	14.46	0.91
2	61083	11.06	0.68	47577	12.94	0.62
3	111303	12.78	1.42	84483	14.42	1.22
4	175428	12.56	2.20	131128	14.06	1.84

From Table 2 we observe that PLB 2 has the lowest area, critical path delay, and area-delay product for both logic and datapath circuits. For the logic circuits, PLB 1 has the second best area, delay and area-delay product. However, PLB 1 has the worst delay across the datapath applications. For further analysis of this result, Figure 6 presents the critical path delay for each of the datapath applications. The difference in performance is particularly distinctive for the last five circuits in Figure 6. Across these applications, PLB 2 has a total critical path delay of 11.1 ns, as opposed to 12.3 ns for PLB 4, 12.5 ns for PLB 1 and 12.6 ns for PLB 3. This represents roughly a 10% performance advantage for PLB 2. To explain this result, we examine the percentage utilization (ratio of utilized to available resources) of the ND3WI gates and 3LUT for each of these applications in Figure 7. From Figure 7 we observe that the ND3WI gate utilization is noticeably higher for the ALU, error correction and priority decoder circuits than for the other applications.

**Fig. 6.** Delay Comparison for Datapath Applications**Fig. 7.** PLB 2 Fabric Utilization Profile

In [8], the authors presented a physically heterogeneous programmable fabric with a logic array consisting of two kinds of PLBs: a homogeneous PLB with four 4-LUTs, and a CPLD style PAL block. Although we cannot include these results due to space limitations, current work in [9] shows that the heterogeneous PLBs

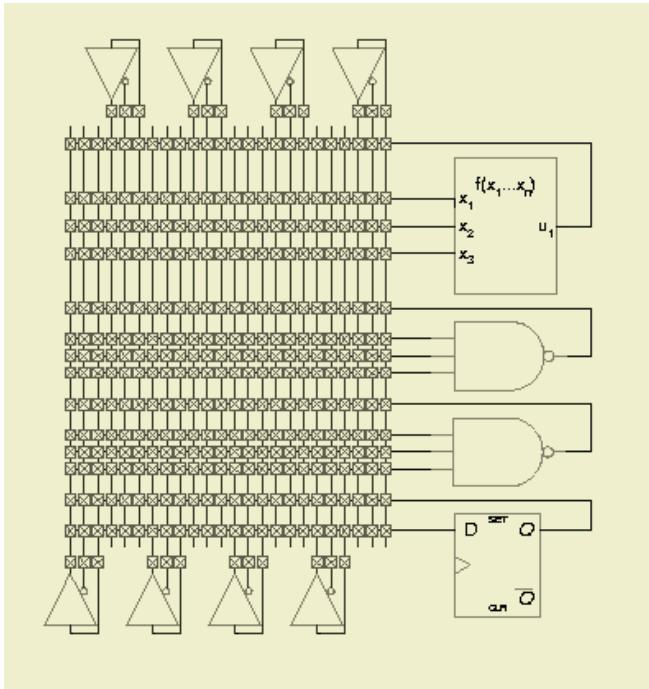


Fig. 8. Selected PLB for the VPGA Fabric

discussed in this paper are more area-efficient than the physically heterogeneous fabric presented in [8] as well.

Based on the results of this section, we select the PLB shown in Figure 8 for the VPGA fabric in this process technology. This cell contains only two ND3WI gates in order to reduce the number of input and output pins. Results presented in [10] show that the VPGA fabric with this logic block is quite competitive with standard cells.

5 Conclusions

In this paper we proposed four new heterogeneous PLBs for via-patterned fabrics and evaluated their performance, density and fabric utilization characteristics for different application domains. Our results suggest employing a heterogeneous PLB consisting of 3-input LUTs and ND3WI gates. Although this analysis was conducted in the context of the VPGA architecture, the proposed heterogeneous PLBs can be employed in any embedded programmable fabric. In future work, we propose to study other via-patterned global routing architectures. We expect the results of these studies to be influenced primarily by variations in delay, since the density benefits of the proposed VPGA PLB are dictated by the high functional

coverage of its logic elements. We also propose to explore heterogeneous PLB architectures for more application domains.

Acknowledgements

We would like to thank Chetan Patel and Kim Yaw Tong for the delay simulations and Anthony Cozzie for the VPGA routing area model.

References

1. L. Pileggi, H. Schmit, J. T. Shah, K. Y. Tong, C. Patel and V. Chandra: A Via Patterned Gate Array (VPGA), Technical Report Series, Center for Silicon System Implementation, No. CSSI 02-15, March 2002
2. A. Koorapaty and L. Pileggi: Modular, Fabric-specific Synthesis for Programmable Architectures, Proceedings of the 12th International Conference on Field Programmable Logic and Applications, September 2002
3. A. Koorapaty, V. Chandra, K. Y. Tong, C. Patel, L. Pileggi, and H. Schmit : Heterogeneous Programmable Logic Block Architectures, Proceedings of Design Automation and Test in Europe, March 2003
4. C. Patel, A. Cozzie, H. Schmit, and L. Pileggi: An Architectural Exploration of Via Patterned Gate Arrays, ACM/SIGDA International Symposium on Physical Design, 2003
5. J. He, and J. Rose: Advantages of Heterogeneous Logic Block Architectures for FPGAs, IEEE Custom Integrated Circuits Conference, pp. 7.4.1-7.4.5, May 1993
6. V. Betz, and J. Rose: Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size, IEEE Custom Integrated Circuits Conference, pp. 551-554, 1997
7. V. Betz, J. Rose, and A. Marquardt: Architecture and CAD for Deep-Submicron FPGAs, Kluwer Academic Publishers, 1999
8. A. Kaviani, and S. Brown: The hybrid field-programmable architecture, IEEE Design & Test of Computers, Vol. 16, Issue 2, pp. 74-83, April-June 1999
9. A. Koorapaty: Modular, Fabric-specific Synthesis and Novel Logic Block Architectures for Regular Fabrics, Ph.D. Thesis, Center for Silicon System Implementation, Carnegie Mellon University, 2003
10. L. Pileggi, H. Schmit, A. J. Strojwas, P. Gopalakrishnan, V. Kheterpal, A. Koorapaty, C. Patel, V. Rovner, and K. Y. Tong: Exploring Regular Fabrics to Optimize the Performance-Cost Tradeoff, Proceedings of Design Automation Conference, June 2003

A Real-Time Visualization System for PIV

Toshihito Fujiwara, Kenji Fujimoto, and Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba,
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 Japan,
fujiwara@darwin.esys.tsukuba.ac.jp

Abstract. Particle image velocimetry (PIV) is a method of imaging and analyzing field of flows. In the PIV method, small windows in an image of the field (time t) are compared with areas around the windows in the another image of the field (time $t + \Delta t$), and the most similar part to the windows are searched using two dimensional cross-correlation function. The computational complexity of the function is very huge, and can not be processed in real-time by micro-processors. In this paper, we describe a real-time visualization system for the PIV method. In the system, an improved direct computation method is used to reduce the computational complexity. The system consists of only one off-the-shelf Virtex-II FPGA board and a host computer, and calculates the complex function without reducing data bit-width, which becomes possible with one latest FPGA.

1 Introduction

Particle image velocimetry (PIV) is a method for visualizing and analyzing field of flows. In the PIV method, a pair of images (time = t and $t + \Delta t$) of the field is compared, and the flows in Δt are visualized and analyzed. In order to compare two images of the field, many small particles with the same specific gravity of the liquid or air in the field are added and the speed and direction of movement of the particles are measured by finding out pairs of small areas (windows) which are the most similar to each other in the two images. In order to find out the pairs, two dimensional cross-correlation function is used in general. The amount of computations in the function is very huge, and can not be processed in real-time by micro-processors.

We have already proposed a basic idea for efficient computation of the cross-correlation function with FPGA[2]. In that implementation, however, data width of pixels in the images and the intermediate data of the function had to be reduced because of the limitation of the hardware resources and memory width (internal and external), which caused errors in the final results. Because of the recent progress of the size of FPGAs and multipliers built in FPGAs, it became possible to compute the function without reducing the data width, which generates the same results with micro-processors. However, the implementation to obtain the correct results in real-time is not straightforward. In our system, an improved direct computation method of the two dimensional cross-correlation function is used to achieve real-time processing.

In this paper, we first discuss computation methods of the cross-correlation function for real-time processing (the best method depends on the requirements by the application and the performance of the FPGA), and then describe a real-time visualization system which consists of one Virtex-II FPGA board and a host computer.

2 Particle Image Velocimetry (PIV) System

Figure 1 shows an overview of a particle image velocimetry (PIV) system. In PIV method, pairs of images of the field (time t and $t + \Delta t$) with many particles are taken by a CCD camera. Then, the speed and direction of movement of the particles in the field are measured by comparing the pairs of images.

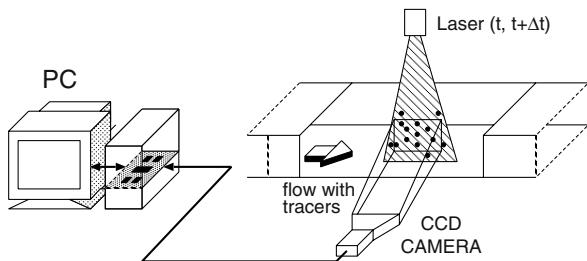


Fig. 1. System Overview

Figure 2 shows a pair of images of the field (t and $t + \Delta t$). Suppose that we want to know where particles in a window (x, y) (a rectangular from (x, y) to $(x+n-1, y+m-1)$) at time t are moving to, and how fast they are moving. Then, the window is compared with all windows in a target area (light gray square in Figure 2(B)), and the most similar window (dark gray square) is searched using cross-correlation function, and the direction of movement and the speed of the particles are obtained.

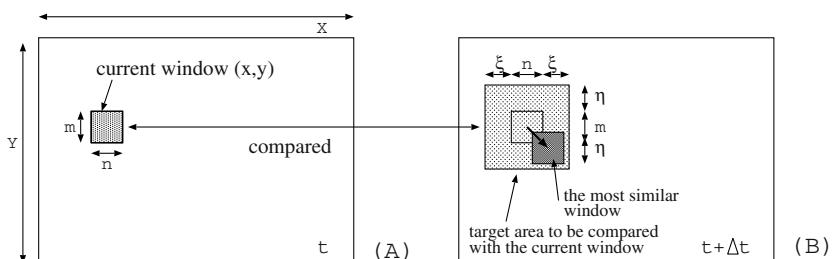


Fig. 2. Particle Image Velocimetry (PIV) Method

The cross-correlation function between two windows (window (x, y) at time t and window $(x + \xi, y + \eta)$ at time $t + \Delta t$) can be described as follows. In this function, I_1 and I_2 show values of pixels of images at time t and $t + \Delta t$ respectively.

$$R(x, y, \xi, \eta) = \frac{\sum_{0}^{n-1} \sum_{0}^{m-1} \{I_1(x+i, y+j) - \bar{I}_1\} \{I_2(x+i+\xi, y+j+\eta) - \bar{I}_2\}}{\sqrt{\sum_{0}^{n-1} \sum_{0}^{m-1} \{I_1(x+i, y+j) - \bar{I}_1\}^2} \sqrt{\sum_{0}^{n-1} \sum_{0}^{m-1} \{I_2(x+i+\xi, y+j+\eta) - \bar{I}_2\}^2}}$$

where $\bar{I}_1 = \sum_{0}^{n-1} \sum_{0}^{m-1} I_1(x + i, y + j) / nm$

$$\bar{I}_2 = \sum_{0}^{n-1} \sum_{0}^{m-1} I_2(x + i + \xi, y + j + \eta) / nm$$

The computational complexity of the cross-correlation is very huge, and can not be processed in real-time by micro-processors. In our system, an FPGA board (ADC XRC-II by Alpha Data with one XC2V6000) with one additional SRAM board is used to support eight memory banks (each bank has 4MB (or 8MB) memory with 32 bit width). These eight memory banks are necessary to realize real-time processing by an improved direct computation method described below without reducing data-width of the cross-correlation function.

3 Computation of the Cross-Correlation Function

3.1 Requirement for Real-Time Processing

In our target PIV system[1], the size of images is 1008×1008 , and 20 pairs of images are compared in one second. The size of windows is up to 30×30 , and the range of ξ and η is $[-10 \text{ to } 10]$. Suppose that the window size is 24×24 , then the number of windows which have to be searched in one pair of images becomes $84(1008/24 * 2) \times 84$ (the position of the windows is moved to right and down by the half of the window size). Under these assumptions, we need to find a pair of the most similar windows in 470 clock cycles in 66 MHz operation frequency in order to realize real-time processing.

3.2 Computation Based on FFT

FFT is often used to compute cross-correlation functions. In FFT, size of the windows has to be $2^k \times 2^l$. Thus, in order to satisfy the requirements for real-time processing, window size must be 64×64 (because the size of the window + the range of ξ and η is larger than 32 and less than 64). As for windows at time t , enlarged area (from 24×24 to 64×64) is filled with 0, while 64×64 pixels are clipped off from images as for windows at time $t + \Delta t$. Figure 3 shows the outline of the computation based on FFT. In Figure 3, FFT values of two windows (time = t and $t + \Delta t$) are first calculated, and the FFT values are

multiplied. Then, IFFT values of the multiplied values are calculated and the maximum value is selected to find out the most similar window. In order to discuss the computation time, we focus on the number of multiply operations which is the most time exhaustive (and hardware resource exhaustive) part in the function. As shown in Figure 3, the number of multiply operations required is

$$(((32 \times 4) \times \log_2 64) \times 64) \times 2 \times 3 + 64 \times 64 \times 4$$

when the row-column two dimensional FFT is used, because one multiply operation of complex numbers requires four integer multiply operations. The key point in the computation based on FFT is that the number of the operations does not change while the size of windows + the range of ξ and η is less than 64 (and larger than 32).

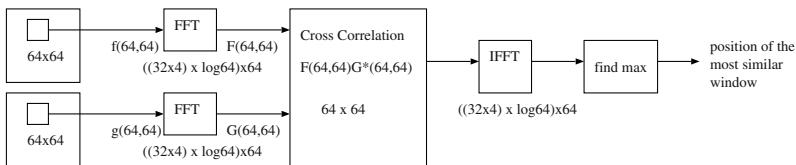


Fig. 3. Outline of the Computation of Cross Correlation Function based on FFT

Our current target FPGA is XC2V6000 (one of the largest FPGAs which is available now). With this FPGA,

1. we can execute 128 multiply operations in parallel because XC2V6000 has 144 built-in multipliers,
2. other add/sub operations can be executed in parallel and in pipeline with the multiply operations,
3. FFT and IFFT can be executed on the same butterfly circuit (with a connection based on omega multi-stage network) by changing only the constant values used for the multiply operations (these values are stored in Block RAMs, and can be easily changed by changing the addresses to the Block RAMs), and
4. 64×64 multiply operations for cross-correlation can be executed on the same butterfly circuit.

Thus, the minimum computation time becomes 2432 clock cycles ignoring any overhead for pipeline processing.

3.3 Direct Computation

Direct computation is often used when window size is small. The total number of multiply operations in this method becomes

$$n \times n \times r \times r$$

where $n \times n$ is the size of windows, and r is the range of ξ and η . When the n is 24 and r is 21, the minimum computation time becomes 1764 clock cycles on XC2V6000 because 144 multiply operations can be executed in parallel, and other add/sub operations can be executed in parallel and pipeline with the multiply operations. This is faster than the computation based on FFT, but still slower than the requirement for real-time processing. The important point in the direct computation is that the computation time depends on the size of windows and the range of ξ and η . If ξ and η vary from -16 to 16, the computation time becomes 4356 clock cycles, which is much slower than the computation based on FFT.

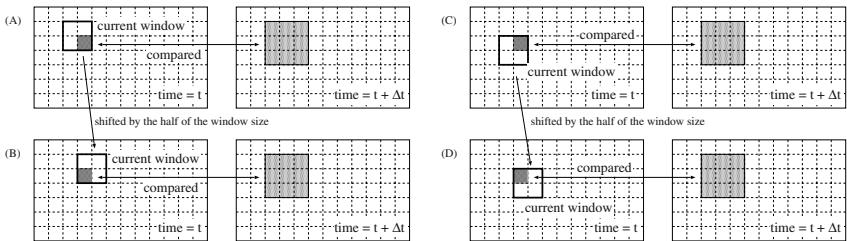


Fig. 4. Redundant Computation in the Direct Computation Method

In case of the direct computation, we can reduce the computation time to 25% of the clock cycles described above, when the CCD camera of the PIV system is placed just above the field of flows and positions of windows are moved to right and down by the half of window size. In Figure 4(A), dark gray part of the current window is compared with light gray part of the image of $t + \Delta t$. Then, the current window is shifted to right by $n/2$, and dark gray part in Figure 4(B) (same with the dark gray part in Figure 4(A)) is compared with the same light gray part again. This dark gray part is compared two more times as shown in Figure 4(C) and (D). In order to avoid these redundant computations, current windows are divided to four sub-windows in our implementation. These sub-windows are calculated first, and their results (intermediate data) are stored in memory. Then, the intermediate data are read back from memory and the cross-correlation for original current windows are calculated. With this optimization, we can reduce the computation clock cycles to 25% of the non-optimized method. Thus, the clock cycles are reduced to 441 clock cycles which satisfy the requirement for real-time processing.

3.4 FPGA Size and Better Computation Method

Because of the reason described above, we used the improved direct computation method for our real-time system. The continuous progress of the size of FPGAs will, however, make it possible to achieve real-time processing by both computation methods in a few years. Then, the computation based on FFT is superior

to the direct computation, because in the computation based on FFT, we do not have to care about window size and the range of ξ and η unless sum of them exceed 64 (which is not common case in PIV applications). Users can change window size and the range of the variables at any time of their experiments without changing circuit on the FPGA.

4 Details of the PIV System

Eight memory banks supported in our system, however, is not enough for naive implementation of the optimized direct computation. In this section, we describe the implementation of the optimized direct computation method.

4.1 Details of the Improved Direct Computation

From the view point of the optimization described above, the cross-correlation function for window (x, y) and window $(x+\xi, y+\eta)$ can be transformed as follows (first term of the denominator in the previous definition is deleted because it is a constant).

$$R(x, y, \xi, \eta) = \frac{MAC12(x, y, \xi, \eta) - SUM1(x, y) \times SUM2(x, y, \xi, \eta) / nm}{\sqrt{MAC22(x, y, \xi, \eta) - SUM2(x, y, \xi, \eta) \times SUM2(x, y, \xi, \eta) / nm}}$$

where

$$\begin{aligned} MAC12(x, y, \xi, \eta) &= mac12(x, y, \xi, \eta) + mac12(x, y + m/2, \xi, \eta) \\ &\quad + mac12(x + n/2, y, \xi, \eta) + mac12(x + n/2, y + m/2, \xi, \eta) \\ MAC22(x, y, \xi, \eta) &= mac22(x, y, \xi, \eta) + mac22(x, y + m/2, \xi, \eta) \\ &\quad + mac22(x + n/2, y, \xi, \eta) + mac22(x + n/2, y + m/2, \xi, \eta) \\ SUM1(x, y) &= sum1(x, y) + sum1(x, y + m/2) + sum1(x + n/2, y) \\ &\quad + sum1(x + m/2, y + n/2) \\ SUM2(x, y, \xi, \eta) &= sum2(x, y, \xi, \eta) + sum2(x, y + m/2, \xi, \eta) \\ &\quad + sum2(x + n/2, y, \xi, \eta) + sum2(x + m/2, y + n/2, \xi, \eta) \\ mac12(x, y, \xi, \eta) &= \sum_{0}^{n/2-1} \sum_{0}^{m/2-1} \{I_1(x+i+\xi, y+j+\eta) \times I_2(x+i+\xi, y+j+\eta)\} \\ mac22(x, y, \xi, \eta) &= \sum_{0}^{n/2-1} \sum_{0}^{m/2-1} \{I_2(x+i+\xi, y+j+\eta) \times I_2(x+i+\xi, y+j+\eta)\} \\ sum1(x, y) &= \sum_{0}^{n/2-1} \sum_{0}^{m/2-1} I_1(x+i, y+j) \\ sum2(x, y, \xi, \eta) &= \sum_{0}^{n/2-1} \sum_{0}^{m/2-1} I_2(x+i+\xi, y+j+\eta) \end{aligned}$$

Therefore, by storing four kinds of values ($mac12$, $mac22$, $sum1$ and $sum2$) in memory banks (one value for $sum1$ and 21×21 values for $mac12$, $mac22$ and $sum2$ in each sub-window because ξ and η vary from -10 to 10) and reading them out afterward, we can compute the cross-correlation of original windows. If we can store the results of four sub-windows in different memory banks, they can be read out at the same time for the computation of original windows. However, eight memory banks in our system is not enough to store them in different memory banks.

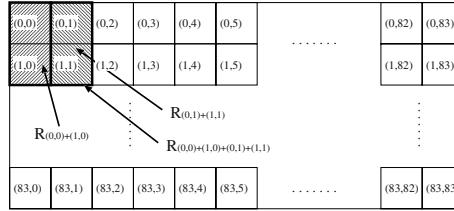


Fig. 5. An Image divided to Sub-windows

Figure 5 shows an image of the field of flows (1008×1008) divided to 84×84 sub-windows (size is 12×12). Sub-windows are processed from $(0, 0)$ to $(83, 83)$, and results of sub-windows $(n, k)_{n \% 2=0}$ are stored in memory banks (A), while results of sub-windows $(n, k)_{n \% 2=1}$ are stored in memory banks (B). In our implementation, results of only two sub-window rows are stored in memory banks (A) and (B) (results of sub-window (n, k) are overwritten by the results of sub-window $(n + 2, k)$).

In Figure 5, suppose that computation of sub-windows $(0, k)_{k=0,83}$ are finished. In the computation of the next sub-window $(1, 0)$,

1. $R_{(1,0)}$ (results of sub-window $(1, 0)$) are calculated and written to memory banks (B),
2. $R_{(0,0)}$ (results of sub-window $(0, 0)$) are read from memory banks (A),
3. $R_{(0,0)}$ and $R_{(1,0)}$ are added ($R_{(0,0)+(1,0)}$ are obtained) and stored in a temporal memory in the FPGA.

Then, in the computation of the next sub-window $(1, 1)$,

1. $R_{(1,1)}$ are calculated and written to memory banks (B),
2. $R_{(0,1)}$ are read from memory banks (A),
3. $R_{(0,1)}$ and $R_{(1,1)}$ are added ($R_{(0,1)+(1,1)}$),
4. $R_{(0,1)+(1,1)}$ are added to $R_{(0,0)+(1,0)}$ in the temporal memory ($R_{(0,0)+(1,1)+(0,1)+(1,1)}$ are obtained), and $R_{(0,1)+(1,1)}$ are stored in the temporal memory instead of $R_{(0,0)+(1,0)}$,
5. the cross-correlation function of the original window is calculated using $R_{(0,0)+(1,0)+(0,1)+(1,1)}$

By repeating this procedure to all sub-windows, we can calculate cross-correlation function of all windows. In this processing, only three banks of memory (memory banks (A), memory banks (B) and one internal temporary memory) are used, which can be supported using eight external memory banks and the FPGA in our system.

4.2 Details of the Circuit

Figure 6 shows the block diagram of the circuit implemented on the FPGA. Pairs of images of the field of flows are sent from the CCD camera and stored in SRAM bank0 and bank1. Two bank of SRAM are used to avoid memory access conflicts by the FPGA and the camera. When one pair of images in bank0(bank1) is processed by the FPGA, next pair of images is transferred to bank1(bank0).

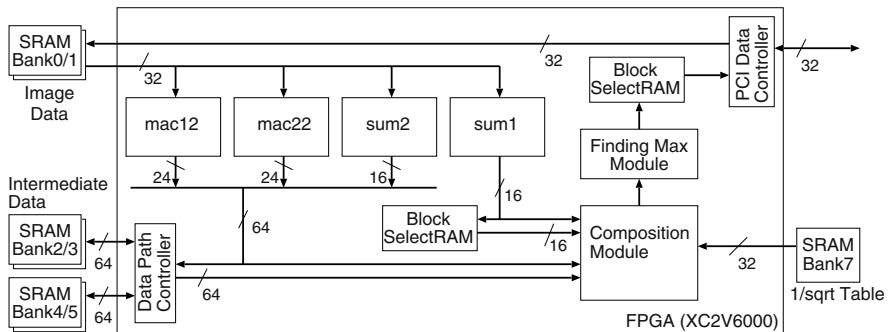


Fig. 6. Block Diagram of the Circuit

The circuit on the FPGA consists of six modules; *mac12* module, *mac22* module, *sum1* module, *sum2* module, composition module and finding max module. In *mac12*, *mac22*, *sum1* and *sum2* modules, values of *mac12*, *mac22*, *sum1* and *sum2* of sub-windows are calculated respectively. Outputs of *sum1* module are stored in one Block RAM, because the number of the outputs for 84 sub-windows ($(n, k)_{k=0,83}$) is 84 with 16 bit-width, and Block RAMs support dual-port access. Two sets of two memory banks (bank2/3 and bank4/5) are used for memory banks (A) and (B) because the number of outputs by *mac12*, *mac22* and *sum2* for 84 sub-windows is $84 \times 21 \times 21$ with 64 bit-width in total. The cross-correlation function for windows is calculated in the composition module using values stored in the Block RAM and the memory banks (bank 2/3 and 4/5). Bank7 is used to compute $1/\sqrt{}$ in the function. In finding max module, the maximum value of the cross-correlation function is selected and stored into the Block RAM, and the values in the Block RAM are sent back to the host computer.

Figure 7 shows the structure of the composition module. Inputs to this module are *sum1* from the Block RAM and *sum1* module, and *mac12/mac22/sum2*

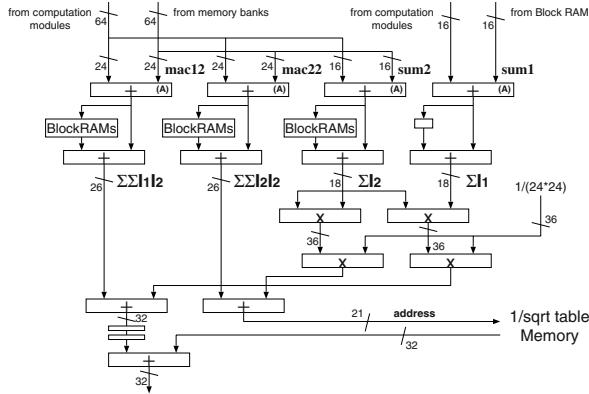


Fig. 7. Composition Module

from the memory banks and *mac12/mac22/sum2* modules. The values from the memories and the computation modules are added (by adders (A)) to obtain $R_{(n,k)+(n+1,k)}$. Then, the results are stored in Block RAMs, and added with the direct outputs of the adders (A) to obtain $R_{(n,k)+(n+1,k)+(n,k+1)+(n+1,k+1)}$. Then, the cross-correlation function of window is calculated. The only difference from the computation by micro-processors is that the address to the $1/\sqrt{t}$ table is reduced to 21 bits because the memory size of the external memory bank (this cause no difference in the results).

Figure 8 shows the basic idea of the implementation of *mac12* module[2]. In Figure 8, the size of sub-windows and the range of ξ and η are reduced to 4×4 to simplify the figure. Data of a sub-window (at time = t) are stored in a shift register array and data of its target area (at time = $t + \Delta t$) are stored in a distributed RAM array. Data on these two arrays are multiplied in parallel using built-in multipliers (in real *mac12* module, 144 multiply operations are executed in every clock cycle), and added by pipelined binary tree adders.

Each element in the distributed RAM array stores nine pixel data (for example, LUTs (8 bit-width) labeled 0 (gray part in Figure 8(1)) store nine pixel data labeled 0 (gray parts in target area)). By shifting data in the shift register array to right and down, and by changing addresses to each row and column of the distributed RAM array, we can compute *mac12* efficiently. In Figure 8(2), multiply-and-accumulate of the sub-window on the shift register array and the gray part in the target area is computed. For this computation, data on the shift register array are shifted five times to right and once to down from Figure 8(1), and addresses to each row and column of the distributed RAM array are given as shown in Figure 8(2) to read out the gray part.

In our implementation, two sets of shift register array and distributed RAM array are used in order to update one of them while the other is used to compute cross-correlation function.

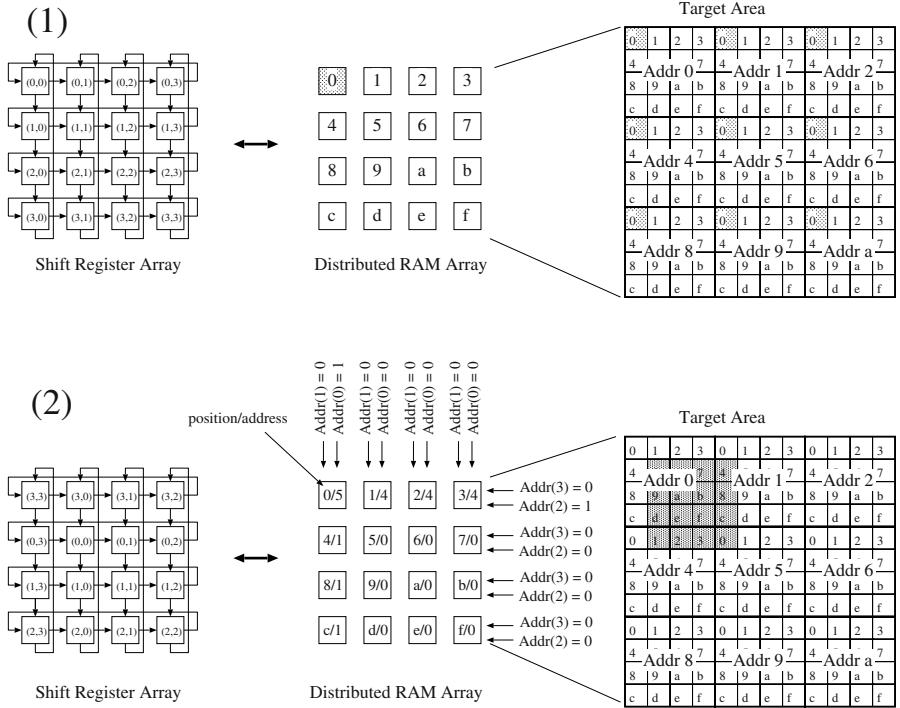
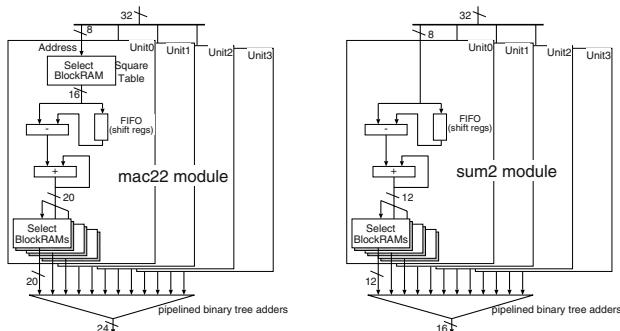
Fig. 8. Basic Idea of *mac12* module

Figure 9 shows the structure of *mac22* and *sum2* modules. In *mac22* and *sum2* modules, multiply-and-accumulate and sum of all columns in all sub-windows in the target area ($\sum_{k=y}^{y+11} I_2(x, k) \times I_2(x, k)$ and $\sum_{k=y}^{y+11} I_2(x, k)$) are calculated when data of the target area are loaded to the distributed RAM arrays, and stored in Block RAMs. During the computation of the cross-correlation

Fig. 9. *mac22* and *sum2* modules

function, 12 column data for $mac22$ and $sum2$ are read out in parallel from Block RAMs respectively, and added by binary tree adders to calculate $mac22$ and $sum2$ of the sub-windows.

5 Performance

In our current implementation on XC2V6000, the number of clock cycles to calculate cross-correlations of a sub-window at time t and all sub-windows in its target area (time $t + \Delta t$) is 916 for the first sub-window (sub-window (0, 0) in Figure 5) to full-fill all pipeline stages of the circuit, and 441 for other $84 \times 84 - 1$ sub-windows. Therefore, the average clock cycles to find a pair of the most similar windows is almost 441 (less than 470 clock cycles; requirement for real-time processing). Because of this clock cycles, the required operational frequency for real-time processing is relaxed to 62.3MHz. The circuit on XC2V6000 runs at 66 MHz (the maximum operational frequency reported by the CAD is 66.5 MHz). This performance satisfies the requirements for real-time processing.

In the current implementation, all multipliers, 32% of slices and 64 Block RAMs (out of 144) in XC2V6000 are used.

6 Conclusions

In this paper, we described a real-time visualization system of PIV method, which consists of one off-the-shelf FPGA board with one Virtex-II FPGA and a host computer. By using one latest FPGA and an improved direct computation method of the two dimensional cross-correlation function, we could realize real-time processing without reducing data bit-width of the function. Because of the built-in multiplier, the circuit which includes parallel and pipelined multiply-and-accumulate operations for 144 data occupied only 32% of the XC2V6000.

The assumptions for the improvement method, however, can not be always satisfied. We are now developing a circuit to transform a view from a camera which is not placed just above the field of flows to a view from just above the field using the rest area of XC2V6000.

References

1. J. Sakakibara and T. Anzai, "Chain-link-fence structures produced in a plane jet", Physics of Fluids 2001
2. T. Maruyama, Y. Yamaguchi and A. Kawase, "An Approach to Read-time Visualization of PIV Method with FPGA", FPL2001

A Real-Time Stereo Vision System with FPGA

Yosuke Miyajima and Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba,
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 Japan,
miyajima@darwin.esys.tsukuba.ac.jp

Abstract. In this paper, we describe a compact stereo vision system which consists of one off-the-shelf FPGA board with one FPGA. This system supports (1) camera calibration for easy use and for simplifying the circuit, and (2) left-right consistency check for reconstructing correct 3-D geometry from the images taken by the cameras. The performance of the system is limited by the calibration (which is, however, a must for practical use) because only one pixel data can be allowed to read in owing to the calibration. The performance is, however, 20 frame per second (when the size of images is 640×480 , and 80 frames per second when the size of images is 320×240), which is fast enough for practical use such as vision systems for autonomous robots. This high performance can be realized by the recent progress of FPGAs and wide memory access to external RAMs (eight memory banks) on the FPGA board.

1 Introduction

The aim of stereo vision systems is to reconstruct the 3-D geometry of a scene from two (or more) images, which we call *left* and *right*, taken by cameras. Many dedicated hardware systems have been developed for real-time processing, and a stereo vision system with FPGA[1] has achieved real-time processing because of the recent progress of the size and the performance of FPGAs.

Compact systems for stereo vision are especially important for autonomous robots. FPGAs are ideal devices for the compact systems. Depending on situations, a robot may try to reconstruct the 3-D geometry, to find out moving objects which are coming to it, and to find out marker objects to check its position. FPGAs can support all these functions by reconfiguration.

In this paper, as the first step toward a vision system for autonomous robots, we describe a compact real-time stereo vision system which

1. supports camera calibration for easily obtaining correct results with simple circuit, and
2. checks *Left-Right Consistency* to find out occlusions without duplicating the circuit (by only adding another circuit for finding minimum value).

These functions are very important to obtain correct 3-D geometry. This system also supports filters for smoothing and eliminating noises to improve the system performance. In order to achieve these functions while exploiting maximum

performance of FPGAs (avoiding memory access conflicts), we need one latest FPGA and eight memory banks on the FPGA board which are just supported on latest off-the-shelf FPGA boards.

The performance of the system is more than 20 frames per second (640×480 inputs and 640×480 output with disparity up to 200), which is much faster than previous works (more than 80 frames if the size of images is 320×240).

This paper is organized as follows. Section 2 describes the overview of stereo vision systems, and details of our system is given in section 3. The performance of the system is discussed in section 4. In section 5, conclusions are given.

2 Overview of Stereo Vision Systems

In order to reconstruct the 3-D geometry of a scene from two images (left and right) taken by two cameras, it is searched which pixels in the two images are projections of the same locations in the scene in the stereo vision systems. In this section, we first discuss the calibration of cameras, which is a very important step to simplify matching computation which is the most time exhaustive part in stereo vision systems, and then discuss matching algorithms and *left-right consistency* to suppress infeasible matches.

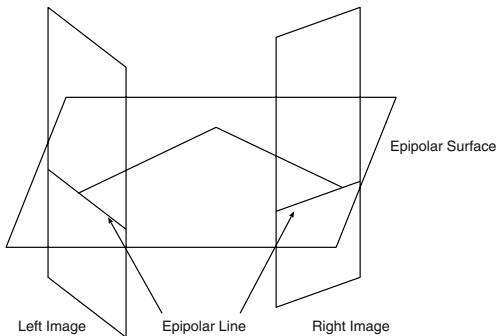
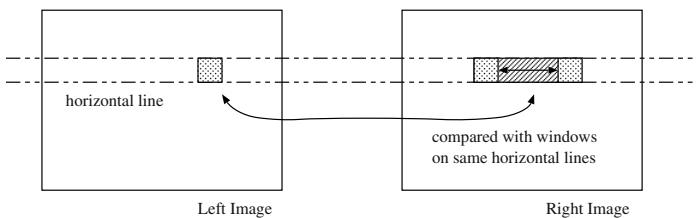
2.1 Calibration

Even if the same type of cameras are used to obtain left and right images, the characteristics of the cameras are different, and horizontal (vertical) lines in real-world may not be horizontal (vertical) in the images taken by cameras. The aim of the calibration is to find out a relationship (perspective projection) between the 3-D points in real-world and their different camera images. This is a crucial stage in order to simplify the following stages in the stereo vision systems and to obtain correct matching.

2.2 Matching Algorithm

Area-based (or correlation-based) algorithms match small windows centered at a given pixel to find corresponding points between the two images. They yield dense depth maps, but fail within occluded areas. Feature-based algorithms match local cues (e.g., edges, lines, corners) and can provide robust, but sparse disparity maps which requires interpolation. In hardware systems, area-based algorithms are widely used, because the operations required in those algorithms are very regular and simple.

In the area-based algorithms, *epipolar restriction* is used in order to decrease computational complexity. As shown in Figure 1, the corresponding point of a given point lies on its epipolar line in the other image, when the two cameras are arranged so that their principal axes are parallel. Corresponding points can then be found by comparing with every points on the epipolar line on the other image. To use this restriction, calibration of cameras is necessary to guarantee

**Fig. 1.** Epipolar Geometry**Fig. 2.** Stereo Matching on Epipolar Constraint

that objects on a horizontal line in real-world also lie in the same horizontal lines in left and right images taken by the cameras. Then, we need to only compare windows on same horizontal lines in left and right images as shown in Figure 2.

The most traditional area-based matching algorithm is normalized cross-correlation [3], which requires more computation time than the following simplified algorithms. The most common pixel-based matching algorithm are squared intensity differences (SSD)[2] and absolute intensity differences (SAD)[4]. We used the SAD (Sum of Absolute Difference) algorithm because the algorithm is the simplest among them, and the results obtained by the algorithm is almost same with other algorithms[4]. In SAD algorithm, the value of d which minimizes the following equation is searched.

$$\sum_{i=-n}^n \sum_{j=-m}^m |I_r(x+i, y+j) - I_l(x+i+d, y+j)|$$

In the equation, I_r and I_l are the right and left image respectively, n and m are the size of the window centered at a given pixel (its position is x and y). d is the disparity, and its range decides how many pixels on the other image are compared with the given pixel. In order to find the corresponding points for an object which is closer to the vision system, larger range for d becomes necessary, though it requires more hardware resources.

2.3 Occlusion and Left-Right Consistency

When pairs of images of objects are taken by two cameras (left and right), some parts of the objects appear in left (right) images and may not appear in right (left) images, depending on the positions and angles between the cameras and the objects. These occlusions are major source of errors in computational stereo vision systems, though it has been reported that these occlusions help the human visual system in detecting object boundaries[5].

In many computational systems, one of left and right images is chosen as the base of the matching. Then, windows which include target pixels are selected in the base image, and the most similar windows are searched in another image. If the role of left and right images is reversed, different pairs of the windows may be selected. The so-called *left-right consistency constraint*[6] states that feasible window pairs are those found with both direct and reverse matching.

In our system, occlusions are detected by checking the left-right consistency. Figure 3 shows left image based matching and right image based matching. These matching can be executed in our system without duplicating whole circuit (by adding only another module which consists of comparators and selectors).

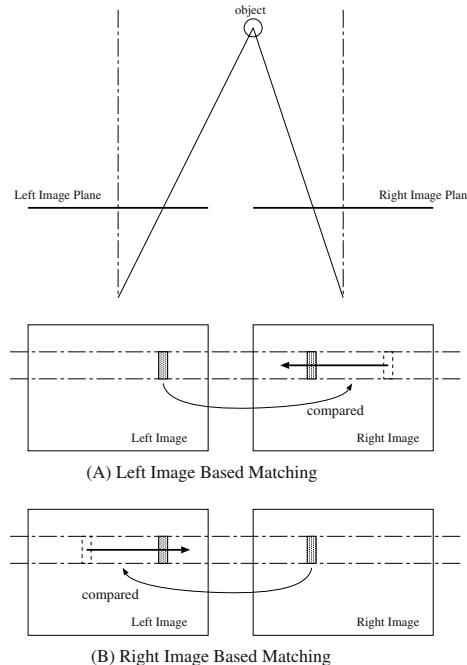


Fig. 3. Left Based / Right Based Matching

2.4 Filters

Filters are often used in the stereo matching for smoothing and eliminating noises to improve the system performance. We prepared Laplacian of Gaussian (LoG) filter for that purpose. Dual-port block RAMs make it possible to implement filters efficiently.

3 Details of the System

3.1 Overview

Figure 4 shows the system overview. Our system consists of a host computer, two cameras and one off-the-shelf FPGA board (ADM-XRC-II by Alpha Data with one additional SSRAM board). Left and right images taken by the two cameras are sent to external RAMs on the FPGA board.

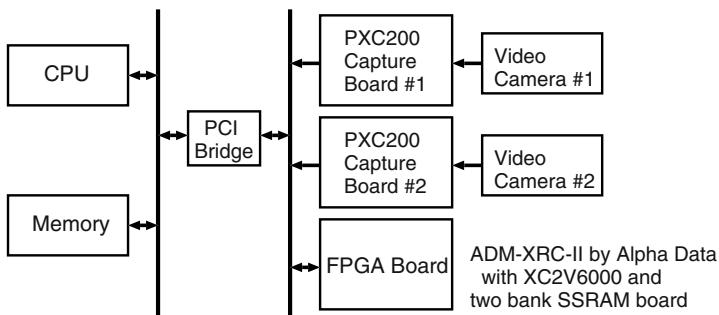


Fig. 4. System Overview

Figure 5 shows the structure of the FPGA board. The board has eight external memory banks (including two memory banks by the additional SSRAM board) which can be accessed independently. The first pair of images (left and right) sent from cameras are stored in bank0 and bank2 respectively, and next pair of images are stored in bank1 and bank3 while the images in bank0 and bank2 are processed by the circuit on the FPGA. The results by the circuit are written back to bank4 and bank5. When the data in bank4 is being sent to the host computer, FPGA writes new results to bank5. In order to exploit maximum performance of FPGA by avoiding memory access conflicts on these external memory banks, we need six memory banks for stereo matching its self. The rest two banks (bank6 and bank7) are used for the calibration described below. Thus, we need at least eight memory banks for our stereo vision system.

3.2 Calibration

In our system, calibration is performed using images of a predefined calibration grid. Before starting the system, the grid is given to left and right cameras (po-

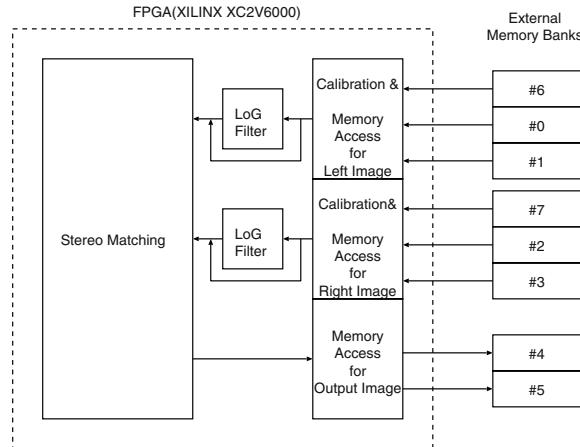


Fig. 5. FPGA Board

sitions of the cameras and the grid are fixed in advance), and the images of the grid taken by both cameras are sent to the host computer. Then, the host computer calculates which pixels on left and right images should be compared, and the positions of the pixels which should be compared are sent back to external RAMs on the FPGA board. These pixel position informations for left and right image are stored in bank6 and bank7, respectively (the size of the information is same with the size of images). In the later matching stages, FPGA first reads out the positions of the pixels which should be compared from the bank6 and bank7, and then the pixel data from bank0/1 and bank2/3 are read out using the positions.

This function is very important to obtain correct 3-D geometry with simple matching circuit, but this allows us to read only one pixel data in each clock cycle from bank0/1 and bank2/3, because the next pixel data which should be compared may not lies in the next address of the image data (when horizontal line in the image do not correspond to the true horizontal lines in real-world owing to the distortion of the lens of the camera and so on). Because of this restriction, the system performance is limited by the access time to the external RAMs on the FPGA board, and can not be improved by providing wider access to the external RAMs.

3.3 Matching and Left-Right Consistency

Figure 6 shows the outline of the matching circuit. In Figure 6, suppose that window size is $n \times n$, and column data of windows (n pixel data which are shown as Li and Ri in Figure 6) are read out at once in order to simplify the figure (in the actual system, only one pixel data is read out at once as described above).

Column data of windows in left image are broadcasted to all column modules, while column data of windows in right image are delayed by registers and then

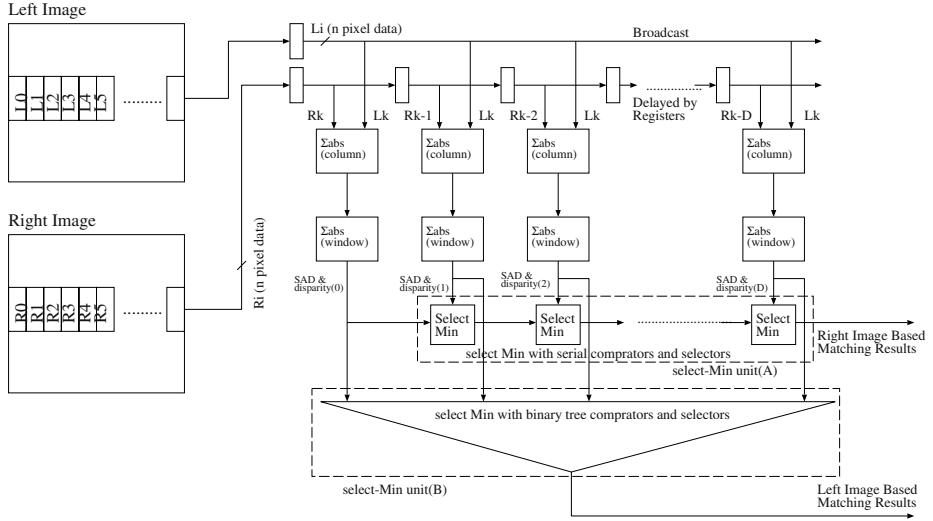


Fig. 6. Outline of the Matching Circuit

given to the column modules. In the column modules, sum of absolute difference of one column data is calculated. The outputs by the column modules are sent to window modules to sum up n values (thus, sum of absolute difference of one window is calculated).

Outputs by the window modules are compared and minimum value and its disparity are selected by two kinds of units. In the select-minimum unit(A), outputs by the window modules are shifted (with delay), and compared with the outputs by the next window module. The smaller value and its disparity are selected and shifted to the next compare module. Then, the output by the last select-min module gives the minimum of sum of absolute difference and its disparity when the right image is chosen as base for the matching. In another select-minimum unit(B), all outputs by the window modules are compared by binary tree comparators and selectors, and minimum value and its disparity are selected. The output by this unit gives the minimum of sum of absolute difference and its disparity when the left image is chosen as base for the matching.

Figure 7 shows the outputs by the window modules when the window size is 5×5 . In Figure 7, by comparing outputs of the window modules with shifting and delaying (parts covered by slanting lines in Figure 7), one window in the right image (window {R6-R10}) is compared with windows in the left image({L6-L10},{L7-L11},{L8-L12}...), while one window in left image({L11-L15}) is compared with windows in the right image({R11-R15},{R10-R14},{R9-R13}...) by comparing outputs of window module at the same time (gray parts in Figure 7).

As described above, left-right consistency check (left image based matching and right image based matching) can be executed by only adding another com-

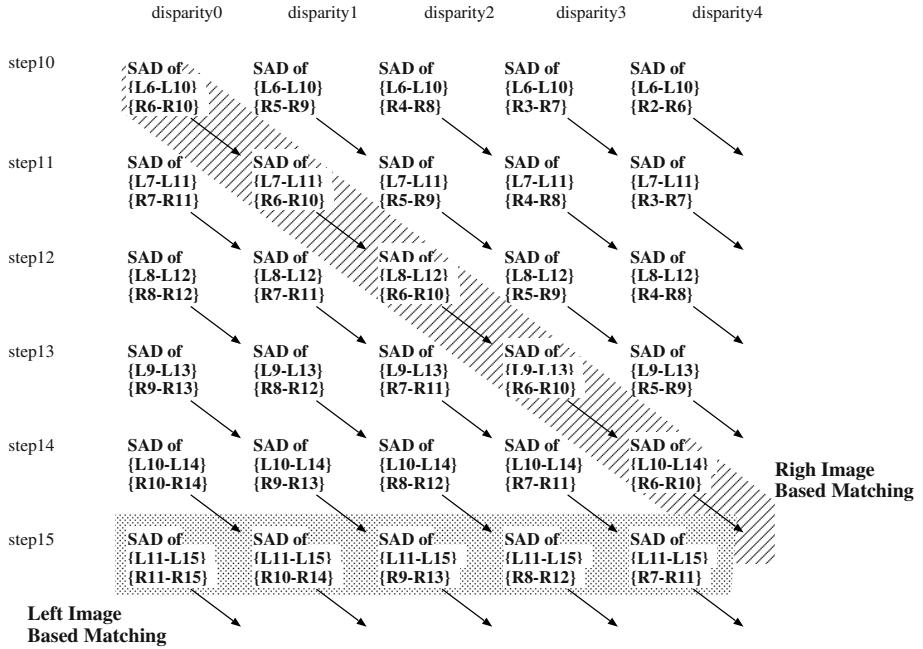


Fig. 7. Left-Right Consistency

pare unit which requires only $D-1$ comparators and selectors when D (number of window modules, namely maximum disparity) is 2^k .

3.4 Details of the Modules

Figure 8 shows the details of the column module and the window module. In the column module, absolute difference of inputs from left image and right image (one pixel in each clock cycle as described above) is calculated, and summed up for n times when the window size is $n \times n$. The outputs of the column module are sent to the window module, and n outputs are summed up again to calculate the SAD(sum of absolute difference) of $n \times n$ window. In the window module, new input value is accumulated, and input at (current_step - n) step is subtracted instead of adding n previous values in order to reduce circuit size (previous values are stored in shift registers)[7]. As shown in Figure 7, the output of the window module is, for example, SAD of window {L6-L10}{R6-R10} at step 10, and SAD of {L7-L11}{R7-R11} at step 11. In this case, SAD of window {L7-L11}{R7-R11} can be calculated by the following equation.

$$\begin{aligned} \text{SAD of } & \{L7-L11\}\{R7-R11\} = \\ & \text{SAD of } \{L6-L10\}\{R6-R10\} - \text{SAD of } \{L6\}\{R6\} + \text{SAD of } \{L11\}\{R11\} \end{aligned}$$

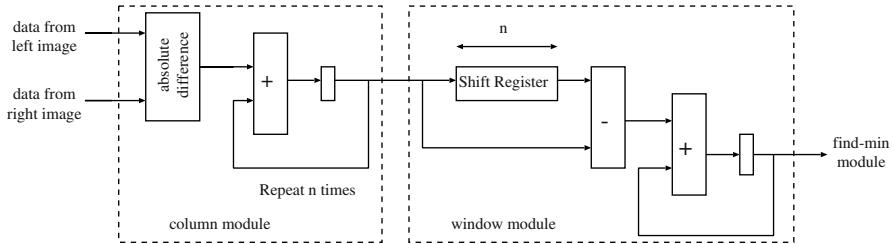


Fig. 8. Details of the Column Module and the Window Module

4 Performance

In our system, major factor that decreases system performance is window size (because only one pixel is read from external memory owing to calibration, it takes n clock cycles to read one column data for the window), and factors that increase the circuit size is the maximum value of disparity which decide the number of modules.

Table 1 shows the system performance against the window size. The image size used for the evaluation is 640×480 . As described above, the performance becomes worse as the window size becomes larger. The most often used window size in stereo vision systems is 7×7 or 9×9 . The performance in Table 1 is calculated based on the maximum frequency reported by CAD. In practice, the system could process 20 frames per second in those window sizes, which is fast enough for autonomous robots. When, we need more performance, we can reduce the image size to 320×240 (which is widely used in other stereo vision systems). Then, the performance becomes four times faster without changing the circuit.

Table 2 shows the performance when we changed the maximum disparity. In this case, the circuit size becomes larger as the maximum disparity becomes larger, though the performance does not change as described above. Maximum disparity 200 is quite large compared with other stereo vision systems.

Table 1. Window Size and the Performance

Window Size	15×15	13×13	11×11	9×9	7×7	5×5
Performance (frames per second)	8.7	10.0	11.8	14.5	18.9	26.0

The size of left and right image is 640×480 , and maximum disparity is 80.

Table 2. Maximum Disparity and Circuit Size

Maximum Disparity	80	160	200
Circuit Size	21%	43%	54%
Performance (FPS)	18.9	18.9	18.9
Operation Frequency (MHz)	40	40	40

The size of left and right image is 640×480 , and window size is 7×7 .

5 Conclusions

In this paper, we described a compact stereo vision system with one off-the-shelf FPGA board with one FPGA. This system supports (1) camera calibration for easy use and for simplifying the circuit, and (2) left-right consistency check for reconstructing correct 3-D geometry. The performance of the system is limited by the calibration (which is a must for practical use) because only one pixel data can be allowed to read in owing to the calibration. The performance is, however, 20 frame per second (when the size of images is 640×480), which is fast enough for practical use such as vision systems for autonomous robots. The operation frequency of the system is still very slow. We are now improving the details of the circuit. We think that we can process more than 30 frames per second by this improvement.

This system became possible because of the continuous progress of FPGAs. We needed at least eight memory banks on the FPGA board to exploit maximum performance of FPGA avoiding memory access conflicts on the memory banks on the FPGA board while supporting calibration. We also needed the latest FPGA to support very large maximum disparity.

References

1. M.Arias-Estrada, J.M.Xicotencatl, "Multiple Stereo Matching Using an Extended Architecture", FPL:2003-212, 2001.
2. P.Anandan, "A computational framework and an algorithm for the measurement of visual motion", IJCV, 2(3):283-310,1989.
3. T.W.Ryan, R.T.Gray, and B.R.Hunt, "Prediction of correlation errors in stereo-pair images", Optical Engineering, 19(3):312-322, 1980.
4. T.Kanade, "Development of a video-rate stereo machine" in IUW, pp. 549-557. 1994.
5. K.Nakayama and S.Shimojo, "Da Vinci stereopsis: Depth and subjective occluding contours from unpaired image points", Vision Research, 30:1811-1825, 1990
6. P.Fua, "Combining stereo and monocular information to compute dense depth maps that preserve depth discontinuities", IJCAI, 1991.
7. O.Faugeras, B.Hotz, H.Mathieu, T.Viville, Z.Zhang, P.Fua, E.Thron, L.Moll, G.Berry, J.Vuillemin, P.Bertin, and C.Proy, "Real time correlation-based stereo: algorithm, implementations and application.", Tech. Rep. 2013, INRIA, August 1993.

Synthesizing on a Reconfigurable Chip an Autonomous Robot Image Processing System*

Jose Antonio Boluda and Fernando Pardo

Departament d'Informàtica, Universitat de València,
Avda. Vicent Andrés Estellés S/N 46100 Burjassot, Spain,
`{Jose.A.Boluda,Fernando.Pardo}@uv.es, http://tapec.uv.es`

Abstract. This paper deals with the implementation, in a high density reconfigurable device, of an entire log-polar image processing system. The log-polar vision reduces the amount of data to be stored and processed, simplifying several vision algorithms and making it possible the implementation of a complete processing system on a single chip. This image processing system is specially appropriated for autonomous robotic navigation, since these platforms have typically power consumption, size and weight restrictions. Furthermore, the image processing algorithms involved are time consuming and many times they have also real-time restrictions. A reconfigurable approach on a single chip combines hardware performance and software flexibility and appears as specially suited to autonomous robotic navigation. The implementation of log-polar image processing algorithms as a pipeline of differential processing stages is a feasible approach, since the chip incorporates RAM memory enough for storing several full log-polar images as intermediate computations. Two different algorithms have been synthesized into the reconfigurable device showing the chip capabilities.

1 Introduction

The incorporation of visual sensing into an autonomous robot is a desirable objective, since the visual information is accurate, and there can be many different image processing algorithms useful for helping its navigation. Unfortunately, a traditional image processing system based on standard board cards needs a lot of hardware resources. Moreover, an autonomous platform may need the hardware implementation of several vision algorithms due to the real-time nature of its own navigation. On the other hand, the platform carries its own resources, giving as result systems with power consumption, size or weight limitations. In this way, the reconfigurable hardware appears as a new trend in autonomous robot design, since it is possible to maintain software flexibility while keeping hardware performance. Furthermore, the increasing density and performance of

* This work has been supported by the Generalitat Valenciana under project CTIDIA/2002/142

reconfigurable devices, specially for the system-on-a-programmable chip families, makes a reconfigurable approach on a single chip as specially suited for a system with hardware restrictions.

Space-variant vision emerges as an attractive image representation, since information reduction is interesting for a system with hardware restrictions. The log-polar mapping shows, as a particular case of space-variant vision, useful properties in addition to the selective reduction of information.

This paper is divided into four sections. Section 2 gives an introduction to log-polar vision and the two differential algorithms useful for robotic navigation. Section 3 gives a brief state-of-the-art about implementation of image processing systems into reconfigurable architectures and explains the advantages that a single chip approach offers. Additionally, section 3 shows how both algorithms have been synthesized into the complex FPGA employed, presenting how they benefit from the use of the on chip RAM for constructing intermediate internal frame-grabbers. Finally, section 4 draws several conclusions.

2 Log-Polar Differential Image Processing Algorithms

A space variant visual sensor distribution presents a selective reduction of information and, in several cases, simplifies geometric computations . The log-polar representation has interesting properties that have been widely studied [1] [2] [3] [4]. As an example of particular computational simplifications, rotations around the sensor center are converted to simple translations along the angular coordinate, and homotheties with respect to the center in the sensor plane become translations along the radial coordinate.

The log-polar transformation can be made directly through a sensor which has the log-polar sensor distribution. Fig. 1 shows the log-polar transformation that is directly performed at the sensor level. The sensor plane is called retinal plane, and the computational plane is called cortical plane. The equations included into the Fig. 1 show the image transformation performed by a log-polar

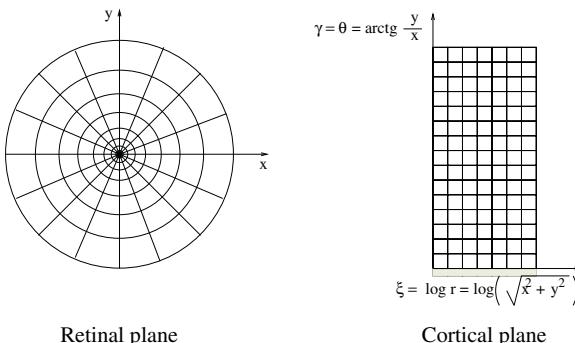


Fig. 1. The log-polar transformation

sensor. As an example of log-polar sensor there is a CMOS log-polar visual sensor with a resolution of 76 rings with 128 cells per ring [5]. This sensor has two different areas: the outer 56 rings, or retina, that follows exactly the log-polar equations, and the inner area, or fovea, that follows a linear growth law.

The log-polar conversion, performed at the presented system, consists on arithmetically calculate the transformation from cartesian to log-polar coordinates, for every pixel coming from the camera by means of the CORDIC algorithm. This conversion stage has been integrated into the chip as a first module previous to the processing stage. The chosen implementation for the log-polar transformation has two stages: the first calculates the polar coordinates (radius and angle) of the cartesian coordinates (x, y) . The second stage then calculates the logarithm of the radius giving the final log-polar coordinates. This approach gives an image resolution of 56 rings with 128 cells per ring (57.344 bits) following exactly the log-polar transformation equations.

Differential algorithms, developed in log-polar coordinates, extract dynamic information from the scene and are useful for a visual-guided platform. Differential approaches use the temporal and spatial derivatives of the image sequence. The operations involved are simple and systematically applied to all the image pixels, thus suitable for hardware implementation. These algorithms are computationally intensive due to the image size, but they benefit from log-polar data reduction. In this way, the implemented algorithms into the reconfigurable board must optimize temporal and spatial differential computations.

The objective is to have a library of algorithms for programming the reconfigurable chip.

Motion Detection Independent of the Log-Polar Camera Movement: Originally developed in Cartesian coordinates [6], and adapted to log-polar coordinates [7], its experimental effectiveness has been already proved [8]. This algorithm detects moving objects with respect to the static background. In a moving platform, there are image variations due to the self camera movement that may appear as moving objects. This algorithm is able to filter the image displacement due to the camera self movement. The constrains are related to several smoothness conditions in the grey level image and in the camera motion. Theoretically, only objects which are moving with respect to the background are detected. The algorithm constrains are related to grey level and movement smoothness, and can be formulated as follows:

$$\frac{\partial^2 E(\xi, \gamma, t)}{\partial \xi^2} = \frac{\partial^2 E(\xi, \gamma, t)}{\partial \gamma^2} = \frac{\partial^2 \xi}{\partial t^2} = \frac{\partial^2 \gamma}{\partial t^2} = 0 \quad (1)$$

For any point of the cortical plane the grey level image $E(\xi, \gamma, t)$ must be smoothed, and the camera movement must be linear along the optical axis. If the focus of expansion is at the center of the sensor, this movement is transformed in a translation along the radial coordinate. Under these constrains, the second temporal derivative of the image becomes a small integer number near zero except for the self-moving objects. Therefore, the algorithm is summarized as fol-

lows: First, the image must be smoothed, next the first order temporal derivative must be calculated from two consecutive images, and finally the second temporal derivative must be computed, selecting the zero values for binaryizing the image and marking self-moving objects. In this way, the algorithm implementation into programmable logic must compute efficiently image temporal differences. In fact, due to the non-exactly accomplishment of the algorithm constrains, the condition for a point that belongs to a self-moving object is that the second temporal derivative of the log-polar images must be larger than a threshold.

Time to Impact Computation: A second differential algorithm based on log-polar vision, that can be useful for helping the navigation of a robot, is the time to impact computation of a camera to an approaching surface [9]. The time to impact (τ) in the case of polar images, supposing an approaching movement along the optical axis at the sensor center is:

$$\tau = K \frac{-\frac{\partial E}{\partial \xi}}{\frac{\partial E}{\partial t}} \quad (2)$$

where K is a constant that depends on the log-polar transformation geometric parameters, $E(\xi, \gamma, t)$ is the log-polar image sequence, $\frac{\partial E}{\partial \xi}$ is the radial gradient and $\frac{\partial E}{\partial t}$ is the first order temporal derivative.

Equation (2) shows that the time to impact can be computed as a division of two differential magnitudes: the radial gradient and the first order temporal derivative. A previous stage for smoothing the original log-polar images is required for avoiding non-sense derivatives, like derivatives at the edges (step functions). A detailed discussion about the accuracy of this log-polar algorithm can be found at [10].

3 Synthesizing Differential Algorithms into the Reconfigurable Device

Reconfigurable architectures have been widely used for implementing image processing algorithms. Modular multiboard architectures are formed of several boards, with various reprogrammable devices into each board [11] [12]. The advantages of this approach are flexibility, throughput and computation power. Unfortunately, these good features have the disadvantage of size, weight and power consumption. There are also other simpler approaches or architectures oriented to autonomous platforms implemented in single boards [13], programmable devices [14], or implementations that use few medium PLDs [15]. This last architecture is a pipeline of small Processing Elements (PEs) oriented to processing log-polar vision algorithms. This machine is modular and scalable as the multi-board architectures and small enough for an autonomous robot. Each PE has a medium PLD, SRAM memory and a small PAL. All these elements are not fully used in each algorithm implemented wasting in this way hardware resources

In the range of single board reconfigurable systems, there are some boards that incorporate high-density and high-performance programmable devices. Moreover, these reprogrammable devices offer complete system integration on a single device called system-on-a-programmable-chip (SOPC). These boards and devices seem particularly useful for applications where there are hardware restrictions, and any gate and bit inclusion must be fully justified. In this way, a single high-density SOPC device, which incorporates the possibility of including exactly the memory size and the modules needed at each algorithm will optimize the hardware utilization.

The APEX PCI board from Altera includes a SOPC APEX 20KC device (EP20K1000C) which has 38.400 Logic Elements (LEs) equivalent to 10^6 gates (or $1,7 \cdot 10^6$ system gates) and 320 Kbits of RAM. Furthermore, the board follows the mechanical and electrical PCI interface specifications and it is designed for the integration of a PCI mega-core as input/output interface. A 64 bits master PCI interface fills around 1.400 LEs, which is less than 4% of the resources. This board and this device has been employed for synthesizing the image processing module for the autonomous robot.

It is possible to extract a common structure from the algorithms presented at section 2 and from other similar differential image processing algorithms. A generic log-polar vision algorithm can be split into different well-balanced stages (similar processing delay) for avoiding a slow stage that becomes the pipeline bottleneck. The input of the first stage is a sequence of log-polar images, but the output of this stage, and therefore the inputs to any other stage, can be an ordered data structure.

Any algorithm implemented into the reconfigurable device can be divided as a pipeline of processing stages. In this way, the image processing algorithms that could be efficiently split into such stages are algorithms which have an unidirectional data flow between all the stages of the pipeline. Moreover, the computation of temporal derivatives are implemented using RAM bits as double port memories between stages, so any stage can simultaneously access the information currently processed by the precedent stage and the data stored from the previous computation. The image storage and the reading of the next stage is made simultaneously, but in different memory blocks. Moreover, there is a switching between memories for each new image. The use of Library Parameterizable Macros (LPMs) available with Quartus II allows the easy definition of the inter-stage double port memories and the local-stage memory for computing differential magnitudes. It must be noticed that the vision algorithms are applied to the retina (the outer area in the log-polar images) since this area follows exactly the log-polar growth law, so the algorithms are accomplished and the total data size is 57.344 bits per log-polar image. The APEX20KC device used has 327.680 RAM bits, so it is shown how this approach, that includes double-port RAM memories between stages is a feasible model.

All the stages have been designed with synthetisable VHDL as autonomous and independent modules, with a Finite State Machine (FSM) for implementing its processing functionality. The communication protocol, among all the pipeline

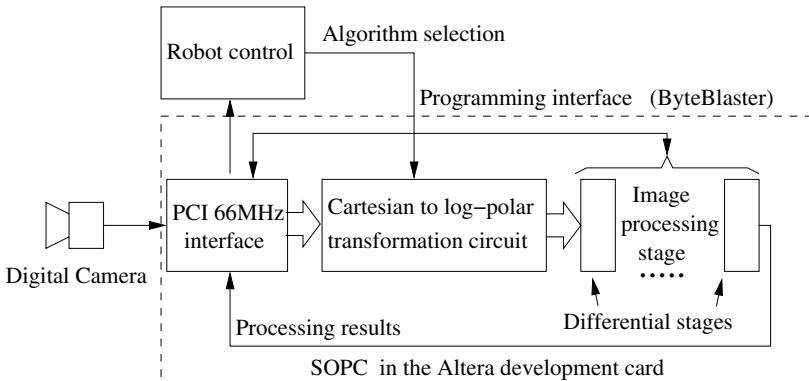


Fig. 2. Overall system

stages, is a simple asynchronous protocol, useful for defining a library of stages that can be mixed for developing different algorithms.

Fig. 2 shows a schema of the overall system organization. A digital cartesian camera acquires the images and send them to the Altera development card by way of the PCI bus. A PCI master/slave megacore is included into the SOPC as image processing system interface as has been already appointed. The chip incorporates a CORDIC module for performing the cartesian to log-polar transformation. Afterwards, the differential pipeline stage processes the images, extracting the information the algorithm delivers, for the robot navigation system. The robot can reconfigure the SOPC selecting the algorithm that is employed for processing the images. The algorithm chosen each time will be the most adequate for helping its navigation.

3.1 Motion Detection Algorithm Synthesis

The algorithm explained in section 2 describes a differential algorithm for detecting objects that move to respect the background, discarding automatically the image displacement due to the camera self movement when this movement is uniform along the camera optical axis. The algorithm has been divided into three stages that work simultaneously as a data pipeline as shown in Fig. 3. Double port memories of exactly the retinal image size (7.168 bytes) are placed between the stages in order to accelerate differential computations. Moreover, the first stage has 128 bytes of local memory for storing a log-polar ring. The ring values are shifted systolically with the aim of computing a grey level average value for accomplishing the smoothed image condition of the algorithm.

1. The first stage smoothes the original log-polar image storing the smoothed image in a double port memory and simultaneously giving it to the next stage. This smooth is made with a simple convolution mask in order to reduce the computations, avoiding a bottleneck at this stage.

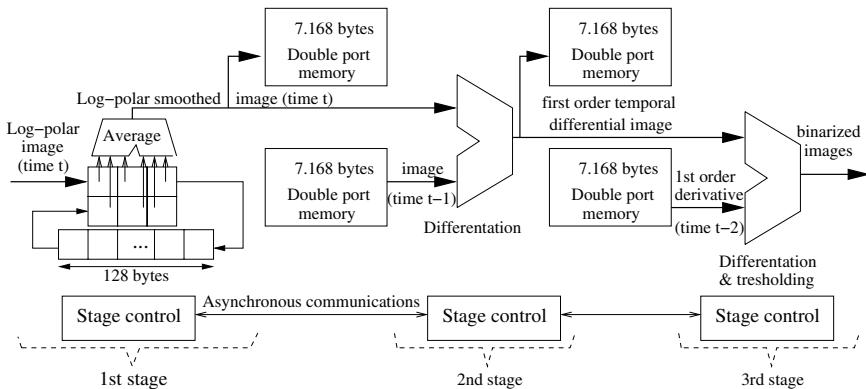


Fig. 3. Motion detection algorithm implementation

2. The second stage computes the first temporal derivative as an image subtraction, accepting the smoothed pixel from the precedent stage. It also reads the double port memory where the precedent smoothed image is stored. Subsequently, first order differences are calculated as a simple pixel subtraction. Finally, the differential pixel is sent to the next stage, being simultaneously stored in a double port memory
3. The third stage computes the second order temporal derivative and binarizes the image. This stage receives first order differences from the preceding one and simultaneously reads the differences previously stored in the double port memories. The second temporal derivative image is computed as differences between two first order temporal derivative images. Moreover, this value is compared to a threshold that is basically related to the robot movement and scene illumination. The final result is a sequence of binarized images which have marks for the points that belong to self-moving objects.

The algorithm has been successfully synthesized into the APEX20K device, occupying 230.400 RAM bits (70% of the total available RAM resources) and less than 1.000 LEs. It is feasible to increase the algorithm accuracy improving the smoothing stage and the derivation computations. The chip clock frequency can reach 200 MHz, but it is limited to 66 MHz that is the system clock frequency. This algorithm has a well balanced number of clock cycles, consuming 4 cycles for computing 1 pixel at each stage. All the stages are working in parallel and the pipeline only needs an additional cycle for the input of each byte and another for the output. In this way, the pipeline yields a processed byte every 6 clock cycles. Taking into account that a complete retinal log-polar image occupies 7 Kbytes it is possible to compute a theoretical processing ratio of more than 250 frames per second. This large high ratio must be limited due to the differential nature of the algorithm. Thus, in order to compute differential magnitudes as are the temporal or spatial derivatives, differences between images must be guaranteed, so a minimum acquisition interval between images must be ensured [8].

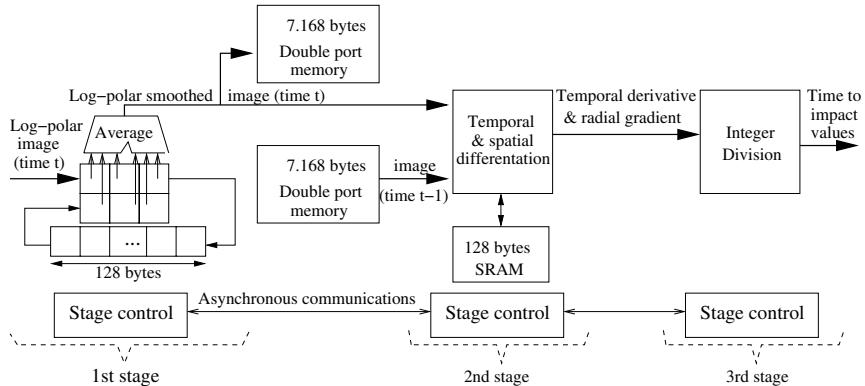


Fig. 4. Time to impact algorithm implementation

3.2 Time to Impact Computation Algorithm Synthesis

The algorithm for time to impact computation has been also implemented with the same methodology of splitting the overall task into a pipeline of stages. Double port memories have been employed for accelerating the computation of the first temporal derivative. Again, the algorithm has been divided into three stages and there are two double port memories as library modules. Fig. 4 shows the algorithm implementation.

1. The first stage is exactly the same smoothing block designed for the previous algorithm. Each stage has been designed with its own control and identical protocol communications. Therefore, any pipeline stage already designed can be re-utilized in any other implementation as a standard library module.
2. The second stage computes the first temporal derivative and the radial gradient. The first order differentiation is computed through the same image subtraction policy described previously. Simultaneously, the radial gradient is computed with the smoothed pixel supplied by the previous stage, and the pixel corresponding to the inferior ring stored in a local small memory of 128 bytes.
3. Finally, the third stage computes the time to impact map for each pixel making an integer division of both values, with a cost of 8 clock cycles.

The algorithm has been also successfully synthesized into the APEX20K device, occupying near 115.000 RAM bits (35% of the total available RAM resources) and less than 1.100 LEs. So, it is also feasible to increase the algorithm accuracy improving the division stage. The clock frequency can be up to 166 MHz, but it is limited to 66 MHz that is the system clock frequency. This algorithm yields a processed byte every 10 clock cycles.

4 Conclusions

Image processing algorithms, involved in visual real-time navigation, benefit from hardware speed. Furthermore, it is not a good solution to have a custom hardware system in an autonomous platform for each desired algorithm due to size, weight and power consumption reasons. Reconfigurable systems on-a-chip appear as a technology that combines hardware performance, software reconfigurability and low resources consumption.

In the other hand, space-variant vision has been employed to reduce the total amount of data to be processed, reducing the memory size for making it possible the implementation of several local frame grabbers inside the chip. Since the involved algorithms are differential, a hardware parallel implementation for computing the temporal differences is a feasible approach that benefits from information reduction. Moreover, the log-polar scheme reduces the computation complexity of the selected algorithms.

A methodology developed previously for splitting differential algorithms into stages for accelerating the temporal difference computations has been applied. All the stages have identical control scheme and communication interface for simplifying the design of algorithms planning, a policy of stage re-usability. Following these ideas, two different algorithms have been implemented in the reconfigurable device, showing its flexibility and performance. Both algorithms take advantage of image reduction and computation simplification, allowing a high rate of processed images per second. The combination of the log-polar formalism, differential algorithms and pipelined architecture shows its good performance for real-time image processing. The synthesis results show that a complete image processing system can be synthesized in a high-density SOPC chip, including a complete PCI interface.

References

1. Bernardino, A., Santos-Victor, J. Vergence Control for Robotic Heads using Log-polar Images. In IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, IROS'96, Osaka, Japan, (1996)
2. Capurro, C., Panerai F., Sandini, G. Dynamic Vergence using Log-polar Images. International Journal of Computer Vision **24-1** (1997) 79-94
3. Daniilidis, K. Attentive Visual Motion Processing: Computations in the Log-polar Plane. Computing **11** (1996) 1-20
4. Ferrari, F., Nielsen, J., Questa, P., Sandini, G. Space Variant Imaging. Sensor Review **15-2** (1995) 17-20
5. Pardo, F., Dierickx, B., Scheffer, D. Space-Variant Non-Orthogonal Structure CMOS Image Sensor Design. IEEE Journal of Solid State Circuits, **33-6** (1998) 842-849
6. Chen W.G., Nandhakumar, N. A Simple Scheme for Motion Boundary Detection. In Proceedings of the IEEE Intl. Conf. on Systems, Man and Cybernetics (1994)
7. Boluda, J.A., Domingo, J., Pardo, F., Pelechano, J. Detecting Motion Independent of the Camera Movement through a Log-polar Differential Approach. In Computer Analysis of Images and Patterns. 7th International Conference CAIP'97, Lecture

- Notes in Computer Science, Vol. 1296. Springer-Verlag Berlin Heidelberg New York (1997) 702-710
- 8. Boluda, J.A., Domingo, J. On the Advantages of Combining Differential Algorithms and Log-polar Vision for Detection of Self-motion from a Mobile Robot. *Robotics and Autonomous Systems* **37-4** (2001) 283-296
 - 9. Tistarelli, M., Sandini, G. On the Advantages of Polar and Log-polar Mapping for Direct Estimation of Time-to-impact from Optical Flow. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **15-4** (1993) 401-410
 - 10. Pardo, F., Boluda, J.A., Coma, I., Mico, F. High Speed Log-polar Time to Crash Calculation for Mobile Vehicles. *Image Processing and Communications*, **8-2** (2002) 23-32
 - 11. Duncan Buell, Jeffrey Arnold and Walter Kleinfelder. Splash2: FPGAs in a Custom Computing Machine. IEEE Computer Society Press, (1996)
 - 12. Demigny, D., Kessal, L., Bourguiba, R., Boudouani, N. How to Use High Speed Reconfigurable FPGA for Real Time Image Processing. In Proceedings of the Fifth IEEE International Workshop on Computer Architecture for Machine perception (2000) 240-246
 - 13. Fross, B., Donaldson, R., Palmer, D. PCI-Based WILDFIRE Reconfigurable Computing Engines, In High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic. Proceedings of the SPIE Vol. 2914 (1996) 170-179
 - 14. Arias-Estrada, M., Rodriguez-Palacio, E. An FPGA Co-processor for Real-Time Visual Tracking. In 12th International Conference on Field-Programmable Logic and Applications. FPL'02, Lecture Notes in Computer Science, Vol. 2438. Springer-Verlag Berlin Heidelberg New York (2002) 710-710
 - 15. Boluda, J.A., Pardo, F. A Reconfigurable Architecture for Autonomous Visual Navigation. *Machine Vision and Applications* **13-5** (2003) 322-331

Reconfigurable Hardware SAT Solvers: A Survey of Systems

Iouliia Skliarova and António B. Ferrari

University of Aveiro, Department of Electronics and Telecommunications, IEETA
3810-193 Aveiro, Portugal
`{iouliia, ferrari}@det.ua.pt`

Abstract. By adapting to computations that are not so well supported by general-purpose processors, reconfigurable systems achieve significant increases in performance. Such computational systems use high-capacity programmable logic devices and are based on processing units customized to the requirements of a particular application. A great deal of research effort in this area is aimed at accelerating the solution of combinatorial optimization problems. Special attention was given to the Boolean satisfiability (SAT) problem resulting in a considerable number of different architectures being proposed. This paper presents the state-of-the-art in reconfigurable hardware SAT satisfiers. The analysis of existing systems has been performed according to such criteria as reconfiguration modes, the execution model, the programming model, etc.

1 Introduction

Although the concept of reconfigurable computing has been known since the early 1960s [1], it is only recently that technologies that allow it to be put into practice became available. The interest started at the beginning of the 1990s as FPGA densities broke the 10K logic gate barrier. Since then, reconfigurable computing became a subject of intensive research. For some classes of applications reconfigurable systems allow very good performance to be achieved compared to general-purpose computers. Other types of applications were mapped to reconfigurable hardware because it offers innovative opportunities to explore. According to the primary objective to be achieved, all these applications can be broadly divided into three categories: hardware emulation and rapid prototyping, evolvable hardware, and the acceleration of computationally intensive tasks. The last category is without doubt the prevalent one.

Recently, a series of attempts have been made to accelerate applications that involve rather complex control flow. In this context special attention was given to problems in the area of combinatorial optimization. Among them, the Boolean satisfiability (SAT) problem stands out. This may be partially explained by the extremely wide range of practical applications in a variety of engineering areas, including the testing of electronic circuits, pattern recognition, logic synthesis, etc. [2]. In addition, SAT has the honor of being the first problem shown to be NP-complete [3]. This means that existing algorithms have an exponential worst-case

complexity. Implementations based on reconfigurable hardware enable the primary operations of the respective algorithms to be executed in parallel. Consequently, the effect of exponential growth in the computation time can be delayed, thus allowing larger size instances of SAT to be solved [2].

SAT is a very well known combinatorial problem that consists of determining whether a given Boolean formula can be satisfied by some truth assignment. The search variant of this problem requires at least one satisfying assignment to be found. Usually, the formula is presented in conjunctive normal form, which is composed of a conjunction of a number of clauses, where a clause is a disjunction of a number of literals. Each literal represents either a Boolean variable or its negation. A survey of algorithmic methods of solving the SAT problem can be found in [2].

In this paper we present the current status of reconfigurable hardware SAT solvers and give an overview of the existing approaches and their tradeoffs. The remaining part of the paper is organized as follows. Section 2 is devoted to the description of the most well known architectures of reconfigurable hardware SAT satisfiers. Analysis and classification of these architectures according to different criteria is performed in section 3. Finally, concluding remarks are given in section 4.

2 Architectures of SAT Solvers

Recently, several research groups have explored different approaches to solve the SAT problem with the aid of reconfigurable hardware [4-5], [9-12], [14-22]. Since names have not typically been given to hardware SAT satisfiers, we will refer to them according to the first author's names of the respective publications.

Suyama et al. [4-5] suggested an architecture of an *instance-specific* SAT solver capable of finding all the solutions (or a fixed number of them) of a given problem instance. The employed algorithm is characterized by the fact that at any moment a full variable assignment is evaluated. A *dynamic decision strategy* based on both experimental unit propagation and a maximum-occurrence-in-clauses-of-minimum-size heuristic has been adopted. A number of circuits have been implemented on an Altera FLEX10K250 FPGA clocked at 10 MHz. *Suyama et al.* were able to achieve a small acceleration compared to the POSIT algorithm [6] executed on an UltraSPARC-II/296 MHz over some instances from DIMACS benchmark suite [7]. However, the time spent in hardware compilation and configuration was not taken into account.

Zhong et al. implemented a version of the well-known Davis-Putnam (DP) algorithm [8]. In their early work [9] they constructed an implication circuit and a state machine for each variable in the formula, all the state machines being connected in a serial chain. As a preprocessing step, all the variables are sorted taking into account the number of their appearances in a given formula. In [10] hardware implementation of *non-chronological backtracking* was proposed. The resulting hardware execution time was quite good but the design had two distinct drawbacks. First, the clock frequency was low (ranging from 700KHz to 2MHz for different formulae). Second, the hardware compilation time took several hours (on a Sun 5/110MHz/64MB) thus canceling all the advantages of fast hardware execution.

In more recent work [11], [12] the basic design decisions were revised. As a result, a regular ring-based interconnecting structure was employed instead of irregular global lines, essentially reducing the compilation time in this way (to an order of

seconds) and increasing the clock rate (to 20-30 MHz) [12]. In addition, a technique enabling conflict clauses to be generated and added was proposed. The experimental results are based on both hardware implementation (on an IKOS emulator containing a number of FPGA array boards) and simulation. The speedups achieved over the software satisfier GRASP [13] executing on a Sun5/110MHz/64MB (in a restricted mode), including the hardware compilation and configuration time, are of an order of magnitude [12] for a subset of the DIMACS SAT benchmarks [7].

Abramovici et al. [15] employed the technique of modeling a formula by a 2-level circuit. The SAT solver proposed in [14] is based on the PODEM algorithm. In [15] an improved architecture is suggested that employs the DP algorithm and implements an enhanced variable selection strategy. For hardware implementation *Abramovici et al.* suggest creating a library of basic modules that are to be used for any formula. The modules have predefined internal placement and routing. In this case the solver circuit will be built from modules, which allows the compilation time to be reduced (to the order of minutes). The authors implemented simple circuits on XC6264 FPGA and simulated the bigger ones. For a circuit occupying the whole area of the XC6264 FPGA the clock frequency is about 3.5 MHz. In [15] *Abramovici et al.* report speedups from 0.01 to 7000 (after time unit justification) achieved over GRASP [13] for a subset of DIMACS SAT benchmarks [7]. In [15] a virtual logic system was proposed allowing circuits to be constructed for solving SAT problems that are larger than the available hardware resources. This is achieved by decomposing a formula into independent sub-formulae that can be processed in separate FPGAs either concurrently or sequentially.

The SAT solver proposed by *Platzner et al.* [16], [17] is similar to that of *Zhong* [9]. It consists of a column of finite state machines, deduction logic and a global control unit. The deduction logic computes the result of the formula based on the current partial variable assignment. All variable assignments are tried in a fixed order. The authors implemented an accelerator prototype on the base of a Pamette board containing 4 Xilinx XC4028 FPGAs. The speedups obtained for *hole6...hole10* SAT benchmarks from DIMACS [7], including hardware compilation and configuration time, range from 0.003 to 7.408 compared to GRASP executing on a PII/300MHz/128MB [16]. The designs for the *holex* problems run at 20 MHz [17].

More recent work in this direction is targeted at avoiding instance-specific layout compilation. *Boyd et al.* [18] proposed an architecture for a SAT-specific programmable logic device that excludes instance-specific placement and routing. The suggested design consumes polynomial hardware resources (with respect to the number of variables and clauses) and requires polynomial time to configure. The authors implemented a small version of their SAT satisfier for a problem having 8 variables and 8 clauses on a Xilinx XC4005XL running at 12 MHz. However, no results on large benchmark problems were reported.

Sousa et al. [19], [20] were the first to propose partitioning the job between software and reconfigurable hardware with the most computationally intensive tasks (such as computing implications and choosing the next decision variable) assigned to hardware, while the control-oriented tasks (such as conflict analysis, backtrack control and clause database management) are performed in software. The suggested SAT solver has an *application-specific* architecture that uses configuration registers for SAT formula instantiation [20]. In order to deal with instances that exceed the available hardware capacity, a virtual hardware scheme with context switching has been proposed. The results reported in [19] are based on a software simulator of the

system under an estimated clock frequency of 80 MHz, assuming that the context-switching device can swap pages in one clock cycle.

Skliarova *et al.* [21], [22] proposed an application-specific SAT solver realizing a DP-based algorithm. The problem was formulated over a ternary matrix by setting a correspondence between clauses and variables of a formula and rows and columns of the matrix. In order to solve various problem instances it is only necessary to download the respective matrix data. All the other components of the satisfier remain unchanged. This allows local reconfigurability to be used and reduces the configuration overhead. The problem is partitioned between software and reconfigurable hardware in such a way that an FPGA is only responsible for processing sub-problems that appear at various levels of the decision tree and satisfy the imposed hardware constraints (such as the maximum allowed number of rows and columns in the matrix). This technique permits problems to be solved that exceed the resources of the available reconfigurable hardware. The SAT satisfier was implemented on an ADM-XRC PCI board containing one XCV812E Virtex-EM FPGA (running at 40MHz). The results of experiments on some of DIMACS benchmarks [7] have shown that it is possible to achieve a significant speedup compared to GRASP (up to 111x, including the FPGA configuration time, with GRASP executed on an AMD Athlon/1GHz/256MB).

3 Analysis of Hardware SAT Solvers

In this section we attempt to analyze the reconfigurable hardware SAT solvers according to such criteria as algorithmic issues, programming model, execution model, reconfiguration modes, logic capacity and performance. Table 1 summarizes the respective characteristics of the architectures considered in the previous section.

3.1 Algorithmic Issues

The majority of the existing reconfigurable hardware SAT solvers employs some variation of the Davis-Putnam algorithm [8]. An exception to this is the SAT satisfier of Abramovici *et al.*, which implements a PODEM-based algorithm [14].

The search process in the DP algorithm is usually organized with the aid of a *decision tree*, whose nodes are characterized by the respective partial variable assignments, and arcs represent the *decisions* taken. There exist two basic approaches to the selection of the decision variables: *static* and *dynamic*. Although dynamic selection has been considered to be a difficult task for hardware implementation, it was realized in a number of architectures [5], [19-22].

In the present-day software SAT solvers a lot of advanced techniques (such as non-chronological backtracking [13]) are employed that enable those regions of the search space that do not contain any solution to be identified and avoided. However, up to now these techniques have been largely ignored by hardware SAT solvers. The few exceptions to this rule are the SAT satisfiers of Zhong *et al.* [12] and Sousa *et al.* [19, 20] (the latter implements them in software).

Table 1. Principal characteristics of the reconfigurable hardware SAT solvers

SAT solver	Algorithmic issues	Program-ming model	Execution model	Reconfi-guration mode	Logic capacity	Perfor-mance (t_{total})
<i>Suyama et al.</i> [12]	DP-like algorithm with dynamic selection	instance-specific	hardware only	static	multi-FPGA system	$t_{comp} + t_{conf} + t_{ex_h}$
<i>Zhong et al.</i> [12]	DP-based algorithm with static selection, non-chronological backtracking, conflict analysis	instance-specific	hardware only	dynamic (global)	multi-FPGA system	$t_{comp} + t_{conf} + t_{comm} + t_{ex_h}$
<i>Platzner et al.</i>	DP-based algorithm with static selection	instance-specific	hardware only	static	use larger device	$t_{comp} + t_{conf} + t_{ex_h}$
<i>Abra-movici et al.</i> [15]	DP-based algorithm with optimized static selection	instance-specific	hardware only	dynamic (global)	logic partitioning in sub-formulae	$t_{comp} + t_{conf} + t_{comm} + t_{ex_h}$
<i>Sousa et al.</i>	DP-based algorithm with dynamic selection and conflict analysis (in software)	application-specific	software/hardware partitioning according to computational complexity	dynamic (partial)	virtual hardware scheme	$t_{conf} + t_{comm} + t_{ex_s} + t_{ex_h}$
<i>Skliarova et al.</i>	DP-based algorithm with dynamic selection	application-specific	software/hardware partitioning according to logic capacity	dynamic (partial)	software/hardware parti-tioning	$t_{conf} + t_{comm} + t_{ex_s} + t_{ex_h}$

3.2 Programming Model

There are two basic approaches to mapping a SAT formula to a reconfigurable system: *instance-specific* and *application-specific*. The first approach has been extensively explored by the SAT research community [4-5], [9-12], [14-17] and assumes the generation of an individual hardware configuration for each problem instance. In this case, a typical design flow is used to describe and implement either a whole instance-specific circuit or a number of primary modules, which are further customized (at compile time) by specially developed software tools to match the respective formula.

In an *application-specific approach* the circuit is designed and optimized only once, after which it can be used for different problem instances [18-22]. This can be achieved with the aid of a hardware template, which is also developed using a typical design flow but is customized with data for a particular problem at run-time (instead of compile-time). It should be noted that in this case a hardware compilation step is completely avoided.

3.3 Execution Model

A SAT problem can be either entirely mapped to reconfigurable hardware (leaving just the tasks of preprocessing and initialization to the host processor) [4-5], [9-12], [14-18] or partitioned between hardware and software [19-22]. There exist different methods of software/hardware partitioning. In the domain of SAT solvers, two are usually employed: *partitioning according to computational complexity* and *partitioning with respect to logic capacity*.

The first method assigns computationally intensive portions of an application to hardware, while the remaining portions that exhibit little parallelism are handled by the host processor [19], [20]. Reconfigurable systems of this type are based on the 90/10 rule, which states that 90% of execution time of an application is spent by 10% of its code. Thus, in order to increase performance it is attempted to accelerate this small portion of an application with the aid of programmable logic devices.

The second method performs partitioning according to the available logic capacity of hardware employed [21], [22]. In this case, if a problem instance does not “fit” to a chosen device (or a number of interconnected devices), it has first to be processed by software up to the point at which it can be transferred to hardware.

3.4 Reconfiguration Modes

In the domain of reconfigurable computing it is common to distinguish between two configuration modes: *static mode* (also known as *compile-time configuration* or *design-time binding*) and *dynamic mode* (frequently referenced as *run-time configuration* or *implementation-time binding*).

Static configuration assumes fixed functionality of the device once it has been programmed [4-5], [9-10], [16-17]. Dynamic reconfiguration allows the functionality of the system to be changed during the execution of an application. Dynamic reconfiguration can in turn be *partial* or *global*. Global reconfiguration reserves all the hardware resources for each step of execution. After a step has been concluded, the device may be reprogrammed for the next step [15]. Partial reconfiguration implies the selective modification of hardware resources [19-22]. This opportunity allows the hardware to be adapted to better suit the actual needs of the application. Since only selected portions are reconfigured, the configuration overhead is less than in the previous case.

A variety of reprogrammable devices can be employed to carry out dynamic reconfiguration. *Single-context* devices require complete reprogramming in order to introduce even a small change. Although many commercially available FPGAs are single-context, there exist techniques (based on hardware templates) that allow partial reconfiguration to take place [19-22]. *Multi-context* devices possess various planes of configuration information with just one of them active at any given moment [19]. The main advantage of such devices is the ability to switch the context very fast. *Partially reconfigurable* devices permit small portions of their resources to be modified without disturbing the remaining parts. Although this kind of devices (such as the XC6200 family of Xilinx) was employed for some SAT solvers [15], the potential for partial reconfigurability has not been explored.

3.5 Logic Capacity

The logic capacity of the employed hardware device is always limited. Thus, efficient techniques are needed to deal with the situation when a problem instance exceeds the available hardware resources. The answers to this issue differ accordingly to the programming and execution models adapted. Basically, four possibilities have been explored.

The first is the expansion of the logic capacity by interconnecting a number of programmable devices and partitioning the circuit between them. It should be noted that fast and efficient multi-device partitioning and routing is quite a difficult task (of course modular design styles [12] can alleviate it). Moreover, the working frequency of such multi-device systems is usually quite limited.

The second method is to partition the problem into a series of configurations to be run either sequentially or in parallel. The partitioning is performed by decomposing an initial formula into a set of independent sub-formulae [15]. Each sub-formula must satisfy the imposed hardware constraints. The main limitation of this method is that the efficiency of the decomposition greatly depends on the characteristics of the formula. As a result, for some problem instances the partitioning time may increase to unacceptable levels.

The third method is based on software/hardware partitioning according to the available logic capacity of hardware that is employed (see section 3.3). In this case just those sub-problems that appear at different levels of the decision tree and respect the capacity limitations are assigned to hardware, the remaining portion of the problem being processed by a software application [21-22].

The last method is based on a virtual hardware scheme proposed in [19-20], which relies on dividing the circuit into a series of hardware pages that are successively run being the intermediate results stored in external memory blocks. Since all the hardware pages have the same structure with only a number of registers being reconfigured, the page switching is performed very fast.

3.6 Performance

The total time (t_{total}) spent by a reconfigurable hardware SAT satisfier to solve a particular problem instance comprises four components: hardware compilation time (t_{comp}), hardware configuration time (t_{conf}), time required for communication between software and hardware (t_{comm}) and actual execution time (t_{ex}). If a problem solution is partitioned between software and hardware then the execution time t_{ex} is composed of software execution time ($t_{ex,s}$) and hardware execution time ($t_{ex,h}$). It should be noted that the values of these components depend on the programming and execution models employed and some of them may be zero. For example, if a problem instance is entirely mapped to hardware, usually there is no communication (except for notifying the final result) between the host processor and the programmable device. In the same manner, if an application-specific approach is followed, the hardware compilation time is zero. Actually, the compilation time may constitute a large portion of the total solving time. For easy problem instances it even dominates and cancels out all the benefits of fast hardware execution [4-5], [9-10], [16-17]. That is why a number of techniques targeted at reducing the hardware compilation time have

been proposed. They are based on exploiting modular design styles and developing customized software tools instead of using commercially available ones [11-12], [15].

One characteristic inherent in reconfigurable hardware SAT solvers is that it is very difficult to analyze and compare their performance accurately. As a rule, the designers present the results achieved in the light of the software SAT satisfier GRASP [13]. However, GRASP is run on different platforms and with dissimilar parameters that heavily influence its performance. Moreover, the parameters set are frequently not published. The majority of the SAT solvers considered involve a hardware compilation step, which is sometimes ignored (or hidden) when presenting the results. It is also difficult to estimate the exact impact of compilation on the total execution time because of the variety of software platforms used. Nevertheless, in all recent designs a clear intention to reduce and even to avoid the hardware compilation step is apparent [12], [15], [18-22].

As shown by the results of the 2002 software SAT competition [23], GRASP has been surpassed by more recent SAT satisfiers such as zChaff [24] and BerkMin [25]. Consequently novel algorithmic and architectural techniques need to be explored in order to put the reconfigurable hardware SAT solvers in a more favorable light comparing to a software solution.

4 Conclusion

This paper is dedicated to the description and comparison of reconfigurable hardware SAT solvers. The analysis leads to the following conclusions:

- The majority of designers implement complete search algorithms derived from the DP algorithm. Conflict analysis is usually not performed and just chronological backtracking is executed (with a few exceptions).
- Practically all the proposed SAT solvers are based on the instance-specific approach. However, the hardware compilation time restricts the range of problems for which a reconfigurable hardware solution is more effective than the software-based approach. That is why all recent efforts have been focused on avoiding instance-specific placement and routing.
- All the reconfigurable SAT solvers considered are loosely coupled systems with the programmable device (usually, a commercially available FPGA) being attached to the host processor via an external interface.
- It is quite difficult to compare the results that have been achieved. First, the hardware compilation and configuration times are not always clearly exposed. Second, the results are usually compared to GRASP, executed on different platforms with dissimilar parameters, which can lead to variations in the solving time of up to an order of magnitude.
- Real-world SAT formulae are quite large, however how instances that do not fit into an available device can be handled efficiently is not always discussed. Recently, in what seems to be a promising solution, it was suggested that the problem should be partitioned between software and reconfigurable hardware. It should also be noted that due to the rapid evolution in FPGA capacity, many

challenging problem instances can now either be fitted into a single FPGA, or at least partitioned more efficiently.

- The speedups achieved by reconfigurable hardware compared to a software solution are significant just for certain classes of SAT instances, for which the optimization techniques proposed and implemented by software SAT satisfiers are not very efficient. Some examples of these techniques are: different decision strategies, exploiting problem symmetry, careful conflict analysis, etc. Consequently, although many interesting and worthwhile architectures have already been proposed, innovative approaches still need to be explored in the reconfigurable hardware domain.

Acknowledgment

This work was supported by the Portuguese Foundation of Science and Technology under grant No. FCT-PRAXIS XXI/BD/21353/99.

References

1. Estrin, G.: Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer. IEEE Annals of the History of Computing. Oct.-Dec. (2002) 3-9
2. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35 (1997) 19-151
3. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Company. San Francisco (1979)
4. Yokoo, M., Suyama, T., Sawada, H.: Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results. In: Proc. of 2nd Int. Conf. on Principles and Practice of Constraint Programming (1996) 497-509
5. Suyama, T., Yokoo, M., Sawada, H., Nagoya, A.: Solving Satisfiability Problems Using Reconfigurable Computing. IEEE Trans. on VLSI Systems, vol. 9, no. 1 (2001) 109-116
6. Freeman, J.W.: Improvements to Propositional Satisfiability Search Algorithms. Ph.D. dissertation. Univ. Pennsylvania (1995)
7. DIMACS challenge benchmarks. [Online]. Available: <http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html>
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM n. 5 (1962) 394-397
9. Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Using Configurable Computing to Accelerate Boolean Satisfiability. IEEE Trans. CAD of Integrated Circuits and Systems, vol. 18, n. 6 (1999) 861-868
10. Zhong, P., Ashar, P., Malik, S., Martonosi, M.: Using reconfigurable computing techniques to accelerate problems in the CAD domain: a case study with Boolean satisfiability. In: Proc. Design Automation Conf. (1998) 194-199
11. Zhong, P., Martonosi, M., Ashar, P., Malik, S.: Solving Boolean satisfiability with dynamic hardware configurations. In Hartenstein, R.W., Kevalik, A. (eds). Field-Programmable Logic: From FPGAs to Computing Paradigm (1998). Springer-Verlag. 326-325
12. Zhong, P.: Using Configurable Computing to Accelerate Boolean Satisfiability. Ph.D. dissertation. Department of Electrical Engineering. Princeton University (1999)

13. Silva, L.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Computers*, vol. 48, n. 5 (1999) 506-521
14. Abramovici, M., Saab, D.: Satisfiability on Reconfigurable Hardware. In: Proc. 7th Int. Workshop on Field-Programmable Logic and Applications (1997), 448-456
15. Abramovici, M., de Sousa, J.T.: A SAT solver using reconfigurable hardware and virtual logic. *Journal of Automated Reasoning*, vol. 24, n. 1-2 (2000) 5-36
16. Platzner, M.: Reconfigurable accelerators for combinatorial problems. *IEEE Computer*. Apr. (2000) 58-60
17. Platzner, M., De Micheli, G.: Acceleration of satisfiability algorithms by reconfigurable hardware. In: Hartenstein, R.W., Keevallik, A. (eds.) *Field-Programmable Logic: From FPGAs to Computing Paradigm*. Springer-Verlag (1998) 69-78
18. Boyd, M., Larrabee, T.: ELVIS – a scalable, loadable custom programmable logic device for solving Boolean satisfiability problems. In: Proc. 8th IEEE Int. Symp. on Field-Programmable Custom Computing Machines - FCCM (2000)
19. de Sousa, J., Marques-Silva, J.P., Abramovici, M.: A configware/software approach to SAT solving". In: Proc. of 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (2001)
20. Reis, N.A., de Sousa, J.T.: On Implementing a Configware/Software SAT Solver. In: Proc. of 10th IEEE Int. Symp. Field-Programmable Custom Computing Machines (2002) 282-283
21. Skliarova, I., Ferrari, A.B.: A SAT Solver Using Software and Reconfigurable Hardware. In: Proc. of the Design, Automation and Test in Europe Conference (2002) 1094
22. Skliarova, I., Ferrari, A.B.: A hardware/software approach to accelerate Boolean satisfiability. In: Proc. of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (2002) 270-277
23. Simon, L., Le Berre, D., Hirsch, E.: The SAT2002 Competition. Technical Report (preliminary draft) [Online]. Available: <http://www.satlive.org/SATCompetition/onlinereport.pdf> (2002)
24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. of the 38th Design Automation Conference (2001) 530-535
25. Goldberg, E., Novikov, Y.: BerkMin: a Fast and Robust SAT-solver. In: Proc. Design, Automation and Test in Europe Conference (2002) 142-149

Fault Tolerance Analysis of Distributed Reconfigurable Systems Using SAT-Based Techniques^{*}

Rainer Feldmann¹, Christian Haubelt², Burkhard Monien¹, and Jürgen Teich²

¹ AG Monien, Faculty of CS, EE, and Mathematics,
University of Paderborn, Germany,
`{obelix, bm}@upb.de`

² Department of Computer Science 12, Hardware-Software-Co-Design,
University of Erlangen-Nuremberg, Germany,
`{haubelt, teich}@cs.fau.de`

Abstract. The ability to migrate tasks from one reconfigurable node to another improves the fault tolerance of distributed reconfigurable systems. The degree of fault tolerance is inherent to the system and can be optimized during system design. Therefore, an efficient way of calculating the degree of fault tolerance is needed. This paper presents an approach based on satisfiability testing (SAT) which regards the question: How many resources may fail in a distributed reconfigurable system without losing any functionality? We will show by experiment that our new approach can easily be applied to systems of reasonable size as we will find in the future in the field of body area networks and ambient intelligence.

1 Introduction

Distributed reconfigurable systems [1, 2] are becoming more and more important for applications in the area of automotive, body area networks, ambient intelligence, etc. The most outstanding property of these systems is the ability of reconfiguration. In terms of system synthesis, this means that the binding of tasks to resources is not static, i.e., the binding changes over time. Recent research was focused on the OS support for FPGAs [3] by dynamically assigning hardware tasks to an FPGA.

In a network of connected FPGAs it is possible to migrate hardware tasks from one node to another. Thus, resource faults can be compensated by *rebinding* tasks to fully functional nodes of the network. The process of rebinding is also called *repartitioning*. Distributed reconfigurable systems that support repartitioning possess an inherent fault tolerance. The degree of fault tolerance is a static property of the system and, hence, can be optimized during system design. In order to evaluate the degree of fault tolerance, we define a new objective called *k-bindability*. A system is called *k-bindable* iff any set of *k* resources is redundant. Note, it may be possible that more than *k* resources are redundant but the *k*-bindability determines that *k* such that any set of *k* arbitrary resources can be removed from the system without losing any functionality.

* Supported in part by the German Science Foundation (DFG), SFB 376 (Massive Parallelität) and SPP 1148 (Rekonfigurierbare Rechensysteme).

The main contribution of this paper is to provide an efficient way based on SAT techniques to determine the k -bindability during system design. This problem is twofold: In a first step, we will reduce the well known binding problem from system synthesis to the satisfiability problem for boolean formulas. Next, we show how to calculate the k -bindability of a system using quantified boolean formulas (QBFs). Therefore, we focus on two particular system synthesis problems:

1. Does there exist a feasible binding for a given specification of a distributed reconfigurable system that supports repartitioning?
2. How many resources may fail in a distributed reconfigurable system that supports repartitioning without losing any functionality?

With this novel approach, we can optimize the fault tolerance of distributed reconfigurable system in an early design phase. In other words, we can maximize the k -bindability of such a system for a limited number of reconfigurable nodes and connections during design space exploration.

The problem to decide the satisfiability of QBFs is an important research issue in Artificial Intelligence, since QBF is the prototypical PSPACE-complete problem. Other PSPACE-hard problems from, e.g., conditional planning [4], non monotonic reasoning [5], and hardware verification [6] have been polynomially reduced to QBF. In the past several decision procedures for QBFs have been proposed in the literature [7–10].

This paper is structured as follows: In Section 2 we introduce the formal specification model of distributed reconfigurable systems used in this paper. The following section shows how to reduce the binding problem to the satisfiability problem of boolean formulas. In Section 4 a QBF-based approach to determine the k -bindability of a distributed reconfigurable systems that supports repartitioning is proposed. Finally, we will show by experiment (Section 5) that problem instances of reasonable size are easily solved by the Davis-Putnam based QBF solver QSOLVE [9].

2 Preliminaries

In order to specify distributed reconfigurable systems, we use a graph-based approach. First, we model the behavior of a system using a directed graph, called *task graph*. The vertices of the task graph represent tasks $t \in T$ where T is a finite set. The edges of the task graph model data dependencies $d \in D$ between the tasks, i.e., $D \subseteq T \times T$.

On the other hand, we model the architecture of our distributed reconfigurable system by a so-called *architecture graph*. An architecture graph is also a directed graph, where vertices correspond to reconfigurable nodes $r \in R$ of the network. Edges of the architecture graph model directed connections $c \in C \subseteq R \times R$ between the nodes.

To relate tasks $t \in T$ and reconfigurable nodes $r \in R$, mapping edges $m \in M$ map tasks to nodes. A mapping edge $m = (t, r)$ indicates that t may be executed on r . Note that more than one mapping edge could be associated with a task t or a reconfigurable node r , modeling possible bindings and resource sharing, respectively. Such graph-based models are also used in commercial systems like VCC [11].

Example 1. Figure 1 shows a specification of a distributed reconfigurable system. The set of tasks T and data dependencies D are given by $T = \{t_0, t_1, t_2\}$ and $D =$

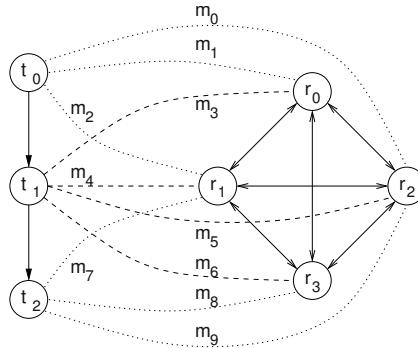


Fig. 1. Distributed control system consisting of a sample task (t_0), a control task (t_1), and a driver task (t_2). The architecture is composed of four reconfigurable nodes (r_0, \dots, r_3). The additional mapping edges (m_0, \dots, m_9) describe possible bindings.

$\{(t_0, t_1), (t_1, t_2)\}$, respectively. This task graph models the coarse grain behavior of a distributed control system, where t_0 corresponds to a sample task sampling a sensor, t_1 corresponds to the control task implementing the actual control, and t_2 models the driver task driving an actuator.

The architecture graph in Figure 1 consists of the reconfigurable nodes $R = \{r_0, r_1, r_2, r_3\}$. Each reconfigurable node could directly communicate with each other, i.e., the architecture graph is a clique. The mapping edges $M = \{m_0, \dots, m_9\}$ indicate that the sample task t_0 may be executed on any reconfigurable node r_0 to r_2 and the driver task t_2 may be performed on any of the reconfigurable node r_1 to r_3 . The controller task t_1 could be bound to any of the reconfigurable nodes in the architecture graph.

3 Binding

With the model introduced previously, the task of system synthesis could be formulated as: “Find a feasible *binding* of the tasks $t \in T$ to reconfigurable nodes $r \in R$, i.e., a subset of mapping edges.” Here, a binding is said to be feasible if:

1. each task $t \in T$ is bound to exactly one reconfigurable node $r \in R$ and
2. required communications given by the data dependencies $d \in D$ can be handled by the given architecture graph, i.e., if there is a directed edge $d = (t_i, t_j)$ between task t_i and task t_j then either t_i and t_j have to be performed on the same reconfigurable node r (intra-node communication) or on reconfigurable nodes r_i and r_j which are directly connected via an edge $c = (r_i, r_j)$ (inter-node communication).

Bickle et al. [12] have reduced the problem of finding a feasible binding to the boolean satisfiability problem which is NP-complete. In this paper, we show how to derive boolean functions from specifications given by a task graph, an architecture graph, and the mapping edges as described in Section 2 such that the boolean function is satisfiable iff the specified distributed reconfigurable system has a feasible binding. These function

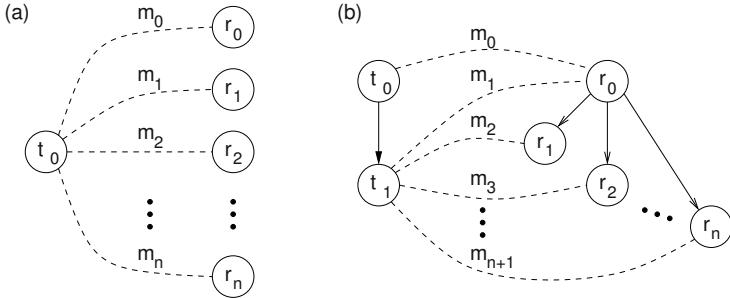


Fig. 2. (a) For each task $t \in T$ exactly one outgoing mapping edge has to be activated. (b) In order to establish the required communication ($d = (t_0, t_1)$), we have to execute the tasks t_0 and t_1 on the same reconfigurable node r_0 or on adjacent reconfigurable nodes.

could be tested by QBF solvers. Later, we extend this idea in order to analyze aspects such as whether a distributed reconfigurable system is fault tolerant and to what degree.

First, we consider the problem of checking the feasibility of a given binding. Therefore, we introduce some notations: Let m_i be a boolean variable, indicating if the mapping edge m_i is part of the binding ($m_i = 1$), or not ($m_i = 0$). The assignment of all variables m_i is denoted by (m) . Note, that (m) is a binary coding of the binding. The set of all possible assignments of (m) is denoted by (M) . With these new notations, we check the feasibility of a given binding represented by the coding (m) by solving a boolean equation. Therefore, we test both criteria of the feasibility as given above.

Example 2. First, we test if there is exactly one outgoing mapping edge for each task $t \in T$ in the binding. As an example consider Figure 2(a). There is a single task t_0 and $n+1$ mapping edges m_0, \dots, m_n . A boolean function that indicates if there is exactly one outgoing mapping edge for t_0 in conjunctive normal form (cnf) is: $(m_0 + m_1 + m_2 + \dots + m_n) \cdot (\overline{m}_0 + \overline{m}_1)(\overline{m}_0 + \overline{m}_2) \dots (\overline{m}_0 + \overline{m}_n) \cdot (\overline{m}_1 + \overline{m}_2) \dots (\overline{m}_1 + \overline{m}_n) \dots (\overline{m}_{n-1} + \overline{m}_n)$. Here, $+$ denotes the boolean OR and \cdot is the boolean AND. The first clause ensures that at least one of the mapping edges is activated ($m_i = 1$). The remainder guarantees that at most one mapping edge is part of the binding. The conjunction of both parts results in the required property.

For each task $t \in T$ we have to establish a formula similar to the one given in Example 2. The logical product results in a boolean function $b_1 : (M) \rightarrow \{0, 1\}$ with $b_1((m)) = 1$ iff (m) contains exactly one mapping edge per task. Hence, we obtain Equation (1) where \prod denotes the boolean AND and \sum denotes the boolean OR.

$$b_1((m)) = \prod_{t \in T} \left[\left(\sum_{\substack{m \in M: \\ m=(t,r)}} m \right) \cdot \left(\prod_{\substack{m_i, m_j \in M: \\ m_i=(t,r_x) \wedge m_j=(t,r_y) \wedge r_x \neq r_y}} (\overline{m}_i + \overline{m}_j) \right) \right] \quad (1)$$

Now, that we are sure that the first criteria is fulfilled, we check the second property of feasible bindings. All data dependencies $d \in D$ must be provided by the architecture

of the implementation. Therefore, let $d = (t_i, t_j)$. If t_i is executed on r_x then t_j has to be performed on r_x also or on an adjacent reconfigurable node r_j , i.e., $(r_i, r_j) \in C$.

Example 3. Consider the example in Figure 2(b). The task t_0 is bound to r_0 by m_0 . The execution of task t_1 must take place on node r_0 itself (by m_1) or on any adjacent reconfigurable node r_1, \dots, r_n . An implication that assures this property is given by $m_0 \mapsto (m_1 + m_2 + \dots + m_{n+1})$. This is equivalent to the clause: $(\bar{m}_0 + m_1 + m_2 + m_3 + \dots + m_{n+1})$. This equation needs to be satisfied for each mapping edge $m \in M$.

A boolean function $b_2 : (M) \rightarrow \{0, 1\}$ to check the second property of feasibility is therefore:

$$b_2((m)) = \prod_{\substack{m=(t,r) \in M, \\ t_i \in T: (t,t_i) \in D}} \left[\bar{m} + \sum_{\substack{m_j \in M: m_j = (t_i, r_x) \wedge \\ (r=r_x \vee (r, r_x) \in C)}} m_j \right] \quad (2)$$

With Equation (1) and (2), we formulate a boolean function $b : (M) \rightarrow \{0, 1\}$ to check the feasibility of a given binding coded by (m) :

$$b((m)) = b_1((m)) \cdot b_2((m)) \quad (3)$$

b is given in cnf and $b((m)) = 1$ iff the system has a feasible binding. In order to check if there is at least one feasible binding for a given specification, a SAT solver may be used to solve the following problem

$$\exists(m) : b((m)) \quad (4)$$

Checking whether a binding is feasible or whether a partial binding may be completed can be an important task during synthesis, but also in dynamically reconfigurable distributed systems. One application of the above SAT-techniques is therefore the domain of fault tolerance.

4 Fault Tolerance

In reconfigurable systems, the binding may change over time. Therefore, it may be possible to compensate resource faults by *rebinding* tasks to fully functional resources. The process of rebinding is called *repartitioning*. Recent research is focused on the OS support for single FPGA architectures [3]. In this section, we show how to model resource faults and how to measure the robustness of a given distributed reconfigurable system that supports repartitioning, also using SAT-based techniques. Therefore, we define the so-called *k-bindability* which quantifies the number of redundant resources in such a system.

4.1 Modeling Resource Faults

If a reconfigurable nodes fails, i.e., we cannot use this node for task execution any longer, the *allocation* of resources nodes may change. The allocation is the set of used resources

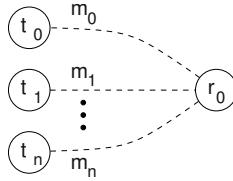


Fig. 3. In case of a resource defect, all incoming mapping edges must be deactivated.

in our implementation. Furthermore, an allocation is said to be feasible if there exists at least one feasible binding for this allocation. As in the case of the binding, we use the term (r) as the coding of an allocation where $r_i = 1$ indicates that the reconfigurable node r_i is part of the allocation. Furthermore, the term (R) describes the set of all possible allocation codings. If a reconfigurable node r_i fails, we have to set the associated binary variable r_i to zero ($r_i = 0$). All adjacent mapping edges m_j to r_i become meaningless, and should not be used in the binding, i.e., $m_j = 0$.

Example 4. Figure 3 shows a single reconfigurable node r_0 and n mapping edges. If r_0 fails m_0, \dots, m_n must not be used in the binding. We express this fact by n implications in the form $\bar{r}_0 \mapsto \bar{m}_j$ with $j = 0, \dots, n$. In cnf we get: $(r_0 + \bar{m}_0)(r_0 + \bar{m}_1)(r_0 + \bar{m}_2) \dots (r_0 + \bar{m}_n)$.

Again, we propose a boolean function to deactivate all mapping edges adjacent to a defect reconfigurable node. This boolean function $e : (M) \times (R) \rightarrow \{0, 1\}$ is satisfiable iff no reconfigurable nodes fail or there exists a feasible binding not using any of the mapping edges to the defect node.

$$e((m), (r)) = \prod_{r \in R, m \in M: m=(t, r)} (r + \bar{m}) \quad (5)$$

With this formula, we can check if a given reconfigurable node is redundant. For example, if we want to test if r_0 is redundant we solve the following SAT formula:

$$\exists(m), (r) : \bar{r}_0 \cdot e((m), (r)) \cdot b((m))$$

4.2 k-Bindability

A frequent question is how many resources could fail in a distributed reconfigurable system that supports repartitioning without losing the desired functionality. Therefore, we define the number of nodes that may fail as *k-bindability*, i.e., k is the maximum number such that any set of k reconfigurable nodes is redundant. Note that we can remove any $n < k$ nodes of our distributed system without losing the specified functionality.

Example 5. Figure 1 shows the specification for a distributed control system. Let us remove one of the reconfigurable nodes r_0, \dots, r_3 in Figure 1. Whatever node we choose, by rebinding the tasks we retain a running system, i.e., our system is at least 1-bindable. If we simultaneously remove any two of the reconfigurable nodes, again, our system remains working through rebinding the tasks. Now, our system is at least 2-bindable.

Let us check for 3-bindability. If we remove the reconfigurable nodes r_0, r_1, r_3 (or they fail simultaneously), we could bind all tasks t_0, t_1 , and t_2 to node r_2 . But the system is not 3-bindable, since if the nodes r_0, r_1 , and r_2 fail simultaneously, we can not find any reconfigurable node to bind task t_0 to. That is: the system is 2-bindable.

In order to check for k -bindability using SAT-based techniques, we formulate a boolean function which encodes all system errors with exactly k reconfigurable node defects:

$$f^{(k)} = (A) \times (R) \rightarrow \{0, 1\} \quad (6)$$

This function depends on the auxiliary variables a_i with $i = 0, \dots, |R| - 1$ where $|R|$ denotes the cardinality of R . If exactly k auxiliary variables are set to zero, we set the k corresponding allocation variables r_i to zero, otherwise all allocation variables may be set to one (i.e., no node fails).

Example 6. For the reconfigurable nodes in Figure 1, we encode the single resource defect of r_0 as: $a_3 a_2 a_1 \bar{a}_0$. The implication $(a_3 a_2 a_1 \bar{a}_0) \mapsto \bar{r}_0$ forces the allocation variable r_0 to zero. In cnf this corresponds to: $(\bar{a}_3 + \bar{a}_2 + \bar{a}_1 + a_0 + \bar{r}_0)$.

For all possible faults with exactly one single resource defect, we have to encode $|R|$ different cases (for each resource):

$$f^{(1)}((a), (r)) = \prod_{j=0}^{|R|-1} \left(\bar{r}_j + a_j + \sum_{i=0, i \neq j}^{|R|-1} \bar{a}_i \right)$$

$f^{(1)}((a), (r))$ does not impose any constraints on (r) if more than one variable a_i is set to false. With this boolean function, we check if for all of these faults there is at least one feasible binding. This is done by the following quantified boolean formula:

$$\forall(a) \exists(r), (m) : f^{(1)}((a), (r)) \cdot e((m), (r)) \cdot b((m))$$

We extend this approach by encoding all faults with exactly k resource defects. This is again an implication and can be written in cnf as:

$$f^{(k)}((a), (r)) = \prod_{i_1=0}^{|R|-1} \cdots \prod_{i_k=i_{k-1}}^{|R|-1} \prod_{l=1}^k \left(\bar{r}_{i_l} + \sum_{\substack{n=0 \\ n=i_1 \vee \dots \vee n=i_k}}^{|R|-1} a_n + \sum_{\substack{n=0 \\ n \neq i_1 \wedge \dots \wedge n \neq i_k}}^{|R|-1} \bar{a}_n \right)$$

Note again, that $f^{(k)}((a), (r))$ does not impose any constraints on (r) if $p > k$ variables a_i are set to false.

Now, that we know how to code resource defects in a boolean function, we formulate the general form of the QBF solving the k -bindability problem:

$$\forall(a) \exists(r), (m) : f^{(k)}((a), (r)) \cdot e((m), (r)) \cdot b((m)) \quad (7)$$

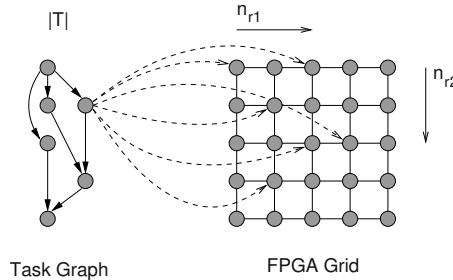


Fig. 4. Example specification of an FPGA grid and an application consisting of $|T| = 6$ tasks. The grid is composed of $n_{r1} \times n_{r2} = 5 \times 5$ FPGAs. The number of mapping edges per task is given by $n_m = 6$ and is shown only for one task.

5 Experimental Results

The k -bindability as defined above specifies the degree of fault tolerance of a distributed reconfigurable system. This fault tolerance can be optimized during system design. In order to compare different implementations during design space exploration, we must evaluate these implementations. In this section, we present first results of our new approach by using QSOLVE [9]. For this purpose, we design a benchmark. Our goal is to evaluate the runtime in dependence of the problem size (e.g., number of tasks, number of resources) which could be easily solved and, hence, could be investigated during automated design space exploration.

In a first step, an array of $n_{r1} \times n_{r2}$ reconfigurable nodes is defined. Communications are established such that the array is a 2-dimensional grid. Here, we use a grid in order to construct scalable architectures of distributed reconfigurable systems. Furthermore, a grid is typical for reconfigurable architectures as FPGAs and coarse grain architectures like PACT [13], Chameleon [14], etc. Next, we define a weakly connected random task graph with $|T| = n_t$ tasks. The probability that there is a data dependency between task t_i and task t_j with $j > i$ is given by another parameter called pb . In a last step, n_m mapping edges are randomly drawn from each task $t \in T$ to reconfigurable nodes $r \in R$, i.e., there are $|M| = |T| \cdot n_m$ mapping edges. Figure 4 shows an example of such a specification. The most meaningful results are presented in the following.

5.1 Feasibility of Binding

In a first test, we solve Equation (4) for randomly generated specifications. Here, three different $n_{r1} \times n_{r2}$ grids of reconfigurable nodes are investigated (5×5 , 10×10 , and 15×15). We map randomly generated task graphs onto each of these distributed architectures, while varying the number of tasks ($n_t = 50, 100, 150$). The tasks are connected with a probability of $pb = 0.5$. The number of mapping edges is chosen in a way, that feasible as well as infeasible systems are constructed. Only the cases of infeasible bindings are documented here (Finding a feasible binding by Equation (4) is the easier case). The average results (100 samples each) using QSOLVE [9] obtained on a PC system with a 1.8 GHz processor are shown in Table 1.

Table 1. Number of recursions recur, number of assignments assign, and computation time time required for solving (unsatisfiable) Equation (4).

	$ T = 50$			$ T = 100$			$ T = 150$		
	5×5	10×10	15×15	5×5	10×10	15×15	5×5	10×10	15×15
n_m	17	65	140	19	75	160	20	80	–
recur	21	70	180	23	116	237	29	176	–
assign	2149	30285	136675	6527	120421	421960	12809	266205	–
time/s	0.07	1.80	14.05	0.46	14.33	69.67	1.57	43.11	–

The number of recursions *recur* corresponds to the number of nodes in the search tree. We see that systems with 225 reconfigurable nodes and 16,000 mapping edges be checked in a reasonable amount of time (≈ 1 min). Note: We only construct weakly connected task graphs. If we were using n weakly connected task subgraphs that are not connected, our binding problem consists of n independent binding problems. A QBF solver solves these (sub)problems independent of each other, which is much easier.

5.2 k-Bindability

With the results above, we consider the k-bindability problem as described in Section 4. Table 2 shows the average results (100 samples each) obtained from solving the boolean functions with the QBF-solver QSOLVE. We have chosen a 4×4 grid of reconfigurable nodes. Different numbers n_t of tasks are mapped onto this architecture. With parameters $n_m = 13$ and $pb = 0.5$, we check the k -bindability for $k = 4, \dots, 1$. Table 2 shows that systems with 50 tasks are still solvable in a reasonable amount of time.

As mentioned above, our approach is not limited to grids of reconfigurable nodes but supports arbitrary topologies. Thus, it is possible to optimize the architecture of distributed reconfigurable systems by using SAT-based techniques during design space exploration.

Table 2. Number of recursions recur, number of assignments assign, and computation time time to solve the k-bindability equation (satisfiable) Equation (7).

$ T $	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$ T $	$k = 1$	$k = 2$	$k = 3$	$k = 4$		
25	recur	532	2759	10230	30106	40	recur	587	4972	19136	49858
	assign	8569	43856	163972	487057		assign	10179	77695	293135	827385
	time/s	0.10	0.60	2.22	6.72		time/s	0.23	1.78	7.06	20.26
30	recur	434	3235	11931	35222	45	recur	649	4524	17107	51307
	assign	7474	54488	201263	594378		assign	12109	86569	318774	933571
	time/s	0.12	0.94	3.62	10.69		time/s	0.35	2.47	9.15	26.28
35	recur	488	3593	13403	39787	50	recur	633	4893	17818	53272
	assign	9031	65010	240039	705878		assign	12573	93306	342441	1007054
	time/s	0.19	1.45	5.89	16.63		time/s	0.37	2.74	10.02	29.90

6 Conclusions

Distributed reconfigurable systems, e.g., arrays of reconfigurable hardware elements including FPGAs or medium and coarse granular reconfigurable systems such as PACT [13] and Chameleon [14], possess an inherent fault tolerance which can be optimized during system design. The main contribution of this paper is to provide an efficient method to determine the degree of fault tolerance of a system, the so-called k -bindability. Two particular problems were considered in this paper: (i) Does there exist a feasible binding for a given specification of a distributed reconfigurable system that supports repartitioning? (ii) How many resources may fail in a distributed reconfigurable system that supports repartitioning without losing any functionality? Both problems were solved by reducing the binding problem to quantified boolean formulas and applying the QBF solver QSOLVE in order to test the satisfiability of these formulas. We have shown by experiment that our new approach can easily be applied to systems of reasonable size as we will find in the future in the field of body area networks and ambient intelligence. Hence, this approach provides a way to optimize the architecture of distributed reconfigurable systems during design space exploration.

References

1. Dick, R., Jha, N.: CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems. In: Proceedigns of ICCAD'98. (1998) 62–68
2. Ouaiss, I., Govindarajan, S., Srinivasan, V., Kaul, M., Vemuri, R.: An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In: IPPS/SPDP Workshops. (1998) 31–36
3. Walder, H., Platzner, M.: Online Scheduling for Block-partitioned Reconfigurable Devices. In: Proceedings of Design, Automation and Test in Europe (DATE03). (2003) 290–295
4. Rintanen, J.: Constructing Conditional Plans by a Theorem-Prover. Journal of Artificial Intelligence **10** (1999) 323–352
5. Egly, U., Eiter, T., Tompits, H., Woltran, S.: Solving Advanced Reasoning Tasks Using Quantified Boolean Formulas. In: Proc. of the 17th Nat. Conf. on Artificial Intelligence. (2000) 417–422
6. Scholl, C., Becker, B.: Checking Equivalence for Partial Implementations. In: Proceedings of 38th Design Automation Conference, Las Vegas, USA (2001) 238–243
7. Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for Quantified Boolean Formulas. Information and Computation **117** (1995) 12–18
8. Cadoli, M., Giovanardi, A., Schaerf, M.: An Algorithm to Evaluate Quantified Boolean Formulae. In: Proc. of the 15th Nat. Conf. on Artificial Intelligence. (1998) 262–267
9. Feldmann, R., Monien, B., Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulas. In: Proc. of the 17th Nat. Conf. on Artificial Intelligence. (2000) 285–290
10. Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for Quantified Boolean Formulas. In: Proc. of the 17th Int. Joint Conf. on Artificial Intelligence. (2001) 275–281
11. Cadence: Virtual Component Co-design (VCC). (2001) <http://www.cadence.com>.
12. Blicke, T., Teich, J., Thiele, L.: System-Level Synthesis Using Evolutionary Algorithms. In Gupta, R., ed.: Design Automation for Embedded Systems. 3. Kluwer Academic Publishers, Boston (1998) 23–62
13. Baumgarte, V., May, F., Nückel, A., Vorbach, M., Weinhardt, M.: PACT XPP - A Self-Reconfigurable Data Processing Architecture. In: ERSA, Las Vegas, Nevada (2001)
14. Chameleon Systems: CS2000 Reconfigurable Communications Processor. (2000)

Hardware Implementations of Real-Time Reconfigurable WSAT Variants

Roland H.C. Yap, Stella Z.Q. Wang, and Martin J. Henz

School of Computing, National University of Singapore
Singapore
`{ryap, wangzhan, henz}@comp.nus.edu.sg`

Abstract. Local search methods such as WSAT have proven to be successful for solving SAT problems. In this paper, we propose two host-FPGA (Field Programmable Gate Array) co-implementations, which use modified WSAT algorithms to solve SAT problems. Our implementations are reconfigurable in real-time for different problem instances. On an XCV1000 FPGA chip, SAT problems up to 100 variables and 220 clauses can be solved. The first implementation is based on a random strategy and achieves one flip per clock cycle through the use of pipelining. The second uses a greedy heuristic at the expense of FPGA space consumption, which precludes pipelining. Both of the two implementations avoid re-synthesis, placement, routing for different SAT problems, and show improved performance over previously published reconfigurable SAT implementations on FPGAs.

1 Introduction

Stochastic local search (SLS) algorithms have been successful for solving prepositional satisfiability problems (SAT). The WalkSAT family (WSAT) of algorithms [1, 2] contains some of the best performing SLS algorithms. SLS algorithms like WSAT have a very simple structure and are composed of essentially three steps which are iterated until a satisfiable solution is found: (i) evaluate clauses; (ii) choose a variable; and (iii) flip the variable's boolean value.

Since each of the steps is simple, and as the SAT clauses can be directly represented in hardware, it is tempting to build a hardware-based SLS solver. There are a number of such hardware designs and implementations [3, 4, 5, 6] using reconfigurable FPGA hardware. Hardware approaches to systematic search procedures for SAT problems are beyond the scope of this paper; see [7] for an overview.

The use of a hardware SAT solver only makes sense if there is a significant performance advantage compared to software. Software can make use of state of the art processors built with the latest processor technology. A hardware SAT solver, on the other hand, is less likely to have the same level of process technology, and hence longer cycle times. Earlier hardware implementations like [3, 4] did not outperform optimized software. For example, a reimplementation of the design in Hamadi and Merceron [3] which was done in Henz et al. [6] had flip rates between 98 – 962 K flips/s. In some problems, this was a bit faster than software and in other cases slower.

In [6], it was shown that GSAT SLS solvers running at one flip per clock cycle was achievable with performance gains of about two orders of magnitude over software. That implementation makes use of the reconfigurable nature of FPGAs to build a custom design specific to a particular SAT problem instance. The contribution of [6] is to show that large speedups are feasible. This approach, however, is not practical as a general SAT problem solver because the time needed to re-synthesize, place and route the specific FPGA design is likely to exceed the runtime improvement from the faster solver.

This paper explores hardware designs for WSAT, which are not instance-specific and thus do not require re-synthesis. In addition to this requirement, a hardware implementation faces interesting design tradeoffs due to the inherently limited logic resources on the chip. We propose two versions of WSAT, which allow real-time reconfiguration. The differences of the WSAT versions lead to different design choices for maximal performance. The first design emphasizes fast cycle times (one flip per clock cycle), employing random variable selection to allow for a pipelined design. The second uses a greedy variable selection heuristic, which precludes pipelining, exemplifying a tradeoff between flip rate and effectiveness of variable selection. Both designs have improved performance over other published non-re-synthesis SLS FPGA implementations.

2 Hardware Implementation Issues

2.1 Cost of Re-synthesis FPGA Implementations

SLS SAT algorithms exhibit large amounts of parallelism and hence are a good match for a hardware solver, which can use the large amounts of parallelism available in the hardware. We focus here on WSAT implementations using Field Programmable Gate Arrays (FPGAs), which provide the benefits of customized hardware but avoid fabrication cost, and thus allow for convenient prototyping of the hardware design. Unlike software, a hardware implementation has to deal with the inherent resource limitations for combinatory logic, memory and routing on an FPGA.

One approach is to maximize performance by making full use of parallelism, exemplified in [6], where clause evaluation and variable selection are parallelized for a GSAT SLS implementation. However, such a high degree of parallelism is expensive in terms of hardware resources. That implementation optimizes the hardware design specifically for a given SAT problem instance, taking advantage of the reconfigurability of FPGAs. This *instance-specific* approach enabled a performance of one flip per clock cycle, more than two orders of magnitude faster than software. The drawback, however, is that a new solver has to be re-synthesized for each SAT problem instance. With current CAD tools, the synthesis, placement and routing for SAT instances with 200 variables can take several hours, while the resulting SAT solver may only take seconds or minutes to find a solution to the instance. Thus, while instance-specific hardware implementations demonstrate the feasibility of very high performance hardware approaches, they are impractical as general-purpose SAT solvers.

A general-purpose hardware SAT solver should instead not require re-synthesis, and be able to handle different SAT instances with only small overheads. One non-re-

synthesis approach is given in [4], which takes advantage of the fact that the FPGA configuration file can be altered directly to modify the design. This provides a shortcut to re-synthesis since only small modifications to the definitions of the SAT clauses are necessary. However, this implementation is mostly sequential and does not outperform optimized software. A more serious issue is that current FPGA chips do not have an open architecture. The configuration file for these chips is a black box, which renders this approach unfeasible.

Leong et al [5] achieved a bitstream reconfigurable FPGA implementation for a WSAT variant. Their implementation stores clauses for a SAT problem in the 16x1-bit ROM available in the Logic Cells (LC) of the Xilinx FPGA. A different SAT instance requires various ROM definitions to be modified. Normally, this would require re-synthesis of the FPGA to generate a new bitstream configuration for downloading. Leong et al were able to achieve a non re-synthesis implementation, using a tool to extract the locations of the relevant LCs in the bitstream, and then directly modify the corresponding data for the ROM values in the bitstream file. This approach requires analysis of the bitstream file to figure out how to rebuild the configuration without re-synthesis.

Both of these implementations [4, 5] simulate re-synthesis in a very efficient fashion. However, they are dependent on the ability to modify the FPGA configuration.

The aim of this paper is to obtain a more portable reconfigurable implementation, which nevertheless is capable of providing good search performance, and which exhibits short reconfiguration times.

2.2 A Clause Evaluator without Re-synthesis

The key to avoid re-synthesis is to be able to handle any SAT instance. Hence the clause evaluator in WSAT must be general rather than instance-specific. Our goal is a general clause evaluator, which fits well within an FPGA architecture and can be reconfigured quickly in a portable fashion.

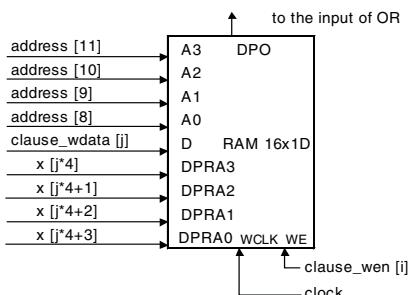
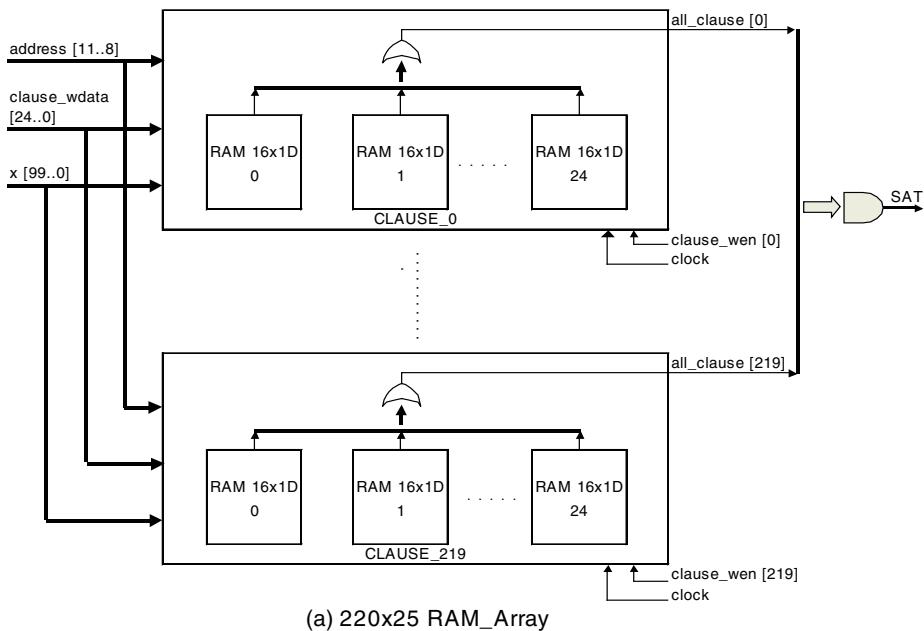
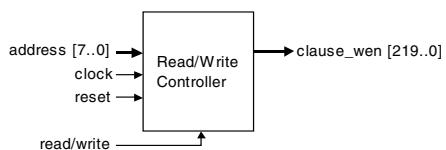
We will focus on the Xilinx Virtex FPGA chips. The basic building block of Virtex FPGA [8] is a LC, which includes a 4-input function generator, carry logic and a storage element. The 4-input function generator is implemented as 4-input look-up table (LUT). Each Virtex CLB (Configurable Logic Block) contains four LCs, organized in two slices. Two LUTs in a slice can be combined to create a 16x1-bit dual port RAM. Our clause evaluator represents the clauses in the SAT instance in a 16x1-bit dual port RAM array, which can be generated from the Xilinx RAM16x1D primitive. The Xilinx RAM16x1D primitive is a 16-word by 1-bit static dual port random access memory with synchronous write capability. The device has two separate address ports; the read address port (DPRA3-DPRA0) and the write address port (A3-A0).

We describe the clause evaluator by example. Consider a SAT clause, c_3 , of the form, $x_1 \vee x_2 \vee \bar{x}_5$, and let us assume that c_3 is a clause of a SAT problem over 8 variables. The clause can be written as a disjunction of two simpler functions,

$$f_{3,1}(x_1, x_2, x_3, x_4) \vee f_{3,2}(x_5, x_6, x_7, x_8)$$

where $f_{3,1}(x_1, x_2, x_3, x_4) = x_1 \vee x_2$ and $f_{3,2}(x_5, x_6, x_7, x_8) = \overline{x_5}$. Thus each SAT clause, c_i , can be decomposed into a disjunction of boolean functions on fewer variables. We map each $f_{i,j}$ arising from the j -th part of clause i to a RAM16x1D primitive, treating the four variables as the address to the read port (DPRA3-DPRA0). The function $f_{i,j}$ is configured by using the write port (A3-A0) to define its truth table.

One advantage of this representation is that negated variables are handled automatically inside the $f_{i,j}$ block. Figure 1(a) shows an overall block diagram of the reconfigurable clause evaluator for 100 variables and 220 clauses. Figure 1(b) shows each $f_{i,j}$ block, which is configured using the controller in Figure 1(c). The result of each RAM primitive is ORed and stored in the array $all_clause[]$. The clause evaluator evaluates all clauses in parallel in one cycle.

(b) Clause_i, RAM_j: $f_{i,j}()$ 

(c) Read/Write Controller

Fig. 1. Block Diagram of the Reconfigurable Clause Evaluator

3 Two FPGA Implementations without Re-synthesis

The reconfigurable clause evaluator requires $O(mn)$ CLBs for an implementation with m clauses and n variables. This component consumes a significant fraction of the available CLBs (as much as 80%). As we would like to be able to handle as large a problem as feasible within the constraints of the FPGA, it is impractical to consider implementations that require multiple clause evaluators. This would consume too much of the chip real estate, even if there is considerable parallelism gain. We present two implementations of WSAT for 3-SAT problems, which represent different tradeoffs in using a single reconfigurable clause evaluator.

3.1 A Pipelined FPGA Implementation Using a Random Selection Heuristic

One strategy is to produce an implementation with a fast cycle time. Given that we are constrained to a single clause evaluator, we are left with pipelining as the only option for increasing the flip rate. For maximal reuse of the clause evaluator, it is important that the pipeline be well balanced with simple pipeline stages. Given that we already have a fully parallel clause evaluator, the most expensive step in WSAT is variable selection. A particularly simple WSAT variant chooses the variable randomly in a selected unsatisfied clause. This strategy is also used in the WSAT implementation of Leong et al [5].

Figure 2 depicts a five-stage pipelined implementation. Stage 1 finds a random unsatisfied clause (this checks all clauses in parallel). Stage 2 generates three variable indices for the selected clause. Stage 3 implements the random selection heuristic, flipping of its input variables. Stage 5 checks for satisfiability. There are a number of storage buffers used. Buffer 1 stores the clause table which gives the mapping of clause to variables used within that clause as represented by a variable index. The SAT problem is initially loaded into buffer 2, which then is used to initialize the $f_{i,j}$ blocks in the clause evaluator. The result is a one flip per cycle implementation.

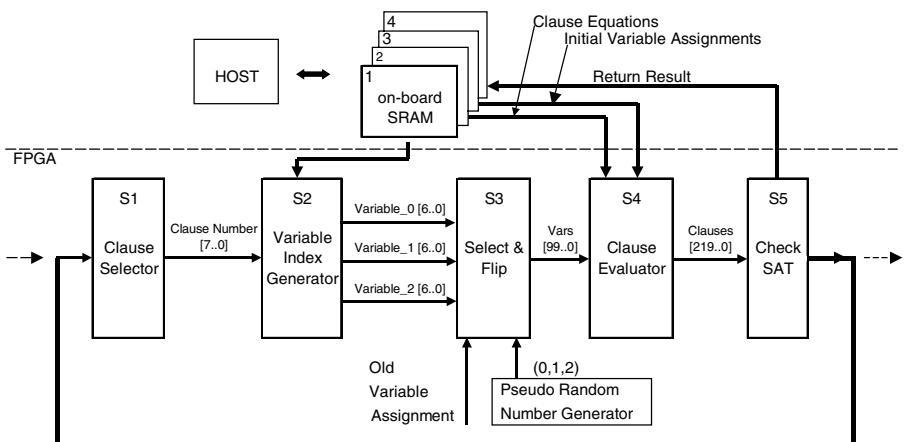


Fig. 2. Pipelined Random WSAT

3.2 A FPGA Implementation Using a Greedy Selection

A more typical WSAT variable selection heuristic is to select the variable, which best improves the score. In terms of the constraints of the hardware, this corresponds to a design with more complex operations. We have chosen to use a pure greedy heuristic without noise (but a noise component can be easily added).

Figure 3 shows the block diagram of a sequential implementation. Since we are dealing with 3-SAT, it is only necessary to determine at most which of the three variables in a clause to select. However, any kind of parallel implementation of this step would require computing the score of each of the three possibilities. This would require three clause evaluator units, which we deem too space consuming for the targeted SAT problem size. Thus, we are restricted to a sequential implementation for the variable selection (Stages 4-6), which reduces the flip rate. Our current implementation performs one flip in nine cycles, as opposed to one cycle achieved by the design for random selection heuristic.

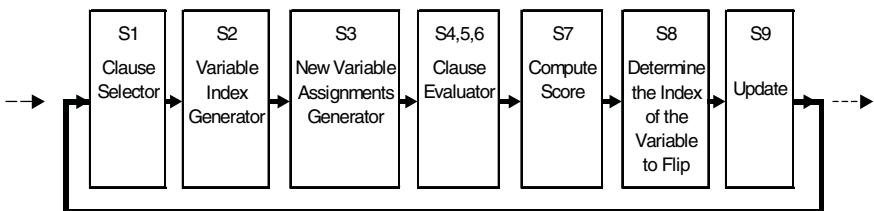


Fig. 3. Sequential Greedy WSAT

4 Results

Our hardware SAT solver is implemented on Celoxica's RC1000-PP standard PCI bus board, which is equipped with a Xilinx XCV1000 FPGA. This board has 8Mb of SRAM directly connected to the FPGA in four 32-bit wide memory banks. Each of the four banks may be granted to either the host CPU or the FPGA at any time. Data can therefore be shared between the FPGA and the host CPU by placing it in the SRAM. It is accessible to the host CPU by DMA transfer across the PCI bus.

As host we use a PC with an AMD Athlon 1.2GHz CPU. Our prototype generates the clause configuration for a new SAT instance in software in about 7ms (this is unoptimized and is probably dominated by file I/O and hence could possibly be faster). Transferring the clause configuration from the host PC to the on-board SRAM takes 0.6ms. The FPGA takes 220 x 16 clock cycles to read the SRAM. With an FPGA clock frequency of 20MHz, this corresponds to 0.176ms. Thus the configuration overhead for solving a new SAT instance is 7.776ms. In contrast, the time to download a new bitstream to the FPGA is around 0.14s.

The prototype implementations investigate the two designs on two SAT problem sizes; a 50 variable/170 clause format and a 100 variable/220 clause format, the latter chosen to such that its reconfigurable clause evaluator fits on the FPGA used. Table 1 gives the hardware costs in terms of slices for the various implementations. The

minimum gate delay is as reported by the Xilinx place and route tools. There is only a small difference in gate delay between the two implementations. The larger influence is the increased delay due to larger problem sizes.

Table 1. Time/Space Cost Comparison of FPGA-based Implementation

	Random-Strategy WSAT		Greedy-Strategy WSAT	
System Size	Delay (ns)	Cost of Slices	Delay (ns)	Cost of Slices
50-var/170-c	24.097	4946 (40%)	24.842	6408 (52%)
100-var/220-c	31.005	10396 (85%)	31.639	11834 (96%)

Table 2 shows the flip rate performance comparison given in number of flips per second (fps). We compare FPGA-based hardware implementations versus software for various 3-SAT benchmarks. The benchmarks used are simply those, which fit within the required problem sizes. As the main purpose of the benchmarks is to measure flip rate performance, the difficulty of the benchmarks is not so relevant, as such a mix of more difficult problems and the easier AIM benchmarks are used. Our FPGA implementations were clocked at 20Mhz. The software WSAT implementation is WalkSAT35 by Kautz and Selman [11] running on a Pentium4 1500Mhz PC.

Table 2. Flip Rate Speedups: FPGA-based Hardware versus Pure Software

SAT Problems	Random-Strategy			Greedy-Strategy	
	Software Flip Rate (Kfps)	Hardware Speedup		Software Flip Rate (Kfps)	Hardware Speedup
		Ours, Pipelined	Leong et al. [5]		
Uf20-9	407.6	49.91	0.89	265.8	8.38
Uf20-31	390.9	52.06	0.93	251.8	9.84
Uf20-37	405.8	50.11	0.90	303.2	7.34
Uf50-01	536.2	37.95	-	409.4	5.45
Uf50-010	466.2	43.70	-	459.6	4.84
aim-50-2_0-yes1-1	865.4	23.49	0.41	775.4	2.87
aim-50-2_0-yes1-2	859.8	23.73	0.45	775.3	2.89
aim-50-3_4-yes1-1	618.6	33.36	0.98	609.2	3.66
aim-50-3_4-yes1-2	612.6	33.56	1.58	596.0	3.74
aim-50-3_4-yes1-3	613.1	33.55	0.65	572.4	3.89
aim-50-3_4-yes1-4	609.3	33.83	0.42	561.2	3.97
aim-100-1_6-yes1-1	962.5	21.37	-	814.2	2.74
aim-100-1_6-yes1-2	968.4	21.09	-	787.4	2.86
aim-100-1_6-yes1-3	972.9	21.05	-	805.4	2.76
aim-100-1_6-yes1-4	1014.4	20.29	-	872.2	2.55
aim-100-2_0-yes1-1	838.5	24.07	-	744.9	3.00
aim-100-2_0-yes1-2	814.3	24.92	-	747.7	3.00
aim-100-2_0-yes1-3	812.6	24.93	-	731.4	3.06
aim-100-2_0-yes1-4	834.4	24.25	-	744.6	3.00

The flip rate for the random and greedy variable selection heuristics is constant throughout the problems – 20M flips for random, and 2.2M flips for the greedy heuristics, due to its 9-stage implementation. We also measured actual timings as a reality check. The "Hardware Speedup" columns represent the ratio of measured flip rate versus the software flip rate. Note that the software flip rate varies with the problem, while it is constant in our implementations.

The fourth column compares our pipelined random strategy with the WSAT reconfigurable FPGA implementation from Leong et al. [5], which also uses a random strategy. Their implementation uses a smaller FPGA with problems of up to 50 variables and hence could be clocked at a faster speed of 33Mhz. The speedup has been recomputed using the average timing results in their paper. Where timings or benchmarks are not available, this is indicated by a (-). A major difference between their implementation and the greedy pipelined one here is that our implementation is based on a constant flip rate. Their implementation, on the other hand, has a variable flip rate, because of the use of sequential clause selection and is bounded by a maximum flip rate of 364Kfps.

With the random variable selection heuristic, the preliminary results show that our reconfigurable FPGA implementation is significantly faster than software and previous hardware implementations. This implementation achieves one flip per clock cycle at 20Mhz. The greedy variable selection implementation has more modest speedups. The speedup is likely comparable to software or slightly faster, if the fastest state of art microprocessors are used, since performance scales at a lower rate with clock speed for microprocessors. However, the reduced flip rate may be offset by the increased effectiveness of the variable selection strategy. The greedy heuristic typically gives a better success rate than a random heuristic for WSAT. A detailed analysis of the effect of different variable selection heuristics is given in [9].

5 Conclusion

We demonstrate two prototype hardware solvers implemented on the Xilinx Virtex XCV1000 FPGA with significantly better performance than software and previous hardware WSAT solvers. Furthermore, the solvers are reconfigurable in real-time, with a reconfiguration time of a few milliseconds for problems with 100 variables. Our two implementations illustrate the tradeoff between time, space and effectiveness of the SLS algorithm. The random solver achieves an optimal flip rate at the cost of a simple variable selection strategy, while the greedy solver uses the more expensive and effective strategy but is not amenable to pipelining and is hence slower.

Both implementations are limited by the size of the Xilinx Vertex XCV1000 chip used, which can accommodate a reconfigurable clause checker only for problems with 100 variables and 220 clauses. This chip, dating from 1999, is fabricated using a 5-layer metal 0.22 μ m CMOS process. In comparison, the current Virtex-II generation uses an 8-layer 0.15 μ m CMOS process. The XC2V10000 has about 10 times more system gates than the XCV1000 and has significantly faster clock speeds. For example, a 100 variable/600 clause evaluator requires about 30K slices and fits in a XC2V6000 which has 6M system gates.

An FPGA implementation will have more limitations on problem sizes even when larger FPGAs are used. A fast hardware based solver can however still be useful for

general SAT solving. One approach is with hybrid search and stochastic solvers. For example, Zhang et al. [10] combine Davis Putnam with stochastic search. Their approach uses Davis Putnam to generate smaller sub-problems which are then solved with WSAT.

Another route to deal with larger problems is to use ASICs rather than FPGAs. Our implementation is not restricted to FPGAs since the reconfiguration for different SAT instances is not dependent on the reconfigurable logic of FPGAs. The prototype uses FPGAs simply because they are more cost effective for development. Given the real-time reconfiguration capability, this may be a promising candidate for direct ASIC implementation, which means higher clock speeds and much more resources for dealing with larger problems.

References

- [1] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. *Proc. National Conference on Artificial Intelligence*, 337-343, 1994.
- [2] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. *Proc. Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.
- [3] Youssef Hamadi and David Merveron. Reconfigurable architectures: A new vision for optimization problems. *Principles and Practice of Constraint Programming*, 209-221, 1997.
- [4] Wong Hiu Yung, Yuen Wing Seung, Kin Hong Lee, and Philip Heng Wai Leong. A runtime reconfigurable implementation of the GSAT algorithm. *Field-Programmable Logic and Applications*, 526-531, 1999.
- [5] P. H. W. Leong, C. W. Sham, W. C. Wong, H. Y. Wong, W. S. Yuen, and M. P. Leong. A bistream reconfigurable FPGA implementation of the WSAT algorithm. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(1): 197-200, 2001.
- [6] Martin Henz, Edgar Tan, Roland Yap. One flip per clock cycle. *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming*, 509-523, 2001.
- [7] M. Abramovici and A. Sousa. A SAT solver using reconfigurable hardware and virtual logic, *Journal of Automated Reasoning* 24(1/2): 5-36, 2000.
- [8] Xilinx. Virtex 2.5 Field programmable gate arrays, 1999.
- [9] H. Hoos and T. Stützle. Local search algorithms for SAT: An empirical evaluation *Journal of Automated Reasoning*, 24:421-481, 2000.
- [10] Wenhui Zhang, Zhuo Huang, Jian Zhang. Parallel Execution of Stochastic Search, Procedures on Reduced SAT Instances. *Proc. of the Seventh Pacific Rim International Conference on Artificial Intelligence*, 108-117, 2002.
- [11] H. Kautz and B. Selman, Walksat homepage,
<http://www.cs.washington.edu/homes/kautz/walksat/>

Core-Based Reusable Architecture for Slave Circuits with Extensive Data Exchange Requirements

Unai Bidarte, Armando Astarloa, Aitzol Zuloaga,
Jaime Jimenez, and Iñigo Martínez de Alegría

University of the Basque Country, E.T.S. Ingenieros,
Department of Electronics and Telecommunications,
Urquijo s/n, E-48013 Bilbao, Spain,
`{jtpbipeu,jtpascua,jtpzuiza,jtpjivej,jtpmamai}@bi.ehu.es`

Abstract. Many digital circuit's functionality is strongly dependant on high speed data exchange between data source and sink elements. In order to alleviate the main processor's work, it is usually interesting to isolate high speed data exchange from all other control tasks. A generic architecture, based on configurable cores, has been achieved for slave circuits controlled by an external host and with extensive data exchange requirements. Design reuse has been improved by means of a software application that helps on configuration and simulation tasks. Two applications implemented on FPGA technology are presented to validate the proposed architecture.

1 Introduction

When analyzing the data path of a generic digital system, three main elements can be distinguished:

- Data source, processor or sink. Any digital system needs some data source, which can be some kind of sensor or an external system. Process units transform data and finally data clients or sinks make some use of it.
- Data buffer blocks or memories and data exchange control units. The described data units cannot usually be directly connected and an intermediate storage element is needed. On the other hand, when transferring from one device to another, a communication channel and a predefined data transfer protocol must be followed. Data exchange control can be a very time consuming task. In order to liberate the main control unit, it is often adequate to use a data exchange control specific unit.
- High level control unit. It is the digital system master that generates all the control signals needed by the previously noted blocks.

This work studies slave digital systems with much data exchange, that is to say, circuits controlled by a host and with high volume data transfers. The following features summarize the system under study:

- There is a communication channel with the host, which sends control commands. The host can also be a data source or sink.
- Data exchange requires complex and high speed control, which makes a specific data exchange module necessary.
- It is needed to attend several data transfer requirements in parallel.
- Data processing is performed on data terminal units, so, for design purposes, data units are supposed to be source or sink. These elements are not synchronized, so an intermediate data storage is needed.

Fig. 1 represents the block diagram corresponding to the described slave system. Several circuits square with the specifications above: industrial machinery like filling or milling machines, polyphonic audio, generation of three dimensional images, video servers, PC equipment like plotters and printers,...

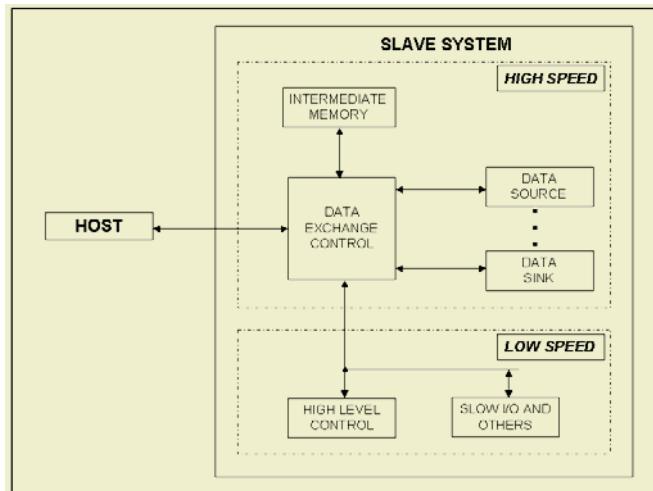


Fig. 1. Slave system's block diagram

The research team's main objective is to achieve a reusable architecture for the presented system. It must be independent of the implementation technology, so a standard hardware description language will be used. In order to facilitate the design reuse, a user friendly software application will be programmed to make the configuration of the architecture parameters and the simulation and verification easier.

2 Slave Digital Systems with Very Time Consuming Data Exchange. Design Alternatives

Traditionally, embedded systems like the one under study have been successfully developed with complex 16 or 32 bits microcontrollers. These process machines

perform millions of instructions per second, and include some communication channels, memory interfaces, direct memory access controllers, ... On the other hand, they present many disadvantages that make impossible to fulfill our design goals:

- As they are general purpose integrated circuits, no feature can be adapted to the application. Sometimes software patches will substitute hardware requirements.
- Although they have different communication interfaces, no frame protocol codification or decodification is usually available, so these tasks become software work.
- No data source or sink interfaces are available, so software and general purpose input and output ports are used.
- The mentioned features force the high level control machine to perform data exchange control tasks, so low speed work is seriously limited. An external circuit dedicated to these tasks can be used to alleviate the data exchange control bottleneck.

An optimum solution requires an architecture focused on high speed data exchange performed in an asynchronous mode between source and sink elements [1]. The complete slave system has been achieved on one chip [2]. The design is modular and based in parameterizable cores to facilitate future reuse [3].

There is an intermediate solution between general purpose microcontroller based solution and one chip solution. High level and low speed tasks can be performed by a microcontroller and high speed data exchange left for an autonomous hardware system. This solution can be adequate when features not available on the FPGA but on the microcontroller are needed, such as A/D or D/A data converters, FLASH or EEPROM memory, ... This is a less integrated and slower solution and it is dependant on the chosen microcontroller.

3 System on a Reprogrammable Chip Design Methodology

With today's deep sub-micron technology, it is possible to deliver over two million usable system gates in a FPGA. The availability of FPGAs in the one million system gate range has started a shift of System on Chip (SoC) designs towards using reprogrammable FPGAs, thereby starting a new era of System on a Reprogrammable Chip (SoRC) [4].

Nowadays market expects better and cheaper designs. The only way electronics industry can achieve these needs in a reasonable amount of time is with design reuse. Reusable modules are essential to design complex circuits [5].

So the goal of this work is to achieve a modular, configurable and reusable architecture that performs very high speed data exchange without damaging the low speed tasks. Hardware and software co-design and co-verification is also one of the objectives [6].

Traditionally IP cores used non-standard interconnection schemes that made them difficult to integrate. This required the creation of custom glue logic to connect each of the cores together. By adopting a standard interconnection scheme, the cores can be integrated more quickly and easily by the end user. A standard data exchange protocol is needed in order to facilitate SoRC design and reuse. Excluding external system buses such as PCI, VME, USB and so forth, there are many SoC interconnection buses. Most of them are proprietary: Advanced Microcontroller Bus Architecture (AMBA, from ARM), CoreConnect (from IBM) [7], FISPbus (from Mentor Graphics and Inventra Business Unit), IP interface (from Motorola), and many more. We looked for an open option, that is to say, a bus which does not need any license agreement and with no need to pay any kind of royalty.

The solution is the Wishbone SoC interconnection architecture for portable IP Cores. Wishbone standard defines the data exchange among IP Core modules, and it does not regulate the application specific functions of the IP Core [8]. It offers a flexible integration solution, a variety of bus cycles and data path widths to solve various system problems, and allows cores to be designed by a variety of designers. It is based on a Master / Slave architecture for very flexible system designs. All Wishbone cycles use a handshaking protocol between Master and Slave interfaces.

4 SoRC Core-Based Architecture

The SoRC architecture shown in Fig. 2 complies with the specifications noted. The main objective of the architecture is to isolate high speed data exchange from any other control tasks in the system. That is why the design has been divided into three blocks, each one with its own specific bus:

- The “Data Exchange” block is responsible of all data transfers and uses the high speed Wishbone SoC interconnection Architecture for Portable IP Cores, which makes possible high speed data exchange. A shared bus interconnection with only one Master has been chosen, which controls all transfers in the bus.
- The “Control” block, usually a microcontroller, is the system high level manager, and performs all other tasks. It uses its specific bus to read and write on input and output blocks and any other devices, as well as to communicate with the Wishbone bus.
- The “Host Communication” block is the communication interface with the host part. It can not directly access to the high speed bus and it exchanges information with the frame receiver and transmitter, which is a module on the Wishbone bus.

On the other hand, the SoRC has these connections with the outside: the host communication channel, the memory bus, the data source and / or sink devices interface and the microcontroller side devices interface.

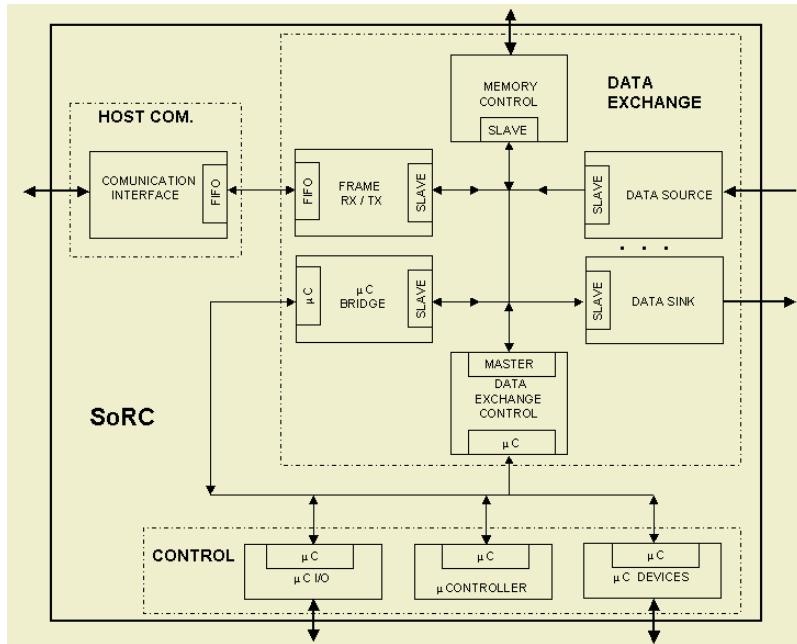


Fig. 2. SoRC architecture

4.1 Communication IP Cores

This core must interface the communication channel. After dealing with a number of different communication channels, the solution we have chosen is:

- If the communication interface needs a very complex controller (Bluetooth, USB) and there is an adequate solution available in ASSP or ASIC format, it is useful and practical to use it. In those cases only the channel interface is implemented into the SoRC.
- For non-complex ones (UARTs, parallel port buses such as IDE or EPP) both the controller and the interface are embedded into the SoC architecture leaving only the physical drivers and protection circuits outside the FPGA.

All the developed cores allow a full duplex communication. These cores have a common interface to the frame receiver/transmitter, which consists of two FIFO memories, one for reception and the other one for transmission. The communication interface presents two control signals to the frame controller to show the status of the FIFOs. The frame controller reads or writes the memories whenever it is needed.

4.2 Data Exchange IP Cores

DATA EXCHANGE CONTROL (DEC): this core allows data transfers between any two Wishbone compatible modules. It is the unique Wishbone master mod-

ule, so it controls all operations on the bus. The system critical task is high speed data exchange, which must be performed in parallel between different origin and destination pairs. To complete any transfer, the DEC must read the data from the origin and then write it in the destination. Many transfer request can be activated concurrently, so the DEC must be capable of serving them. In order to guaranty that no request is blocked by another one, the DEC priorities them following a round robin scheme.

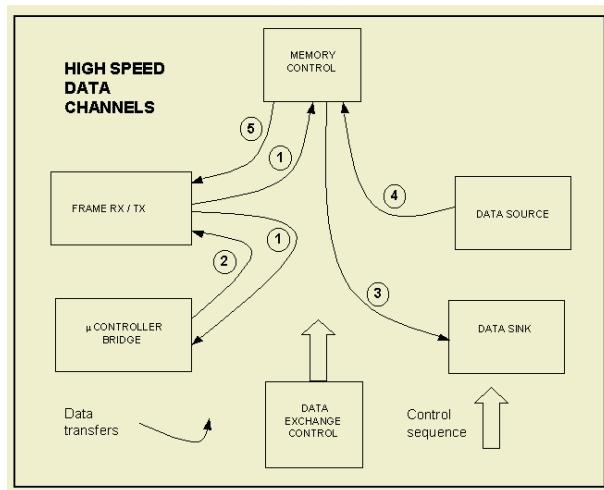


Fig. 3. High speed bus data exchange channels

The key to the control is to manage the right number of data channels, which must be exactly the number of concurrent data movements that can be accepted. Fig. 3 summarizes the solution adopted for hypothetical case with one data source and one data sink. The data channels are as follows:

- From the frame controller to the microcontroller bridge, in case of commands transmission, or to the memory, if data transmission to a sink.
- From the microcontroller bridge to the frame controller.
- From the memory to a data sink.
- From a data source to the memory.
- From the memory to the frame controller.

The number of channels is three plus one additional channel for each data source or sink. Each channel has three control registers: origin, destination and transfer length. Some of them are fixed and others must be configured by the microcontroller before starting data exchange. Additionally, there are two common registers which contain one bit associated to each channel: the control register to enable or disable transfers, and the start register, which must be asserted by

the microcontroller, after correctly configuring the three registers of the channel, to start the data transference.

An interruption register is used to acknowledge the termination of the data exchange to the microcontroller. Once the requested data transfer is accomplished, the DEC asserts the interruption register bit associated with the channel. There is only one interruption line, so whenever an interruption occurs, the microcontroller must read the interruption register, process it and deactivate it.

Partial address decoding has been used, so each slave decodes only the range of addresses that it uses. This is accomplished using an address decoder element, which generates all chip select signals. The advantages introduced are: it facilitates high speed address decoding, uses less redundant address decoding logic, supports variable address sizing and supports variable interconnection scheme.

FRAME RX/TX: frame information contains a header, commands for the microcontroller, file configuration, data and a check sequence. The receiver part of this module decodes data frames and sends data to correct destination under the DEC control. The transmitter part is responsible of packaging the outgoing information. This core permits full duplex communication, so receiver and transmitter parts are completely independent.

The high level control block must know about command reception because it configures all transfers. This core generates two interruptions, one in case a command is received and another one whenever a communication error is detected. These are the two only interruptions not generated by the DEC.

When data intensive communication is performed, some kind of data correctness check must be performed. The frame controller is able to perform different kinds of checksum coding and decoding.

MICROCONTROLLER BRIDGE: the microcontroller can not access data on the Wishbone bus directly, so an intermediate bridge between the high speed bus and the microcontroller low speed bus is needed. It must adapt data and control interfaces. Usually the data bus on the high speed side is wider than on the microcontroller side, so one data transfer on the Wishbone side corresponds to more than one operations on the microcontroller side.

MEMORY INTERFACE: data exchange between data source and sink elements is supposed to be performed in asynchronous mode. This is possible using an intermediate data buffer. Large block of RAM memory is needed in data exchange oriented systems and stand alone memories provide good design solutions. The design must be capable of buffering several megabytes, so dynamic memory is needed, and in order to optimize memory access, it must also be synchronous. So synchronous and dynamic memory (SDRAM) controller has been developed [9].

High speed systems like the one presented here must follow synchronous methodology rules. The generation, synchronization and distribution of clock signals is essential. FPGAs designed for SoRC provide high speed, low skew clock distributions through dedicated global routing resources and Delay Locked Loop (DLL) circuits. A DLL works by inserting delay between the input clock and the feedback clock until the two rising edges align. It is not possible to use one DLL

to provide both the FPGA and SDRAM clocks. Using two DLLs with the same clock input and separate feedback signals achieves zero delay between input clock, the FPGA clock, and the SDRAM clock.

DATA SOURCE / SINK: data from a source is written to memory and then transmitted to a sink. The interface to external data source or sink is application dependant.

4.3 High Level Control Unit

This is the slave system central process unit. We have assumed that it is orientated to data exchange, which means that this task is very time consuming and it justifies the specific data exchange control unit. All other tasks can be controlled by a general purpose machine, usually a microcontroller, and it will be chosen in accordance to the application. This multifunction machine uses its own bus to access memory, input/outputs, user interface, other devices, and the high speed bus as well. This bus must be coincident with the one on the bridge core.

A command reception interruption from the frame controller tells the microcontroller about the request from the host. The microcontroller reads it from the bridge, through the DEC, and processes it. If it is a control command, it will send back the answer command. If it is a data command, it will configure the corresponding data channel on the DEC and after this it will send back the acknowledge or answer command to the host. The DEC core will generate the data transfer end interruption when this operation is finished.

Whenever it is detected that data coming from the host is corrupted, the frame controller activates the error interruption. The microcontroller will tell back the host about the failure.

5 Configuration and Verification User Interface

In order to make the reuse and verification of the proposed architecture easier, a user interface application has been developed [10].

The use of a hardware description language like VHDL has allowed doing a parameterizable design. The specifications can be kept open and design alternatives can be evaluated, due to the fact that the design parameters can be modified. Design modularity and cores parameterization greatly improve future reuse possibilities.

To do a generic design, the effort needed at the beginning of the project is bigger than to do a closed design, in which component functionality is fully fixed. But this technique, apart from the advantages mentioned above, could greatly alleviate the unexpected problems that arise in the final stages of the design process.

Some hard coded values in the design have been replaced with constants or generics. In this way, even if the parameter is not going to be changed in the future, code readability is increased. A global package containing the definition of

all parameters has been used. The designer can configure the application specific architecture writing on the global package or using the software interface.

Once the desired architecture is configured, and after designing the application specific cores, the complete system functionality must be validated. All the cores, as well as the high level control machine code, must be co-simulated and co-verified. Modelsim from Mentor Graphics is the simulation tool used. It provides a Tool Command Language and Toolkit (Tcl/Tk) environment offering script programmability. A custom testbench has been created to facilitate the visualization of simulation results. A Tcl/Tk program creates a new display based on the simulator's output data, where a selection of the signals can be visualized with data extracted from the simulation results. Some buttons have been added so that new functionality is accessible. Dataflow can be graphically analyzed and design depuration is much easier. Host, communication channel, SDRAM and data source and sink functionality have been described using VHDL behavioural architectures. Data transfers on Wishbone bus are automatically analyzed by a supervisor core, which dramatically simplifies simulation.

6 Results and Conclusions

The following lines describe the application of the proposed architecture to the design of two digital systems.

The first one is a video system connected to a host via ethernet communication channel. The slave system controls two motors related to the camera movement and processes incoming control commands. The DEC core performs image data exchange between the analog to digital converter and the host. The SDRAM is used as a ping-pong memory: while a video frame is being captured, the previous one is being transmitted to the host. A general purpose evaluation board from Altera containing the 20K200EFC484 device has been used for prototyping. System features are summarized in Table 1.

The second one corresponds to an industrial plotter that provides high efficiency on continuously working environments. The microcontroller manages one stepping motor, three dc motors, many input/output signals and a simple user interface. The DEC module is dedicated to high volume data exchange from the host to the printer device using a parallel communication channel. Printing information is buffered in the SDRAM. This circuit is based on a Spartan II

Table 1. Implementation results

Features	Video Processor	Industrial Plotter
Equivalent gates	127.000	182.000
Internal RAM bits	16 Kbits	28 Kbits
User I/O pins	85	94
Max. DEC freq.	47 MHz	65 MHz
High level Proc.	Nios 32 bits	MicroBlaze 32 bits

family device from Xilinx which offers densities up to 200.000 equivalent gates. System features are summarized in Table 1.

The size, speed, and board requirements of today's state-of-the-art FPGAs make it nearly impossible to debug designs using traditional logic analysis methods. Flip-chip and ball grid array packaging do not have exposed leads that can be physically probed. Embedded logic analysis cores have been used for system debugging [11].

The results show that our SoRC architecture is suitable for generating hardware/software designs for slave digital systems with much data exchange. The multicore architecture with configuration parameters is oriented to reuse and a user friendly software application has been developed to help on application specific configuration and system hardware/software coverification. The use of a standard and open SoC interconnection architecture on the high speed bus improves the portability and reliability of the system and results in faster time to market.

References

1. Cesario, W., Baghdadi, A.: Component-Based Design Approach for Multicore SoCs. Design Automation Conference. Proceedings of the 39th conference. New Orleans, Louisiana (2002)
2. Bergamaschi, R., Lee, W.: Designing Systems on Chip Using Cores. Design Automation Conference. Proceedings of the 37th conference (2000)
3. Gupta, R., Zorian, Y.: Introducing core-based system design. IEEE Design and Test of Computers (October-December 1997) 15-25
4. Xilinx Design Reuse Methodology for ASIC and FPGA Designers. <http://www.xilinx.com/ipcenter/designreuse/docs/Xilinx-Design-Reuse-Methodology.pdf>
5. Bursky, D.: Core-based design leads the way to flexible system solutions. Electronic Design (May 1997)
6. Balarin, F.: Hardware-Software Co-design of Embedded Systems: The POLIS approach. Kluwer Academic Press (1997)
7. The Connect TM Bus Architecture. <http://www-3.ibm.com/chips/techlib/techlib.nsf/productfamilies/CoreConnect-Bus-Architecture>
8. Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores (rev. B.2), Silicore Corporation, Corcoran (USA), October 2001
9. Gleerup, T.: Memory Architecture for Efficient Utilization of SDRAM: A Case Study of the Computation/Memory Access Trade Off. Int'l Workshop on Hardware-software Codesign (2000) 51-55
10. Quinnell, R.: Development tool suits core-based design. Electronic Design (August 1996)
11. Chakrabarty, K.: Optimal test access architectures for system-on-a-chip. Transactions on Design Automation of Electronic Systems (January 2001) 26-49

Time and Energy Efficient Matrix Factorization Using FPGAs

Seonil Choi and Viktor K. Prasanna

Electrical Engineering-Systems, University of Southern California, Los Angeles, USA,
`{seonilch,prasanna}@usc.edu`, <http://ceng.usc.edu/~prasanna>

Abstract. In this paper, new algorithms and architectures for matrix factorization are presented. Two fully-parallel and block-based designs for LU decomposition on configurable devices are proposed. A linear array architecture is employed to minimize the usage of long interconnects, leading to lower energy dissipation. The designs are made scalable by using a fixed I/O bandwidth independent of the problem size. High level models for energy profiling are built and the energy performance of many possible designs is predicted. Through the analysis of design tradeoffs, the block size that minimizes the total energy dissipation is identified. A set of candidate designs was implemented on the Xilinx Virtex-II to verify the estimates. Also, the performance of our designs is compared with that of state-of-the-art DSP based designs and with the performance of designs obtained using a state-of-the-art commercial compilation tool such as Celoxica DK1. Our designs on the FPGAs are significantly more time and energy efficient in both cases.

1 Introduction

FPGAs have become an attractive option for implementing digital signal processing applications because of their high processing power and customizability [7]. The inclusion of new features in the FPGA fabric, such as a large number of embedded multipliers, further enhance their suitability. Recent FPGAs such as Xilinx Virtex-II(pro) [15] and Altera Stratix [1] offer hundreds of multipliers and large memory on a single chip. FPGAs can now be considered for implementing massively parallel and computationally demanding applications [12]. Also, with the proliferation of portable and mobile devices [2], it has become increasingly important that systems are not only fast, but also energy efficient. Even though state-of-the-art configurable devices offer very few features for power control, we show how to effectively use them to improve energy performance.

In this paper, we consider one of the important signal processing kernels: matrix factorization. For example, matrix factorization is a fundamental kernel in adaptive beamforming [9]. Approaches to future wireless communications such as software defined radio (SDR), require the mapping of such signal processing kernels onto reconfigurable hardware like FPGAs [14]. Moreover, the implementations have to be time and energy efficient. First, we develop a linear array

architecture based design for matrix factorization. Then we investigate and apply algorithmic techniques that use a block based approach to obtain time and energy efficient designs in FPGAs. Performance estimation (based on the time and energy performance models) is used for rapid design space exploration. Since block matrix factorization can be realized using various block sizes, we identify an optimal block size that minimizes the total energy dissipation based on the estimation. Candidate designs are implemented. To the best of our knowledge, there are no FPGA based designs for LU decomposition. Hence for the sake of comparison, we implement the matrix factorization on FPGAs using the state-of-the-art compilation tool, Celoxica DK1, with Handel-C [4] and also implement it in software on TI DSP devices. The performance of our designs is compared with that of the DK1 based design and the TI DSP benchmarks.

The remainder of this paper is organized as follows. In Section 2, we present our algorithms and architectures for matrix factorization. In Section 3, time and energy performance is estimated for the proposed algorithms and architectures. Section 4 presents the implementation details and the performance of these synthesized designs. Also, a comparison with Handel-C based and TI DSP-based implementations is made. Finally, Section 5 summarizes our work and discusses possible areas for future work.

2 Time and Energy Efficient Designs for Matrix Factorization

Several methods are known for factoring a given matrix [5]. In this paper, we choose to implement LU decomposition on FPGAs. Essentially, LU decomposition factors a $b \times b$ matrix into a $b \times b$ lower triangular matrix L (the diagonal entries are all 1) and a $b \times b$ upper triangular matrix U .

We propose two designs using two theorems. In Theorem 1, a new algorithm and architecture for LU decomposition is developed for a linear array of processing elements (PEs). Each PE performs computations on the input or intermediate matrix and the results are fed to the neighboring PE. Data dependencies between input and intermediate matrices are solved by efficient and regular scheduling. Each PE uses only two input ports: one for feeding input or intermediate matrices and the other for outputting the decomposed matrix. With this fixed I/O bandwidth regardless of problem size, we achieve an optimal latency of $b^2 + b - 1$ with leading coefficient of 1. The best latency of previously proposed designs [3] is $2b(b+1)$. In Theorem 2, a new parallel design on FPGAs for block LU decomposition is proposed. The design partitions a large matrix into multiple smaller blocks. To perform a computation for the smaller blocks, the architecture/algorithim in Theorem 1 is re-used. By varying the block size, we achieve time and energy efficient designs.

2.1 LU Decomposition

Let A be a $b \times b$ matrix. $a_{x,y}$ denotes an element of matrix A , where x is the row index and y is the column index. Similarly, $l_{x,y}$ ($u_{x,y}$) denotes an element of

matrix L (U). We assume that matrix A is a non-singular matrix and, further, we do not consider pivoting. The sequential algorithm in [8] consists of three main steps:

Step 1: The column vector $a_{x,1}$ where $2 \leq x \leq b$ is multiplied by the reciprocal of $a_{1,1}$. The resulting column vector is denoted $l_{x,1}$.

Step 2: $l_{x,1}$ is multiplied by the row vector $a_{1,y} (= u_{1,y})$ where $2 \leq y \leq b$. The product $l_{x,1} \times u_{1,y}$ is computed and subtracted from the submatrix $a_{x,y}$ where $2 \leq x, y \leq b$.

Step 3: Step 1 and 2 are recursively applied to the new submatrix formed in Step 2. An *iteration* denotes an execution of Step 1 and 2. During the k -th iteration, the column vector $l_{x,k}$ and the row vector $u_{k,y}$ where $k+1 \leq x, y \leq b$ are generated. The product $l_{x,k} \times u_{k,y}$ is subtracted from the submatrix $a_{x,y}$ where $k \leq x, y \leq b$ obtained during the $(k-1)$ -th iteration.

The time complexity of the sequential algorithm is $\Theta(b^3)$. We propose an architecture and algorithm on a linear array shown in Figure 1 using b PEs. The number of PEs p is the same as the problem size b . Essentially, PE_j performs computations for the j -th column of matrices L and U . Each PE consists of an adder/subtractor, a multiplier, a division lookup table, and a storage LU (p entries per PE). The storage LU of PE_j is used to store the j -th column of matrices L and U . Each PE has two input ports (a_{in} , LU_{in}) and two output ports (a_{out} , LU_{out}). a_{in} and a_{out} are used to feed in and out $a_{x,y}$ or $l_{x,y}$. LU_{in} and LU_{out} are used to output resulting matrices L and U to PE_b in a pipelined manner. Figure 1 (c) shows our algorithm by describing the operations in each PE during each cycle.

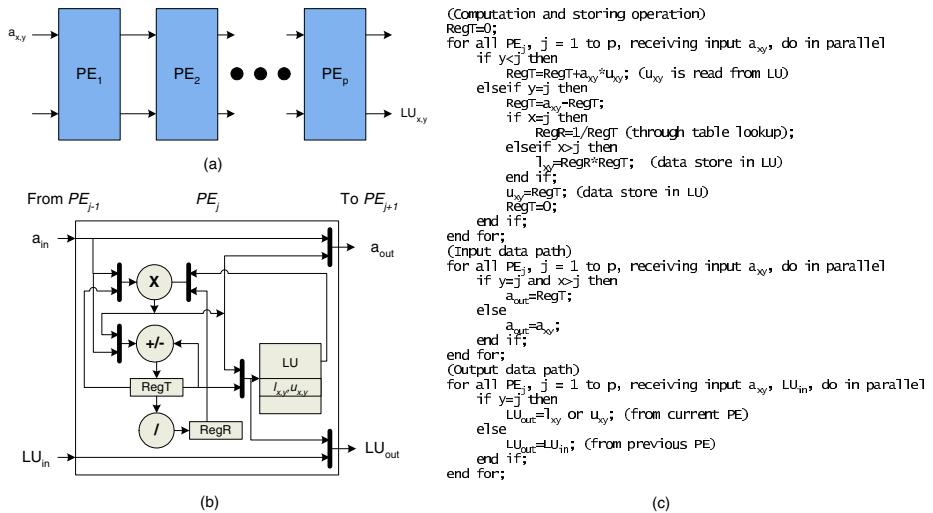


Fig. 1. (a) Overall architecture, (b) architecture of PE, and (c) algorithm for LU decomposition

Theorem 1. *LU decomposition without pivoting of a non-singular $b \times b$ matrix can be performed in $b^2 + b - 1$ cycles using the architecture and the algorithm in Figure 1 using b PEs.*

Proof. The elements $a_{x,y}$ of matrix A are fed in row major order ($a_{1,1}, a_{1,2}, a_{1,3}, \dots, a_{1,b}, a_{2,1}, \dots, a_{b,b}$) to \mathbf{a}_{in} of PE_1 . All data are fed from left to right. $a_{x,y}$ arrives at PE_j at cycle $b(x-1) + y + j - 1$ where $1 \leq j \leq b$. Seven operations are performed based on indices x, y and index j of PE_j . The indices x and y can be realized using counters in each PE. They also can be fed to PE_1 and propagated in a pipelined manner.

Op 1) Data propagation: $a_{x,y}$ is passed from PE_{j-1} to PE_{j+1} via PE_j except when $y = j$ and $x > j$. If $y = j$ and $x > j$, $l_{x,y}$ is generated at PE_j and is passed to PE_{j+1} via port \mathbf{a}_{out} . $l_{x,y}$ is also stored in the LU of PE_j .

Op 2) Multiplication/Accumulation: If $y < j$, a multiplication and an accumulation are performed in PE_j . $a_{x,y}$ ($= l_{x,y}$ generated at PE_{j-1}) is fed via port \mathbf{a}_{in} . During the k -th iteration, PE_j computes the product of the column vector $l_{x,k}$ and the j -th entry from $u_{k,y}$, where $l_{x,k}$ is a column vector generated in PE_k and $u_{k,y}$ is a row vector generated from PE_{k+1} to PE_b during the k -th iteration ($k+1 \leq x, y \leq b$). $u_{k,y}$ are stored in the LUs of PE_{k+1} to PE_b . An accumulation, $a_{x,y}^{(k)} = l_{x,k} \times u_{k,y} + a_{x,y}^{(k-1)}$, is performed after the multiplication during the same clock cycle. $a_{x,y}^{(k)}$ denotes the intermediate element of submatrix generated during the k -th iteration. $a_{x,y}^{(k)}$ is used either for another accumulation or for normalization (*Op 6*) and is stored in \mathbf{RegT} . \mathbf{RegT} is a temporary storage to hold $a_{x,y}^{(k)}$ during the accumulation. Note that accumulation and subtraction share one adder/subtractor since they do not occur simultaneously.

Op 3) Subtraction: If $y = j$, a subtraction is performed after all accumulations are complete. This ensures that $a_{x,y}^{(k)}$ is subtracted from the submatrix $a_{x,y}$ where $k \leq x, y \leq b$ during k -th iteration. For example, $u_{3,3}$ is computed as $\{-(l_{3,1}u_{1,3} + l_{3,2}u_{2,3}) + a_{3,3}\}$. In Step 2 of the sequential algorithm, the subtraction is performed after multiplication and the result is stored for the next subtraction. These operations are done repeatedly. For example, $u_{3,3}$ is computed as $\{(a_{3,3} - l_{3,1}u_{1,3}) - l_{3,2}u_{2,3}\}$.

Op 4) Storing: If $y = j$, $l_{x,y}$ or $u_{x,y}$ is generated in PE_j . If $x \leq j$, $u_{x,y}$ is stored in LU. If $x > j$, $l_{x,y}$ is stored in LU after normalization (*Op 6*). This operation ensures that the j -th column of the decomposed matrices L and U is stored in PE_j .

Op 5) Reciprocal: Division is required since the normalization is performed by $u_{k,k}$ for the column vector $a_{x,k}^{(k-1)} = (a_{k+1,k}, \dots, a_{b,k})$ during the k -th iteration ($1 \leq k \leq b$). $u_{k,k}$ is stored in \mathbf{RegT} after subtraction (*Op 3*) and the reciprocal value of $u_{k,k}$ is stored in \mathbf{RegR} . The reciprocal operation occurs if $x = y = j$.

Op 6) Normalization: After the subtraction (*Op 3*), the value is stored in \mathbf{RegT} . If $y = j$ and $x > j$, the values in \mathbf{RegT} and \mathbf{RegR} are multiplied. This operation generates the column vector $l_{x,k}$ where $k+1 \leq x \leq b$ in PE_k during the k -th iteration.

Op 7) Output: This operation sends out the results $l_{x,y}$ and $u_{x,y}$ in LU in a pipelined manner. If $y = j$, $l_{x,y}$ or $u_{x,y}$ is sent to port LU_{out} . Otherwise, $l_{x,y}$ or $u_{x,y}$ from PE_{j-1} is passed to PE_{j+1} via port LU_{out} .

To satisfy the data dependency of $l_{x,k}$ being generated during the k -th iteration and used during the $(k+1)$ -th iteration and to obtain the minimum latency, two conditions have to be satisfied. Note that the column vector $l_{x,k}$ ($k+1 \leq x \leq b$) is produced in PE_k during the k -th iteration. The first condition is that $l_{x,k}$ has to propagate from PE_k to PE_{k+1} after $l_{x,k-1}$ (generated during the $(k-1)$ -th iteration) propagates to PE_{k+1} and before $l_{x,k+1}$ is generated in PE_{k+1} during the $(k+1)$ -th iteration. Let $T_k(l_{x,j})$ be the sum of the time when $l_{x,j}$ is generated in PE_j and the propagation time when it reaches PE_k , which is $T_k(l_{x,j}) = b(x-1) + 2(j-1) + 1 + k - j$. Then, $T_{k+1}(l_{x,k-1}) = b(x-1) + 2k - 1$, $T_{k+1}(l_{x,k}) = b(x-1) + 2k$, and $T_{k+1}(l_{x,k+1}) = b(x-1) + 2k + 1$. Since $T_{k+1}(l_{x,k-1}) < T_{k+1}(l_{x,k}) < T_{k+1}(l_{x,k+1}) \rightarrow -1 < 0 < 1$, the condition is satisfied for all x where $k+1 \leq x \leq b$. To define the second condition, let $lu_{x,y}$ be $l_{x,y}$ if $x > j$, or $u_{x,y}$ if $x \leq j$ in PE_j . Note that $lu_{x,j}$ is computed in PE_j every b cycles. To output the resulting matrices L and U without delay, the second condition that $lu_{x,j}$ arrives at PE_k via port LU_{in} and LU_{out} before PE_k produces any $lu_{x,k}$ is required to satisfy. We assume $j < k$. Then, $T_k(lu_{x,j}) = b(x-1) + j + k - 1$ and $T_k(lu_{x,k}) = b(x-1) + 2k - 1$. Since $T_k(lu_{x,j}) < T_k(lu_{x,k}) \rightarrow j < k$, the second condition is satisfied for all x where $1 \leq x \leq b$. Total latency is calculated as the time taken for the last result $lu_{b,b}$ to be available as output: $T_b(lu_{b,b}) = b(b-1) + 2(b-1) + 1 = b^2 + b - 1$. \square

Since our design is a pipelined architecture, the first b cycles of the computations on the next matrix can be overlapped with the last b cycles of the computations on the current matrix. For a stream of matrices, one matrix can be decomposed every b^2 cycles. Thus the *effective latency* becomes b^2 , which is the time taken to obtain the first output data to the last output data during the current computation.

2.2 Block LU Decomposition

For large matrices, block LU decomposition can be performed. The sequential algorithm is given in [5]. An $n \times n$ matrix A is partitioned into four matrices: A_{11} , A_{12} , A_{21} , and A_{22} . A_{11} is a $b \times b$ matrix, A_{12} is a $b \times (n-b)$ matrix, A_{21} is an $(n-b) \times b$ matrix, and A_{22} is an $(n-b) \times (n-b)$ matrix. The algorithm is to decompose A into two $n \times n$ matrices, L and U , such that $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L'_{11} & 0 \\ L'_{21} & L'_{22} \end{pmatrix} \begin{pmatrix} U'_{11} & U'_{12} \\ 0 & U'_{22} \end{pmatrix}$. The steps of the algorithm are as follows:

Step 1: Perform a sequence of Gaussian eliminations on the $n \times b$ matrix formed by A_{11} and A_{21} in order to calculate the entries of L'_{11} , L'_{21} , and U'_{11} .

Step 2: Calculate U'_{12} as the product of $(L'_{11})^{-1}$ and A_{12} .

Step 3: Evaluate $A'_{22} \leftarrow A_{22} - L'_{21}U'_{12}$.

Step 4: Apply Step 1 to 3 recursively to matrix A'_{22} . During the k -th iteration, the resulting submatrices $L_{11}^{(k)}$, $U_{11}^{(k)}$, $L_{21}^{(k)}$, $U_{12}^{(k)}$, and $A_{22}^{(k)}$ are obtained. An *iteration* denotes an execution of Step 1 to 3.

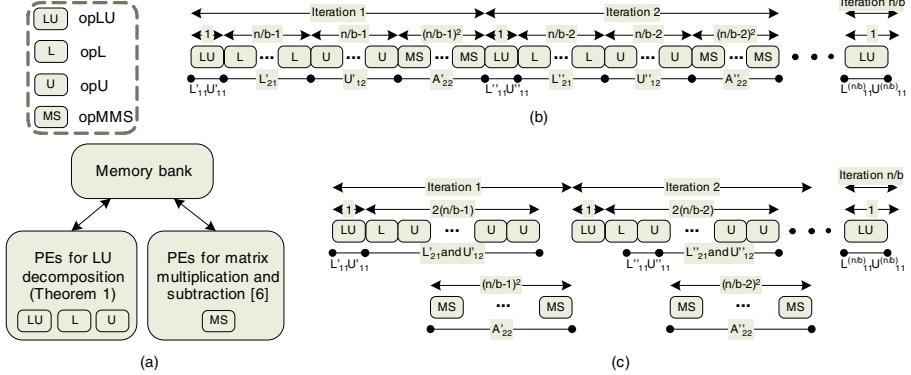


Fig. 2. (a) Overall architecture for block LU decomposition, (b) a schedule for Theorem 2 and (c) a schedule for Corollary 1

By utilizing the architecture and algorithm in Theorem 1 in combination with a matrix multiplication/subtraction architecture, we propose an architecture for block LU decomposition on FPGAs as shown in Figure 2. The block size b is later used as the parameter to realize time and energy efficient designs. There are two sets of PEs: one set performing a $b \times b$ LU decomposition and the other performing a $b \times b$ matrix multiplication/subtraction. Each set of PEs is linearly pipelined and both sets are connected to a memory bank. The input matrix is stored in the memory bank and fed to both sets of PEs. After computation, the results are stored back to the memory bank and used for next computation. Four different operations are identified: *opLU*, *opL*, *opU*, and *opMMS*. *opLU* is performed to obtain $L_{11}^{(k)}, U_{11}^{(k)}$. *opLU* from Step 1 is realized by using the algorithm and architecture proposed in Theorem 1. *opL* from Step 1 is performed to obtain $L_{21}^{(k)}$. The same architecture in Theorem 1 is used. However, the matrix $U_{11}^{(k)}$ and the reciprocal of its diagonal entries are required to perform *opL*. Since *opL* is performed after *opLU*, all PEs already hold the reciprocals in RegRs. $U_{11}^{(k)}$ is fed via port LU_{in} . We add one more data path that feeds the data from port LU_{in} to storage LU . *opU* from Step 2 is performed to obtain $U_{12}^{(k)}$. *opU* also uses the same architecture. It requires $L_{11}^{(k)}$ from *opLU*. $L_{11}^{(k)}$ are fed via port LU_{in} to storage LU . In Step 3, matrix multiplication/subtraction (*opMMS*) is performed. Once *opL* and *opU* are complete, $L_{21}^{(k)}$ and $U_{12}^{(k)}$ are available for *opMMS*. We have proposed an architecture for this operation [7]. Since there is matrix subtraction after matrix multiplication, additional subtraction logic is added. The matrix multiplication algorithm takes two $b \times b$ submatrices C from $L_{21}^{(k)}$ and D from

$U_{12}^{(k)}$ and computes the product $E \leftarrow C \times D$. Another $b \times b$ submatrix F is taken from $A_{22}^{(k)}$. Then the final values are obtained by $E \leftarrow F - E$. If b is large, the $b \times b$ matrix multiplication can be decomposed into $(\frac{b}{r})^3 r \times r$ matrix multiplications, where r is the sub-block size.

Theorem 2. *LU decomposition of $n \times n$ matrix can be performed in $n^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) \times (\frac{2n}{b} - 1) + b - 1$ cycles using the architecture in Figure 2 (a) using b PEs for $b \times b$ LU decomposition and r PEs for $r \times r$ matrix multiplication/subtraction, where b is block size and r is sub-block size.*

Proof. At a given time, only one operation is performed and the schedule is shown in Figure 2 (b). As all matrices are fed as streaming input, the computation on the current matrix and the next matrix can be overlapped. Therefore, each of $opLU$, opL , and opU has an effective latency of b^2 . $opMMS$ has an effective latency of $\frac{b^3}{r}$ [7]. There are $\frac{n}{b}$ iterations to complete the block LU decomposition. During each iteration, only one $b \times b$ $opLU$ is performed. The effective latency of all $opLU$ is $(\frac{n}{b})b^2$. During the k -th iteration, $(\frac{n}{b} - k)$ opL and opU for $b \times b$ block size are performed. The effective latency of all opL and opU is $b^2 \sum_{k=1}^{n/b} (\frac{n}{b} - k)$. During the k -th iteration, $(\frac{n}{b} - k)^2$ $opMMS$ for $b \times b$ block size are performed. Since $b \times b$ matrix multiplication can be decomposed to $(\frac{b}{r})^3 r \times r$ matrix multiplications, the effective latency of all $opMMS$ is $\frac{b^3}{r} \sum_{k=1}^{n/b} (\frac{n}{b} - k)^2$. The total latency is $(\frac{n}{b})b^2 + 2b^2 \sum_{k=1}^{n/b} (\frac{n}{b} - k) + \frac{b^3}{r} \sum_{k=1}^{n/b} (\frac{n}{b} - k)^2 + b - 1$, which includes the time to fill the pipeline stages. \square

Theorem 2 uses a straightforward schedule since only one set of PEs performs computations at a given time. We can utilize the two sets of PEs in parallel to reduce the total latency.

Corollary 1. *LU decomposition of $n \times n$ matrix can be performed in $3bn - 2b^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) (\frac{2n}{b} - 1) + b - 1$ cycles using the schedule in Figure 2 (c).*

Proof. After opL and opU for the first blocks are performed, the input matrices for $opMMS$ are ready. Thus, opL and opU can be performed in parallel with $opMMS$. The effective latency to complete the k -th iteration is $3b^2 + (\frac{n}{b} - k)^2 \cdot \frac{b^3}{r}$. During the $\frac{n}{b}$ -th iteration, only one $opLU$ is performed. Thus, the total latency is $\sum_{k=1}^{n/b-1} \{3b^2 + (\frac{n}{b} - k)^2 \cdot \frac{b^3}{r}\} + b^2 + b - 1 = 3bn - 2b^2 + \frac{1}{6} \frac{nb^2}{r} (\frac{n}{b} - 1) (\frac{2n}{b} - 1) + b - 1$, which includes the time to fill the pipeline stages. \square

While Corollary 1 reduces the total latency compared with Theorem 2, it does not reduce the amount of computation. Total energy is the sum of the energy used for computation and quiescent energy (the energy for configuration memory, static energy, etc.) used by the device even when the logic is idle. The quiescent energy depends only on the total latency. Since Corollary 1 reduces the latency, the quiescent energy and hence the total energy are reduced. Thus we use the architecture and algorithm in Corollary 1 to obtain both time and energy efficient designs.

3 Performance Estimation and Design Trade-Offs

For a given problem size n , varying the parameters such as block size b and sub-block size r creates a large design space. Before implementing the designs and performing low level simulation, we estimate the performance of possible designs, prune the design space, and finally identify “good” candidate designs for time and energy efficiency. The candidate designs were implemented using VHDL (See Section 4).

3.1 High Level Performance Model

To estimate the performance of our designs, we have employed domain-specific modeling proposed in [6]. Domain-specific modeling is a hybrid (top-down plus bottom-up) approach to performance modeling that allows the designer to rapidly evaluate candidate algorithms and architectures in order to determine the design that best meets criteria such as energy, latency, and area. An architecture is divided into *RModules* and *Interconnects*. RModules are hardware elements that are assumed to dissipate the same amount of power no matter where they are instantiated on the device and Interconnects are the wires connecting the RModules. From the algorithm, we know when and for how long each RModule is active. With this knowledge, we can calculate the latency of the design. Additionally, with estimates for the power dissipated by each RModule and the Interconnect, we can estimate the energy dissipated by the design. In the top-down portion of the hybrid approach, the designer’s knowledge of the architecture and the algorithm is incorporated, by deriving the performance models to estimate energy, area, and latency. The bottom-up portion is the power estimation of RModules and Interconnects from low level simulations. In our designs, the RModules are multipliers, adders, multiplexers, RAM, reciprocal lookup tables, and registers. The power values of each RModule are as follows. $P_{Mult}(= 11.25 \text{ mW})$ is the power dissipation for a 16×16 multiplier, $P_{Add}(= 1.34 \text{ mW})$ for a 16-bit adder/subtractor, $P_{Div}(= 7.31 \text{ mW})$ for a division lookup table (1024×16 bit), $P_{BSRAM}(= 7.31 \lceil x/1024 \rceil \text{ mW})$ for an on-chip memory where x is the number of entries, $P_R(= 1.17 \text{ mW})$ for a 16-bit register, and $P_{Store}(= 0.126 \lceil x/16 \rceil + 2.18 \text{ mW})$ for a storage LU where x is the number of entries. Table 1 lists the performance models of our designs. The latencies are converted to seconds by dividing them by the clock frequency.

3.2 Design Trade-Offs for Time and Energy Efficiency

To achieve time and energy efficient designs, we explore the various design parameters such as frequency, block size, precision, and number of PEs. All parameters contribute to energy dissipation, latency, and area of a design. For example, the latency and energy of Corollary 1 are a function of the block size b and the sub-block size r . By choosing $b = r = n$, the minimum latency of n^2 (546.1 μ sec at 120 MHz for $n = 256$) can be achieved. However, this design does not necessarily have minimum energy dissipation. We explore the parameters,

Table 1. Time and energy performance models

Design	Metric	Performance model
Theorem 1	Latency	$L_{Thm1} = b^2$
	Power	$P_{Thm1} = 6bP_R + bP_{Add} + bP_{Mult} + bP_{Store} + bP_{Div} + 2P_{BSRAM}$
	Energy	$E_{Thm1} = L_{Thm1} \cdot P_{Thm1}$
Theorem 2	Latency	$L_{opLU} = \left(\frac{n}{b}\right)b^2, L_{opU} = L_{opL} = \frac{1}{2}\left(\frac{n}{b}\right)\left(\frac{n}{b}-1\right)b^2$ $L_{opMMS} = \frac{1}{6}\left(\frac{n}{b}\right)\left(\frac{n}{b}-1\right)\left(2\frac{n}{b}-1\right) \cdot \frac{b^3}{r}$ $L_{Thm2} = L_{opLU} + L_{opL} + L_{opU} + L_{opMMS}$
	Power	$P_{opLU} = P_{opL} = P_{Thm1}$ $P_{opU} = 6bP_R + bP_{Add} + bP_{Mult} + bP_{Store} + 2P_{BSRAM}$ $P_{opMMS} = 8rP_R + rP_{Add} + rP_{Mult} + 2rP_{Store} + 3P_{BSRAM}$
	Energy	$E_{Thm2} = L_{opLU} \cdot P_{opLU} + L_{opL} \cdot P_{opL} + L_{opU} \cdot P_{opU} + L_{opMMS} \cdot P_{opMMS}$
Corollary 1	Latency	$L_{Cor1} = L_{opLU} + 2\left(\frac{n}{b}-1\right)b^2 + L_{opMMS}$
	Power	the same as the ones in Theorem 2
	Energy	$E_{Cor1} = E_{Thm2}$

b and r , and determine their values that minimize the energy dissipation. The estimates are based on 120 MHz designs by considering the operating frequency that can be achieved after implementation (See Section 4). Figure 3 (a) shows the energy dissipation as a function of b and r for $n = 256$. The minimum energy is obtained around $r = 16$ and $b = 16$ while the latency is 2743.5 μ sec. Note that the energy optimal design runs 5 times longer than the latency optimal design. Figure 3 (b) shows the energy distribution over four operations when $n = 256$ and b varies. When $b = 16$, we achieve the minimum energy design and $opMMS$ is the dominant source of energy dissipation. Through design space exploration, we found that the energy efficient designs are obtained when $b = r = 16$ for $n \geq 32$. Another interesting results are the energy distribution on the core operation, MAC (multiply-and-accumulate) and the rest of operations. In Figure 3 (c), approximately 50% of the total energy is used by MAC, which is relatively high compared with the a general purpose processor or DSP processor.

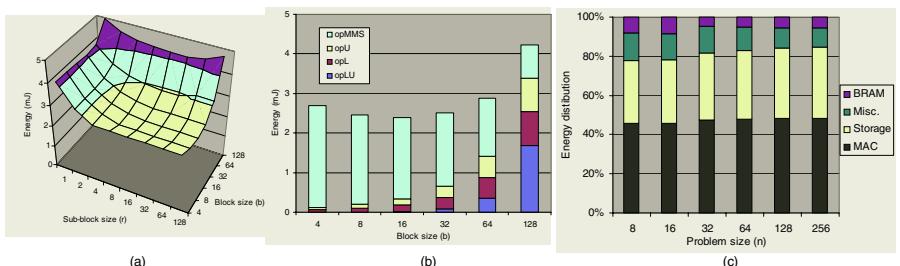


Fig. 3. (a) Energy dissipation as function of b and r for $n = 256$, (b) energy distribution as function of b for $n = 256$, and (c) energy distribution as function of n

4 Design Synthesis and Simulation Results

To obtain time and energy efficient designs, we briefly discuss the optimization techniques used in our designs. Then the synthesized designs for various problem sizes and the results from low level simulations are presented.

4.1 Optimizations for Time and Energy Efficiency

In this section, we summarize the energy efficient design techniques [7] employed in our designs. First, we have chosen a linear array of PEs. In FPGAs, long wires dissipate a significant amount of power [10]. For energy efficient designs, it is beneficial to minimize the number of long wires using a linear array since each PE communicates only with its nearest neighbors. Additionally, the linear array architecture facilitates the use of *parallel processing* and *pipelining*. Both techniques decrease the effective latency of a design and can lead to lower energy dissipation. Another technique is *block disabling*. We design the algorithm such that it utilizes the clock gating technique [15] to disable modules that are not in use during the computation. In our designs, since *opMMS* takes longer time than other operations, a set of PEs for *opLU*, *opL*, and *opU* becomes idle and is disabled to save energy. Another technique is choosing the appropriate *bindings*. In the Xilinx Virtex-II, the storage LU can be implemented as registers, distributed RAM, or embedded Block RAM. When the number of entries > 64, Block RAM is used since it is energy efficient for large memory; otherwise, distributed RAM is used. Similar decisions can be made such as choosing between (embedded) Block multiplier or configured multiplier. We choose Block multiplier since it is energy efficient when both inputs are not constant. To implement the division unit, a lookup table approach is used. This technique is faster and uses less energy compared with other division algorithms [11]. To calculate a/b , we first obtain $1/b$ via a lookup table and perform the multiplication $a \times (1/b)$. The approach is effective if the multiplication is fast. Using Block multipliers, fast multiplication (within one cycle) can be performed. The lookup table for reciprocal is generated as $Inv(b) = Round(2^m/b)$ where b is the value to be inverted, m is the number of bits used to represent the output, and *Round* is the rounding function.

4.2 Simulation Results

Using the performance models defined in Section 3, we identified the energy and time efficient designs based on the parameters. By considering different criteria such as area, latency, and energy, we identified several designs. The minimal energy designs are chosen as candidate designs and are implemented in VHDL. The precision of all designs was 16 bits. These designs were synthesized using XST in Xilinx ISE 4.2i and the frequency achieved was 120 MHz. The place-and-route file as an .ncd file was obtained for the Virtex-II XC2V1500 bg575-6 device. The input test vectors for the simulation were randomly generated such that their average switching activity was 50%. Mentor Graphics ModelSim 5.6b was used to

simulate the designs and generate the simulation results as a .vcf file. These .vcf and .ncd files were then used by the Xilinx XPower tool to evaluate the average power dissipation. Energy dissipation was obtained by multiplying the average power by latency. We also compared estimates from Section 3 against actual values based on implemented designs to test the accuracy of the performance estimation. We observed that the energy estimates (See Table 2) were within 10% of the simulation results. The the average power dissipation of the designs on Virtex-II included the quiescent power of 150 mW (from XPower).

Table 2. Performance of the designs based on Corollary 1 (from low-level simulation)

n	TI DSP (600MHz)			DK1 based design (50MHz)			Our design (120MHz)						
	b	L (usec)	E (uJ)	b	L (usec)	Slice/Mult	E _m (uJ)	b	L (usec)	Slice/Mult	E _{est} (uJ)	E _m (uJ)	Err in E _{est}
8	8	2.9	4.2	4	9.6	810/6	2.52	8	0.5	835/8	0.27	0.29	6.8%
16	16	7.2	10.6	4	35.0	810/6	10.27	8	2.1	1955/16	1.3	1.4	7.1%
32	16	31.5	46.6	4	218.2	810/6	42.86	16	10.7	3550/32	8.6	8.2	6.8%
64	16	152.7	229.7	8	1004.8	1318/10	342.7	16	51.2	3550/32	52.3	50.0	5.3%
128	16	836.4	1282.1	8	7087.9	1318/10	1485.7	16	345.6	3550/32	368.8	355.8	4.3%
256	16	5171.3	8068.0	8	56157	1318/10	6863.5	16	2743.5	3550/32	2801.6	2717.2	3.7%
512	16	35335	55857	8	453583	1318/10	34825	16	22421	3550/32	21927	21330	3.3%
1024	16	258619	412251	8	3658982	1318/10	198275	16	182473	3550/32	173710	169236	3.2%

* n is problem size. b is block size. L is latency. Slice is area. Mult is the number of embedded multipliers.

E_{est} is the estimated energy using the performance model. E_m is the measured energy from low level simulation.

We were not aware of any prior FPGA based designs for LU decomposition. Hence, for the sake of comparison, we implemented two baseline designs: one on FPGAs using a state-of-the-art commercial compilation tool and the other, a software implementation on state-of-the-art TI DSPs. The FPGA design was implemented using Handel-C and synthesized using Celoxica DK1.1 [4]. The compilation tool can automatically exploit the parallelism of the algorithm. The synthesized design, with a frequency of 50 MHz, was then implemented with the Xilinx ISE 4.2i. Our designs dissipated 8.8x to 1.2x less energy than the Handel-C based designs.

We also compared the performance of LU decomposition on FPGAs and DSPs. FPGAs are known to be better than DSPs in terms of time and energy performance. Since many target applications for DSP devices and FPGAs are similar, comparing their time and energy performance is beneficial to designers. We chose the TI TMS320C6415 running at 600 MHz as a representative DSP. TMS320C6415 is a high performance DSP and has eight 16-bit MAC units. The LU decomposition was implemented in C and its precision was 16 bits. The matrix multiplication was performed using the function call *DSP_mat_mul* from the TI DSP library. The latency was obtained by using the TI Code Composer 2.1. To compute the energy dissipation, we assumed the 75% high / 25% low activity category of power dissipation for the function call *DSP_mat_mul* since it is a hand-optimized code [13]. The power dissipation for the rest of C code is based on the 50% high / 50% low activity category since the code is optimized by the TI compiler. For the DSP, we chose the block size *b*, $0 < b < \min(n, 16)$ so as to minimize the energy dissipation. As seen from the results in Table 2,

our FPGA implementations perform LU decomposition faster using less energy. While we used the high performance DSP processor, TI also provides low power devices, namely the TMS320VC55xx series. Based on the datasheets, the 55xx series dissipate 150 mW at 300 MHz while the 64xx series dissipate 1500 mW at 600 MHz. The 55xx series have two MACs (600 MIPS) while the 64xx series have eight MACs (4800 MIPS). Thus the scaling factor from 64xx series to 55xx series for energy dissipation can be defined as: $s_e = \frac{P_{55xx}}{P_{64xx}} \times \frac{MIPS_{64xx}}{MIPS_{55xx}} = 0.78$. By applying this scaling factor, the energy dissipation of our designs was determined to be 12.1x to 1.8x less than the TI 55xx series.

5 Conclusion

We developed time and energy efficient designs for LU decomposition on FPGAs. Before implementing the designs, we analyzed the architecture and algorithm to understand the design trade-offs. After pruning the design space, selected designs were implemented using VHDL in the Xilinx ISE design environment. Currently, state-of-the-art FPGAs (e.g., Virtex-II/pro) do not provide low power features such as control for multiple power states or lower static power. The proposed architectures and algorithms are parameterized based on several design parameters. Hence, when more features such as dynamic voltage scaling with dynamic frequency scaling are available, the operations opL and opU can be executed slower than the operation $opMMS$. This might provide the opportunity to use a lower frequency and lower voltage.

Acknowledgements

This work is supported by the DARPA Power Aware Computing and Communication Program under contract F33615-C-00-1633 monitored by Wright Patterson Air Force Base and in part by the National Science Foundation under award No. 99000613. The authors wish to thank Gokul Govindu, Ronald Scrofano, and Zack Baker for helpful discussions and contributions to the low level simulation results.

References

1. Altera Corporation. <http://www.altera.com>. 2002.
2. J. Becker, T. Pionteck, M. Glesner. DReAM: A Dynamically Reconfigurable Architecture for Future Mobile Communication Applications. *FPL*, 2002.
3. E. Casseau, D. Degruillier. A Linear Systolic Array for LU Decomposition. *VLSI Design*, 1994.
4. Celoxica Corporation. DK1.1 Design Suite. <http://www.celoxica.com>. 2003.
5. J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, R. C. Whaley. The Design and Implementation of the Scalapack LU, QR, and Cholesky Factorization Routines. *Scientific Programming*, 5:173–184, 1996.

6. S. Choi, J. Jang, S. Mohanty, V. K. Prasanna. Domain-Specific Modeling for Rapid System-Wide Energy Estimation of Reconfigurable Architectures. *ERSA*, 2002.
7. S. Choi, R. Scrofano, V. K. Prasanna, J.-W. Jang. Energy Efficient Signal Processing using FPGAs. *Field Programmable Gate Array*, 2003.
8. T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*, McGraw-Hill, 2nd edition, 2001.
9. S. Haykin. *Adaptive Filter Theory*, Prentice Hall, 4th edition, 2002.
10. L. Shang, A. Kaviani, K. Bathala. Dynamic Power Consumption in Virtex-II FPGA Family. *Field Programmable Gate Arrays*, 2001.
11. N. Shirazi, A. Walters, P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. *FCCM*, 1995.
12. H. Styles, W. Luk. Customising Graphics Application: Techniques and Programming Interface. *Field Programmable Custom Computing Machines*, 2000.
13. Texas Instruments. TMS320C64xx Power Consumption Summary, <http://www.ti.com>.
14. W. Tuttlebee. *Software Defined Radio: Enabling Technologies*, J. Wiley, 2002.
15. Xilinx Incorporated. <http://www.xilinx.com>.

Improving DSP Performance with a Small Amount of Field Programmable Logic

John Oliver and Venkatesh Akella

Department of Electrical & Computer Engineering
University of California, Davis
{jyoliver, akella} @ece.ucdavis.edu

Abstract. We show a systematic methodology to create DSP + field-programmable logic hybrid architectures by viewing it as a hardware/software codesign problem. This enables an embedded processor architect to evaluate the trade-offs in the increase in die area due to the field programmable logic and the resultant improvement in performance or code size. We demonstrate our methodology with the implementation of a Viterbi decoder. A key result of the paper is that the addition of a field-programmable data alignment unit (FPDAU) between the register-file and the computational blocks provides 15%-22% improvement in the performance of a Viterbi decoder on the state-of-the-art TigerSHARC DSP. The area overhead of the FPDAU is small relative to the DSP die size and does not require any changes to the programming model or the instruction set architecture.

1 Introduction

Can we improve the performance and power or memory requirements of a state-of-the-art DSP with programmable logic? Many researchers have addressed this question in the past and many solutions have been proposed including customized instructions, loops [1], reconfigurable functional units, [2] and co-processor [3,4,5,6,7,8]. However most of the existing approaches do not factor the cost of the programmable logic in their evaluation - they tacitly assume that the die size penalty of adding programmable logic is not important. However, in embedded applications where DSPs are used, cost is a very critical factor. So, we would like to find a sweet spot for the programmable logic where a small addition to the die size in the form of programmable logic realizes maximum return in terms of improvements to performance (throughput), power, or memory requirements. For this we believe that the integration of programmable logic with a DSP should be viewed as a hardware/software co-design problem.

We illustrate our proposal using a Viterbi decoder as an example. First we analyze the optimized assembly code for Viterbi decoding on state-of-the-art DSPs and show that it is not the functional units that are the problem but the restrictions on the connection between the register file and the computational units that are the bottleneck which can be elegantly overcome by using a flexible interconnect network that can be realized using field-programmable logic. We call this new hardware block - FPDAU (Field Programmable Data Alignment Unit). This is situated between the register file of a processor and the computational units. This block dynamically re-configures the dataflow between the register file and the functional unit and hence

eliminates a significant fraction of the instructions in the kernels of many important signal-processing algorithms. In order to determine the configuration of the FPDAU the implementation has to be approached as a hardware/software co-design problem. We will show the details of our implementation again using the Viterbi decoder on a TigerSHARC DSP as an example.

The techniques presented are general enough to be used with any other DSP that supports SIMD style processing such as the AltiVec and TI's TMS320c62xx. Also, we show that this approach is not just meant for Viterbi decoding, it can be used with other algorithms as well. In fact, a variety of DSP oriented computations like vector and matrix operations like transposing a matrix, finding the determinant of a matrix can benefit with the proposed architecture.

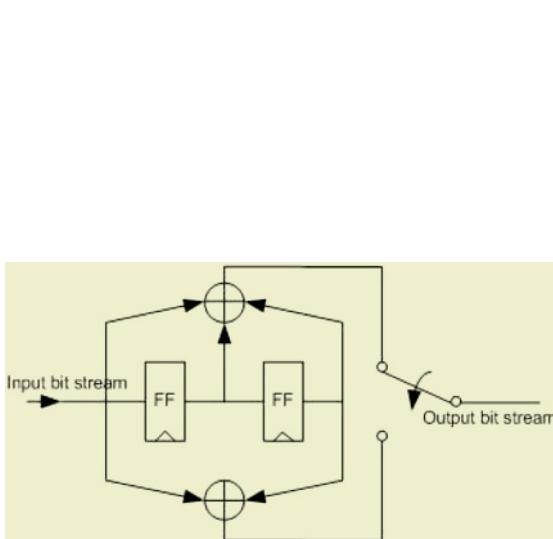
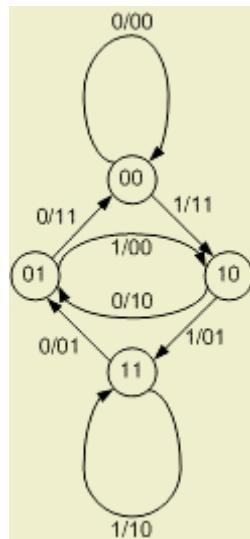
1.1 Organization of This Paper

First we will introduce Viterbi decoding and how it is efficiently implemented in assembly language on the TigerSharc that already has support for ACS computation. Then we will show what the bottleneck of the implementation is and its impact on the execution time and memory for K=5, 7, and 9 Viterbi decoding. We then propose a simple scheduler and programmable interconnect to rectify the problem. The design of the scheduler is described and its cost in terms of equivalent look-up-tables and die size is estimated. We show how the field programmable interconnect is used by the DSP programmer. The improvements in performance are then presented. We then describe other algorithms that can benefit by the proposed solution to demonstrate that this is not just for Viterbi Decoding. Finally, we compare our approach to related solutions in the area of DSP+PL hybrids and show why our approach is more promising.

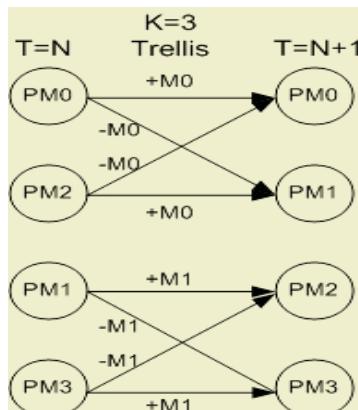
2 Overview of Viterbi Decoding and Its DSP Implementation

Viterbi decoding is a critical application in embedded communication systems like 802.11-based wireless LAN; CDMA based cellular technologies and host of other applications that require data communication over noisy channels. It is part of the EEMBC benchmark suite. In spite of special support to execute Viterbi algorithm efficiently modern DSP are unable to meet the high data rate Viterbi decoding requirements imposed by standards such as the 3G and 802.11(a). So, we use Viterbi algorithm as an example in this paper to illustrate our technique.

First, to understand the computational requirements of a Viterbi decoder, it is useful to start with a convolutional encoder. Fig. 1 shows a $\frac{1}{2}$ rate convolutional encoder for constraint length K=3. In this encoder, for every input bit, two output bits are transmitted. Each input is convolved through XOR operations with the previous two bits. The circuit in Fig. 1 can also be represented as a state-machine shown in Figure 2.

**Fig. 1.** K=3 Convolutional Encoder**Fig. 2.** State Diagram of K=3 Convolutional Encoder

The goal of a Viterbi decoder is to determine what the *most likely* inputs were, given an output data stream corrupted by a noisy transmission channel. A trellis is a map of all of the states from the encoder, drawn out to show each step in time. For the K=3 encoder shown in Figure 2, there would be four states in each time instance of the trellis. Fig. 3 shows a trellis used for Viterbi decoding for the K=3 convolutional encoder. Viterbi decoding consists of two tasks - the population of the trellis and the trace back through the trellis to find the path that yields the most likely sequence of states. Population of the trellis works as follows. For each pair of input bits, the distance between the input bits and the expected output for each transition between states is calculated for each of the possible state transitions. In the

**Fig. 3.** K=3 Trellis Diagram

K=3 state machine shown in Figure 2, there are a total of 8 possible transitions, two to each state. This distance is represented by the +/- M0 and M1 in the trellis diagram in Fig. 3. The smallest distance to each state is chosen and saved for each state. For the next time instance, the same procedure is used except the chosen smallest distance to each state is added to the previous metric saved for that state. These accumulated distances are referred to as *path metrics*. This process of adding the input bits against the local path value, comparing the two local path values to find the smallest distance, and selection of the smallest path distance is often referred to as an Add-Compare-Select, or ACS. Many DSPs have custom ACS instructions to accelerate this process.

The traceback of a Viterbi decoder is simply the selection of the smallest accumulated state metric for each of states of the trellis. This computation is mostly serial, and relatively inexpensive in terms of instructions for Viterbi decoders of constraint lengths of 7 or more as a fraction of the total time.

2.1 AltiVec Implementation

The AltiVec DSP co-processor[9] is a vector processor that operates on 128-bit vectors in 8, 16, or 32 bit SIMD mode. Assuming that path metrics are 32-bit values, we could store the four path-metrics PM0 to PM3 for the the ACS kernel for K=3 trellis (shown in figure 3) in one 128bit vector. Figure 4 shows the pseudo-assembly code for the implementation of the ACS kernel for K=3 where V0, V1, V2, V3, V4 and V5 are 128 bit vector registers. PM(x) denotes a 32-bit value that holds the accumulated path metric of state x. M0 and M1 represent the magnitude of the two different possible distances that may be generated for any input pair of bits. Figure 5 illustrates the flow of data between the registers and the result of the computation. For example, it shows that the least significant 32 bits of register V0 are obtained by adding PM(3) and M1 and so on. Now, in order to compute the new value PM(0) we need to find the minimum of PM(0)+M0 and PM(2)-M0 (please refer to Figure 3), but the vector-min instruction expects the two operands to be in adjacent locations in the vector register. This is an alignment restriction in SIMD processing and is results in simplification of the hardware.

So, the data needs to re-ordered so that the pairs of candidate path metrics are in adjacent sub-word locations in a vector register. This necessitates the need for the two *vec_merge* instructions shown in the pseudo code in Fig. 4.

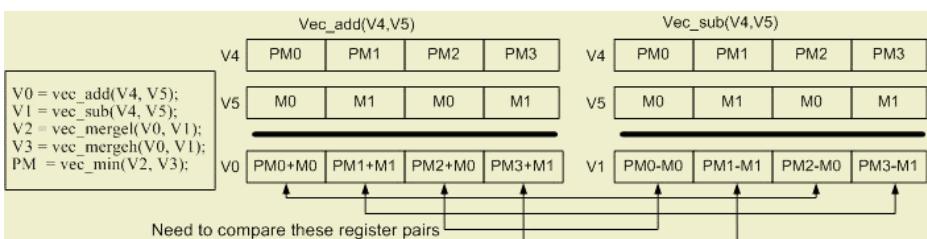


Fig. 4. AltiVec Register Mapping of ACS and Pseudo Code

2.2 TigerSHARC Implementation

Is this restriction just a limitation of the AltiVec processor or is it more general? To investigate this we looked at other DSP architectures (with a completely different architecture style), namely, the TigerSHARC [10] from Analog Devices, which is a statically scheduled superscalar with various SIMD modes of computation and two independent functional units, that operate on 64-bit data. A block diagram of the TigerSHARC computational block is shown in Fig. 7. Each computational block is fed by a 32 entry, 32-bit register file with 4 read ports and 4 write ports. Within each computational block, there are 3 different SIMD modes, allowing for sub-word computations on 32-bit, 16-bit or 8-bit boundaries similar to the Altivec. There are restrictions on which registers may be used in a SIMD instruction. 32-bit SIMD calculations may be completed only on adjacent 32-bit registers. Similar restrictions are placed on 16-bit and 8-bit SIMD computations.

The K=3 trellis (presented in Fig. 3) can be mapped to one of the TigerSHARC computational blocks. Again the pseudo assembly code is shown in Figure 7 and the register dataflow is shown in Figure 8. $PM(x)$ denotes a 32-bit value that holds the accumulated path metric of state x . $M0$ and $M1$ represent the magnitude of the two different possible distances that may be generated for any input pair of bits. $PM3$ and $PM2$ are stored in register pair $R5:4$, $PM1$ and $PM0$ are stored in register pair $R3:2$ and $M0$ and $M1$ are assumed stored in register pair $R1:0$. This grouping of registers allows the TigerSHARC to use its 32-bit SIMD mode and represents an efficient implementation of Viterbi on TigerSHARC.

```

R15:12 = Add/Subtract(R5:4, R1:0);
R11:8  = Add/Subtract(R3:2, R1:0);
R15:12 = Merge(R15:14, R11:10);
R11:8  = Merge(R13:12, R9:8);
R15:12 = VectorMin(R15:14, R11:10);
R11:8  = VectorMin(R13:12, R9:8);

```

Fig. 5. TigerSHARC Viterbi ACS Pseudo-Code

On the right half of Fig. 6, both $M0$ and $M1$ as well as $PM0$ and $PM1$ can be fetched from the register file in a given cycle. Next, using a special instruction that allows addition and subtraction to operate on a pair of registers, the TigerSHARC can then produce half of all of the possible transitions for this stage of the trellis. The result of this computation is shown in the 128-bit result register $R11:8$. Likewise, the left half of Fig. 6 shows a similar computation for the other four possible transitions for the same stage of the trellis. Next we need to find the path with the minimum metric for each of the pairs of transitions to each state in the trellis, which can be done by the special *vector_min* instruction which also supports SIMD mode. However, to utilize the SIMD mode, the vector minimum instruction expects the data to be compared in the same bit locations in both operands. The overlapping arrows in the

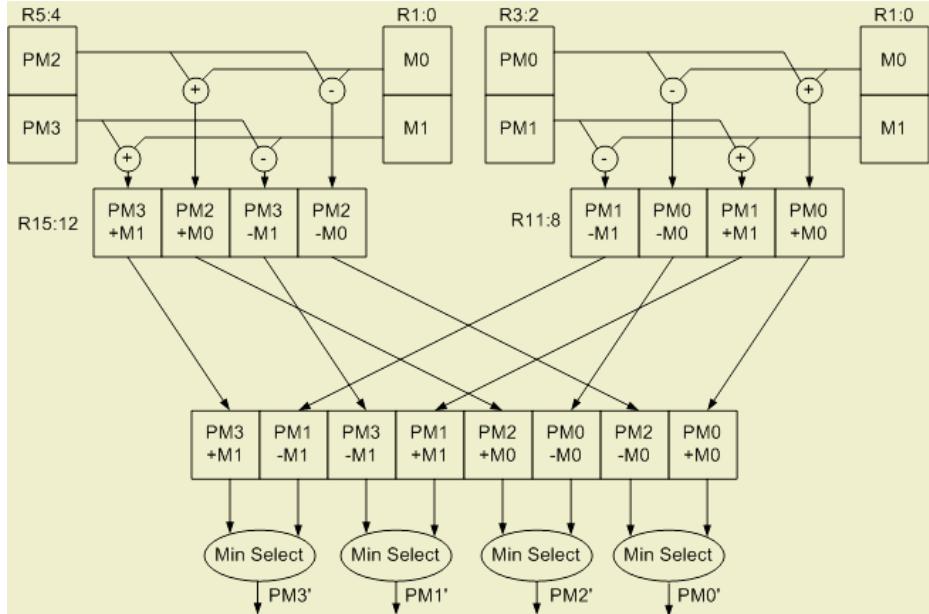


Fig. 6. Mapping of Viterbi ACS Dataflow to the TigerSHARC DSP

middle of Fig. 6 indicate the required data movement in order to utilize the SIMD vector minimum instruction. The overlapping arrows are realized by the *permutation* instructions that are similar in spirit to the *vec_merge* instructions in AltiVec, i.e., they rearrange data in the register file. Finally, we analyzed the Texas Instrument'sC62xx DSP and found that a similar permute instructions are needed to overcome the SIMD restriction [11]. Table 1 shows the performance of the TigerSHARC on various different Viterbi decoders. The %ACS row indicates the fraction of the total cycles the TigerSHARC DSP spends on the trellis population and the %Permutates row indicates the fraction of the total cycles spent on permutations. These cycles are for the entire implementation of the GSM decoder. The fraction of the total cycles spent on ACS and permutations is similar for TI C6x DSP [12]and the AltiVec vector processor. From here on out, we will focus on the TigerSHARC architecture as we had access to the simulation tools for this platform.

Is there a more efficient way to address this problem? To investigate this we decided to profile the TigerSHARC implementation of a Viterbi decoder developed for the GSM wireless handset standard, which requires K=5, 16-bit data and 189 bit data frame.

Table 1. TigerSHARC Viterbi ACS Performance

For ½ Rate Viterbi Decoder, L=190 Bits	K=5	K=7	K=9
ACS Cycles	1960	4191	8459
Traceback Cycles	960	1245	1625
% Execution Cycles in ACS	67.1%	77.1%	83.9%
% of ACS Instructions which are Permutates	23.3%	25.0%	26.9%

3 HW/SW Co-design and the FPDAU

The data in Table 1 shows that a significant fraction of the computation cycles in the Viterbi decoder are spent in permute instruction, which are actually not doing anything useful in terms of the Viterbi algorithm. They are merely there to overcome the data flow restrictions to the function units in a typical DSP. So, the problem is not that the DSP do not have the appropriate instructions or the memory bandwidth (as shown in the previous section, most DSP do have special instructions to support Viterbi), but it is the data alignment restriction.

We propose a Field Programmable Data Alignment Unit (FPDAU) to circumvent the need for these permutation instructions. So, the data rearrangement will be done in hardware instead of software as it is being done now. This gives us two key benefits. It eliminates the instructions from the critical kernel of the computation and thereby provides improvements in performance and memory requirements and possibly reduces power and instruction cache pollution. It also gives us additional flexibility, because with a field-programmable hardware unit we can customize the dataflow to the specific algorithm being implemented.

Next we describe the details of the FPDAU and its integration with the DSP architecture and its programming model. We will illustrate this with the TigerSHARC DSP because we have access to their simulation tools. As noted before, a similar structure would work with other DSP as well; the programming model and the interface will differ.

The FP-DAU consists of two parts - a flexible interconnect that connects the register file to the ALU, Shifter and MAC units and a dynamically programmable state-machine to control the configuration of the flexible interconnect. The controller has configuration register that is mapped into the TigerSHARC's memory space. The placement of the FP-DAU is shown in Fig. 8.

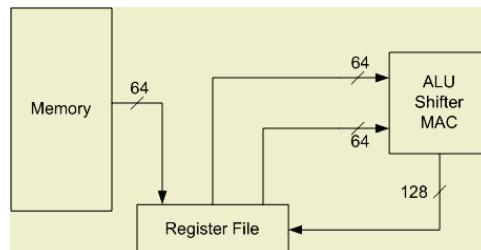


Fig. 7. Block Diagram of a TigerSHARC Computational Block

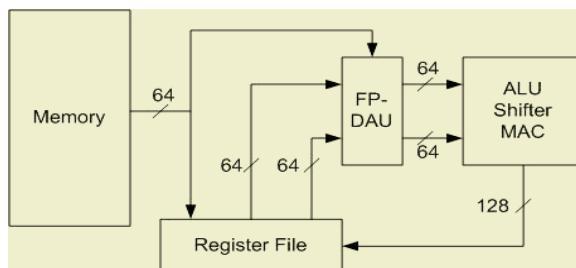


Fig. 8. Block Diagram of a TigerSHARC Computational Block with FPDAU

The detailed block diagram of the FPDAU is shown in Fig. 9. To support the data alignment required for Viterbi (the overlapping arrows in the middle of Fig. 6), we need an interconnect that is flexible only on word i.e. 32-bit boundaries. However, since the TigerSHARC does supports operations on bytes, we will design the FPDAU to support byte-wide granularity. The TigerSHARC register file has two 64-bit read ports, as shown in Fig. 8. The FPDAU needs to select one of 16 bytes from the register file and connect each of those bytes to a byte input of the computational unit. This interconnect can be built with 128 16-to-1 multiplexers. The dynamically programmable state machine inside the FPDAU controls the configurations of the multiplexers. As far as the impact of the FPDAU on the DSP critical path goes, there is a delay of an additional 16:1 multiplexer which does endanger the 300 MHz operating frequency of the TigerSHARC DSP. In the future as we move to finer geometries, we expect this to be less of an issue. We propose an identical FPDAU in both of the independent computational blocks of the TigerSHARC DSP.

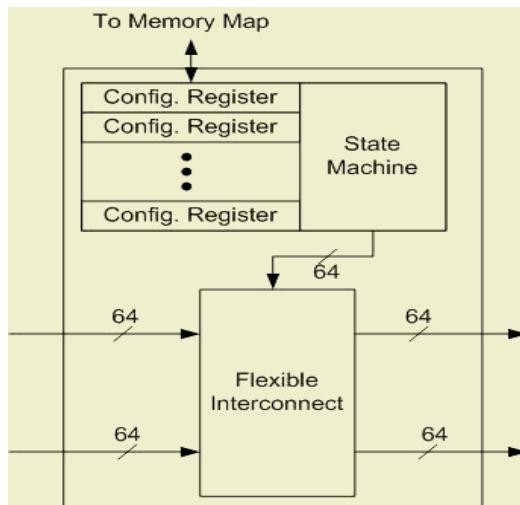


Fig. 9. FP-DAU Block Diagram

Next, the design of the dynamically programmable controller or the state machine shown at the top of Fig. 9 is described. The purpose of the controller is to define the configuration of the flexible interconnect of the FPDAU. This controller will be realized on traditional LUT-based fabric to give it maximum flexibility. This state machine will be clocked by the *read enable* signal of the TigerSHARC register file. In order to minimize the impact of the FPDAU on the instruction set architecture we require that the state machine does not have any additional inputs. Therefore, every time that the *read enable* is clocked and the FPDAU is active, the state machine will proceed to the next state. This has two consequences. First, we need as many states as there are register reads in the inner most loop of the algorithms that utilize the FPDAU. Secondly, it precludes us from using the FP-DAU in inner loops that have non-linear flow, such as branches or jumps.

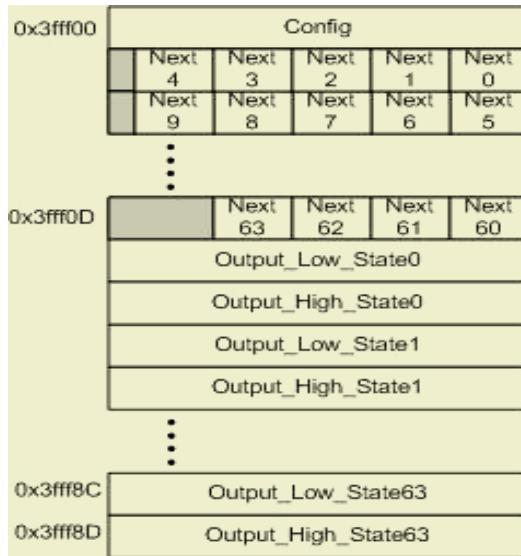


Fig. 10. FP-DAU memory map

However, with predicated execution and the tight loops in DSP kernels this is not much of a restriction. Note that this is a design decision to minimize the impact of the FPDAU on the instruction set architecture. If one has the ability to slightly modify the instruction set architecture (define new opcodes) more efficient and more general-purpose programmable state machines can be realized inside the FPDAU, without the restrictions listed above.

The configuration of the state machine has to be generated during the compilation of the application to the DSP processor. This will allow the data flow between the register file and the ALU to change (in customized way) every clock. The configuration space is memory mapped into the TigerSHARC's internal memory address space, as shown in Fig. 10. This allows the state of the programmable logic to be saved to memory, and also allows new states to be saved and restored by the TigerSHARC. In addition, the FPDAU needs a control register (one bit) that defines whether the FPDAU is active or not.

As noted in the beginning of the paper, the main objective of our work is to minimize the amount of programmable logic to achieve a certain level of performance improvement. That is why we did not advocate a new functional unit or a coprocessor to execution. So, how much area is required for the FPDAU? This requires the estimation of the area for the programmable state machine that is implemented in LUTs. For flexible interconnect structure (that gives us byte-wide data realignment), we need 128 16:1 multiplexers that results in 64 bits for each state of the state machine. Let us assume we have 64 states for the state machine, which should be sufficient to cover a wide range of applications (the inner loops for most applications have fewer than 64 instructions). Each state must have 6 bits to indicate which is the next state. Fig. 10 shows how the state machine of the FP-DAU is memory mapped into the TigerSHARC architecture. Table 2 summarizes the overhead of the FPDAU. The maximum initialization overhead should only be incurred if all 64 states of the

FPDAU's controller are used. The start overhead is the overhead of writing to the configuration register of the FPDAU's controller to start or stop the operation of the FPDAU. The hardware overhead cost is relatively minor when compared to a typical DSP die area of about 1 sq. cm². In the hardware overhead, we assumed that the FPDAU's controller is resident inside 4-LUTs. However, the controller could also use configuration SRAM, which would decrease the number of 4-LUTs needed dramatically. Finally, other functions could be included in the FPDAU, like zero/one insertion, bit reversal or any other simple operation. These added functions could marginally increase the needed hardware for the FPDAU, but would allow us to leverage the strengths of programmable logic on a DSP platform.

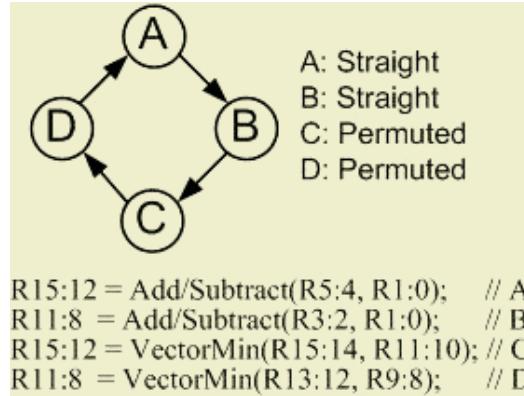
Table 2. FPDAU Overhead

Initialization Overhead	72 cycles, Maximum
Start Overhead	2 cycles per ACS Trellis Frame
Hardware Overhead	128 16:1 Muxes and 200 4-LUTs

3.1 Programming the FPDAU

Next we will describe how the programmer or the compiler uses the FPDAU, using the ACS computation of Viterbi as an example. The configuration for the FPDAU is generated from the register-transfer level assembly code. From the pseudo code shown in Fig. 5 (Viterbi decoder with K=3) we can see that if we omit the permute instructions; we only need four cycles to complete a single stage of the ACS trellis update. Since each of these four instructions accesses the register file, we will need a state-machine with four states to control the flexible interconnect. Note that typically a branch would be executed at the bottom of the loop, but it is omitted from the pseudo code in Fig. 5 for simplicity. Since the branch does not access the register file, it will not clock the state machine so we can ignore it from the perspective of configuring the FPDAU. Figure 11 shows the resultant state machine for Viterbi ACS derived from Figure 7. The register reads for the two add/subtract instructions are done in normal i.e. without any permutation. They are indicated by state A and state B in Figure 11. The two *vector_min* instructions are executed in states C and D of the state machine which requires the FPDAU to program the flexible interconnect to permute the data corresponding to the pattern shown at the bottom of Fig. 6. The new pseudo code required to complete a single stage of the ACS trellis update is also shown in Figure 11, as expected it eliminates the two permute instructions.

Finally, it is important to note that the FP-DAU should be disabled and the configuration of the FP-DAU is saved and restored upon entering interrupt routines. If the FP-DAU is to be utilized inside an interrupt service routine, the states of the FP-DAU must be saved and restored to the TigerSHARC's on-chip memory upon entering and exiting the interrupt, respectively. In most cases the entire configuration memory is not utilized, so the overhead of saving the FPDAU is typically a few cycles, especially given that the TigerSHARC has the ability to read/write 128-bits to memory in a given cycle. However, if the entire configuration space of the FP-DAU does indeed have to be saved, the maximum penalty is around 72 cycles to save the entire FP-DAU state.

**Fig. 11.** Example FP-DAU Configuration for Viterbi ACS

4 Results from Viterbi Implementation

In this section we will summarize the results of the implementation of the Viterbi decoder on the TigerSHARC enhanced with the FPDAU. As noted before, the programmable logic is configured at compile time i.e. statically by analyzing the kernel of the computation, which in the case of this decoder has 20 instructions. Using the simple compilation scheme described above would translate into 20 states for the FPDAU programmable state machine. The one time overhead of writing to the FP-DAU configuration register and programming the states in the FP-DAU is 16 cycles. Two additional cycles are needed to turn on/off the FP-DAU when entering/exiting the ACS inner loop. Table 3 shows the performance improvements of the TigerSHARC DSP with the FP-DAU on Viterbi decoders of different lengths. Note that the improvement in terms of cycles saved is quite impressive (15 % to 23%) given that the TigerSHARC is already optimized to implement Viterbi efficiently. Also, note improvement also results in improvements to code density, which is quite useful in embedded applications. It may also result in power savings but the FPDAU itself will consume some power but we do not have access to the gate-level netlists of the TigerSHARC to evaluate exactly what the savings would be.

The additional area required for 64 16-to-1 Muxes is Y, incurring a total delay of Z in W process technology. The state machine in the FP-DAU requires the equivalent of X number of CLBs, at an area estimate of A um² in W process technology.

Table 3. TigerSHARC Viterbi Performance with FP-DAU

L=190 Bits	K=5	K=7	K=9
ACS	1506	3146	6183
Traceback	960	1245	1625
Total Cycles	2466	4391	7808
% Speed Up	15.5%	19.2%	22.6%

5 Other Applications of FPDAU

Even though the focus of this paper was the implementation of a Viterbi decoder, it should be pointed out that the FPDAU concept is quite general and it has many applications. Basically, the FPDAU restores some flexibility of a Vector, VLIW, or SIMD mode processor by allowing the functional units to operate on any data in the register file. Without the FPDAU one has to waste valuable CPU cycles and power in rearranging the data so that a given instruction can execute properly. We have found applications for FPDAU in a variety of DSP applications especially those that involve matrix operations like Reed-Solomon decoding, finding the minimum or maximum in a vector, data interleaving and de-interleaving and matrix transpose. In each of these applications the FPDAU can be used, but exactly how it is used is determined by the hardware/software co-design of the application, as illustrated in this example. The interface and the programming model of the FPDAU will be the same but the configuration of the state machine will be different in each case and depending on the application the amount of improvement will also vary. For example, in an experiment with matrix transpose on the AltiVec we found that only half the merge instructions in the inner loop can be eliminated with the FPDAU. So, it is important to note that *not all permute (or data rearrangement) operations* can be eliminated with the FPDAU; this is the trade-off between the amount of configurable logic inside the FPDAU and its interface and the amount of flexibility. We deliberately restrict the inputs to the FPDAU to two and byte-level reconfigurability to minimize the area overhead of the FPDAU and its impact on the critical path of the processor.

6 Related Work and Conclusions

The idea of utilizing field programmable logic to accelerate computations is not new. Starting with the PRISC project in Harvard [13] and the work in BYU[3] on integration of DSP and reconfigurable logic and more recently the reconfigurable functional unit idea in the Chimera project in Northwestern University[2], there have been numerous efforts at integrating programmable logic with a processor. The key difference between those efforts and the proposed solution is in two areas (a) we treat DSP + programmable logic integration as a hardware/software co-design problem, hence what we propose is a methodology rather than a specific solution. So, it can be applied to any processor and any application (b) unlike the previous efforts we focus on the cost issue, which precludes us from using a co-processor or a new functional unit because that would add to the cost and change the instruction set architecture of the underlying processor – which poses problems in terms of adoption in embedded processors especially in the commercial arena. We believe that the solution proposed here finds a sweet spot in terms of return on investment in terms of the amount of programmable logic and the improvement in performance achieved. Also, it has minimal impact on the instruction set architecture and the programming model of a DSP, so it can be ignored without significant penalty if the application domain does not require it.

Also, if one has more chip area to spend on the programmable logic the FPDAU can be expanded to include other operations in the LUT area that is currently being

used to only implement the programmable state machine. For example, one could have a bit-level operations support that could help in encryption algorithms like DES and AES. So, again the proposal here is a co-design methodology for the DSP + programmable logic platform, where the architect can choose how much chip area to spend on programmable logic and what operations to implement there with the FPDAU providing a general framework for programming and interface. If it is expanded further it will resemble the RFU idea in Chimera or the co-processor concept in the BYU project or Riverside project[1].

References

- 1 Stitt, G. and Vahid, F.: Energy Advantages of Microprocessor Platforms with On-chip Configurable Logic. *IEEE Design and Test*, 2002
- 2 Ye, Z., Moshovos, A., Hauck, S., Banerjee, P.: CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit, *Computer Architecture News*, (2000).
- 3 Graham, P., Nelson, B.: Reconfigurable Processors for High-Performance, Embedded Digital Signal Processing, *Field Programmable Logic and Applications*. (1999)
- 4 Fisher, J., Faraboschi, P., Desoli, G.: Custom-Fit Processors: Letting Applications Define Architectures. *Hewlett-Packard Laboratories Cambridge*, Cambridge, MA, (1996)
- 5 Compton, K., Hauck, S.: Reconfigurable Computing: A Survey of Systems and Software, <http://www.ee.washington.edu/faculty/hauck/publications/ConfigCompute.pdf>
- 6 Dehon, A.: The Density Advantage of Configurable Computing. *IEEE Computer Magazine*, (2000)
- 7 Tessier, R., Burleson, R.: Reconfigurable Computing for Digital Signal Processing: A Survey. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, (2001)
- 8 Hartenstein, R.: Reconfigurable Computing: A New Business Model – and its Impact on SoC Design. *Proceedings Euromicro Symposium on Digital Systems Design*. IEEE Comput. Soc. (2001)
- 9 Ollmann, I.: Altivec. http://www SIMDtech.org/apps/group_public/documents.php
- 10 Analog Devices: TigerSHARC DSP Hardware Specification. http://www.analog.com/Analog_Root/static/library/dspManuals/Tigershare_hardware.html
- 11 Fridman, J.: Data Alignment for Sub-Word Parallelism in DSP. *IEEE Signal Processing Magazine*, IEEE, (2000). p.27-35.
- 12 Texas Instruments: TMS320C6000 CPU and Instruction Set Reference Guide. (2000), <http://www-s.ti.com/sc/psheets/spru189f/spru189f.pdf>
- 13 Razdan, R., Smith, M.: High-Performance Microarchitectures with Hardware-Programmable Functional Units, Proc. 27th Annual IEEE/ACM Intl. Symp. on Microarchitecture, pp. 172-180, November (1994)

Fully Parameterized Discrete Wavelet Packet Transform Architecture Oriented to FPGA

Guillermo Payá, Marcos M. Peiró, Francisco Ballester, and Francisco Mora

Universidad Politécnica de Valencia. Department of Electronic Engineering.

Camino de Vera s/n, 46022, SPAIN.

guipava@doctor.upv.es, {mpeiro, fballest, fmora}@eln.upv.es

Abstract. The present paper describes a fully parameterized Discrete Wavelet Packet Transform (DWPT) architecture based on a folded Distributed Arithmetic implementation, which makes possible to design any kind of wavelet bases. The proposed parameterized architecture allows different CDF wavelet coefficient with variable bit precision (data input and output size, and coefficient length). Moreover, by combining different blocks in cascade, we can expand as many complete stages (wavelet packet levels) as we require. Our architecture need only two FIR filters to calculate various wavelet stages simultaneously, and specific VIRTEX family resources (SRL16E) have been instantiated to reduce area and increase frequency operation. Finally, a DWPT implementation for CDF(9,7) wavelet coefficients is synthesized on VIRTEX-II 3000-6 FPGA for different precisions.

1 Introduction

For years, the Discrete Wavelet Transform (DWT) has been used in a wide range of applications, including signal analysis and coding, data compression, image and video compression, numerical analysis, statistics, physics... Recently, new studies propose opening the high-pass branches in the DWT (see Fig. 1). This fact trades new data compression capability when the information is distributed in low and higher frequency ranges. New designs with this structure have been successfully applied in 1D data processing, such as in medical audio processing [1], in ECG compression [2], and in digital modulations systems like CDMA or OFDM [3]. Two-dimensional applications of these structures are employed when images have strong high-pass components, e.g., the fingerprint images and high-contrast medical images. This wavelet structure is called Wavelet Packet Transform (WPT) due to the frequency ‘packets’ obtained in the output of its binary tree. In [4], Coifman introduced the adaptive tree structure concept using packets as an evolution of wavelet for signal and image compression. He proposed to expand the wavelet tree selecting the best wavelet bases.

This work describes a new methodology to implement different DWPT structures, depending on bit precision (data input, coefficients and data output) and wavelet bases selected. Distributed Arithmetic (DA) technique has been applied to implement

easily-parameterized structures and gives optimum results on FPGA devices. This work is organized as follows: next section introduces the DA by using a finite-impulse response (FIR) filter implementation. In section three, we explain the DWPT architecture based on DA and Polyphase Decomposition. Fourth and fifth sections present the modular DWPT architecture and results. Finally, the conclusions are exposed.

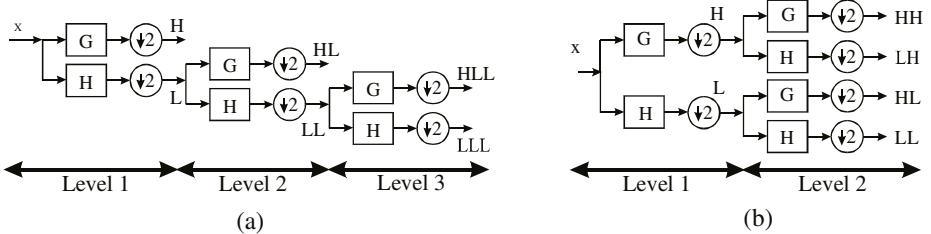


Fig. 1. (a) Three stages Discrete Wavelet analysis filter bank and (b) Two stages Packet Wavelet analysis filter bank

2 Distributed Arithmetic Technique on FPGA Device

The Distributed Arithmetic technique is an efficient procedure for computing sum-of-products (inner products) between a fixed and a variable data vector. The basic principle is owed to Croisier et al. [5], but Peled and Liu [6] have independently presented a similar method. This arithmetic trades memory for combinatory elements, resulting ideal to implement custom digital signal processors in look-up table (LUT-based) FPGA [7]. In addition to a DA implementation, the designer also can select from a bit-serial to a full-parallel implementation [8].

DWT and its packet version are based on a cascade of FIR filters. The operation of these filters involves inner products of the equation 1 type. The inner product can be rewritten as equation 2 with two's complement representation for coefficient (α_i) and data input (x_i). The data input are scaled so that $|x_i| \leq 1$.

$$y = \sum_{i=1}^N \alpha_i \cdot x_i \quad (1)$$

$$y = \sum_{i=1}^N \alpha_i \cdot \left[-x_{i0} + \sum_{k=1}^{Wd-1} x_{ik} \cdot 2^k \right] \quad (2)$$

The symbols x_{ik} represent the k th bit in x_i data input, and Wd is the number of bits of the data. By modifying the order of the summations we get

$$y = -\left[\sum_{i=1}^N \alpha_i \cdot x_{i0} \right] + \sum_{k=1}^{Wd-1} \left[\sum_{i=1}^N \alpha_i \cdot x_{ik} \right] \cdot 2^{-k} \quad (3)$$

The elements in brackets take only a finite number of values, 2^N , so we compute and store these values in a look-up table (LUT).

Fig. 2 shows a block diagram for computing an inner product according to previous equation. Since the output is divided by 2, by the inherent shift, the circuit is called *shift-accumulator*. Bits x_{ik} are the address of the LUT which store the binary coefficient additions. Data inputs x_i are shifted one-bit at a time (1BAAT) generating a bit-serial DA structure. Of course, we can implement a parallel form of DA by allocating a LUT to each term in brackets in equation 3. Our work computes DA in bit-serial fashion using the minimum resources into the FPGA. On the other hand, we save half of the area resources in DA implementations, when a FIR filter has symmetric coefficients.

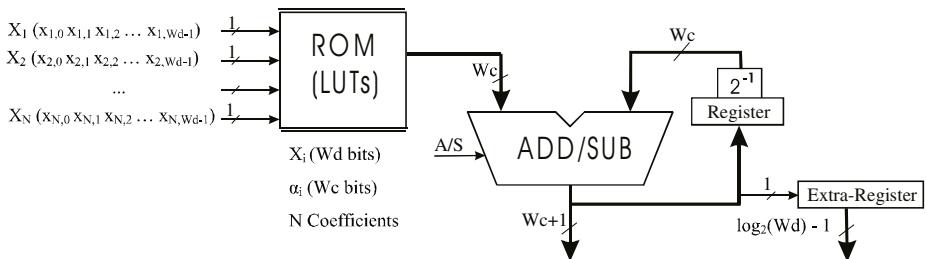


Fig. 2. Bit serial Distributed Arithmetic basic cell

Moreover, using shift registers LUT mode (SRL16E and SRLC16E primitives, see Fig. 3.) we can reduce a 50% on area when original design has a higher number of flip-flops. Virtex family can configure any LUT as a 16-bit shift register without using the flip-flops available in each slice. Shift-in operations are synchronous with the clock, and output length is dynamically selectable. A dedicated output allows the cascading of any number of 16-bit shift registers to create whatever size shift register is needed. Nevertheless, the configurable 16-bit shift register cannot be set or reset.

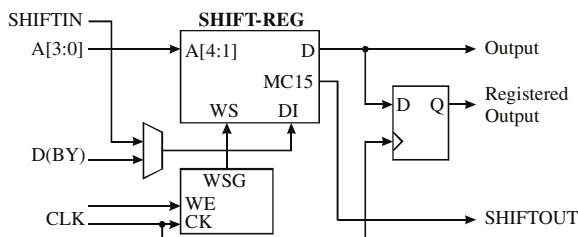


Fig. 3. Shift Register mode LUT Configuration

3 Wavelet Packet and Polyphase Decomposition

The theory of wavelet signal decomposition was firstly introduced by S.G. Mallat [9]. In his work, he computes the wavelet representation with a Pyramidal Algorithm (PA) based on convolutions with Quadrature Mirror Filters (QMF) filters (the basic wavelet cell showed in Fig. 4a) and decimators. In this figure, G represents the high-pass filter that obtains the signal details whereas H obtains the coarser resolution (low-pass component) of the input signal. In addition, the decimator can be translated to the input of the filters obtaining the commutator model [10] for the QMF cell (Fig. 4b).

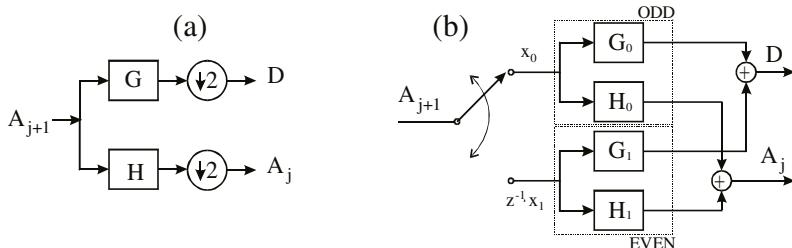


Fig. 4. (a) The QMF of the wavelet decomposition and (b) the commutator model of the Polyphase QMF bank

By repeating in cascade this algorithm the wavelet representation of a signal A on J resolution levels (or stages) is computed. Since the filter outputs are decimated in the basic DWT elementary cell, we apply the polyphase decomposition of the filter banks. For example, the algorithm expresses the symmetric 9-tap filter $H(z)$,

$$H(z) = \alpha_0 \cdot z^{-4} + \alpha_1 \cdot (z^{-3} + z^{-5}) + \alpha_2 \cdot (z^{-2} + z^{-6}) + \alpha_3 \cdot (z^{-1} + z^{-7}) + \alpha_4 \cdot (1 + z^{-8}) \quad (4)$$

using the biphasic decomposition in the form

$$H(z) = H_0(z^2) + z^{-1} \cdot H_1(z^2) \quad (5)$$

where

$$\begin{aligned} H_0(z) &= \alpha_0 \cdot z^{-2} + \alpha_2 \cdot (z^{-1} + z^{-3}) + \alpha_4 \cdot (1 + z^{-4}) \\ H_1(z) &= \alpha_1 \cdot (z^{-1} + z^{-2}) + \alpha_3 \cdot (1 + z^{-3}) \end{aligned} \quad (6)$$

To compute the DWT, M.Vishwanath [11] proposed an alternative to the PA algorithm called Recursive Pyramidal Algorithm (RPA). Basically, RPA consists of rearranging the order of the outputs such that an output is scheduled at the earliest instance that it can be scheduled. To compute the 2-level DWPT, we propose an RPA-modified algorithm. It consists of rearranging the low-pass and high-pass outputs in order to open 2-levels DWPT (see Fig. 5).

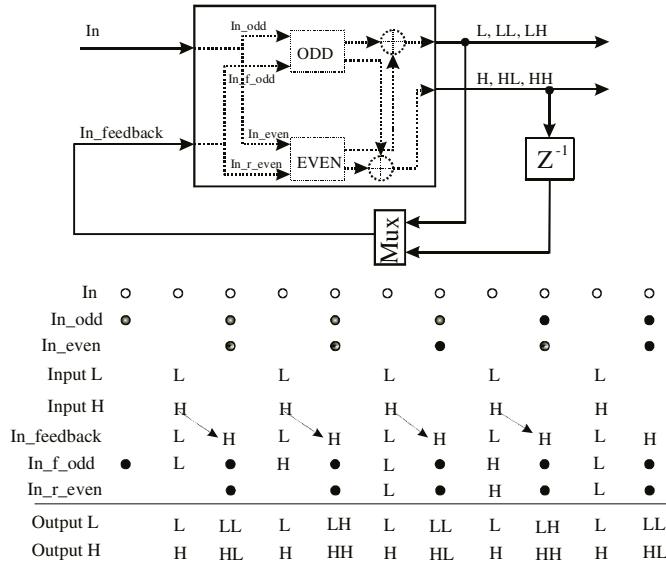


Fig. 5. 2-levels DWPT structure based on the Recursive Pyramid Algorithm

The sampling grid for the DWPT obtains the output schedule by push down all the horizontal lines of samples until they form a single line. The order of the outputs obtained gives us the output schedule.

Taking profit of bit rate, we can expand as many complete levels as we need, by replicating blocks in cascade (see Fig. 6). Using the free time slots, we can expand two additional stages by means of 4-Stages block (4-S). This 4-Stages block has similar structure than 2-Stages block (2-S) described before. The differences are located on the number and length of the registers, as we will see in next section.

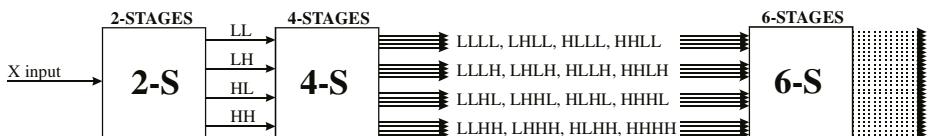


Fig. 6. J-Stages DWPT structure.

4 FPGA-Based Fully Parameterized DWPT Architecture

Our proposed architecture is fully parameterized and modular. We can select the different modules to implement different discrete wavelet packets. In Fig. 7, we present a DWPT structure for a CDF(9,7) wavelet bases. CDF is chosen because it has been applied in most of the previously mentioned applications. These fixed coefficients can be referenced as CDF(G,H), representing the number of taps in the high-pass and the low-pass filter coefficients.

In Fig. 7 the parallel input sequence for even (x_0) and odd (x_1) paths are serialized by the Parallel to Serial converter (P/S) and then introduced to the z^{-l} filter registers of Wd bits. The adders perform both the symmetry coefficients in each filter and the shared additions between two sub-filters. Next, the LUT and the scaling-accumulator are performed for each sub-filter.

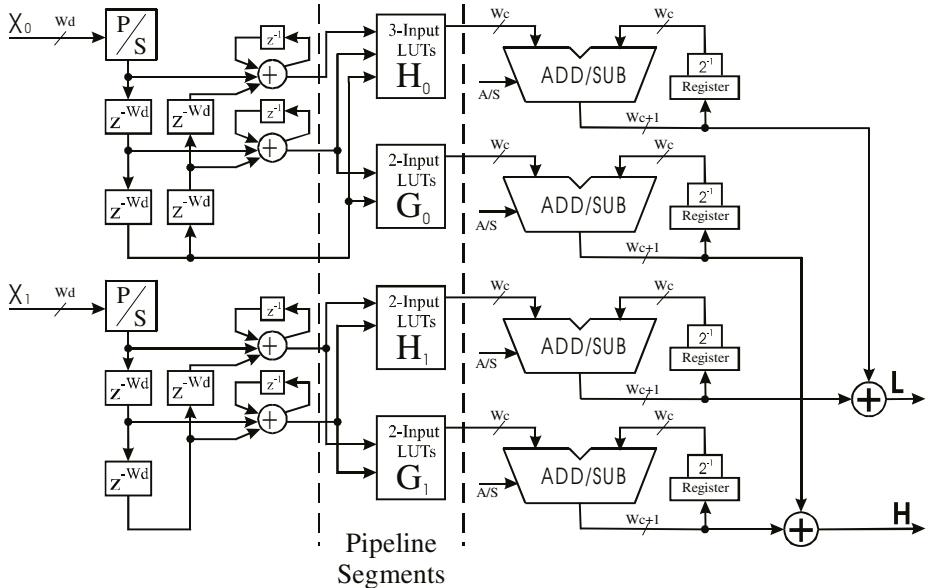


Fig. 7. DWPT structure for a CDF(9,7) wavelet bases

Our basic cell can be RPA characterized by increasing the registers in both the input feedback data from the previous stage and the registers in the scaling-accumulator of the DA structure (see Fig. 5 scheduling). The idea is that filters work at double frequency than the odd/even input data-rate by using free time slots. These registers allow us to accept the input at a uniform rate taking profit of decimation by two. Applying the RPA principles, the output to next level is connected to feedback input (x_i). The scheduling generated from these additional registers is described in next figures.

4.1 Parallel to Serial Converters

The Parallel to Serial Converters have a special functionality as Fig. 8 shows. In 2-Stages block, it converts parallel data (X_i inputs with $T \cdot Wd$ period, really $2 \cdot T \cdot Wd$ because we select the odd or even inputs) and L_i and H_i with $2 \cdot T \cdot Wd$ period ($4 \cdot T \cdot Wd$ odd/even period) to bit serial data with T period. Fig. 8 shows a detailed version of the P/S converter with its synthesis results.

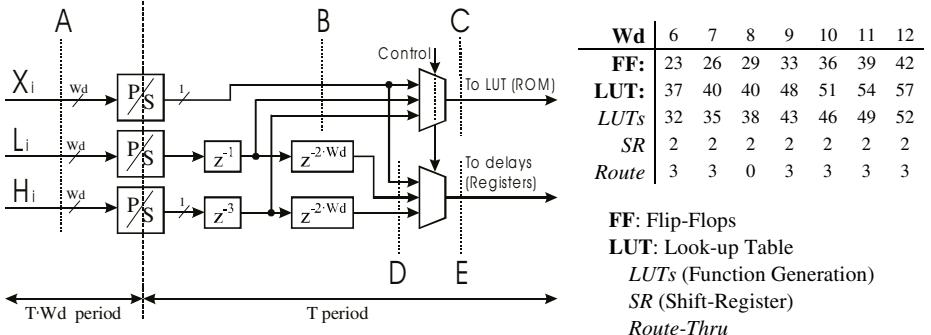


Fig. 8. Parallel to Serial Converter with scheduling registers for 2-Stages block. Notation (for $Wd = 4$ bits): $X_i = (X_{3,i} X_{2,i} X_{1,i} X_{0,i})$

The z^{-1} and z^{-3} are included to fix the low-pass and high-pass feedback inputs (L_i and H_i , respectively) between x_i bits of the data. The vertical dotted lines represent the scheduling points (A, B, C, D, E) explained in Fig. 9. The cutset D represents the z^{-2wd} delays needed to synchronize new data inputs x_i with the inputs from the previous stages.

Time	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cutset B	$X_{3,j+2}$		$X_{2,j+2}$		$X_{1,j+2}$		$X_{0,j+2}$		$X_{3,i}$		$X_{2,i}$		$X_{1,i}$		$X_{0,i}$	
	$H_{3,i}$		$L_{3,i}$		$H_{2,i}$		$L_{2,i}$		$H_{1,i}$		$L_{1,i}$		$H_{0,i}$		$L_{0,i}$	
Cutset C	$H_{3,i}$	$X_{3,j+2}$	$L_{3,i}$	$X_{2,j+2}$	$H_{2,i}$	$X_{1,j+2}$	$L_{2,i}$	$X_{0,j+2}$	$H_{1,i}$	$X_{3,i}$	$L_{1,i}$	$X_{2,i}$	$H_{0,i}$	$X_{1,i}$	$L_{0,i}$	$X_{0,i}$
Cutset D		$X_{3,j+2}$		$X_{2,j+2}$		$X_{1,j+2}$		$X_{0,j+2}$		$X_{3,i}$		$X_{2,i}$		$X_{1,i}$		$X_{0,i}$
		$H_{1,i}$		$L_{1,i}$		$H_{0,i}$										
Cutset E	$H_{1,i}$	$X_{3,j+2}$	$L_{1,i}$	$X_{2,j+2}$	$H_{0,i}$	$X_{1,j+2}$	$L_{0,i}$	$X_{0,j+2}$		$X_{3,i}$		$X_{2,i}$		$X_{1,i}$		$X_{0,i}$

Fig. 9. Scheduling of P/S Converter for $Wd = 4$ bits. Notation: $X_i = (X_{3,i} X_{2,i} X_{1,i} X_{0,i})$, $L_i = (L_{3,i} L_{2,i} L_{1,i} L_{0,i})$ and $H_i = (H_{3,i} H_{2,i} H_{1,i} H_{0,i})$

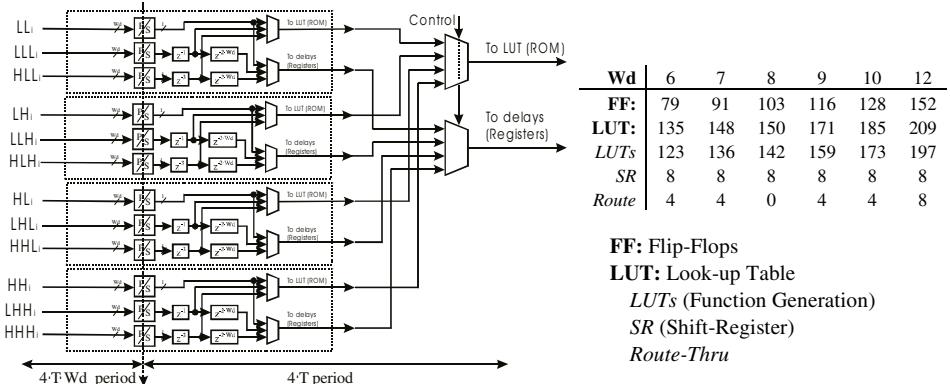


Fig. 10. Parallel to Serial Converter for 4-Stages block

The P/S Converters for the 4-Stages block have 4 inputs and 8 feedbacks inputs. We have to replicate four times the Fig. 8 structure. The input rates are divided by 4 because the decimators. Fig. 10 represents the structure with the synthesis results.

4.2 Delay Blocks

The delay blocks are detailed in Fig. 11a, and Fig. 12 describes its scheduling. Paying attention on the synchronization between F outputs and the previous C outputs, the delay blocks for the 4-Stages block only differ in the register length (see Fig. 11b).

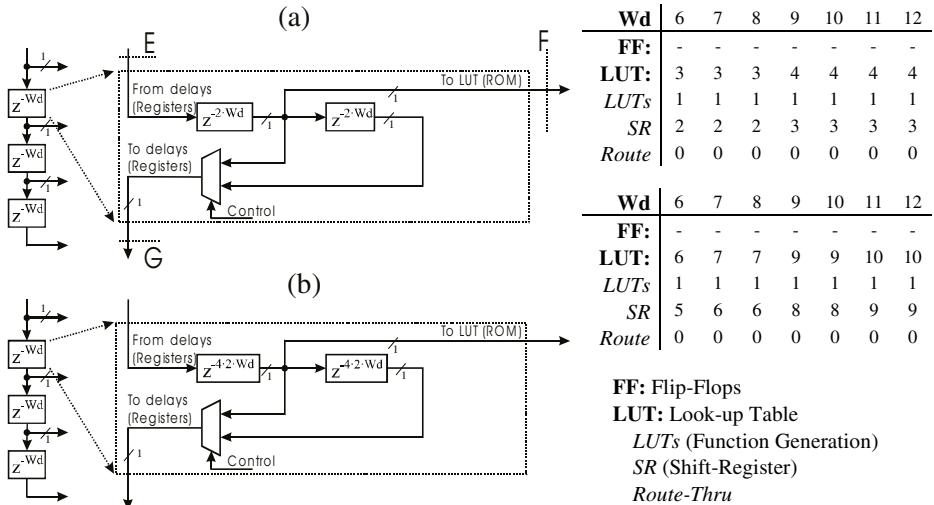


Fig. 11. (a) Delay blocks for the 2-Stages block and (b) Delay blocks for the 4-Stages block

Time	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Cutset E	$H_{1,i}$	$X_{3,i+2}$	$L_{1,i}$	$X_{2,i+2}$	$H_{0,i}$	$X_{1,i+2}$	$L_{0,i}$	$X_{0,i+2}$		$X_{3,i}$				$X_{1,i}$	$X_{0,i}$	
Time	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8
Cutset F	$H_{1,i}$	$X_{3,i+2}$	$L_{1,i}$	$X_{2,i+2}$	$H_{0,i}$	$X_{1,i+2}$	$L_{0,i}$	$X_{0,i+2}$		$X_{3,i}$		$X_{2,i}$	$X_{1,i}$	$X_{0,i}$		
Time	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Cutset G	$H_{1,i}$	$X_{3,i+4}$	$L_{1,i}$	$X_{2,i+4}$	$H_{0,i}$	$X_{1,i+4}$	$L_{0,i}$	$X_{0,i+4}$		$X_{3,i+2}$		$X_{2,i+2}$	$X_{1,i+2}$	$X_{0,i+2}$		

Fig. 12. Delay blocks scheduling for Wd = 4 bits

4.3 Symmetrical Adders

The use of different data flows (X , L and H) implies the design of three registers (one per flow) in the symmetrical adders (see Fig. 13). We have pipelined the adder's output to increase the frequency operation.

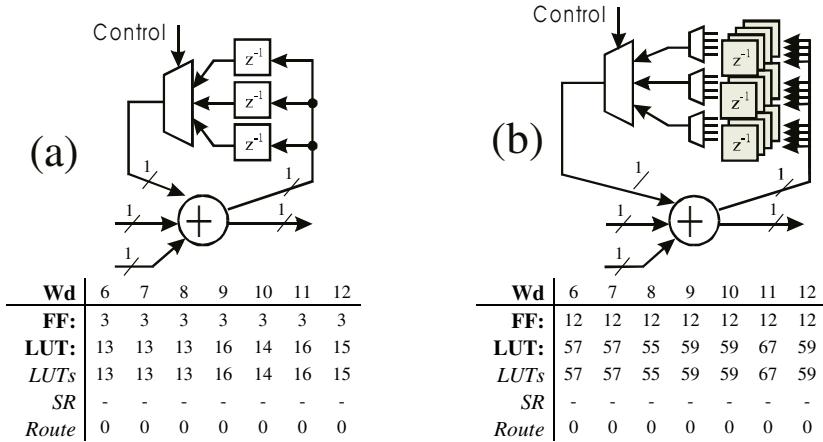


Fig. 13. (a) Symmetrical-adders structure for the 2-Stages block. (b) Symmetrical-adders structure for the 4-Stages block.

4.4 Memory ROMs (LUTs)

As usual, in DA technique, the symmetry of G_o , G_i , H_o and H_i halves the LUT size in their physical implementation. LUT-area depends on coefficient length (W_c), not on number of stages. Only four memories of W_c LUT are needed in the overall design.

4.5 Adder/Subtract Accumulators

Finally, we have to use the accumulator registers to keep the inner products results of the different subbands. The structure (see Fig. 14) is similar then the Fig. 13.

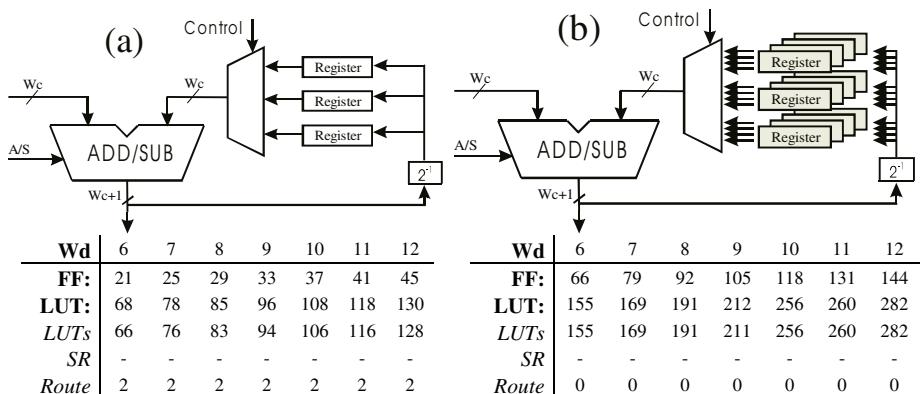


Fig. 14. (a) Adder/Subtract Accumulator registers structure for 2-Stages blocks. (b) Adder/Subtract Accumulator registers structure for 4-Stages blocks

5 Implementation Results and Conclusions

We have synthesized and implemented a CDF(9,7) DWPT into a XILINX Virtex-II 3000-6 FPGA device. For 2-Stages DWPT and $W_d=W_c=8$, the clock frequency reaches 130 MHz with a hardware cost of 255 Flip-flops and 495 LUTs.

In Fig. 15, we have estimated the occupation area in Slice FF and LUTs for different CDF DWPT structure (CDF(2,2) with 5 and 3 coefficients and CDF(9,7) with 9 and 7 coefficients). The maximum clock frequency results always over 100 MHz.

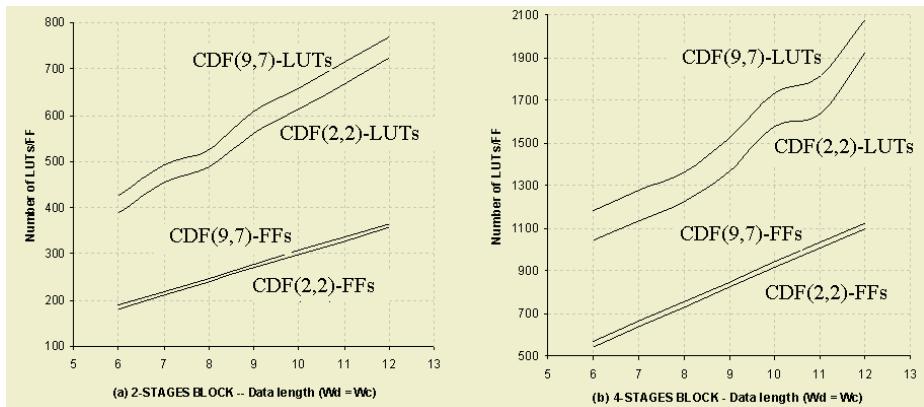


Fig. 15. Data length vs. Number of LUTs and Flip-flops in (a) 2-Stages blocks and (b) 4-Stages blocks

References

1. Trenas, M..A, Lopez, J., and Hongyi, C.: A Configurable Architecture for the Wavelet Packet Transform. IEE Electron Letters (1999) 499-500
2. Hilton, M.L.: Wavelet and Wavelet Packet Compression of Electrocardiograms. Technical Report TR9505, Department of Computer Science, University of South Carolina (1997)
3. Jamin, A., Mähönen, P.: FPGA Implementation of the Wavelet Packet Transform for High Speed Communications. Conference on Field-Programmable Logic and Applications (2002)
4. Coifman, R., Meyer, Y., Quake, S., Wickerhauser, V.: Signal Processing an Compression with Wave Packets. Numerical Algorithms Research Group, Yale University (1990).
5. Croiser, A., Esteban, D.J., Levilion, M.E., Rizo, V.: Digital Filter for PCM Encoded Signals. U.S. Patent 3 777 130 (1973).
6. Peled, A., Liu, B.: A New Approach to the Realization of NonRecursive Digital Filter. IEEE Trans. On Audio and Electroacoustic, vol. 21, no. 6 (1973) 477-485.
7. LB_2DFDWT – Line-Based Programmable Forward DWT, AllianceCore™ (2001)
8. White, S.A.: Applications of Distributed Arithmetic to Digital Signal Processing: a Tutorial Review. ASSP Magazine, vol. 6, Issue. 3 (1989) 4-9
9. Mallat, S.: Multifrequency Channel Decompositions. IEEE Trans. On Acoustics, Speech and Signal Processing, Vol.37, no. 12 (1989)
10. Vaidyanathan, P.P.: Multirate Systems and Filters Banks, Prencitce-Hall Inc (1993)
11. Vishwanath, M.: The Recursive Pyramid Algorithm for the Discrete Wavelet Transform. IEEE Trans. On Signal Processing, vol. 42, no. 3 (1994) 673-677

An FPGA System for the High Speed Extraction, Normalization and Classification of Moment Descriptors

Stavros Paschalakis¹, Peter Lee², and Miroslaw Bober¹

¹ Mitsubishi Electric ITE-VIL, The Surrey Research Park
20 Frederick Sanger Road, Guildford, Surrey GU2 7YD, UK
{Stavros.Paschalakis, Miroslaw.Bober}@vil.ite.mee.com

² Department of Electronics, University of Kent at Canterbury
Canterbury, Kent CT2 7NT, UK
P.Lee@kent.ac.uk

Abstract. We propose a new FPGA system for the high speed extraction, normalization and classification of moment descriptors. Moments are extensively used in computer vision, most recently in the MPEG-7 standard for the region shape descriptor. The computational complexity of such methods has been partially addressed by the proposal of custom hardware architectures for the fast computation of moments. However, a complete system for the extraction, normalization and classification of moment descriptors has not yet been suggested. Our system is a hybrid, relying partly on a very fast parallel processing structure and partly on a custom built, low cost, reprogrammable processing unit. Within the latter, we also propose FPGA circuits for low cost double precision floating-point arithmetic. Our system achieves the extraction and classification of invariant descriptors for hundreds or even thousands of intensity or color images per second and is ideal for high speed and/or volume applications.

1 Introduction

The theory of moments is among the most commonly used methodological frameworks in computer vision applications such as document analysis and OCR [1], object recognition [2] and, recently, for the specification of the MPEG-7 region shape descriptor [3]. Moments are projections of the image function onto a basis function and different basis functions have given rise to different types of moments, such as geometric, Zernike and Legendre [4], each type with distinct characteristics with regards to its information content, ease of normalization to image transformations, etc. Geometric moments are among the most commonly used, mainly due to their ease of calculation in relation to other types of moments. Nevertheless, even geometric moments are computationally demanding in terms of their extraction, in spite of the increasing performance of computer systems. This is especially troublesome for high speed and/or volume systems. A lot of effort has been devoted to the development of algorithmic modifications and custom hardware structures to alleviate this computational complexity problem. Thus, there are techniques which allow the fast calculation of moments based solely on simple integer arithmetic operations. Nevertheless, in order for moment descriptors to be used in image processing applications, e.g. object

recognition and image retrieval, they usually require a set of normalization procedures and a classification framework. Such processes are usually assigned to “host” microprocessors, such as CPUs, due to their extended arithmetic processing capabilities and their reprogrammability, which facilitates algorithmic enhancements over time. This partly negates the benefit of the custom hardware implementation, e.g. for applications where a CPU does not actually exist, such as intelligent sensors.

In this paper we propose a new FPGA architecture for the high speed extraction, normalization and classification of moment descriptors. We have chosen an FPGA as our implementation vehicle because it combines the reprogrammability advantage of general purpose processors with the parallel processing and speed advantages of custom hardware. The proposed system has a hybrid form, comprising a parallel processing structure working alongside a low cost, custom built, reprogrammable processing unit. The latter can be reprogrammed for different normalization and classification functions, or even different image processing problems. In the context of this general processing unit we also propose low cost FPGA circuits for 64-bit double precision floating-point arithmetic operations, i.e. addition/subtraction, multiplication, division and square root. Such circuits have been investigated by researchers but only for 32-bit single precision or, more commonly, lower precision custom formats. Our system achieves the extraction and classification of invariant moment descriptors for hundreds or even thousands of intensity or color images per second, making it ideal for high speed and/or volume applications.

2 Algorithmic Framework

The theory and normalization procedures of moments are only briefly presented here to place the subsequent designs in context. A more detailed analysis can be found in [2]. For a grayscale image $g(x,y)$, with $x = 0, 1, \dots, M$ and $y = 0, 1, \dots, N$, the geometric moments m_{pq} of order $p+q$ are defined as

$$m_{pq} = \sum_{x=0}^M \sum_{y=0}^N x^p \cdot y^q \cdot g(x, y) \quad (1)$$

Translation invariance is achieved by calculating the central moments given by

$$\mu_{pq} = \sum_{r=0}^p \sum_{s=0}^q \binom{p}{r} \cdot \binom{q}{s} \cdot (-\bar{x})^r \cdot (-\bar{y})^s \cdot m_{p-r, q-s} \quad \text{with } \bar{x} = \frac{m_{10}}{m_{00}}, \bar{y} = \frac{m_{01}}{m_{00}} \quad (2)$$

Invariance with respect to isometric scale and to scalar intensity changes (which arise from uniform illumination intensity changes) is achieved by n_{pq} , using

$$n_{pq} = \frac{\mu_{pq}}{\mu_{00}} \cdot \left(\frac{\mu_{00}}{\mu_{20} + \mu_{02}} \right)^{\frac{p+q}{2}} \quad (3)$$

Invariance with regards to in-plane rotations and/or reflections is more involved, and a number of feature sets exist. The complex moment magnitudes give rise to a concise yet powerful rotation and reflection invariant descriptor, and will be considered here. The complex moments C_{pq} can be calculated from n_{pq} of using

$$C_{pq} = \sum_{r=0}^p \sum_{s=0}^q \binom{p}{r} \cdot \binom{q}{s} \cdot i^{p+q-r-s} \cdot (-1)^{q-s} \cdot n_{r+s, p+q-r-s} \quad (4)$$

The complex moment magnitudes are then calculated as

$$|C_{pq}| = \sqrt{(C_{pq}^{real})^2 + (C_{pq}^{imag})^2} \quad (5)$$

Thus, the above descriptor is normalized with respect to translation, isometric scale, illumination intensity changes, in-plane rotation and reflection. A classification function is also required to compare it to the descriptors of the given templates, e.g. for image retrieval. With moment-based methods, distance-based classification functions are most commonly used. An example is the weighted Euclidean metric, defined between an unknown sample descriptor X and the descriptor of the t^{th} template Y_i as

$$d_i = \sqrt{\sum_{j=1}^n w_{ij} (X_j - Y_{ij})^2} \quad (6)$$

where n is the dimensionality of the descriptors. The weight w is commonly given by an expression which takes into account the variance for each feature and for the different templates when multiple samples represent each template, or it can be 1 for single sample templates, giving rise to the simple Euclidean metric.

The algorithms presented here have been used mostly in the processing of binary and grayscale images. In [2], a framework is proposed for the processing of color images based on the above methodology or, indeed, using any type of moments and normalization procedures. This entails the treatment of each color plane as an isolated grayscale image for the derivation of invariant descriptors followed by a fusion stage. Different fusion schemes are examined in [2], such as descriptor aggregation followed by classification or single plane classification followed by decision fusion. Furthermore, because for RGB images changes in the intensity or spectral power distribution of the incident illumination result in an independent scalar change of each color plane, this framework achieves invariance not only to geometric and illumination intensity changes but also to changes in the color of the incident illumination.

3 Parallel Moment Computation Circuit

The fast calculation of the geometric moments of an intensity image relies on a parallel computation structure. Our module is based on Hatamian's work [5] towards the implementation of a VLSI moment calculation chip and is, effectively, a cascaded accumulator structure. The organization of the moment computation module for the calculation of moments up to the fifth order can be seen in Figure 1. The row processing elements (RPEs) process each pixel of each image row y and produce the outputs $Y_0(y) \dots Y_5(y)$. The column processing elements (CPEs) process the results produced by the RPEs for each row y and produce outputs for the entire image.

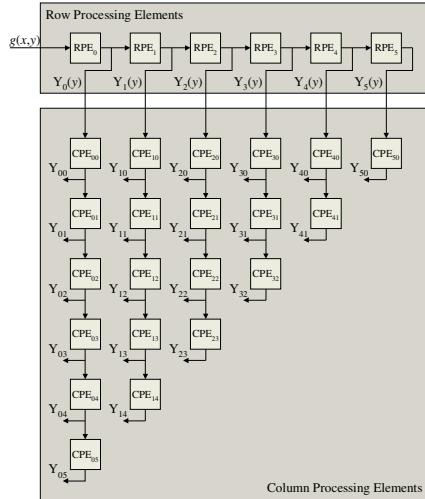


Fig. 1. Organization of the parallel moment computation circuit

The RPEs have been implemented as parallel accumulators, to achieve a pixel per cycle processing rate. Each RPE has a different size, from 17 bits for RPE_0 up to 53 bits for RPE_5 . These values were chosen so that an overflow is guaranteed not to occur for an 8-bit intensity image with rows of up to 512 pixels. The CPEs have been implemented as identical 64-bit serial accumulators, giving rise to a CPE systolic array. This facilitates a straightforward implementation and simple control logic. The serial implementation is the preferred design choice, since the CPEs operate at the image row level and need not be as fast as the RPEs. The serial accumulators have been implemented using the FPGA's function generator RAMs (LUTRAMs) instead of registers, drastically reducing the circuit size. The 64-bit accumulator size has been chosen so that an overflow is guaranteed not to occur for 512×512 pixel 8-bit intensity images. As for the calculation of moments of higher than fifth order, this can be easily achieved by extending the RPE chain and the CPE systolic array. The reconfigurability of the FPGA device allows the redeployment of such modified modules. The CPE structure considered here is more efficient than the one proposed by Hatamian, which requires almost twice the resources because it calculates not only the moments up to the required order, but also incomplete sets of higher order moments.

Note that the values produced by this circuit are not, in fact, the geometric moments of the image. The moments m_{pq} are derived through the simple algebraic combination of the Y_{pq} outputs. These equations are not included here, but they are can be found in [2]. Of relevance here is the fact that the calculation of m_{pq} from Y_{pq} involves a total of 45 multiplications of Y_{pq} values with 8-bit integers followed by 45 additions of these products. Although this processing is quite simple, no custom component was created because it will performed by the processing units described next.

Table 1 shows the implementation statistics of the complete moment computation module on a XILINX® XCV1000 Virtex™ FPGA of -6 speed grade [6]. At 4.56% usage, the circuit is quite small. These figures also include 53 I/O synchronization

Table 1. Circuit statistics for the moment computation module

Slices	535 (4.35%)
Slice flip-flops	628
4-input LUTs	664
Dual Port LUTRAMs	84
GCLKs	1 (25%)
Total equivalent gate count	21,287

registers. The circuit can operate at up to 50MHz, the critical path lying on the RPE chain and comprised of 62.3% and 37.7% logic and routing delays respectively. Thus, for an image of an $M \times N$ pixels, the frame rate is $(50 \times 10^6)/(M \times N)$ frames/sec., e.g. ~3051 frames/sec. for 128×128 pixel images and ~190 frames/sec. for 512×512 pixel images. This is ~750 times faster than a direct implementation and ~100 faster than a recursive addition implementation on a SUN UltraSPARC10 server.

4 Floating-Point Arithmetic Unit

The design and implementation of FPGA floating-point arithmetic circuits has been examined by various researchers [7-11]. Although such circuits cannot match the performance of the floating-point units of state-of-the-art microprocessors, they are extremely useful in systems and applications which benefit from custom parallel processing structures, but also require floating-point capabilities, such as the system considered in this paper. Thus, this section presents FPGA circuits for all the common floating-point operations, i.e. addition/subtraction, multiplication, division and square root, the last two being the least investigated in the literature. While previous work addressed the implementation of operators in the 32-bit single precision or even lower precision custom formats, in order to reduce the circuit costs and increase their speed, our aim was the creation of low cost operators that provide 64-bit double precision arithmetic. Furthermore, our circuits are IEEE-754 [12] compliant, implementing round-to-nearest-even rounding, able to handle infinities and NaNs, etc. Because the detailed treatment of this material is not feasible here, this section will focus only on the circuit statistics and performance of the units, while a very detailed description of these designs and implementations can be found in [2]. The implementation statistics of all four operators on the aforementioned device are shown in Table 2.

Addition and subtraction are, in general, the most frequent floating-point operations in scientific computing. Our double precision floating-point adder aims at a low implementation cost combined with a low latency. A non-pipelined design was adopted, so that key components may be reused, with a fixed latency of three clock cycles. At 5.49% usage, the circuit is quite small. These figures also include 194 I/O synchronization registers. The circuit can operate at up to 25MHz, the critical path lying on the significand processing path and comprised of 41.1% and 58.9% logic and routing delays respectively. Since the design is not pipelined and has a fixed latency of three clock cycles, this gives rise to a performance in the order of 8.33MFLOPS. Obviously, the implementation considered here is small enough to allow multiple instances to be incorporated in a single FPGA device if needed.

Table 2. Double precision floating-point operator circuit statistics

	Adder	Multiplier	Divider	Square Root
Slices	675 (5.49%)	495 (4.03%)	343 (2.79%)	347 (2.82%)
Slice flip-flops	336	460	400	316
4-input LUTs	1,118	604	463	399
GCLKs	1 (25%)	2 (50%)	1 (25%)	1 (25%)
Total equiv. gate count	10,334	8,426	6,464	5,366

The double precision floating-point multiplier also aims at a low implementation cost while maintaining a relatively low latency, considering the scale of the significand multiplication involved. A non-pipelined design was adopted with a fixed latency of ten cycles. The circuit operates on two clocks, a primary or global clock (CLK_1), to which the ten clock cycle latency corresponds, and an internal secondary clock (CLK_2), which is twice as fast as the primary clock and is used by the significand multiplier. The overall circuit is quite small, occupying only 4.03% of the device. These figures also include 193 I/O synchronization registers. The primary clock CLK_1 can be set to a frequency of up to 40MHz, its critical path comprised of 36.4% and 63.6% logic and routing delays respectively, while the secondary clock CLK_2 can be set to a frequency of up to 75MHz, its critical path comprised of 36.8% and 63.2% logic and routing delays respectively. Since the circuit is not pipelined with a fixed latency of ten CLK_1 cycles, a frequency of 33MHz and 66MHz for CLK_1 and CLK_2 respectively gives rise to a performance in the order of 3.3MFLOPS.

In general, division is much less frequent than the previous operations. Because of this, our double precision floating-point divider aims mainly at a low implementation cost. A non-pipelined design was adopted, incorporating an economic significand divider, with a fixed latency of 60 clock cycles. The circuit is very small, occupying only 2.73% of the device, which also includes 193 I/O synchronization registers. This circuit can operate at up to 60MHz, the critical path comprised of 42.8% and 57.2% logic and routing delays respectively. Since the design is not pipelined and has a fixed latency of 60 clock cycles, this gives rise to a performance in the order of 1MFLOPS.

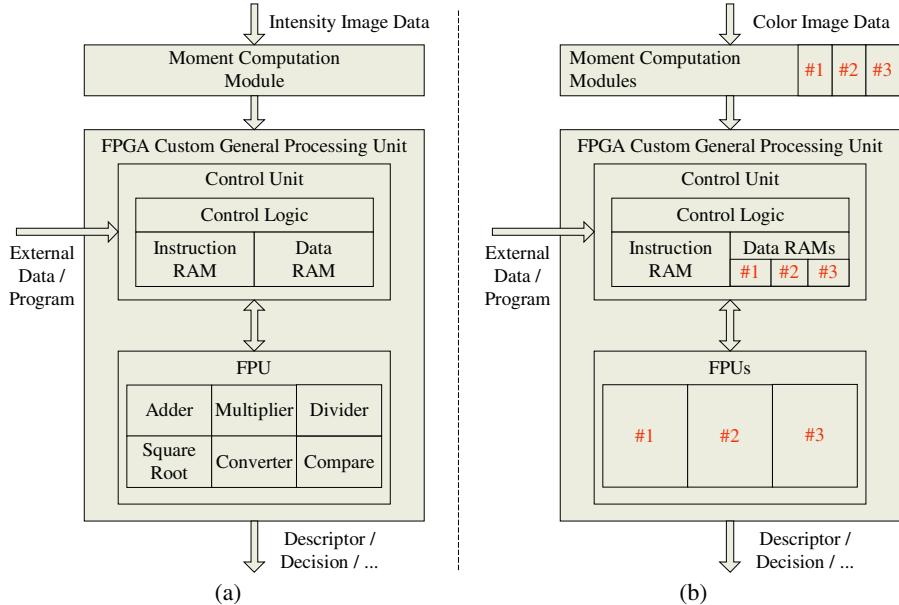
The square root function is the least frequent of the operations considered here. As for the divider, our double precision floating-point square root circuit aims at a low implementation cost. A non-pipelined design was adopted with a fixed latency of 59 cycles. The circuit is very small, occupying only 2.83% of the device, which also includes 129 I/O synchronization registers. The circuit can operate at up to 80MHz, the critical path comprised of 53.0% and 47.0% logic and routing delays respectively. Since the implementation considered here is not pipelined and has a fixed latency of 59 clock cycles, this gives rise to a performance in the order of 1.36MFLOPS.

The above discussion and the associated implementation statistics illustrate that our aim was the implementation of high precision operators at a low cost rather than with a very high performance. Our floating-point unit can perform a few million double precision operations per second and is meant to complement custom FPGA architectures, such as the one of Section 3, eliminating the need for integrating a microprocessor into the system. Furthermore, the circuits sizes allow the integration of multiple operators on the same device if necessary, while their self-contained implementation allows one to choose only the required operators for a given system.

5 System Architecture

The system that we have implemented for the processing of intensity images based on the components of the previous sections is illustrated in Figure 2(a). The first main stage in our architecture is the parallel processing structure of Section 3, the moment computation module. The input to this module can be a video signal directly from a camera, some other image delivery mechanism, or some other pre-processing module, e.g. a noise filter. The second main stage is a custom FPGA general processing unit which, in turn, comprises two main parts. The first main part is a double precision floating-point unit (FPU) which includes the circuits of the previous section. Two additional modules, which we did not describe earlier due to the simplicity of their implementation, are an integer to floating-point converter and a floating point comparator. The converter is very small, occupying ~1% of the device, has a variable latency of 1 to 15 clock cycles and a maximum clock speed of 145MHz. The comparator also occupies ~1% of the device, has a fixed latency of 1 cycle (input buffering) and a maximum clock speed of 135MHz. The second main part of the general purpose processing unit is a controller module, the main components of which are a control logic unit, an instruction RAM and a data RAM. The data RAM is an 8-bit address 64-bit data memory, implemented using the dedicated RAM blocks of the FPGA (BRAMs). This memory is used by the FPU, as well as for externally provided constants and statistics. Clearly, external RAM modules may very easily be used if large storage is required or if no RAM is available in the FPGA device. The instruction RAM is a 10-bit address 28-bit data memory, also implemented using BRAMs. This memory stores all the operations required for the calculation and classification of the invariant descriptors in the form of FPU commands.

In order to assess how the custom processing unit compares to the fast and parallel moment computation circuit, consider the following scenario. Assume that all the normalization procedures of Section 2 are desired. Obviously, these equations are not stored in the closed forms examined earlier, but must be expressed in an explicit form. Once expressed in such a form, one also discovers that there exists a great amount of redundancy and duplication of calculations, which can be eliminated. Note that the system is not restricted to these normalization and classification functions. By changing the contents of the instruction and/or data RAMs, different functions may be implemented or even different types of moments based on geometric moments. Furthermore, assume that the target application is a 10-way classification problem, i.e. assigning an unknown image to one of ten different templates. This affects the complexity of the classification function. The entire general processing unit has a fixed latency, with the floating-point converter being the only exception, for which we can assume a constant worst case latency. With the above scenario, all the processing, i.e. from the delivery of the results of the moment computation circuit up to a classification decision being delivered, requires 6806 clock cycles. The derivation of this figure is not included here but may be found in [2]. During all of this time, the moment computation module can carry on with the processing of the next image. Now, compare that with the processing time of the moment computation module, which has a pixel per cycle processing rate, e.g. 16384 cycles for 128×128 pixel images and 262144 cycles for 512×512 pixel images. In both cases, the latency of the moment

**Fig. 2.** (a) Binary/grayscale descriptor system (b) Color descriptor system**Table 3.** Implementation statistics for the intensity and color descriptor systems

	Intensity descriptor system	Color descriptor system
Slices	3,437 (27.97%)	9,825 (79.96%)
Slice flip-flops	3,108	8,821
4-input LUTs	3,775	10,975
Dual port LUTRAMs	84	252
Bonded IOBs	77 (19.06%)	79 (19.55%)
TBUFs	1,600 (12.76%)	4,800 (38.27%)
BRAMs	11 (34.38%)	19 (59.38%)
GCLKs	2 (50%)	2 (50%)
GCLKIOBs	2 (50%)	2 (50%)
Total equiv. gate count	248,531	509,541

computation module completely overshadows the latency of all the other circuits. The timing analysis indicates that the entire system can be clocked at up to 25MHz, the speed of the floating point adder being the determining factor. However, one can adopt a dual clock strategy and clock the moment computation circuit at 50MHz, which is feasible based on the results of Section 3. In that case, the entire system has the same processing rates with the moment computation module, i.e. ~3051 frames/sec. for 128×128 pixel images and ~190 frames/sec. for 512×512 pixel images. These speeds certainly satisfy the requirements for real-time systems or for multiple camera systems or for other high volume applications. The implementation statistics of this system can be seen in Table 3. Occupying 27.97% of the aforementioned XCV1000 device, the design is not negligible, but it is certainly feasible.

With regards to the processing of color images based on the information fusion framework briefly described in Section 2, a number of choices are available. Thus, one can use the hardware described above (its speed certainly allows that, even for real-time processing) for the sequential processing of the color planes, storing different sets of statistics for each plane for use by the classifier. Alternatively, one can employ three moment computation modules and a single general processing unit, and so on. The system outlined in Figure 2(b) is yet another possibility. There, three moment computation modules are used along with three complete FPUs, the FPUs sharing a common control logic and instruction register but each having its own data RAM. The sharing of the control and instruction logic is possible because all the color planes are processed in the same manner and the different components of the system have a fixed latency (the worst case latency must be enforced on the floating-point converters). Assuming a descriptor aggregation scheme for the fusion step, the actions performed by this system are basically the same as before, with some additional steps in the instruction RAM to implement the fusion framework. With a dual clock strategy, this system can also process ~3051 frames/sec. and ~190 frames/sec. for 128×128 pixel and 512×512 pixel 24-bit RGB images respectively. Alternatively, the same organization can be used for the parallel processing of grayscale images, achieving ~9153 frames/sec. and ~570 frames/sec. for 128×128 pixel and 512×512 pixel grayscale images respectively. The statistics of this system are also shown in Table 3. Occupying 79.96% of the device, the system requires the best part of the chosen FPGA, but is still feasible within a single chip.

6 Discussion and Conclusions

We have presented a new FPGA architecture for the high speed extraction, normalization and classification of moment descriptors. FPGAs are ideal for the implementation of such systems because they combine the reprogrammability advantage of general purpose processors with the parallel processing and speed advantages of custom hardware. Our system relies on a custom parallel moment computation module working alongside a custom low cost general processing unit. The specific characteristics of the moment computation module, such as highest moment order and maximum image dimensions, can be easily changed and the module redeployed by reconfiguring the FPGA. The reprogrammable general processing unit, on the other hand, allows the easy deployment of different normalization and classification functions and can be programmed for different recognition problems without any design modifications. Even if such changes are required, the reconfigurability of the FPGA makes such a task feasible. In this manner, one can always implement the exact processing unit required for a system. It is interesting to note that the design of this general processing unit was actually straightforward once all the modules had been created. In terms of design effort, the design and implementation of the double precision floating-point operators was probably the most involved task, these circuits also involving the most laborious testing procedures. It is for this reason that we have implemented each operator as a low cost standalone unit instead of implementing an FPU that shares hardware components between the operators. That is, so that one can choose only the operators needed for a given system, image processing or not, and also have multiple instances of individual operators if necessary. The performance of our overall archi-

ture, in the order of hundreds or thousands of binary, grayscale or color images per second, makes it suitable for high speed and/or volume applications, e.g. for image recognition, detection and retrieval, while its self-contained implementation also allows its deployment in small standalone systems.

References

1. Rahman, A.F.R.: Study of Multiple Expert Decision Combination Strategies for Handwritten and Printed Character Recognition, Ph.D. Thesis, Electronic Engineering Laboratory, University of Kent at Canterbury, UK (1997)
2. Paschalakis, S.: Moment Methods and Hardware Architectures for High Speed Binary, Greyscale and Colour Pattern Recognition, Ph.D. Thesis, Department of Electronics, University of Kent at Canterbury, UK (2001)
3. Bober, M., Preteux, F., Kim, W.Y.: Shape Descriptors. In Manjunath, B.S., Salembier, P., Sikora, T. (eds.): *Introduction to MPEG-7*, John Wiley & Sons (2002) 231-260
4. Teh, C.H., Chin, R.T.: On Image Analysis by the Method of Moments, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 4 (1988) 496-513
5. Hatamian, M.: A Real-Time Two-Dimensional Moment Generating Algorithm and Its Single Chip Implementation, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-34, no. 3 (1986) 546-553
6. Virtex™ 2.5V Field Programmable Gate Arrays, XILINX® Corporation (2000)
7. Ligon, W.B., McMillan, S., Monn, G., Schoonover, K., Stivers, F., Underwood, K.D.: A Re-Evaluation of the Practicality of Floating Point Operations on FPGAs, In Proc. IEEE Symposium on FPGAs for Custom Computing Machines (1998) 206-215
8. Louca, L., Cook, T.A., Johnson, W.H.: Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs, In Proc. IEEE Symposium on FPGAs for Custom Computing Machines (1996) 107-116
9. Li, Y., Chu, W.: Implementation of Single Precision Floating Point Square Root on FPGAs, In Proc. Fifth IEEE Symposium on Field Programmable Custom Computing Machines (1997) 226-232
10. Tangtrakul, A., Yeung, B., Cook, T.A.: Signed-Digit On-Line Floating-Point Arithmetic for FPGAs, In Proc. High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic (1996) 2-13
11. Shirazi, N., Walters, A., Athanas, P.: Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines, In Proc. IEEE Symposium on FPGAs for Custom Computing Machines (1995) 155-162
12. ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic, IEEE (1985)

Design and Implementation of a Novel FIR Filter Architecture with Boundary Handling on Xilinx VIRTEX FPGAs

A. Benkrid, K. Benkrid, and D. Crookes

School of Computer Science, The Queen's University of Belfast, UK
a.benkrid@qub.ac.uk

Abstract. This paper presents the design and implementation of a novel architecture for FIR filters on Xilinx Virtex FPGAs. The architecture is particularly useful for handling the problem of signal boundaries filtering, which occurs in finite length signal processing (e.g. image processing). It cleverly exploits the Shift Register Logic (SRL) component of the Virtex family in order to implement the necessary complex data scheduling, leading to considerable area savings compared to the conventional implementation (based on a hard router), with no speed penalty. Our architecture uses bit parallel arithmetic and is fully scalable and parameterisable. A case study based on the implementation of the standard low filter of the Daubechies-8 wavelet on Xilinx Virtex-E FPGAs is presented.

1 Introduction

Finite Impulse Response (FIR) filters are widely used in digital signal processing. An N-tap FIR filter is defined by the following input-output equation [1]:

$$\text{out}(n) = \sum_{i=0}^{N-1} x(n-i) h(i) \quad (1)$$

where $\{h(i) : i = 0, \dots, N-1\}$ are the filter coefficients.

Figure 1 shows the two conventional structures (the direct and the inverse form) of an FIR filter [2]. Both structures of Fig 1 seek to align the products $x(n-i)h(i)$ of equation (1) in time before accumulation.

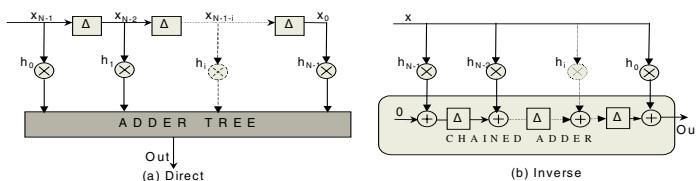


Fig. 1. Two conventional FIR filter structures

An FIR filter implements a convolution operation [1], which is often built on the assumption of infinite length signals e.g. continuous audio signal. Finite length signals (e.g. images) on the other hand, have discontinuities at the boundaries (see Fig 2). Thus emerges the problem of which values to use at these regions.

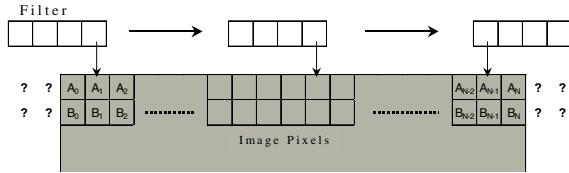


Fig. 2. Boundary problem when filtering an image

Although, this problem could be ignored for a one-stage convolution, it cannot be discarded when implementing a multi-stage convolution as in Discrete Wavelet Transform [3]. A usually recommended solution to this problem is to extend each row by reflection at the signal boundary [4] as shown in Fig 3.

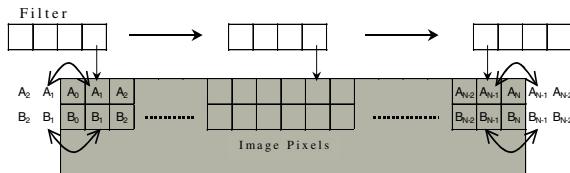


Fig. 3. A finite 2-D signal (image) filtering with boundary extension by reflection

For an N-Tap FIR, the minimum number of extra samples to be introduced is constant and equal to $N-1$. This is because $P+(N-1)$ input samples are required to generate P output samples. However, the number of samples to be added at the left border of the input signal (referred to by α) or the right one (referred to by μ) can be variable, i.e. not a constant.

To handle the problem of boundary processing in hardware, Chakrabati [5] proposed the use of a router (or switcher) to feed the appropriate data, in parallel, to the multipliers (see Fig 4).

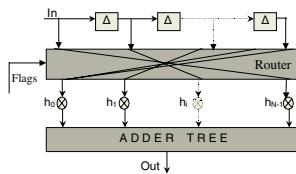


Fig. 4. A conventional FIR architecture with a hard-router to handle the boundary processing

The hard router is implemented using multiplexers. A controller is needed to drive the appropriate multiplexers' selection signals. A minimum of $W(\sum_{i=2}^{\alpha+1} \lceil i/2 \rceil + \sum_{j=2}^{N-\alpha} \lceil j/2 \rceil)$

LUTs are required to implement the router for an N-tap FIR filter[6] if its input signal is extended by α samples at its left side. This represents an $O(N^2W)$ hardware complexity¹ and therefore requires considerable area and routing resources which will have a negative effect on the speed performance of the implementation.

To overcome this high area cost of the Hard Router, we have presented in [6] a novel architecture for symmetric FIR filter family. The suggested structure is parameterised and scalable. It led to considerable area saving but with speed penalty as it requires the use of clock doubler. The actual paper overcomes this problem and addresses basically a general FIR filter. Nonetheless, the results provided in this paper can be tailored for the symmetric FIR filter type. Unfortunately, because of the paper size limit, this will not be detailed in this paper. The following will explain our approach for a general FIR filter.

Our suggested architecture is tailored to the Xilinx Virtex FPGA family. It exploits mainly the Virtex Shift Register Logic component: SRL16 [7]. SRL16 is implemented by the Virtex slices' LUT (see Fig 5). There are two LUTs in every Virtex slice. Each one can be configured to create a shift register (SRL16) that varies in length from 1 to 16 bits. Longer shift register length can be implemented with multiple chained SRL16. As shown in Fig 5, the SRL16 configuration consists of a chained delay with a multiplexer at the output. The input address $Addr[3:0]$ selects which bit in the chained delay to be output hence controlling the length of the shift register from 1 to 16. Note that each SRL16 can be immediately pipelined by using the flip-flop available on the same slice logic cell [7].

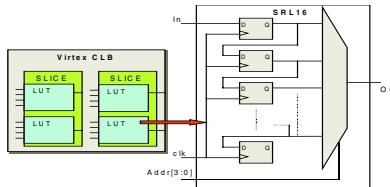


Fig. 5. Virtex CLB and SRL16 configuration.

The remaining of this paper will first present the basic architecture of our novel FIR architecture regardless of the signal boundaries filtering. It then shows how this basic architecture can be easily extended to handle signal boundaries processing with little hardware penalty. A detailed approach will be given. Then, area measurements of our novel architecture will be presented and compared with the corresponding results from a conventional hard-router based FIR implementation. Timing and area results of a case study implementation will be provided. Finally conclusions will be drawn.

Throughout the remaining of the paper, we will assume the use of bit parallel arithmetic. The term SRL will correspond to either one SRL16 component or a chained SRL16 components if required. The abbreviation 'cc' denotes the term clock cycle and the term $SRL_{(i)}$ refers to the SRL associated with the filter coefficient h_i .

¹ $\sum_{i=1}^N i = N(N+1)/2 = O(N^2)$

2 Our Novel FIR Filter Architecture

In order to handle the boundary filtering efficiently, we suggest the novel architecture shown in Fig 6. The input data samples (X) in this figure structure are first multiplied in parallel with the filter coefficients. Then, the multiplication results $x(n-i) \cdot h(i)$ of equation (1) are skewed and aligned in time properly using the SRL to produce the filter output.

The SRL layer in Fig 6 structure aims to skew the products and align them properly in time before parallel accumulation as shown in Fig 7. In fact, unlike Fig 1.a structure, our structure does not include an input samples chained delay and therefore the supply of the products onto the adder tree needs to be synchronised by the SRL layer before *parallel* accumulation.

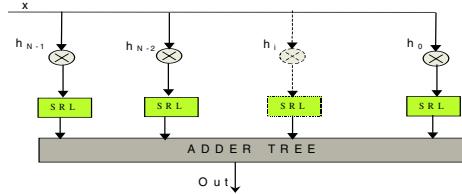


Fig. 6. Our novel FIR filter structure for signal borders processing using symmetry extension

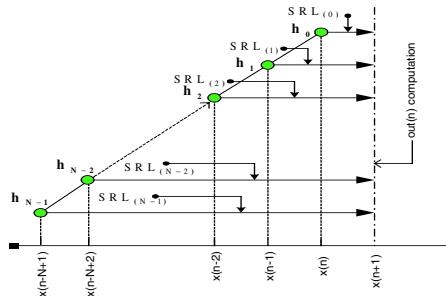


Fig. 7. Data Dependence Graph (DDG) of N-tap FIR filter. The horizontal arrows show the SRLs' delays length (the filled circles represent the relevant products $x(n-i) \cdot h(i)$)

Table 1 gives the SRLs delay length. Each SRL delay length is equal to the projection length of its associated product instant on the time axis abscise “ $n+1$ ”(see Fig 7).

Table 1. The SRLs' delay length for an FIR filter with no boundary processing

SRL ₍₀₎	SRL ₍₁₎	SRL ₍₂₎	SRL _(N-2)	SRL _(N-1)
1	2	3		N-1	N

It is worth noting that the SRLs in Fig 6 structure can be placed before or after the multipliers (thus skewing the product $x(n-i)h(i)$ or the multiplicands x). If the filter coefficients are fractional and truncation is carried out at the output of the multipliers, the multipliers' output word lengths could be smaller than their input ones. Thus, for

area optimisation, the SRLs should be placed at the multipliers output. On the other hand, if the filter coefficients values are greater than 1, the multipliers' output word lengths will be greater than their inputs. Therefore, placing the SRLs before the multipliers will lead to a more compact implementation.

Besides from being able to implement a convolution operation, our structure seeks primarily to handle signal boundaries processing. This is handled efficiently with very little hardware overhead, thanks to the SRLs dynamic skewing feature. The following sections will detail our own approach to achieve this goal.

Throughout, the term P_i denotes the FIR DDG's product associated with the multiplier h_i . In particularly, P_{N-1} (P_0) denotes the first (last) FIR DDG's node.

3 An Extension to Handle Signal Boundaries

To handle *signal boundary filtering*, no alteration on the architecture of Fig 6 is necessary. In fact, it is handled efficiently by a proper skewing of the products P_i through *a dynamic SRLs addressing*. For this purpose, we will present an algorithm, which allows us to find the required SRLs delay length values according to the filter length, N , and the symmetry-filtering axis. The latter is defined through the number of input samples, α , which should be added at the left side of the signal, and to the number of input samples, μ , which should be introduced at its right side, where $\alpha+\mu=N-1$.

Fig 8 shows a 4-tap FIR filter DDG at the boundary regions. The dashed arrows show where the signal extension using symmetry reflection is applied. The deduction of such graphs will be detailed later in the section. In this figure, the filled circles refer to the filter products whereas the shaded rectangle shows the boundary region between the two sequences I and (I-1). Sequence-I refers to the actual sequence of samples, whereas Sequence-(I-1) to its previous one. The negative values (-1, -2, -3...) will denote all instants before instant 0 of sequence I. The boundary DDGs associated with sequence-I represent actually the DDGs at the left side signal boundary, whereas the ones of sequence-(I-1) represent the ones at its right side edge.

From Fig 8, and as a general rule we can see that the *regular* DDG (a straight line as depicted in Fig 7) of a generic N -tap filter ends at the P_{N-1} of instant “- N ” (= -4 in Fig 8), and starts from the P_0 of instant $N-1$ (= 3 in Fig 8). Between those two values, the DDG becomes irregular.

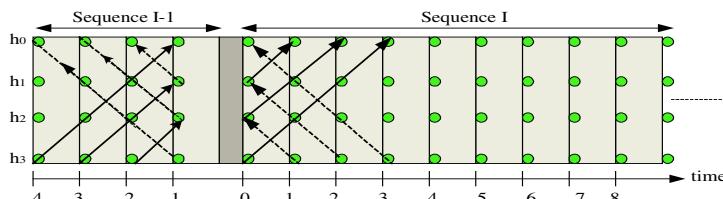


Fig. 8. Data dependence graph of a 4-tap FIR filter ($N=4$) using boundary processing with symmetry extension

We will refer to the irregular DDG *deflection point* by the term **Hub**. The term P_{Hub} will represent its associated product whereas $P_{\text{Hub},i}$ ($P_{\text{Hub},+i}$) represent the i^{th} DDG's product that comes before (after) the Hub (i is a positive integer).

Because of the DDG irregularity at the signal boundary, the SRLs delay length given in table 1 should then be updated. We therefore suggest the following approach.

An Approach to Determine the SRLs Delay Length when Using the Symmetry Signal Extension

Once the filtering symmetry axis (i.e. α and μ , where $\alpha+\mu=N-1$) is determined, the updated SRLs' delays length could be deduced using two main steps associated respectively with the left side and the right side input signal.

Fig 9.a (Fig 9.b) shows the DDG at the left (right) side boundary of sequence-I input signal, where α (β) samples are introduced through symmetry reflection (see the arced arrows). This was needed since the $P_{\text{Hub},i}$ ($P_{\text{Hub},+i}$) multiplicands at this boundary region correspond to sequence-(I-1) (sequence-(I+1)) samples (see the dashed line).

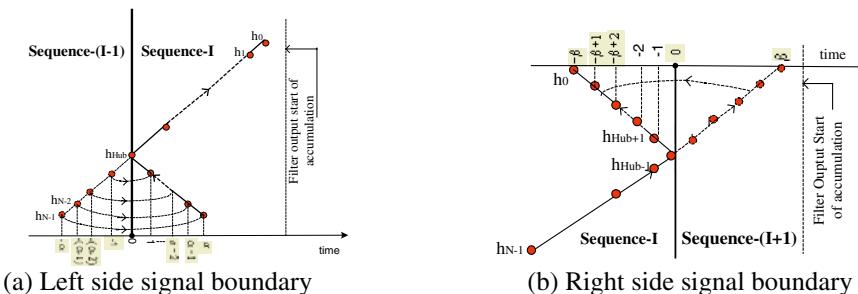


Fig. 9. The irregular DDG for an FIR filter when using symmetry extension.

For the proper functionality of the Fig 6 structure, all the DDG products should feed the adder tree at the same time. In fact, the parallel accumulation can't start till all the products have been computed. Table 1 gives the SRLs delay length when no boundary processing is handled. The values given assume implicitly the instant of computation of P_0 as a reference (see Fig 7). The question that arises: will this reference needs to be changed if we process the border filtering?

Fig 10 shows two cases where P_0 and P_{N-1} are computed in different relative order. We can see from Fig 10.b that the accumulation can't start one cc after the computation of P_0 if this latter is calculated earlier than P_{N-1} . Instead, the accumulation can start one cc after the computation of P_{N-1} (instant t_2 rather than t_1 , see Fig 9.b). The change in time (t_2-t_1) denotes the difference between the computation instants of P_{N-1} and P_0 . We therefore define a variable ξ such that:

$$\xi = \max[(t[P_{N-1}]-t[P_0]), 0]$$

This variable value should then be added to the SRLs delay length values given in table 1. In fact, when $t[P_{N-1}]-t[P_0]<0$ (see Fig 10.a), ξ is equal to zero, since no update is needed (the accumulation starts one cc after the computation of P_0 as assumed

implicitly in table 1). However, when $t[P_{N-1}] - t[P_0] > 0$ (see Fig 10.b), $\xi = t[P_{N-1}] - t[P_0]$ which is equal to the required SRLs delays length increment as explained in the last paragraph.

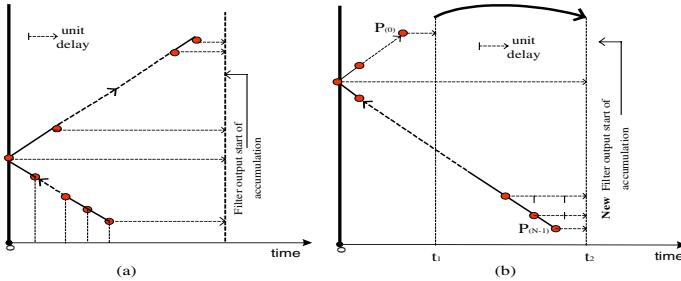


Fig. 10. The effect of P_{N-1} and P_0 relative time order on the start of accumulation instant

It is worth noting that if we extend the input signal by α samples at its left side, $\xi = \max[2\alpha + 1 - N, 0]$ since $t[P_{N-1}] = \alpha$ and $t[P_0] = N - (\alpha + 1)$. This will correspond to the SRLs delay length update value for the first filter output. However, since all the filter outputs are produced regularly in time (delayed by one cc), the same update value (ξ) needs to be added to the SRLs delays length associated with the second, third etc filter outputs, and in particular to the SRLs delays length associated with the non-boundary regions. Therefore, we can represent the delays length of the SRLs list in the *non-boundary* region by a *RegSkew* list where:

$$\text{RegSkew} = [\lambda, \lambda-1, \lambda-2, \dots, \xi+3, \xi+2, \xi+1], \text{ where } \lambda = N + \xi \quad (2)$$

This list values are applied on tap-(N-1) to tap-0.

Nonetheless, the individual SRLs delay length at the boundary regions need still to be determined because of the DDGs irregularity. For this sake, we consider in the following the left side of the signal and then its right side.

Left Side Signal Boundary Extension

Taking into account Fig 7 structure as a reference, we can see from Fig 9.a that the $P_{\text{hub},i}$ products are advanced in time relatively to the accumulation instant. Thus, their associated regular delays length should be *decreased*. From this figure, we can see that P_{N-1} is computed at instant α instead of $(-\alpha)$. This represents a change of 2α cc's delay. Logically, the associated SRL delay length should be *decremented* by this value. Similarly P_{N-2} is computed at instant $\alpha-1$ instead of $(-\alpha+1)$. This represents a change of $2(\alpha-1)$ cc's delay. The associated SRL delay length should be then *decremented* by this value. Following the same reasoning, we can conclude easily that if α samples are introduced at the left side of the input signal, the $P_{\text{hub},i}$ ($i \in [\alpha, \alpha-1, \dots, 0]$) delays length value for the first filter output should be *updated* by the L_i list values:

$$L_i = [-2\alpha, -2(\alpha-1), \dots, -2, 0]$$

The same reasoning can be applied on the second boundary filter output. The latter corresponds to $P_{N,i}$ of instant $(\alpha-1)$. The reader can verify easily that the update list L_2 for the $P_{hub,i}$ SRLs delay length is:

$$L_2 = [-2(\alpha-1), -2(\alpha-2), \dots, -2, 0]$$

These updates lists L_1, L_2, \dots should be added to the regular skew list RegSkew (equation 2) to deduce the final SRLs delay length.

By applying the same reasoning on the remaining α boundary outputs, the SRLs delay length at the boundary region can be represented with a *Left-Matrix* matrix expression such that:

$\text{Left-Matrix} = [\alpha_Matrix | Tail(\alpha, N-\alpha-1)]$, where:

- o $\text{Tail}(\alpha, N-\alpha)$ is a matrix of size $[\alpha, N-\alpha-1]$, where each of its rows is equal to:

$$[\lambda-\alpha-1, \lambda-\alpha-2, \lambda-\alpha-3, \dots, \xi+2, \xi+1]$$
- o α_Matrix is a matrix of size $[\alpha, \alpha+1]$, where:

$$\alpha_Matrix = \begin{bmatrix} \lambda-2a & \lambda-2a+1 & \lambda-2a+2 & \dots & \lambda-a-2 & \lambda-a-1 & \underline{\lambda-a} \\ \lambda-2a+2 & \lambda-2a+3 & \lambda-2a+4 & \dots & \lambda-a+2 & \underline{\lambda-a+1} & \lambda-a \\ \lambda-2a+4 & \lambda-2a+5 & \lambda-2a+6 & \dots & \lambda-a+2 & \lambda-a+1 & \lambda-a \\ \cdot & \cdot & \cdot & \dots & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \dots & \cdot & \cdot & \cdot \\ \lambda-6 & \lambda-5 & \lambda-4 & \underline{\lambda-3} & \lambda-a+2 & \lambda-a+1 & \lambda-a \\ \lambda-4 & \lambda-3 & \underline{\lambda-2} & \lambda-3 & \lambda-a+2 & \lambda-a+1 & \lambda-a \\ \lambda-2 & \underline{\lambda-1} & \lambda-2 & \lambda-3 & \lambda-a+2 & \lambda-a+1 & \lambda-a \end{bmatrix}$$

The underlined elements in this matrix expression refer to the delay length of the SRL_{Hub} . All the matrix elements put on bold refer to the pre-hub SRLs delay length. They form an upper triangular matrix.

Right Side Signal Boundary Extension

We can see from Fig 9.b that because of the irregularity of the DDG, P_0 is computed at instant $(-\beta)$ instead of β . This represents a change of 2β cc's delay. Logically the associated SRL delay length should be increased by this value. Similarly P_1 is computed at instant $(-\beta+1)$ instead of $(\beta-1)$. This represents a change of $2(\beta-1)$ cc's delay. Logically the associated SRL delay length should be incremented by this value. Following the same reasoning we can conclude easily that if β samples are introduced at the right side of a signal, the SRL_{hub+i} ($0 \leq i \leq \beta$) delays length of the first output should be *updated* by the following list, R_1 ,

$$R_1 = [0, 2, 4, 6, \dots, 2\beta]$$

This update list should be added to the RegSkew list expression (equation 2) to deduce the final SRLs delays length. By doing so for every β value in the $[1, \mu]$ interval, the SRLs delay length at the boundary region can be represented by a *Right-Matrix* matrix expression such that:

$\text{Right-Matrix} = [\text{Head}(\mu, N-\mu-1) | \mu_Matrix]$ where:

- o $\text{Head}(\mu, N-\mu-1)$ is a matrix of size $[\mu, N-\mu-1]$, such that each of its row is equal to:

$[\lambda, \lambda-1, \dots, \xi + \mu + 2]$

μ -Matrix a matrix of size $[\mu, \mu+1]$, where:

$$\mu\text{-Matrix} = \begin{bmatrix} \xi+\mu+1 & \xi+\mu & \xi+\mu-1 & \xi+\mu-2 & \dots & \xi+3 & \underline{\xi+2} & \xi+3 \\ \xi+\mu+1 & \xi+\mu & \xi+\mu-1 & \xi+\mu-2 & \dots & \underline{\xi+3} & \xi+4 & \xi+5 \\ \xi+\mu+1 & \xi+\mu & \xi+\mu-1 & \xi+\mu-2 & \dots & \xi+5 & \xi+6 & \xi+7 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \xi+\mu+1 & \xi+\mu & \underline{\xi+\mu-1} & \xi+\mu & \dots & \xi+2i-5 & \xi+2i-4 & \xi+2i-3 \\ \xi+\mu+1 & \xi+\mu & \xi+\mu+1 & \xi+\mu+2 & \dots & \xi+2i-3 & \xi+2i-2 & \xi+2i-1 \\ \xi+\mu+1 & \xi+\mu & \xi+\mu+2 & \xi+\mu+3 & \dots & \xi+2i-1 & \xi+2i & \xi+2i+1 \end{bmatrix}$$

The underlined elements in this matrix expression refer to the delay length of the SRL_{hub} . All the matrix elements put in bold refer to the post-hub SRLs delay length. They form a lower triangular matrix. By using the *Left_Matrix*, *Right_Matrix* matrices and the *RegSkew* list expressions, all the required delays length for the SRL layer of Fig 6 structure are determined.

4 Area Measurements

In this section, we will compare the area cost of our structure and Fig 4's architecture. Table 2 lists the resources used by those two structures where N is the filter length and W is the input wordlength.

Table 2. Logic resources consumed by different FIR structures

	Number of Multipliers	Number of Word Delays	Number of Adders	Router Hardware (LUTs)
Fig 4 Architecture	N	N-1	AdderTree(N)	$W\left(\sum_{i=2}^{\alpha+1} \lceil i/2 \rceil + \sum_{j=2}^{N-\alpha} \lceil j/2 \rceil\right)$
BenKrid Architecture	N	0	AdderTree(N)	SRLs Layer

From table 2, we can see clearly that our architecture consumes as many multipliers as Fig-4 structure. Our structure does not infer input samples delays unlike with Fig-4 structure saving thus a $(N-1)$ parallel word delays resource. However, it consumes as much hardware as Fig 4 structure for the accumulation task. The last remaining resource in table 2 is related to the router functionality. As explained in section 1, the hard router consumes $W\left(\sum_{i=2}^{\alpha+1} \lceil i/2 \rceil + \sum_{j=2}^{N-\alpha} \lceil j/2 \rceil\right)$ LUTs. With our structure, the

routing functionality is implemented through the SRL layer. For each SRL, we need to determine its maximum delay length (DL) value. This can be deduced easily from the expression of the *Left_Matrix*, *Right_Matrix* and *RegSkew* expression. If this value is less or equal to 16, one SRL16 (one LUT) can be used. However, if more depth is needed, more SRL16 should be chained, thus increasing the required number of LUTs. The number of the required LUTs is simply equal to $\lceil \frac{DL}{16} \rceil$. If we omit the area cost of the SRLs' address generators (which indeed can be considered negligible

in front of the final filter area), our SRL layer cost area will depend solely on the number of SRLs used.

For different α , μ and N values, Fig 11 depicts the router functionality area cost evolution. It shows clearly that the hard router consumes much more area than our suggested structure.

Therefore, we can conclude that our novel architecture consumes fewer resources than the conventional structure of Fig 4. The next section presents the real hardware implementation results for the standard low filter of the Daubechies-8 wavelet (8 taps) [3] on the Xilinx Virtex family FPGAs. These will be compared to an implementation of Fig 4 architecture (i.e. a conventional FIR architecture with a hard router).

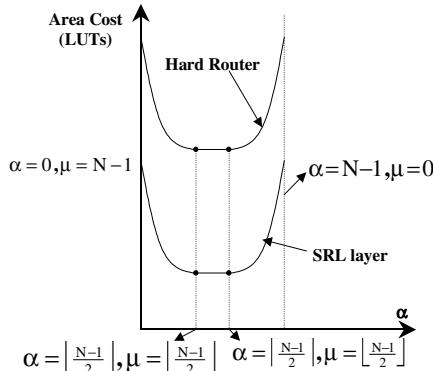


Fig. 11. Router area cost evolution when implemented using an SRL layer and a multiplexer layer (Hard Router), $W=1$

5 Case Study

In the following, we assume a bit parallel arithmetic. The FIR filters were implemented using 9-bit input word length, 8-bit coded coefficients and a 2-bit intermediate and final precision results. Our adders and multipliers were designed using the dedicated carry logic of the Virtex Xilinx CLBs. Since the filter coefficients are constant, the Canonic Signed Digit (CSD) representation based approach was used to design the multipliers [8]. Timing constraints were applied to determine the maximum achievable frequency.

Table 3 shows the performances achieved from implementing the Daubechies-8 FIR filter with no boundary processing. The structures of Fig 1 are used as well as ours (Fig 6). We can see that when using Fig 1.a structure, the implementation delivers higher speed but requiring more area comparing to the inverse form structure of Fig 1.b. Those two effects are due to the input chained delay of Fig 1.a structure, which will increase its area (since Fig 1.b structure does not include such resources) and will avoid using long routing line to feed the multipliers as in Fig 1.b structure. Our structure with no boundary processing delivers the same speed as for the conventional inverse FIR structure. However, it consumes more area because of the SRL layer.

Table 3. Performance of a low Daubechies-8 FIR filter implementation using two different architectures on Xilinx XCVE50-8 FPGA with no boundary processing

	Fig1-a architecture	Fig 1-b architecture	Benkrid architecture
Area(Slices)	147	113	146
Speed(Mhz)	~167	~159	~159

When handling the signal boundaries processing, the architecture of Fig 4 is implemented and compared to our structure. The SRLs Layer in our structure get addressed dynamically by the values of the *Left_Matrix*, *Right_Matrix* and *RegSkew* expression values. To implement the SRLs' address generators, we can conclude from section 3 that the SRL addressing can be subdivided into two categories:

Boundary regions: that will corresponds to N-1 states (corresponding to *Left_Matrix*, *RightMatrix* rows). These states can be stored in the slices distributed RAMs.

Non-boundary regions: the address will be constant. It corresponds to a *RegSkew* element value.

A counter line and multiplexer are needed to flag the boundary transition instant.

Table 4 lists the performances achieved. The second column of table 3 and 4 shows the effect of the Hard Router on Fig 1.a structure. It involves the use of 78 extra slices with ~6 Mhz speed penalty. On the other hand, by comparing the performance of our structure with and without boundary processing, we can see clearly that the dynamic skewing of the SRL layer does introduce a slight area penalty (12 slices) with no speed penalty.

Table 4 shows clearly that our architecture is more compact in area (almost 70 slices less) and run at the same speed range. It is worth noting that the Hard Router has been implemented at word level therefore seeking its highest speed implementation.

Table 4. Performance of a low Daubechies-8 FIR filter implementation using two different architectures on Xilinx XCVE50-8 FPGA with boundary processing

	Fig 4 architecture	Benkrid architecture
Area(Slices)	225	158
Speed(Mhz)	~161	~159

6 Conclusion

In this paper, we have presented a novel architecture for FIR filters with signal boundary handling. This architecture is tailored to Xilinx Virtex FPGAs family. It cleverly exploits the SRL16 component to implement the FIR filters. The problem of signal boundary processing, which occurs in finite length signals filtering, is smartly and efficiently handled by an appropriate skewing of input data, rather than using the conventional brute force hard-router. The architecture is fully scalable and parameterisable. Its real hardware implementation on FPGAs leads to a very compact configuration compared to a hard-router based implementation, with no speed penalty. Moreover, unlike the conventional architecture, our architecture allows the use of multiplier blocks or sub-expression sharing, leading therefore to more compact implementation.

References

- [1] Proakis. J.G., Manolakis .D.G, 'Introduction to Digital Signal Processing', McMillan Publishing, USA, 1989.
- [2] Peter Pirsch, 'Architectures for Digital Signal Processing', John Wiley & Sons, 1999.
- [3] Vetterli M, Kovacevic M, 'Wavelets and Subband Coding' Prentice Hall, New Jersey, USA, 1995
- [4] Smith M.J.T, Eddins S 'Analysis/Synthesis techniques for subband image coding', IEEE Trans On Acoustics, Speech and Signal Processing, 1446-1456, August 1990
- [5] Chakrabarti C, 'A DWT based encoder architecture for symmetrically extended images', Proceedings of the International Symposium on Circuits and Systems, 1999.
- [6] Benkrid A, Benkrid K, Crookes D, 'Design and Implementation of a Novel Architecture for Symmetric FIR filters with Boundary Handling on Xilinx Virtex FPGAs', IEEE Conference on Field-Programmable Technology, FPT'2002, December 16
- [7] <http://www.xilinx.com/partinfo/ds022.pdf>
- [8] Keshab K. Parhi, 'VLSI Digital Signal Processing Systems: Design and Implementation', John Wiley & Sons, 1999

A Self-reconfiguring Platform

Brandon Blodget¹, Philip James-Roxby², Eric Keller²,
Scott McMillan¹, and Prasanna Sundararajan¹

¹ Xilinx, 2100 Logic Drive, San Jose, CA, 95124, USA,

{brandonb, mcmillan, prasanna}@xilinx.com

² Xilinx, 3100 Logic Drive, Longmont, CO, 80503, USA,

{jamespb, keller}@xilinx.com

Abstract. A self-reconfiguring platform is reported that enables an FPGA to dynamically reconfigure itself under the control of an embedded microprocessor. This platform has been implemented on Xilinx Virtex IItm and Virtex II Protm devices. The platform's hardware architecture has been designed to be lightweight. Two APIs (Application Program Interface) are described which abstract the low level configuration interface. The Xilinx Partial Reconfiguration Toolkit (XPART), the higher level of the two APIs, provides methods for reading and modifying select FPGA resources. It also provides support for relocatable partial bitstreams. The presented self-reconfiguring platform enables embedded applications to take advantage of dynamic partial reconfiguration without requiring external circuitry.

1 Introduction

This paper presents a self-reconfiguring platform (*SRP*) for Xilinx Virtex IItm and Virtex II Protm devices[1]. It begins by reviewing the motivation for developing the *SRP*, before presenting the first detailed description of the two core software components, the ICAP API and the Xilinx Partial Reconfiguration Toolkit(XPART). Recent improvements and revisions to the *SRP* hardware architecture are also described in more detail.

Dynamic reconfiguration and self-reconfiguration are two of the more advanced forms of FPGA reconfigurability. Dynamic reconfiguration implies that an active array may be partially reconfigured, while ensuring the correct operation of those active circuits that are not being changed. Self-reconfiguration extends the concept of dynamic reconfigurability. It assumes that specific circuits on the logic array are used to control the reconfiguration of other parts of the FPGA. Clearly the integrity of the control circuits must be guaranteed during reconfiguration, so by definition self-control is a specialized form of dynamic reconfiguration.

Both dynamic reconfiguration and self-reconfiguration rely on an external reconfiguration control interface to boot an FPGA when power is first applied or the device is reset. Once initially configured, self-control requires an internal reconfiguration interface that can be driven by the logic configured on the logic array. On Xilinx Virtex II and Virtex II Pro parts, this interface is called the internal configuration access port (ICAP)[2].

The hardware component of *SRP* is composed of the ICAP, control logic, a small configuration cache, and an embedded processor. The embedded processor can be Xilinx's MicroBlazetm, which is a 32-bit RISC soft microprocessor core[3]. The hardcore

PowerPC on the Virtex II Pro can also be used as the embedded processor. The embedded processor provides intelligent control of device reconfiguration at runtime. The integration of this functionality is especially attractive for embedded systems. This lightweight approach maximizes flexibility while minimizing additional external circuitry.

The software component of *SRP* defines two APIs. The lower level one is the ICAP API. The ICAP API provides access to the configuration cache and controls reading and writing the cache to the device. The higher level API is Xilinx Partial Reconfiguration Toolkit(XPART). XPART is derived from the JBits API work[4]. Like the JBits API it abstracts the bitstream details providing seemingly random access to select FPGA resources. XPART also contains methods for relocating partial bitstreams.

SRP opens up a number of interesting possibilities. Firstly it gives more reconfiguration options to the designer. Adding *SRP* to a system allows an application to get reconfiguration data from any peripheral and partially reconfigure the device. For example if the system has a network connection partial bitstreams could be pulled off the network. Secondly the embedded processor could manipulate the data before reconfiguring the device. This could permit custom encryption or compression of bitstreams. Thirdly the XPART API enables fine grain reconfiguration control over select FPGA resources. This allows tuning of MGTs (multi-gigabit transceiver) , or constant folding achieved by modifying LUTs. Finally it is envisioned that a subsystem like *SRP* could play an important role in an embedded operating system running on a platform FPGA. *SRP* could help the OS manage hardware tasks. It could provide the capability to swap tasks in and out of hardware. It would also allow these tasks to be relocated to different regions on the device[5].

The paper is arranged as follows: Section 2 reviews previous work relating to *SRP*. Section 3 looks at the details of the ICAP and the reconfiguration mechanisms of the Virtex line of FPGAs. This provides the background necessary for an appreciation of the *SRP* hardware and software infrastructure that are described in sections 4 and 5. Sections 5.1 and 5.2 describe the APIs in the software layers. Section 6 presents future work and concludes the paper.

2 Related Work

Dynamic reconfiguration implies that an active array may be partially reconfigured, while ensuring the correct operation of those active circuits that are not being changed. Much research has been done on dynamic reconfiguration which shows it can be used to reduce circuit complexity, increase performance and simplify system design[6].

Self-reconfiguration is a special case of dynamic reconfiguration where the configuration control is hosted within the logic array that is being dynamically reconfigured. The part of the array containing the configuration control remains unmodified throughout execution. A formal definition of self-reconfiguration is presented in [7]. There are several desirable features of such an arrangement. Firstly the control logic is as close to the logic array as possible, thus minimizing the latencies associated with accessing the configuration port. Secondly, fewer discrete devices are required, reducing the overall system complexity[8].

For a device to support self-reconfiguration it must be dynamically reconfigurable. A second desirable, but not required, characteristic is the device provides internal access to the configuration port. This way, the configuration stream does not have to exit the chip and use board level resources to gain access to the configuration port. The Xilinx XC6200 family of devices first provided both of these features, newer devices from Xilinx such as the Virtex II and Virtex II Pro families are both dynamically reconfigurable and provide access to the configuration port to internal logic.

The first references to self-reconfiguration using FPGAs in the literature was in [9] where a small amount of static logic is added to a reconfigurable device in order to produce a self-reconfiguring processor. The Flexible URISC[10] defined an abstract model of virtual circuitry that had self-configuring capability. A pattern-matching algorithm was used to investigate the viability of systems that exhibit self-control of reconfiguration management[8]. A number of applications mapped using self-reconfiguration have been shown to improve performance over existing approaches[9][11].

Xilinx application note 662 describes a method for using self-reconfiguration on a Virtex II Pro device to update MGT (multi-gigabit transceiver) parameters[12]. These attributes must be modified to optimize the MGT signal transmission prior to and after a system has been deployed in the field. The work presented in this paper extends the framework described in Xilinx application note 662 to enable a general purpose self-reconfiguring platform.

The XPART software layer is derived from Xilinx's JBits API work. XPART provides a lightweight, minimal set of JBits API features implemented in the C language. XPART also provides some basic functionality for supporting relocatable modules. A relocatable module is a partial bitstream that can be relocated to multiple places on the FPGA. This functionality has also been called Dynamic Hardware Plugins(DHP) and was used for implementing multiple networking applications in hardware for high-performance programmable routers[13]. The methods that XPART provide are very similar to PARBIT (PARtial Bitfile Transformer)[14]. The tool JBitsDiff also provides the ability to create relocatable modules directly from bitstreams[15]. The added benefit XPART provides is these functions can run on an embedded processor.

3 Virtex I/II/II Pro Configuration Architecture

SRAM based FPGAs are configured by loading application specific data into configuration memory. All Virtextm series devices (Virtex I, Virtex II and Virtex II Pro) have their configuration memory segmented into *frames*. These devices are partially reconfigurable and a frame is the smallest unit of reconfiguration. There are multiple frames per CLB column. For example Virtex devices have 48 frames per CLB column. Frames are one bit wide and differ in length depending on the number of CLB rows in the device. For example an XC2V40 has 832 bits per frame and an XC2V1000 has 3392 bits per frame.

Virtex II and Virtex II Pro devices have an internal reconfiguration access port (ICAP) which can be controlled by internal FPGA logic. The ICAP interface is a subset of the SelectMAPtm interface. Figure 1 shows a comparison between the two interfaces. The ICAP interface has fewer signals than the SelectMAP interface because it does not have to do full configurations and it does not have to support different configuration modes.

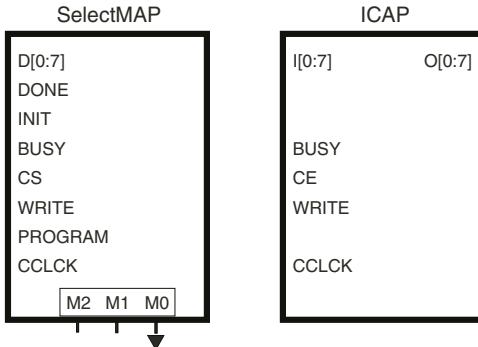


Fig. 1. SelectMap vs ICAP

It also differs in that the SelectMAP bi-directional D port is split into an I data port and an O data port. One other item to note is the functionality provided by the ICAP CE pin is equivalent to the SelectMAP CS pin.

The eight bit data I port on the ICAP allows faster reconfiguration than serial modes of reconfiguration. The maximum frequency that the SelectMAP and ICAP interfaces can be clocked at without checking the $BUSY$ signal is 66MHz[16].

Virtex I/II/II Pro FPGAs require a pad frame be added to the end of the configuration data. This pad frame flushes out the reconfiguration pipeline. Therefore to write one frame to the device it is necessary to clock in two frames, the data frame plus a pad frame. Thus the minimal time to reconfigure over ICAP a single XC2V40 frame at 66MHz is 3.2us. It would take 13us to reconfigure a single XC2V1000 frame at 66MHz.

4 The SRP Hardware Architecture

SRP's hardware component is composed of the ICAP, some control logic, a small configuration cache, and an embedded processor. These peripherals communicate over the CoreConnecttm Open Peripheral Bus (OPB)[17]. Figure 2A shows a block diagram of the current hardware subsystem implementation.

In this implementation a 32 bit register is used to interface to the ICAP port. Figure 2A shows how this register is mapped to the ICAP signals. The control logic for reading and writing data to the ICAP is implemented in a low level software driver. This driver defines methods for reading and writing to the ICAP interface register. There are also methods for reading and writing blocks of data to the ICAP. This methodology is similar to the XVPI register JBits SDK used to access the SelectMAP interface[18].

The BlockRAM (BRAM) shown in Figure 2A is used to cache configuration data. To keep the SRP hardware as lightweight as possible only one BRAM is used. One Virtex II BRAM can store 16K bits of data. Since the largest Virtex II Pro device, the XC2VP125, has a frame length of 11K bits, one BRAM can easily store a whole frame of even this largest Virtex II Pro device.

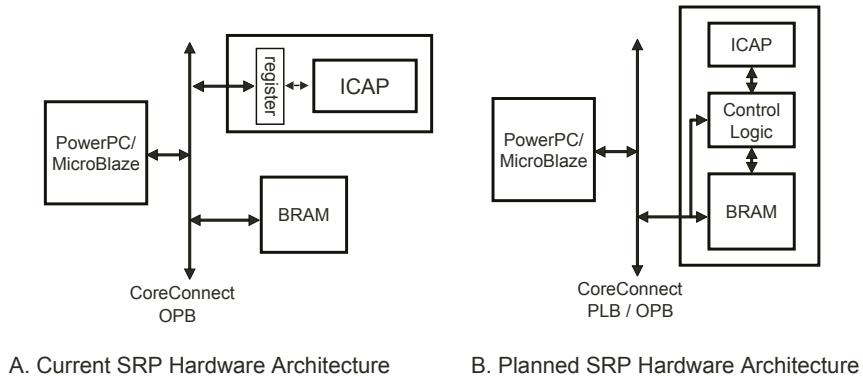


Fig. 2. Current and Planned SRP Hardware Architecture

This hardware system has been implemented on both Virtex II and Virtex II Pro devices. On the Virtex II device the MicroBlaze soft processor was used. On Virtex II Pro the embedded PowerPC was targeted. The Xilinx Embedded Developer Kit (EDK) and Xilinx 5.1i ISE tools were used to build these hardware platforms.

The next generation of the hardware system will provide much better performance. Figure 2B shows this system. This will be achieved via a few changes. Firstly the ICAP control logic will move from being in software (hardware drivers) to being implemented directly in hardware. This move also means there will be less communication over the system bus. Secondly, the configuration cache BRAM will be moved inside the ICAP control peripheral. This will allow the dual ported nature of the BRAM to be exploited. One port of the BRAM will be exposed to the system bus. The other port will be accessed by the ICAP control logic. This way the ICAP control logic will not take up system bus cycles to transfer data to and from the configuration cache. The embedded processor will still be able to access the configuration cache BRAM over the system bus. Thirdly, the ICAP control peripheral will be a master peripheral. This way the peripheral will be able to fetch configuration data directly from external memory without requiring the processor be involved. Lastly, we plan to implement the peripheral as a Xilinx IPIF (IP Interface) peripheral. IPIF peripherals can interface to both the OPB and the faster PLB (Processor Local Bus).

5 The SRP Software Architecture

The software components of *SRP* are designed in layers. Figure 3 shows the different software layers. The software layers are divided into hardware dependent and hardware independent parts. This division is enabled by the ICAP API. This API defines methods for transferring data between the configuration cache implemented in BRAM and the active configuration memory. It also provides methods for accessing the configuration cache.

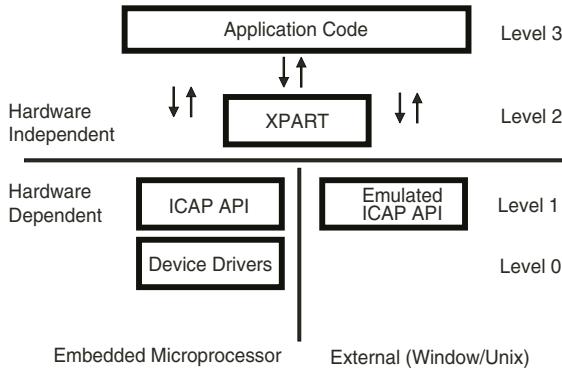


Fig. 3. SRP Software Layers

We have created two implementations of the ICAP API. The first uses the SRP hardware described in the previous section to enable embedded applications to readback or modify the active configuration of the FPGA. The second implementation emulates the active configuration store and the configuration cache entirely in software. This emulated version of the ICAP API can then be compiled for a Windows or Unix workstation.

The Xilinx Partial Reconfiguration Toolkit (XPART) is built on top of the ICAP API. Thus XPART is hardware independent and can be compiled for an embedded system, or for a Windows or Unix system. This portability also applies to applications that use XPART and/or the ICAP API.

One advantage of this portability is one can do a degree of debug on a standard workstation before moving to the target embedded platform. It is also possible to use XPART on a workstation, manipulate partial bitstreams, then write them to the computer's hard drive instead of the physical FPGA device.

5.1 The ICAP API

The ICAP API defines methods for accessing configuration logic through the ICAP port. The main methods move data between the configuration cache (BRAM) and the active configuration memory (the device). Other methods allow the processor to read and write to the configuration cache. Finally the setDevice() method indicates which device is being targeted. This method allows the designer to retarget their application to a different device by changing one line in the code. All Virtex II and Virtex II Pro family members are supported. Table 1 gives an outline of the methods in the ICAP API.

5.2 XPART – Xilinx Partial Reconfiguration Toolkit

The Xilinx Partial Reconfiguration Toolkit (XPART) has been built on top of the ICAP API. The purpose of this toolkit is to allow embedded processors to modify resources and relocate modules. Currently this toolkit has four methods. Table 2 gives an overview of these methods.

Table 1. The ICAP API

Routines	Description
setDevice()	Indicates the target device.
storageBufferWrite()	Writes data to the configuration storage buffer
storageBufferRead()	Reads data from the configuration storage buffer
deviceWrite()	Transfers specified number of bytes from storage buffer to ICAP_IN
deviceRead()	Transfers specified number of bytes from ICAP_OUT to storage buffer
deviceAbort()	Aborts the current operation
deviceReadFrame()	Reads one or more frames from the device into the storage buffer
deviceWriteFrame()	Writes one or more frames to the device from the storage buffer
setConfiguration()	Loads a configuration from memory
getConfiguration()	Writes current configuration to memory

Table 2. XPART Methods

Routines	Description
getCLBBits()	Reads back the state of a selected CLB resource.
setCLBBits()	Reconfigures the state of a selected CLB resource.
setCLBModule()	Place the module at a particular location on the device
copyCLBModule()	Given a bounding box copy it to a new location on the device

XPART provides functions for on the fly resource modification. This is done through the getCLBBits() and setCLBBits() methods. These methods abstract the bitstream and provides seemingly random access to selected resources. Currently XPART has the capability to read or modify all Virtex II CLB logic and routing resource. It currently does not have access to IOB, DCM, MGT, BRAM or other non CLB resources.

The strategy employed is to do a read/modify/write of configuration data. The following lists the steps that setCLBBits() would perform to update a resource like a LUT:

1. Calculate the target frame
2. Find LUT bits in target frame
3. read target frame from device and put in storage buffer using deviceReadFrame()
4. Modify the LUT bits in the storage buffer using writeStorageBuffer()
5. Reconfigure the device with new LUT bits using deviceWriteFrame()

The memory elements that define the content of a LUT in Virtex II are all located in one frame. This is not true with all resources. Some resources have their configuration bits scattered across multiple frames. The setCLBBits() and getCLBBits() methods are optimized so they will not have to read or write the same frame twice during the same call. Figure 4 shows sample code that updates the content of a LUT.

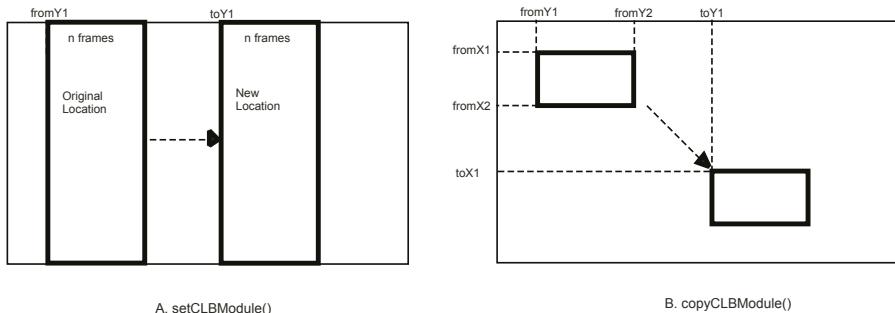
XPART provides some basic functionality for supporting relocatable modules. A relocatable module is a partial bitstream that can be relocated to multiple places on the FPGA. XPART provides two methods for dealing with relocatable modules. The two methods are setCLBModule() and copyCLBModule(). The setCLBModule() method works on regular partial bitstreams that contains information about all of the rows in the

```
#include <XPART.h>
#include <LUT.h> /* Bitstream resource library for LUTs */

int main(int argc, char *args[]) {
    char* value;
    int error, i, row, col, slice;
    setDevice(XC2VP7); // Set the device type

    ... [initialize row,col,slice and value to desired values] ...

    error = setCLBBits(row, col, LUT.RES[slice][LE_F_LUT], value, 16);
    return error;
} /* end main() */
```

Fig. 4. Code to update LUT contents**Fig. 5.** Illustration of Module Methods

included frames. It works by modifying the header of the partial reconfiguration packet. Figure 5A illustrates the setCLBModule() command.

The copyCLBModule() function copies any sized rectangular region of configuration memory and writes it to another location. The copied region contains just a subset of the rows in a frame. This allows the designer to define dynamic regions that have static regions above or below it. Figure 5B shows an illustration of copyCLBModule().

The copyModule() function employees a read/modify/write strategy like the resource modification functions. This technique enables changing select bits in a frame and leaving the others bits to their current configured state. The Virtex II provides glitchless configuration logic, meaning if a bit stays the same between two configurations no glitch will occur. For example a frame may be modified that contains a routing resource. If the value of the bits controlling that routing resource remain the same between the old and new configurations, no glitch will occur on that routing resource.

6 Future Work and Conclusions

Several extensions and applications of the system are in progress. As noted earlier we are working on a more efficient implementation of our *SRP* hardware architecture. In the current implementation the ICAP control logic is implemented in software via device drivers. This implementation requires 10ms to modify a LUT value on an XC2V1000 device with MicroBlaze running at 50MHz. The next version of the *SRP* hardware should allow us to clock configuration data into and out of the device at the maximum no handshake frequency of 66MHz. Modifying one LUT on the XC2V1000 will take 13us for the read, negligible time for the modify, and 13us for the write for a total of approximately 26us. This is over two orders of magnitude faster than the existing *SRP* hardware architecture.

We are currently using *SRP* to develop a reconfiguration controller for a high I/O reconfigurable crossbar switch[19]. The original reconfiguration controller was implemented using external logic and memory. The data for the reconfiguration controller was generated offline using JBits SDK. *SRP* should allow us to implement the reconfigurable crossbar and controller on the same chip.

In conclusion, we have described a self-reconfigurable platform. *SRP* is an intelligent subsystem for lightweight reconfiguration of Xilinx Virtex II and Virtex II Pro FPGAs in embedded systems. The system enables self-reconfiguration under software control within a single FPGA. *SRP* has a layered hardware and software architecture that permits a variety of different interfaces to maximize flexibility and ease-of-use.

References

1. Blodget, B., McMillan, S., Lysaght, P.: A lightweight approach for embedded reconfiguration of fpgas. In: Design, Automation and Test in Europe (DATE03), IEEE (2003) 399–400
2. Xilinx, Inc.: Xilinx 5.1i Libraries Guide. (2002)
3. : Xilinx web site. <http://www.xilinx.com/ipcenter/processorcentral/microblaze> (2003)
4. Sundararajan, P., Guccione, S.A., Levi, D.: JBits: Java based interface for reconfigurable computing. In: 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies (MAPLD99), Laurel, MD (1999)
5. Nollet, V., Coene, P., Verkest, D., Vernalde, S., Lauwereins, R.: Designing an operating system for a heterogeneous reconfigurable soc. In: RAW'03 workshop. (2003) (Accepted)
6. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. In: ACM Computing Surveys, Vol 34, No. 2. (2002) 171–210
7. Sidhu, R., Prasanna, V.K.: Efficient metacomputation using self-reconfiguration. In: Field Programmable Logic and Applications (FPL02), Springer (2002) 698–709
8. McGregor, G., Lysaght, P.: Self controlling dynamic reconfiguration: A case study. In: Field Programmable Logic and Applications (FPL99), Springer (1999) 144–154
9. French, P.C., Taylor, R.W.: A self-reconfiguring processor. In: IEEE Symposium on File-Programmable Custom Computing Machines (FCCM93), Napa Valley, California (1993) 50–59
10. Donlin, A.: Self modifying circuitry - a platform for tractable virtual circuitry. In: Field Programmable Logic and Applications (FPL98), Springer (1998) 199–208
11. Sidhu, R.P.S., Mei, A., Prasanna, V.K.: Genetic programming using self-reconfigurable fpgas. In: Field Programmable Logic and Applications (FPL99), Springer (1999)

12. Eck, V., Kalra, P., LeBlanc, R., McManus, J.: In-circuit partial reconfiguration of RocketIO attributes. Xilinx Application Note XAPP662, version 1.0, Xilinx, Inc. (2003)
13. David E. Taylor, Jonathan S. Turner, J.W.L.: Dynamic hardware plugins (DHP): exploiting reconfigurable hardware for high-performance programmable routers. In: Open architectures and Network Programming Proceedings, IEEE (2001) 25–34
14. Horta, E., Lockwood, J.W.: PARBIT: a tool to transform bitfiles to implement partial reconfiguration of field programmable gate arrays (FPGAs). Technical Report WUCS-01-13, Washington University in Saint Louis, Department of Computer Science (July 6, 2001)
15. James-Roxby, P., Guccione, S.A.: Automated extraction of run-time parameterisable cores from programmable device configurations. In: IEEE Symposium on File-Programmable Custom Computing Machines (FCCM00), Napa Valley, California, IEEE Computer Society (2000) 153–161
16. Xilinx, Inc.: Virtex-II Platform FPGA User Guide. (2002)
17. : IBM web site. <http://www.chips.ibm.com/products/coreconnect> (2003)
18. Sundararajan, P., Guccione, S.A.: XVPI: A portable hardware/software interface for virtex. In: Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212, SPIE – The International Society for Optical Engineering (2000) 90–95
19. Young, S., Alfke, P., Fewer, C., McMillan, S., Blodget, B., Levi, D.: A high i/o reconfigurable crossbar switch. In: IEEE Symposium on File-Programmable Custom Computing Machines (FCCM03), Napa Valley, California, IEEE Computer Society (2003)
20. Carmichael, C.: Virtex FPGA series configuration and readback. Xilinx Application Note XAPP138, version 1.1, Xilinx, Inc. (1999)
21. Lim, D., Peattie, M.: Two flows for partial reconfiguration: Module based or small bit manipulations. Xilinx Application Note XAPP290, version 1.0, Xilinx, Inc. (2002)
22. McMillan, S., Guccione, S.A.: Partial run-time reconfiguration using JRTR. In: Field Programmable Logic and Applications (FPL00), Springer (2000) 352–360
23. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using fpgas. In: IEEE Symposium on File-Programmable Custom Computing Machines (FCCM01), Rohnert Park, California (2001)
24. Horta, E.L., Lockwood, J.W., Taylor, D.E., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In: Design Automation Conference (DAC), New Orleans, LA (2002)

Heuristics for Online Scheduling Real-Time Tasks to Partially Reconfigurable Devices*

Christoph Steiger, Herbert Walder, and Marco Platzner

Swiss Federal Institute of Technology (ETH) Zurich, Switzerland,
platzner@tik.ee.ethz.ch

Abstract. Partially reconfigurable devices allow to configure and execute tasks in a true multitasking manner. The main characteristics of mapping tasks to such devices is the strong nexus between scheduling and placement. In this paper, we formulate a new online real-time scheduling problem and present two heuristics, the *horizon* and the *stuffing* technique, to tackle it. Simulation experiments evaluate the performance and the runtime efficiency of the schedulers. Finally, we discuss our prototyping work toward an integration of scheduling and placement into an operating system for reconfigurable devices.

1 Introduction

Todays reconfigurable devices provide millions of gates capacity and partial reconfiguration. This allows for true multitasking, i.e., configuring and executing tasks without affecting other, currently running tasks. Multitasking of dynamic task sets can lead to complex allocation situations which clearly asks for well-defined system services that help to efficiently operate the system. Such a set of system services forms a *reconfigurable operating system* [1] [2]. This paper deals with one aspect of such an operating system (OS), the problem of online scheduling hard real-time tasks.

Formally, a task T_i is modeled as rectangular area of reconfigurable logic blocks given by its width and height, $w_i \times h_i$. Tasks arrive at arbitrary times a_i , require execution times e_i , and carry deadlines d_i , $d_i \geq a_i + e_i$. The reconfigurable device is also modeled as rectangular area $W \times H$ of logic blocks. We focus on non-preemptive systems – once a task is loaded onto the device it runs to completion.

The complexity for mapping tasks to such devices depends heavily on the used *area model*. We use two different area models, a 1D and a 2D model as shown in Figure 1. In the simpler 1D area model, tasks can be allocated along the horizontal device dimension; the vertical dimension is fixed. The 1D area model suffers badly from two types of fragmentation. The first type is the area wasted when a task does not utilize the full device height. The second type occurs when the remaining free area is split into several unconnected vertical

* This work was supported by the Swiss National Science Foundation (SNF) under grant number 2100-59274.99.

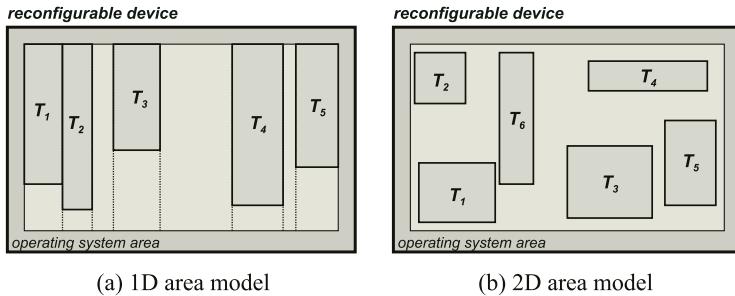


Fig. 1. Reconfigurable resource models

stripes. Fragmentation can prevent the placement of a further task although a sufficient amount of free area exists. The more complex 2D area model allows to allocate tasks anywhere on the 2D reconfigurable surface and suffers less from fragmentation.

The main contribution of this paper is the development of online hard real-time scheduling heuristics that work for both the 1D and the 2D area model. The limitations of the models and related work are discussed in Section 2. Section 3 states the online scheduling problem and presents the two heuristics. An experimental evaluation is done in Section 4. Section 5 shows our work toward a prototype implementation and, finally, Section 6 summarizes the paper.

2 Limitations and Related Work

Our task and device models are consistent with related work in the field [1] [3] [4] [5] [6]. However, we also have to discuss the limitations when it comes to practical realization on currently available technology. The main abstraction is that tasks are modeled as relocatable rectangles. While the latest design tools allow to constrain tasks to rectangular areas, the relocatability raises questions concerning the i) device homogeneity, ii) task communication and timing, and iii) partial reconfigurability.

We assume surface uniformity which is in contrast with modern FPGAs that contain special resources such as block memories and embedded multipliers. However, a reconfigurable OS takes many of these resources (e.g., block RAM) away from the user task area. Tasks must use predefined communication channels to access such special resources [7][2]. Further, the algorithms in this paper can easily be extended to handle additional placement constraints for tasks, e.g., to relocate tasks at different levels of granularity or even to place tasks at fixed positions. The basic problems and approaches will not change, but the resulting performance.

Arbitrarily relocated tasks that communicate with each other and with I/O devices would require online routing and delay estimation of their external signals, neither of which is sufficiently supported by current tools. The state-of-the-art in reconfigurable OS prototypes [7] [2] overcomes this problem by using a

slightly different area model that partitions the reconfigurable surface into fixed-size blocks. These OSs provide predefined communication interfaces to tasks and asynchronous intertask communication. The same technique can be applied to our 1D area model. For the 2D model, communication is an unresolved issue. Related work mostly assumes that sufficient resources for communication are available [4].

The partial reconfiguration capabilities of the Xilinx Virtex family, which reconfigures a device in vertical chip-spanning columns, fits perfectly the 1D area model. While the implementation of a somewhat limited 2D area model on the same technology seems to be within reach, ensuring the integrity of non-reconfigured device areas during task reconfiguration is tricky.

In summary, given current technology the 1D area model is realistic whereas the 2D model faces unresolved issues. Most of the related work on 2D models targets the (meanwhile withdrawn) FPGA series Xilinx XC6200 that is reconfigurable on the logic block level and has a publicly available bitstream architecture. Requirements for future devices supporting the 2D model include block-based reconfiguration and a built-in communication network that is not affected by user logic reconfigurations. As we will show in this paper, the 2D model has great advantages over the 1D model in terms of scheduling performance. For these reasons, we believe that it is worthwhile to investigate and develop algorithms for both the 1D and 2D area models.

3 Scheduling Real-Time Tasks

3.1 The Online Scheduling Problem

The online scheduler tries to find a placement and a starting time for a newly arrived task such that its deadline is met. In the 1D area model a *placement* for a task T_i is given by the x coordinate of the left-most task cell, x_i , with $x_i + w_i \leq W$. The *starting time* for T_i is denoted by s_i . The main characteristics of scheduling to dynamically reconfigurable devices is that a scheduled task has to satisfy intertwined time and placement constraints:

Definition 1 (Scheduled Task). A scheduled task T_i is a task with a placement x_i and a starting time s_i such that:

- i) $[(x_i + w_i) \leq x_j] \vee [(s_i + e_i) \leq s_j] \vee [x_i \geq (x_j + w_j)] \vee [s_i \geq (s_j + e_j)]$
 $\forall T_j \in \mathcal{T}, T_j \neq T_i$ (scheduled tasks must not overlap in space and time,
 \mathcal{T} denotes the set of scheduled tasks)
- ii) $s_i + e_i \leq d_i$ (deadline must be met)

We consider an online *hard real-time* system that runs an acceptance test for each arriving task. A task passing this test is guaranteed to meet its deadline. A task failing the test is rejected by the scheduler in order to preserve the schedulability of the currently guaranteed task set. The scheduling goal is to minimize the number of rejected tasks. Accept/reject mechanisms are typically adopted in dynamic real-time systems [8] and assume that the scheduler's environment

can react properly on a task rejection, e.g., by migrating the task to a different computing resource.

Our scheduling problem shares some similarity with orthogonal placement problems, so-called strip packing problems. Strip packing tries to place a set of two dimensional boxes into a vertical strip of width W by minimizing the total height of the strip. Translated to our scheduling problem, the width of the strip corresponds to the device width and the vertical dimension corresponds to time. The offline strip packing problem is NP-hard [9] and many approximation algorithms have been developed for it. There are also some online algorithms with known competitive ratios. However, to the best of our knowledge, there is no published online algorithm with a proven competitive ratio for the problem described in this paper which differs in following characteristics: First, our optimization goal is to minimize the number of rejected tasks, based on an acceptance test that is run at task arrival. Second, time proceeds as tasks are arriving. We cannot schedule tasks beyond the current timeline, i.e., into the past. Finally, tasks must not be rotated or otherwise modified.

The simplest online method is to check whether a newly arrived task finds an immediate placement. If there is none, the task is rejected. This crude technique needs to know only about the current allocation situation but will show a low performance. We include this method as a reference point in our experimentation. Sophisticated online methods increase the acceptance ratio by *planning*, i.e., looking into the future. We may delay starting a task for its laxity (until $s_{i-\text{latest}} = d_i - e_i$) and still meet its deadline. The time interval $[a_i, s_{i-\text{latest}}]$ is the planning period for a task. In the following sections we discuss two online methods, the *horizon* technique and *stuffing* technique. These planning methods are runtime efficient as they do not reschedule previously guaranteed tasks.

3.2 The Horizon Technique

The horizon technique implements scheduling and placement by maintaining two lists, the *scheduling horizon* and the *reservation list*. The scheduling horizon is a set of intervals that fully partition the spatial resource dimension. Each horizon interval is written as $[x_1, x_2]@t_r$, where $[x_1, x_2]$ denotes the interval in x -dimension and t_r gives the last release time for the corresponding reconfigurable resources. The set of intervals is sorted according to increasing release times.

Figure 2 shows an example for a device of width $W = 10$. At time $t = 2$, two tasks (T_1, T_2) are running on the device and further four tasks (T_3, T_4, T_5, T_6) are scheduled. The resulting scheduling horizon is given by the four intervals shown in Figure 3a) and indicated as dotted lines in Figure 2.

When a new task arrives, the scheduler walks through the list of intervals and checks whether the task can be appended to the horizon. When a horizon interval $[x_1, x_2] @ t_r$ is hit that is large enough to accommodate the task, the task is scheduled to placement x_1 and starting time t_r and the planning process stops. Otherwise, the scheduler tries to merge the interval with already released and adjacent horizon intervals to form a larger interval. If this larger interval does not fit either, the next interval in the horizon is considered. Should several

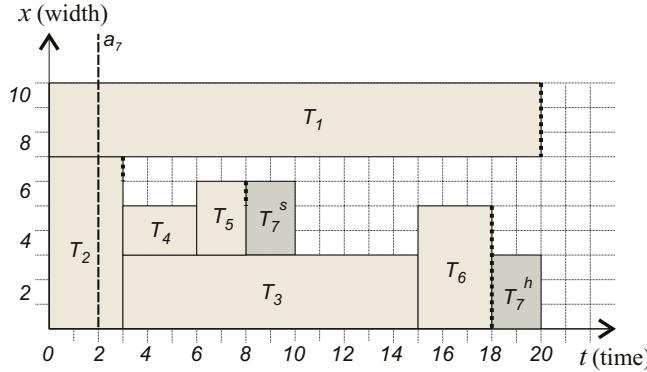


Fig. 2. 1D allocation with scheduling horizon (*dotted lines*) before accepting T_7 and placements for T_7 in the horizon (T_7^h) and stuffing methods (T_7^s)

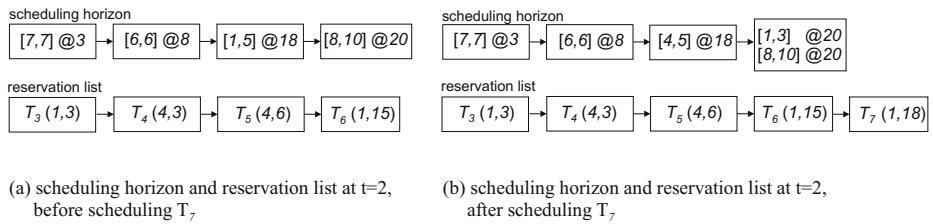


Fig. 3. Horizon method: scheduling horizon and reservation list

intervals fit at some point, the scheduler applies the best-fit rule to select the interval with the smallest width.

In the example of Figure 2, a new task T_7 arrives with $(a_7, w_7, e_7, d_7) = (2, 3, 2, 20)$ which gives a planning period of $[a_7, (d_7 - e_7)] = [2, 18]$. The first horizon interval to be checked is $[7, 7] @ 3$, which is too small to fit T_7 . The scheduler proceeds to $[6, 6] @ 8$, which is too small again. Then, these two intervals are temporarily merged to $[6, 7] @ 8$ which is still insufficient. The next interval is $[1, 5] @ 18$ which allows to schedule T_7 to $(x_7, s_7) = (1, 18)$.

The *reservation list* stores all scheduled but not yet executing tasks. The list entries are denoted as $T_i(x_i, s_i)$ and hold the placement and starting time. The reservation list is sorted in order of increasing starting times. The horizon technique ensures that new tasks are only inserted into the reservation list when they do not overlap in time or space with other tasks in the list. The scheduler is activated whenever a new task arrives, a running task terminates, or a scheduled task is to be started. On each event, the horizon is updated. For the example of Figure 2, the reservation list at time $t = 2$ is displayed in Figure 3a). Figure 3b) shows the updated horizon and reservation lists after scheduling and accepting T_7 at time $t = 2$.

The central point of the horizon scheduler is that tasks can only be *appended* to the horizon. Particularly, it is not possible to schedule tasks before the hori-

zon, as the procedure maintains no knowledge about the time-varying allocation between the current time and the horizon. The advantage of this technique is that maintaining the horizon is simple compared to maintaining the complete future.

3.3 The Stuffing Technique

The stuffing technique schedules tasks into arbitrary free rectangles that will exist in the future. The implementation uses again two lists, the *free space list* and the reservation list. The free space list is a set of intervals $[x_1, x_2]$ that denote currently unused resource rectangles with height H . The free spaces are ordered according to their x -coordinates.

On arrival of T_i , we assume n tasks are executing on the device and m tasks are waiting in the reservation list. Two types of events occur during the planning period of T_i . First, $n', n' \leq n$ tasks terminate which generates new free areas. Second, $m', m' \leq m$ previously guaranteed tasks are started which reduces the free area. When a new task T_i arrives, the scheduler starts walking through the task's planning period, simulating all future allocations of the device by mimicking task terminations and placements together with the underlying free space management. On arrival of T_i and the termination of the n' placed tasks, the placer is called to find a feasible interval. If one is found, the scheduler accepts and adds a new reservation for T_i , and planning stops. If no sufficient interval is found, the scheduler proceeds until $s_{i-\text{latest}}$. During the planning process, the scheduler merges adjacent free spaces. Should several intervals fit, the best-fit rule is applied.

For T_7 in the example of Figure 2 planning proceeds until time $t = 8$, where the free space list contains the interval $[4, 7]$ which fits T_7 . The stuffing technique leads to improved performance over the horizon method. The drawback is the increased complexity as we need to simulate future task terminations and planned starts to identify free space. Both schedulers use two lists. They differ, however, in the planning process for an arriving task. While the horizon method updates the horizon list only once per task arrival, the stuffing method updates the free space list for $n' + m'$ times.

3.4 Extension to the 2D Area Model

The concepts and methods discussed so far extend naturally to the 2D area model. A 2D placement is given by the coordinates of the task's bottom-left cell, (x_i, y_i) , with $x_i + w_i \leq W$ and $y_i + h_i \leq H$, and the resulting scheduling problem relates to 3D strip packing problems [6]. The main difference compared to the 1D model lies in the placer. Instead of keeping lists of intervals, we need to manage lists of rectangles. The 2D scheduling horizon is a set of rectangles that fully partition the device rectangle, together with the last release time for each rectangle. The 2D stuffing method maintains the free space as a list of free rectangles. The placers we use to implement the 2D horizon and stuffing

methods are based on the approach presented by Bazargan et al. [5] and have been described in [10].

3.5 Runtime Efficiency

The runtime efficiency of the schedulers depends largely on the underlying placer implementation. Both our 1D free space list and the 2D Bazargan placer [5] keep $\mathcal{O}(n)$ free areas for n currently placed tasks. Thus, the asymptotic worst-case runtime complexity is the same for 1D and 2D area models. The reference scheduler that considers only immediate placements has a complexity of $\mathcal{O}(n)$. It can be shown that the horizon scheduler's complexity is given by $\mathcal{O}(n + m)$, where m denotes the number of guaranteed but not yet scheduled tasks. The complexity of the stuffing method amounts to $\mathcal{O}(n^2m)$.

4 Evaluation

4.1 Simulation Setup

To evaluate the online schedulers, we have devised a discrete-time simulation framework. Tasks are randomly generated. We have simulated a wide range of parameter settings. The results presented in this section are typical and are based on following settings: The simulated device consists of 96×64 reconfigurable units (Xilinx XCV1000). The task areas are uniformly distributed in $[50, 500]$ reconfigurable units; task execution times are uniformly distributed in $[5, 100]$ time units. The aspect ratios are distributed between $[5, 0.2]$. We have defined three task classes A, B, C , with laxities uniformly distributed in $[1, 50]$, $[50, 100]$, and $[100, 200]$ time units, respectively. Runtime measurements have been conducted on a Pentium-III 1000MHz, taking advantage of Visual C++'s profiling facilities. All the simulations presented below use 95 % confidence level with an error range of ± 3 percent.

4.2 Results

Scheduling to the 1D Area Model: Figure 4a) compares the performance of the reference scheduler with the horizon and stuffing techniques. The aspect ratios are distributed such that 50 % of the tasks are taller than wide (standing tasks) and 50% are wider than tall (lying tasks). The reference scheduler does not plan into the future. Hence, its performance is independent of the laxity class. As expected, the stuffing method performs better than the horizon method which in turn is superior to the reference scheduler. The differences between the methods grow with increasing laxity because longer planning periods provide more opportunity for scheduling. For laxity class C, the horizon scheduler outperforms the reference by 14.46 %; the stuffing scheduler outperforms the reference by 23.56 %.

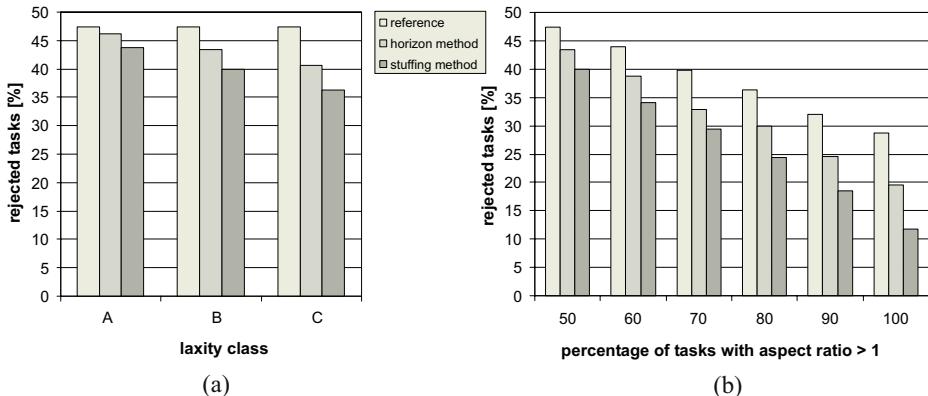


Fig. 4. Performance of the scheduling heuristics for the 1D area model

Figure 4b) shows the number of rejected tasks as function of the aspect ratio, using laxity class B. For the 1D area model standing tasks are clearly preferable. The generation of such tasks can be facilitated by providing placement and routing constraints. In Figure 4b), a percentage of tasks with aspect ratio > 1 of 100 % denotes an all standing task set. The results demonstrate that all schedulers benefit from standing tasks. The differences again grow with the aspect ratio. For 100 % standing tasks, the horizon method results in a performance improvement over the reference of 32%, the stuffing method even in 58.84 %.

Comparison of 1D and 2D Area Models: Figure 5a) compares the performance between the 1D and 2D area models for the stuffing technique. The aspect ratios are distributed such that 50 % of the tasks are standing. The results clearly show the superiority of the 2D area model. For laxity class A, the performance

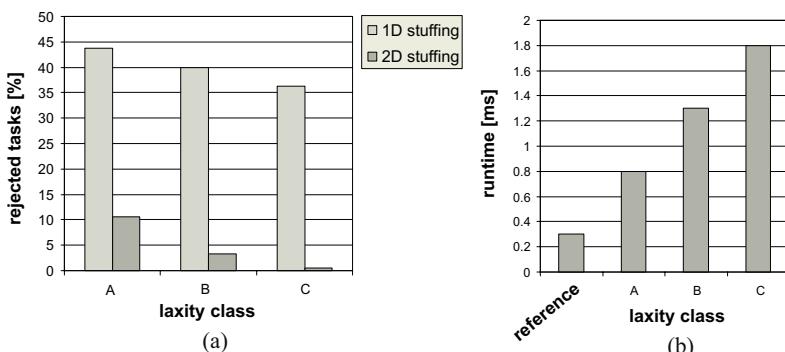


Fig. 5. (a) performance of the 1D and 2D stuffing methods; (b) runtimes for scheduling one task for the reference scheduler and the 2D stuffing method

improvement in going from 1D to 2D is 75.57 %; for laxity class C it is 98.35 %. An interesting result (not shown in the figures) is that the performance for the 2D model depends only weakly on the aspect ratio distribution. Due to the 2D resource management, both mixed and standing task sets are handled well.

Runtime Efficiency: Figure 5b) presents the average runtime required to schedule one task for the 2D stuffing method, the most complex of all implemented techniques. The runtime increases with the length of the planning period. However, with 1.8 ms at most the absolute values are small. Assuming a time unit to be 10 ms, which gives us tasks running from 50 ms to 1 s, the runtime overheads for the online scheduler and for reconfiguration (in the order of a few ms) are negligible. This justifies our simulation setup which neglects these overheads.

5 Toward a Reconfigurable OS Prototype

Figure 6(a) shows the 1D area model we use in our current reconfigurable OS prototype. The reconfigurable surface splits into an OS area and a user area which is partitioned into a number of fixed-size task slots. Tasks connect to predefined interfaces and communicate via FIFO buffers in the OS area. The hardware implementation is done using Xilinx Modular Design [11]. The prototype runs an embedded networking application and has been described elsewhere in more detail [2]. The application is packet-based audio streaming of encoded audio data (12kHz, 16bit, mono) with an optional AES decoding. A receiver task checks incoming Ethernet packets and extracts the payload to FIFOs. Then, AES decryption and audio decoding tasks are started to decrypt, decode and stream the audio samples. The task deadlines depend on the minimal packet inter-arrival time and the FIFO lengths. Our prototype implements rather small FIFOs and guarantees to handle packets with a minimal inter-arrival time of 20 ms.

Our prototype proves the feasibility of multitasking on partially reconfigurable devices and its applicability to real-time embedded systems. However,

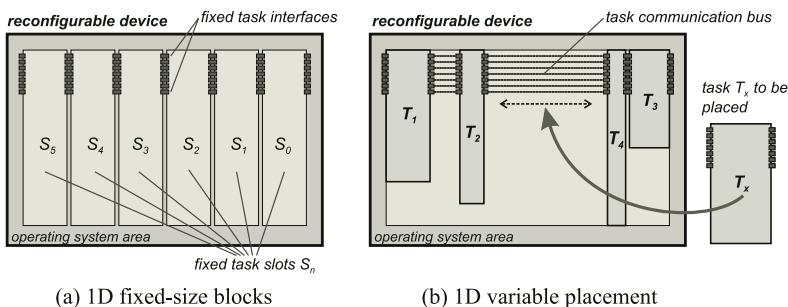


Fig. 6. Prototype OS structure

the 1D block-partitioned area model differs from the model used in this paper which requires task placements to arbitrary horizontal positions as shown in Figure 6(b). While task relocation is solved, prototyping a communication infrastructure for variably-positioned tasks, as proposed by [3], remains to be done.

6 Conclusion and Further Work

We discussed the problem of online scheduling hard real-time tasks to partially reconfigurable devices and developed two online scheduling heuristics for 1D and 2D area models. Simulations show that the heuristics are effective in reducing the number of rejected tasks. While the 1D schedulers depend on the tasks' aspect ratios, the 2D schedulers do not. In all cases, the 2D model dramatically outperforms the 1D model. Finally, the scheduler runtimes are so small that the more complex stuffing technique will be the method of choice for most application scenarios.

References

1. G. Brebner. A Virtual Hardware Operating System for the Xilinx XC6200. In *Int'l Workshop on Field-Programmable Logic and Applications (FPL)*, pages 327–336, 1996.
2. H. Walder and M. Platzner. Reconfigurable Hardware Operating Systems: From Concepts to Realizations. In *Int'l Conf. on Engineering of Reconfigurable Systems and Architectures (ERSA)*, 2003.
3. G. Brebner and O. Diessel. Chip-Based Reconfigurable Task Management. In *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, pages 182–191, 2001.
4. O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt. Dynamic scheduling of tasks on partially reconfigurable FPGAs. 147(3):181–188, 2000.
5. K. Bazargan, R. Kastner, and M. Sarrafzadeh. Fast Template Placement for Reconfigurable Computing Systems. 17(1):68–83, 2000.
6. S. Fekete, E. Köhler, and J. Teich. Optimal FPGA Module Placement with Temporal Precedence Constraints. In *Design Automation and Test in Europe (DATE)*, pages 658–665, 2001.
7. T. Marescaux, A. Bartic, Verkest D., S. Vernalde, and R. Lauwereins. Interconnection Networks Enable Fine-Grain Dynamic Multi-tasking on FPGAs. In *Int'l Conf. on Field-Programmable Logic and Applications (FPL)*, pages 795–805, 2002.
8. G.C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer, 2000.
9. B.S. Baker, E.G. Coffman, and R.L. Rivest. Orthogonal packings in two dimensions. *SIAM Journal on Computing*, (9):846–855, 1980.
10. H. Walder, C. Steiger, and M. Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Reconfigurable Architectures Workshop (RAW)*, 2003.
11. D. Lim and M. Peattie. Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. XAPP 290, Xilinx, 2002.

Run-Time Minimization of Reconfiguration Overhead in Dynamically Reconfigurable Systems

Javier Resano¹, Daniel Mozos¹, Diederik Verkest^{2,3,4},
Serge Vernalde², and Francky Catthoor^{2,4}

¹ Dept. Arquitectura de Computadores, Universidad Complutense de Madrid, Spain.
javier1@fdi.ucm.es, mozos@dacya.ucm.es

² IMEC vzw, Kapeldreef 75, 3001, Leuven, Belgium
{Verkest, Vernalde, Catthoor}@imec.be

³ Professor at Vrije Universiteit Brussel, Belgium

⁴ Professor at Katholieke Universiteit Leuven., Belgium.

Abstract. Dynamically Reconfigurable Hardware (DRHW) can take advantage of its reconfiguration capability to adapt at run-time its performance and its energy consumption. However, due to the lack of programming support for dynamic task placement on these platforms, little previous work has been presented studying these run-time performance/power trade-offs. To cope with the task placement problem we have adopted an interconnection-network-based DRHW model with specific support for reallocating tasks at run-time. On top of it, we have applied an emerging task concurrency management (TCM) methodology previously applied to multiprocessor platforms. We have identified that the reconfiguration overhead can drastically affect both the system performance and energy consumption. Hence, we have developed two new modules for the TCM run-time scheduler that minimize these effects. The first module reuses previously loaded configurations, whereas the second minimizes the impact of the reconfiguration latency by applying a configuration prefetching technique. With these techniques reconfiguration overhead is reduced by a factor of 4.

1 Introduction and Related Work

Dynamically Reconfigurable Hardware (DRHW), that allows partial reconfiguration at run-time, represents a powerful and flexible way to deal with the dynamism of current multimedia applications. However, compared with application specific integrated circuits (ASICs), DRHW systems are less power efficient. Since power consumption is one of the most important design concerns, this problem must be addressed at every possible level; thus, we propose to use a task concurrency management (TCM) approach, specially designed to deal with current dynamic multimedia applications, which attempts to reduce the energy consumption at task-level.

Other research groups have addressed the power consumption of DRHW, proposing a technique to allocate configurations [1], optimizing the data allocation both for improving the execution time and the energy consumption [2], presenting an energy-

conscious architectural exploration [3], introducing a methodology to decrease the voltage requirements [4], or carrying out a static scheduling [5]. However, all these approaches are applied at design-time, so they cannot tackle efficiently dynamic applications, whereas our approach selects at run-time among different power/performance trade-offs. Hence, we can achieve larger energy savings since our approach prevents the use of static mappings based on worst-case conditions.

The rest of the paper is organized as follows: sections 2 and 3 provide a brief overview of the ICN-based DRHW model and the TCM methodology. Sect. 4 discusses the problem of the reconfiguration overhead. Sect. 5 presents the two modules developed to tackle this problem. Sect. 6 introduces some energy considerations. Sect. 7 analyses the experimental results and Sect. 8 presents the conclusions.

2 ICN-Based DRHW Model

The Interconnection-Network (ICN) DRHW model [6] partitions an FPGA platform into an array of identical tiles. The tiles are interconnected using a packet-switched ICN implemented using the FPGA fabric. At run-time, tasks are assigned to these tiles using partial dynamic reconfiguration. Communication between the tasks is achieved by sending messages over the ICN using a fixed network interface implemented inside each tile. As explained in [6], this approach avoids the huge Place & Route overhead that would be incurred when directly interconnecting tasks. In the latter case, when a new task with a different interface is mapped on a tile, the whole FPGA needs to be rerouted in order to interconnect the interface of the new task to the other tasks. Also the communication interfaces contain some Operating System (OS) support like storage space and routing tables, which allow run-time migration of tasks from one tile to another tile or even from an FPGA tile to an embedded processor.

Applying the ICN model to a DRHW platform greatly simplifies the dynamic task allocation problem, providing a software-like approach, where tasks can be assigned to HW resources in the same way that threads are assigned to processors. Thus, this model enables the use of the emerging Matador TCM methodology [7].

3 Matador Task Concurrency Management Methodology

The matador TCM methodology [7, 9] proposes a task scheduling technique for heterogeneous multiprocessor embedded systems. The different steps of the methodology are presented in figure 1. It starts from an application specification, composed of several tasks, called Thread Frames (TF). These TFs are dynamically created and deleted and can even be non-deterministically triggered. Nevertheless, inside each TF only deterministic and limited dynamic behavior is allowed.

The whole application is represented using the grey-box model, which combines Control–Data Flow Graphs (CDFG) with another model (called MTG*) specifically designed to tackle dynamic non-deterministic behavior. CDFGs are used to model the

TFs, whereas MTG* models the inter-TF behavior. Each node of the CDFG is called a Thread Node (TN). TNs are the atomic scheduling units.

TCM accomplishes the scheduling in two phases. The first phase generates at design-time a set of near-optimal scheduling solutions for each TF called a Pareto curve. Each solution represents a schedule and an assignment of the TNs over the available processing elements with a different performance/energy tradeoff. Whereas this first step accomplishes separately the design-space exploration of each TF, the second phase tackles their run-time behavior, selecting at run-time the most suitable Pareto point for each TF. The goal of the methodology is to minimize the energy consumption while meeting the timing constraints of the applications (typically, highly-dynamic multimedia applications). TCM has been successfully applied to schedule several current multimedia applications on multiprocessor systems [8, 9].

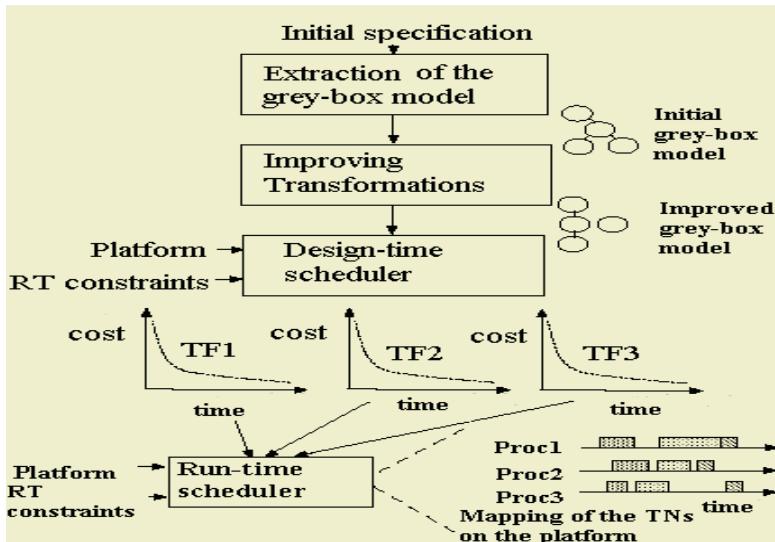


Fig. 1. Matador Task Concurrency Management Methodology

4 Dynamic Reconfiguration Overhead

Fig. 2 represents the Pareto curve obtained using the existing TCM design scheduler for a system with a SA-1110 processor coupled with a Virtex2 v6000 FPGA. The TF corresponds to a motion JPEG application. In the figure optimal, and non-optimal schedules are depicted, but only the optimal ones are included in the Pareto curve. This curve is one of the inputs for the run-time scheduler. Thus, the scheduler can select at run-time between different energy/performance trade-offs. For instance, if there is not a tight timing constraint, it will select the least energy consuming solution, whereas, if the timing constraint changes (for instance if a new task starts), the run-time scheduler will look for a faster solution which meets the new constraint (with the consequent energy penalty).

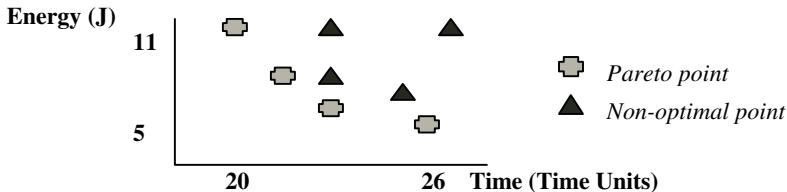


Fig. 2. Pareto curve of the motion JPEG

In this example, assuming that 80% of the time the most energy-efficient solution can be selected TCM can achieve a 47% energy saving (on average $11*0.2+5*0.8=5.78$ J/iteration) than a static worst-case approach while providing the same peak performance (the worst-case consumes 11 J/iteration). Hence, the TCM approach can drastically reduce the overall energy consumption while still meeting hard real-time constraints. However, current TCM scheduling tools neglect the task context-switching overhead, since for many existing processors it is very low. However, the overhead due to the load of a configuration on a FPGA is much greater than the aforementioned context-switching overhead, e.g. reconfiguring a tile of our ICN-based FPGA consumes 4 ms (assuming that a tile occupies one tenth of a XC2V6000 FPGA and the configuration frequency is 50 MHz). The impact of this overhead on the system performance greatly depends on the granularity of the TNs. However, for current multimedia applications, TN average execution time is likely to be in the order of magnitude of milliseconds (a motion JPEG application must decode a frames in 40 ms). If this is the case, the reconfiguration overhead can drastically affect both the performance and the energy consumption of the system, moving the Pareto curve to a more energy and time consuming area. Moreover, in many cases, the shape of the Pareto curve changes when this overhead is added. Therefore, if it is not included the TCM schedulers cannot take the optimal decisions. To address this problem, we have added two new modules to the TCM schedulers, namely: configuration reuse, and configuration prefetch. These modules are not only used to accurately estimate the reconfiguration overhead, but also to minimize it. Configuration reuse attempts to reuse previously loaded configurations. Thus, if a TF is being executed periodically, at the beginning of each iteration the scheduler checks if the TNs loaded in the previous iteration are still there, if so, they are reused preventing unnecessary reconfigurations.

Configuration prefetch [10] attempts to overlap the configurations of a TN with the computation of other TNs in order to hide the configuration latency. A very simple example is presented in figure 3, where 4 TNs must be loaded and executed on an FPGA with 3 tiles. Since current FPGAs do not support multiple simultaneous reconfigurations, configurations must be load sequentially. Without prefetching, the best on-demand schedule result is depicted in 3(a) because TN3 cannot be loaded before the loading of TN2 is finished and TN4 must wait until the execution of both TN2 and TN3 is finished. However, applying prefetch (3b), the loads of TN2 and TN4 overlap with the execution of TN1 and TN3. Hence, only the loads of TN1 and TN3 penalize the system execution time.

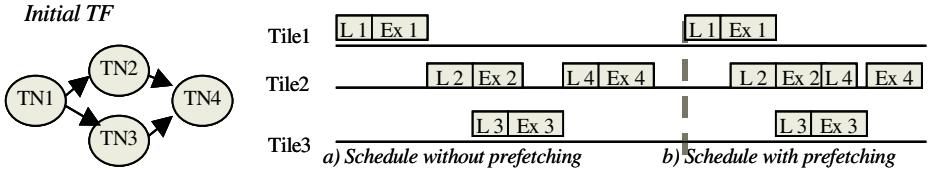


Fig. 3. Configuration prefetch on a platform with 3 FPGA tiles. **L:** loading, **Ex:** executing

Clearly, this powerful technique can lead to significant execution time savings. Unfortunately, deciding the best order to load the configurations is a NP-complete problem. Moreover, in order to apply this technique in conjunction with the configuration reuse technique, the schedule of the reconfiguration must be established at run-time, since the number of configurations that must be loaded depends on the number of configurations that can be reused and typically this number will differ from one execution to another if the system behavior is non-deterministic. Therefore, we need to introduce these techniques in the TCM run-time scheduler while attempting to keep the resulting overhead to a minimum.

5 Run-Time Configuration Prefetch and Reuse

The run-time scheduler receives as an input a set of Pareto curve and selects a point on them for each TF according to timing and energy considerations. Currently, at run-time we never alter the scheduling order imposed by the design-time scheduler, thus, in order to check if a previous configuration can be reused we just look for the first TN assigned to every FPGA tile in the given schedule (we use the term *initial* for those TNs). Since all tiles created by the use of the ICN in the FPGA are identical, the actual tile in which a TN is executed is irrelevant. When the run-time scheduler needs to execute a given TF, it will first check whether the *initial* TNs are still present in any of the FPGA tiles. If so, they are reused (avoiding the costly reconfiguration). Otherwise the TNs are assigned to an available tile. For instance in the example of Fig. 3, the run-time scheduler will initially check if the configurations of TN1, TN2, and TN3 are still loaded in the FPGA, in this case TN1, and TN3 will be reused. The run-time scheduler will not check if it can reuse the configuration of TN4 since it knows that even when this TN could remain loaded in the FPGA it will be overwritten by TN2.

The run-time configuration reuse algorithm is presented in figure 4.a, its complexity is $O(NT*I)$, where NT is the number of tiles and I the number of *initial* TNs. Typically I and NT are small numbers and the overhead of this module is negligible.

Once the run-time scheduler knows which TN configurations can be reused, it has to decide when the remaining configurations are going to be loaded. The pseudo-code of the heuristic developed for this step is presented in figure 4.b. It starts from a given design-time schedule that does not include the reconfiguration overhead and updates it according to the number of reconfigurations needed. Then, it schedules the reconfigurations by applying prefetching to try to minimize the execution time.

```

a) for (i=0; i < Number of initial TNs; i++) {
    while (not found) and(j < Number of FPGA tiles) {
        found = look_for_reuse(i, j);
        if(found){assign the TN i to actual tile j; j++;}}
    Assign the remaining TNs to actual tiles;

b) if there are TN configurations to load{
    schedule the TNs that do not need to be loaded
    for (i=0; i < Number of configurations to load; i++) {
        select&schedule a configuration;
        schedule its successors that do not need to
        be loaded on to the FPGA;}}

```

Fig. 4. a) Configuration reuse pseudo-code. **b)** Configuration prefetch pseudo-code

We have developed a simple heuristic based on an enhanced list-scheduling. It starts by scheduling all the nodes that do not need to be configured on the FPGA, i.e. those assigned to SW or those assigned to the FPGA whose configuration can be reused. After scheduling a TN, the heuristic attempts to schedule its successors. If they do not need to be loaded on to the FPGA the process continues, otherwise, a reconfiguration request is stored in a list. When it is impossible to continue scheduling TNs, one of the requests is selected according to the following criteria:

- If, at a given time t , there is just one configuration ready for loading this configuration is selected. A configuration is ready for loading if the previous TN assigned to the same FPGA tile has already finished its execution.
- Otherwise, when several configurations are ready, the configuration with the highest weight is selected.

The weight of a configuration is assigned at design-time and represents the maximum time-distance from a TN to the end of the TF. This weight is computed carrying out an ALAP scheduling in the TF. Those configurations corresponding to nodes in the critical path of the TF will be heavier than the other ones. The complexity of this module is $O(N*C)$ where N is the number of TNs and C the number of configurations to load.

Since these two techniques must be applied at run-time, we are very concerned about their execution time. This time depends on the number of TN, FPGA tiles and configurations that must be loaded. The configuration-prefetch module is much more time-demanding than the configuration-reuse module. However we believe that the overhead is acceptable for a run-time scheduler. For instance a TF with 20 nodes, 4 FPGA tiles and 13 configurations to be loaded is scheduled in less than 2.5 μ s using a Pentium-II running at 350MHz. This number has been obtained starting from a C++ initial code, and disabling all the compiler optimizations. We expect that this time will be significantly reduced when starting from C code and applying optimization compiler techniques. But even if is not reduced, it will be compensated by the prefetching time-savings if the heuristic hides the latency of one reconfiguration at least once every 1600 executions (assuming that the reconfiguration overhead is 4ms). In our experiments the heuristic has exhibited much better results than this minimum requirement. Although we believe that the overhead due to the prefetch module is acceptable, it is not always worthwhile to execute it. For instance, when all the configurations of a design are already loaded in the FPGA there will be no gains applying

prefetch. Thus, in this case the module will not be executed, preventing unnecessary computation. There is another common case where this run-time computation can be substituted by design-time computation; this is when all the configurations must be loaded. This case happens at least once (when the TF is loaded for the first time), and it can be very frequent if there is a great amount of TNs competing for the resources. Hence, we can save some run-time computation analyzing this case at design-time. However, this last optimization duplicates the storage space needed for a Pareto point, since now for each point in the Pareto curve two schedules are stored. Hence, if the system has a very limited storage space, this optimization should be disabled. The run-time scheduling will follow the steps depicted in figure 5.

```

for each Pareto point to evaluate {
    apply configuration reuse
    If there are not configurations to load {
        read energy and execution time of scheduling 1}
    Else if all the configurations must be loaded{
        read energy and execution time of scheduling 2 }
    Else {
        actualize scheduling 1 applying configuration prefetching} }
```

Fig. 5. Run-time evaluation process pseudo-code. Schedules 1 and 2 are computed at design time. Both of them share the same allocation of the TNs on the system processing elements, but they have different execution time and energy consumption since schedule 1 assumes that all the TNs assigned to the FPGA have been previously loaded, whereas schedule 2 includes the reconfiguration overhead of these TNs

6 Energy Considerations

TCM is an energy-aware scheduling technique. Thus, these two new modules should not only reduce the execution time but also the energy consumption. Clearly, the configuration reuse technique can generate energy savings, since loading a configuration to an FPGA involves both an execution time and an energy overhead. According to [5], when a FPGA is frequently reconfigured, up to 50% of the FPGA energy consumption is due to the reconfiguration circuitry. Hence, reducing the number of reconfigurations is a powerful way to achieve energy savings.

Configuration prefetch can also indirectly lead to energy savings. If we assume that loading a configuration on to a tile has a constant overhead E_c , and a given schedule involves 4 reconfigurations, the energy overhead due to the reconfigurations will be $4*E_c$ independently of the order of the loads. However, configuration prefetch reduces the execution time of the TFs and the run-time scheduler can take advantage of this extra time to select a slower and less energy consuming Pareto point. Fig. 6 illustrates this idea with one TF.

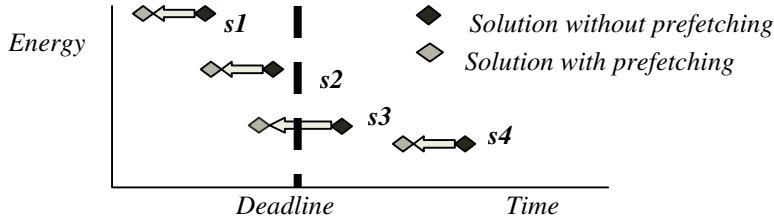


Fig. 6. Configuration prefetch technique for energy savings. Before applying prefetching, s2 was the solution that consumes less energy meeting the timing constraint. After applying prefetching the time saved can be used to select a more energy efficient solution (s3)

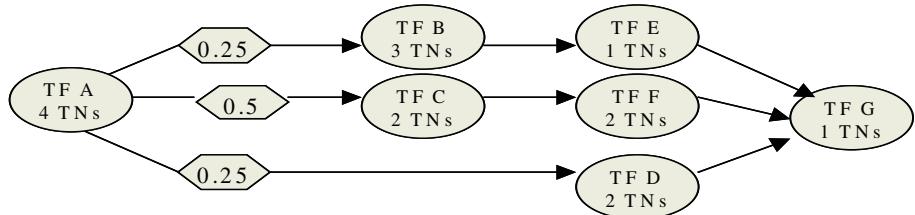


Fig. 7. Set of TFs generated using the Task Graph For Free (TGFF) system [11]. Inside each node its name (a letter from A to G), and the number of TNs assigned to HW are depicted. When there are different execution paths, each one is tagged with the probability of being selected. We assume that this set of TF is executed periodically

7 Results and Analysis

The efficiency of the reuse module depends on the number of FPGA tiles, the TNs assigned to these tiles and on the run time events. Figure 8 presents the average reuse percentage for the set of TFs depicted in figure 7 for different number of FPGA tiles. This example contains 15 TNs assigned to HW, although not all of them are executed every iteration. The reuse percentage is significant even with just 5 FPGA tiles (29%). Most of the reuse is due to the TFs A and G, since they are executed every iteration. For instance, when there are 8 tiles, 48 % of the configurations are reused and 30% are due to TF A and TF G (5 TNs). When there are 17 tiles, only the 71 % of configurations are reused, this is not an optimal result since as long as there are more tiles than TNs it should be possible to reuse 100% of the configurations. However, we are just applying a local reuse algorithm to each TF instead of applying a global one to the whole set of TFs. We have adopted a local policy because it creates almost no run-time overhead, and can be easily applied to systems with non-deterministic behavior. However, we are currently analyzing how a more complex approach could lead to higher percentages of reuse with an affordable overhead.

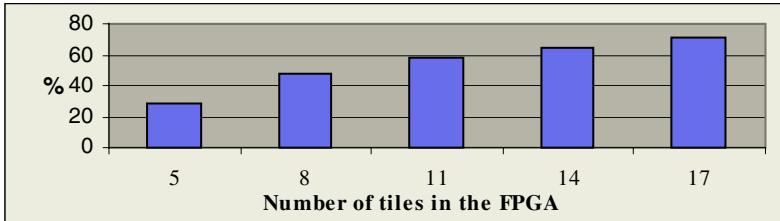


Fig. 8. Percentage of configurations reused vs Number of tiles in the FPGA

We have performed two experiments to analyze the prefetch module performance. Firstly, we have studied how good the schedules computed by our heuristic are. To this end we have generated 100 pseudo-random TFs using the TGFF system, and we have scheduled the configuration loads both with our heuristic, and with a branch&bound (b&b) algorithm that accomplishes a full design space exploration. This experiment shows that the b&b scheduler finds better solutions (on average 10% better) than our heuristic. However, for TFs with 20 TNs, it needs 800 times more computational time to find these solutions. Hence, our heuristic generates almost optimal schedules, in an affordable time.

The second experiment presents the time-savings achieved due to the prefetch module for three multimedia applications (table 1). It assumes that the whole application is executed on an ICN-like FPGA, with 4 tiles that can be reconfigured in 4 ms, which is currently the fastest possible speed. However, even with this fast reconfiguration assumption, it is remarkable how the reconfiguration overhead affects the system performance, increasing up to the 35% the overall execution time. This overhead is drastically reduced (in average a factor of 4) when the prefetch module is applied.

8 Conclusions

The ICN-based DRHW model provides the dynamic task reallocation support needed to apply a TCM approach to heterogeneous systems with DRHW resources. With this model both the DRHW and the SW resources can be handled in the same way, simplifying the tasks of the TCM schedulers. We have identified that the DRHW reconfiguration overhead significantly decreases the system performance and increases the energy consumption. Hence, we have developed two new modules for the TCM run-time scheduler (namely configuration reuse and configuration prefetch) that can reduce this problem while improving at run-time the accuracy of the execution time and energy consumption estimations.

Configuration reuse attempts to reuse previously loaded configurations, leading to energy and execution time savings. Its efficiency depends on the number of FPGA tiles, and TNs assigned to these tiles. However, the module exhibits significant reuse percentage even with 15 TNs competing for 5 FPGA tiles. Configuration prefetch schedules the reconfigurations to minimize the overall execution time. The results show that it reduces the execution time reconfiguration overhead by a factor of 4.

Table 1. Reconfiguration overhead with and without applying the prefetch module for three actual multimedia applications. **TNs** is the number of TNs in the application, **Init T** is the execution time of the application assuming that no reconfigurations are required. **Prefetch** and **Overhead** are the percentage of the execution time increased due to the reconfiguration overhead, when all the configurations are loaded in the FPGA, with and without the prefetch module. The first two applications are different implementations of a JPEG decoder, the third application computes the Hough transform of a given image in order to look for certain patterns

	TNs	Init T	Overhead	Prefetch
JPEG decoder	4	81ms	+25%	+5%
Enhanced JPEG decoder	8	57ms	+35%	+7%
Pattern Recognition	6	94ms	+17%	+4%

Acknowledgements

The authors would like to acknowledge all our colleagues from the T-Recs and Matador groups at IMEC, for all their comments, help and support. This work has been partially supported by TIC 2002-00160 and the Marie Curie Host Fellowship HPMT-CT-2000-00031.

References

1. R. Maestre et al., “Configuration Management in Multi-Context Reconfigurable Systems for Simultaneous Performance and Power Optimizations”, ISSS’00, pp 107-113, 2000.
2. M. Sánchez-Elez et al “Low-Energy Data Management for Different On-Chip Memory Levels in Multi-Context Reconfigurable Architectures”. DATE’03, pp. 36-41, 2003.
3. M. Wan et al. “Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System”, Journal of VLSI Signal Processing 28, pp. 47-61, 2001.
4. A. D. Garcia et al., “Reducing the power Consumption in FPGAs with keeping a high Performance Level”, WVLSI00, pp 47-52, 2002.
5. Li Shang et al., “Hw/Sw Co-synthesis of Low Power Real-Time Distributed Embedded Systems with Dynamically Reconfigurable FPGAs”, ASP-DAC’02, pp. 345-360, 2002.
6. T. Marescaux et al., “Interconnection Network enable Fine-Grain Dynamic Multi-Tasking on FPGAs”, FPL’02, pp. 795-805, 2002.
7. P. Yang et al., “Energy-Aware Runtime Scheduling for Embedded-Multiprocessors SOCs”, IEEE Journal on Design&Test of Computers, pp. 46-58, 2001.
8. P. Marchal et al., “Matador: an Exploration Environment for System-Design”, Journal of Circuits, Systems and Computers, Vol. 11, No. 5, pp. 503-535, 2002.
9. P. Yang et al, “Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia systems”, ISSS’02, pp. 112-119, 2002.
10. Z. Li and S. Hauck, “Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation” Int’l Symp. FPGAs, pp. 187-195, 2002.
11. R.P. Dick et al, “TGFF: Task Graphs for Free”, Int’l Workshop HW/SW Codesign, pp. 97-101, 1998.

Networks on Chip as Hardware Components of an OS for Reconfigurable Systems*

T. Marescaux¹, J-Y. Mignolet¹, A. Bartic¹, W. Moffat¹,
D. Verkest^{1,2,3}, S. Vernalde, and R. Lauwereins^{1,3}

¹ IMEC vzw, Kapeldreef 75, B-3001 Leuven, Belgium,
marescau@imec.be

² also Professor at Vrije Universiteit Brussel

³ also Professor at Katholieke Universiteit Leuven

Abstract. In complex reconfigurable SoCs, the dynamism of applications requires an efficient management of the platform. To allow run-time allocation of resources, operating systems and reconfigurable SoC platforms should be developed together. The operating system requires hardware support from the platform to abstract the reconfigurable resources and to provide an efficient communication layer. This paper presents our work on interconnection networks which are used as hardware support for the operating system. We show how multiple networks interface to the reconfigurable resources, allow dynamic task relocation and extend OS-control to the platform. An FPGA implementation of these networks supports the concepts we describe.

1 Introduction

Adding reconfigurable hardware resources to an Instruction Set Processor (ISP) provides an interesting trade-off between flexibility and performance in mobile terminals. Because these terminals are dynamic and run multiple applications, design-time task allocation is clearly not an option. Additional dynamism may arise from changing bandwidth availability in networked applications and from intra-application computation variation as in MPEG-4. Tasks must therefore be mapped at run-time on the resources. We need an operating system to handle the tasks and their communications in an efficient and fair way at run-time.

In addition to supporting all the functionality of traditional OSes for ISPs, an Operating System for Reconfigurable Systems (OS4RS) has to be extended to manage the available reconfigurable hardware resources. Hardware support for an OS targeting reconfigurable SoCs is required for two reasons. On the one hand, we have to avoid inefficiencies inherent to software management of critical parts of the system, such as inter-task communication. On the other hand, the ISP needs physical extensions to access, in an unified way, the new functions of all components of a reconfigurable SoC. Interconnection networks are the solution we advocate as hardware support for the operating system.

* Part of this research has been funded by the European Commission through the IST-AMDREL project (IST-2001-34379) and by Xilinx Labs, Xilinx Inc. R&D group.

In this paper we use a system composed of an ISP running the software part of the OS4RS, connected to an FPGA containing a set of blocks, called tiles, that can be individually reconfigured to run a hardware task, also called an IP-block. The concepts developed herein are not restricted to FPGAs and are meant to be extended to other reconfigurable SoC architectures as well.

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 lists the requirements of operating systems for reconfigurable SoCs and introduces a novel Network on Chip (NoC) architecture able to fulfill these requirements. Section 4 describes our implementation of this NoC architecture to give efficient OS4RS hardware support. Section 5 discusses our implementation results. Finally, conclusions are drawn in Section 6.

2 Related Work

Dally advises in [7] the usage of NoCs [10] in SoCs as a replacement for top-level wiring because they outperform it in terms of structure, performance and modularity. Because we target reconfigurable SoCs we have an extra-reason to use NoCs: they allow dynamic multitasking [2] and provide HW support to an OS4RS.

Simmler addresses in [6] “multitasking” on FPGAs. However, in his system only one task is running on the FPGA at a time. To support “multitasking” he sees the need for task preemption, which is done by readback of the configuration bitstream. The state of the task is extracted by performing the difference of the read bitstream with the original one, which has the disadvantages of being architecture dependent and adding run-time overhead. We address the need for high-level task state extraction and real dynamic heterogeneous multitasking.

In [5], Rijpkema discusses the integration of best-effort and guaranteed-throughput services in a combined router. Such a combined system could be an interesting alternative to our physically separated data and control networks (Sect. 3.3).

The following two papers are tightly related to the work presented here. In [1], Nollet explains the design of the SW part of an OS4RS by extending a Real-Time OS with functions to manage the reconfigurable SoC platform we present in this paper. He introduces a two-level task scheduling in reconfigurable SoCs. The top-level scheduler dispatches tasks to schedulers local to their respective processors (HW tiles or ISP). Local schedulers order in time the tasks assigned to them. Task relocation (Sect. 3.1) is controlled in SW by the top-level scheduler.

Finally, Mignolet presents in [3] the design environment that allows development of applications featuring tasks relocatable on heterogeneous processors. A common HW/SW behavior, required for heterogeneous relocation is obtained by using a unified HW/SW design language such as OCAPI-XL [8]. OCAPI-XL allows automatic generation of HW and SW versions of a task with an equivalent internal state representation. Introduction of switch points is thus possible and allows a high level abstraction of task state information.

3 Multiple NoCs Are Required for OS4RS HW Support

This section first lists the requirements of an OS4RS in terms of hardware support. It then recalls how a single NoC enabled us to partially support an OS4RS and demonstrate dynamic multitasking on FPGAs in [2]. A proposal for complete OS4RS HW support is discussed in the last sub-section.

3.1 OS4RS Requirements in Terms of HW Support

In a heterogeneous reconfigurable platform, traditional tasks of operating systems are getting more complex. The following paragraph enumerates typical functions of the OS and explains why hardware support is required when adding reconfigurable hardware computing elements to an ISP.

Task creation/deletion: This is clearly the role of an operating system. In addition to the traditional steps for task setup in an operating system, we need to partially configure the hardware and to put it in an initial state. OS access to the reconfiguration mechanism of the hardware is therefore required.

Dynamic heterogeneous task relocation: Heterogeneous task relocation is a problem that appears when dealing with the flexible heterogeneous systems, that we target (ISP + reconfigurable hardware). The problem is allowing the operating system to seamlessly migrate a task from hardware to software (or vice-versa) at run-time¹ [1]. This involves the transfer of internal state of the task (contents of internal registers and memories) from HW to SW (or vice-versa).

Inter-task communication: Inter-task communication is traditionally supported by the operating system. A straightforward solution would be to pass all communications (HW to HW as well as HW to SW) through the OS running on the ISP. On a heterogeneous system, this solution clearly lacks efficiency, since the ISP would spend most of its time copying data from one location to another. Hardware support for intra-task data transfers, under control of the OS, is a better solution [2].

Debug ability: Debugging is an important issue when working with hardware/software systems. In addition to normal SW debug, the operating system should provide support to debug hardware tasks. This support, in terms of clock stepping, exception generation and exception handling is local to the HW tile and cannot be implemented inside the ISP running the OS. Specific hardware support is thus required.

Observability: To keep track of the behavior of the hardware tasks, in terms of usage of communication resources and of security, the operating system requires access to various parts of the SoC. It is inefficient for the central ISP to monitor the usage of communication resources and check whether the IPs are not creating security problems by inappropriate usage of the

¹ HW to HW relocation may also be required to optimize platform resource allocation and keep communications local within an application.

platform. A hardware block that performs this tracking and provides the OS with communication statistics and signals security exceptions is therefore essential.

3.2 Single NoC Allows Dynamic Multitasking on FPGAs, but Has Limitations

In [2] we demonstrated that using a single NoC enables dynamic multitasking on FPGAs. Separating communication from computation enables task creation/deletion by partial reconfiguration. The NoC solves inter-task communication by implementing a HW message-passing layer. It also partially solves the task relocation issue by allowing dynamic task migration thanks to run-time modification of the Destination Look-up Tables (Sect. 4.1) located in the network interface component (NIC)². These concepts have been implemented in the T-ReCS Gecko demonstrator [3, 9].

As explained in [1] dynamic task relocation requires preemption of the task and the transfer of its state information (contents of its internal registers and memories) to the OS. This state information is then used to initialize the relocated task on a different computation resource (another HW tile or a software thread on the ISP [3]) to smoothly continue the application.

Experimentation on our first setup showed some limitations in the dynamic task migration mechanism. During the task-state transfer, the OS has to ensure that pending messages, stored in the network and its interfaces, are redirected in-order to the computation resource the task has been relocated to.

This process requires synchronization of communication and is not guaranteed to work on our first platform [2]. Indeed, OS Operation And Management (OAM) communication and application data communication are logically distinguished on the NoC by using different tags in the message header. Because application-data can congest the packet-switched NoC, there is no guarantee that OS OAM messages, such as those ensuring the communication synchronization during task relocation, arrive timely.

To support general dynamic task relocation, we have to enhance our system described in [2] to allow the OS to synchronize communications within an application. An interesting approach is to physically separate OS communication from application communications by means of separate NoCs and is discussed in the following section. Additional extensions are required to provide full HW support to the OS4RS as defined in Section 3.1. We need mechanisms to retrieve/restore state information from a task, to control communication load, handle exceptions and provide security and debug support. These extensions are discussed in Sections 4.1 and 4.2.

² This acronym overloads Network Interface Card because our NIC serves the similar role of abstracting a high-level processor from the low level communication of the network.

3.3 Reconfigurable Hardware Multitasking Requires Three Types of Communication

On the reconfigurable platform we presented in [2], the FPGA executes a task per reconfigurable tile and is under the control of an operating system running on the ISP. The OS can create tasks both in hardware and software [1, 3]. For such a system, as Section 3.2 explains, there are two distinct types of communication: OS OAM data and application data. Furthermore, reconfigurable systems have a third logical communication channel to transmit the configuration bitstreams to the hardware tasks.

Each tile in our reconfigurable SoC has therefore three types of communication: reconfiguration data, OS OAM data and application data. Because application data requires high bandwidth whereas OS OAM data needs low latency, we have decided to first implement each communication type on a separate network to efficiently interface the tiles to the OS running on the ISP: a reconfiguration network, a data network and a control network (Fig. 1). The services implemented on these three networks compose the HW support for the OS4RS. In addition to efficiency, a clean logical separation of the three types of communications in three communication paths, ensures independence of application and OS. The OS does not need to care about the contents of the messages carried on the data network and an application designer does not need to take into account OS OAM interactions.

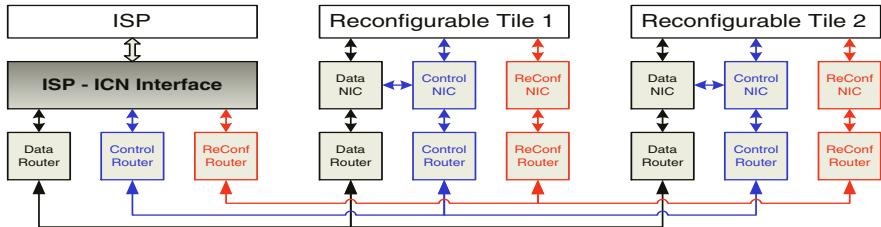


Fig. 1. Three NoCs are required: reconfiguration network, a data network and a control network.

4 Implementation of a Novel NoCs Architecture Providing HW Support to an OS4RS

This section explains how each of the NoCs introduced in section 3.3 plays its role as HW support for an OS4RS and gives the implementation details.

4.1 Application Data Network

By application data, we mean the data transferred from one task to another inside an application. Tasks communicate through message passing. These mes-

sages are sent through the Data Network³ (DN) if the sender and/or the receiver are in a HW tile. A similar message passing mechanism is used for two software tasks residing in the ISP [2]. For performance reasons, application data circulates on the NoC independently of the OS. Nevertheless, the DN must provide hooks for the OS to enable platform management. These hooks, detailed in the next subsections, are implemented in the NIC of the DN and compose a part of the HW support for OS4RS.

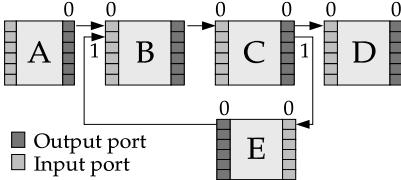


Fig. 2. Application Task Graph showing Input-Output port connections.

Task	src_out_port	dst_in_port	dst_logic_addr	dst_phys_addr
A	0	0	logic(B)	physical(B)
B	0	0	logic(C)	physical(C)
C	0	0	logic(D)	physical(D)
	1	0	logic(E)	physical(E)
D	{}	{}	{}	{}
E	0	1	logic(B)	physical(B)

Fig. 3. Destination Look-up Tables for every task in the graph.

Data NIC supports dynamic task relocation. Inter-task communication is done on an input/output port basis [2]. Figure 2 shows an example of an application task graph with the input/output port connections between tasks. Each application registers its task graph with the OS upon initialization [1].

For each task in the application, the OS assigns a system-wide unique logic address and places the task on the platform, which determines its physical address (Fig. 3). For every output port of a task the OS defines a triplet (destination input port, destination logic address, destination physical address). For instance, task C (Fig. 2) has two output ports, hence is assigned two triplets, which compose its Destination Look-Up Table (DLT) (Fig. 3). In our system a task may have up to 16 output ports, thus there are 16 entries in a DLT.

The OS can change the DLT at run-time, by sending an OAM message on the Control Network (CN) (Sect. 4.2). Dynamic task relocation in reconfigurable SoCs is enabled by storing a DLT in the data NIC of every tile in the system [2].

Data NIC monitors communication resources. The usage of communication resources on the DN is monitored in the data NIC of every tile. Figures such as number of messages coming in and out of a specific tile are gathered in the NIC in real time and made available to the OS. Another important figure available is the average number of messages that have been blocked due to lack of buffer space in the NIC. These figures allow the OS to keep track of the communication usage on the NoC. Based on these figures and on application priorities, the OS4RS can manage communication resources per tile and thus ensure Quality of Service (QoS) on the platform [1].

³ The data network is very similar to the NoC we described in a previous FPL paper [2].

Data NIC implements communication load control. The maximum amount of messages an IP is allowed to send on the network per unit of time can be controlled by the OS. To this end we have added an injection rate controller in the data NIC. As explained in [2], outgoing messages from an IP are first buffered in the NIC and are then injected in the network as soon as it is free (Best Effort service).

The injection rate controller adds an extra constraint on the time period when the messages may be injected in the NoC. It is composed of a counter and a comparator. The OS allows the NIC to inject messages only during a window of the counter time. The smaller the window, the less messages injected into the NoC per unit of time, freeing resources for other communications. This simple system, introduces a guarantee on average bandwidth⁴ usage in the NoC and allows the OS to manage QoS on the platform.

Data NIC adds HW support for OS security. Security is a serious matter for future reconfigurable SoCs. Thanks to reconfiguration, unknown tasks may be scheduled on HW resources and will use the DN to communicate. We must therefore perform sanity checks on the messages circulating on the DN and notify the OS when problems occur. Communication related checks are naturally performed in the NIC. We check whether the message length is smaller than the maximum transfer unit, that messages are delivered in order and especially that IPs do not breach security by sending messages on output ports not configured in the DLT by the OS.

4.2 Control Network

The control network is used by the operating system to control the behavior of the complete system (Fig. 1). It allows data monitoring, debugging, control of the IP block, exception handling, etc. OS OAM messages are short, but must be delivered fast. We therefore need a low bandwidth, low latency CN.

CN uses Message-Based Communication. To limit resource usage and minimize latency we decided to implement the CN as a shared bus, where the OS running on the ISP is the only master and all control network NICs of tiles are slaves. The communication on this bus is message-based and can therefore be replaced by any type of NoC.

The control NIC of every tile is memory-mapped in the ISP. One half of this memory is reserved for ISP to control-NIC communication and the other one for NIC to ISP communication. To send a control OAM message to a tile, the OS first writes the payload data, such as the contents of a DLT (Fig. 3) and finishes by writing a command code on the CN, in this case *UPDATE_DLT*. The control NIC reads the command opcode and processes it. When done, it writes a status opcode in the NIC to NoC memory, to indicate whether the command was successfully processed and posts an interrupt. The OS retrieves this data and clears the interrupt to acknowledge the end of command processing.

⁴ As long as the data NIC buffers are not permanently saturated.

CN Controls the DN Section 4.1 lists the capabilities of the data NIC in terms of control the OS has over the communication circulating on the DN. The OS commands, to enforce load control or synchronize DN communication, are actually sent over the CN to avoid interference with application data. It is in the control NIC, that statistics and security exceptions from the data NIC are processed and communicated to the OS. It is also through the CN that the OS sends destination look-up tables or injection-rate windows to the data NIC.

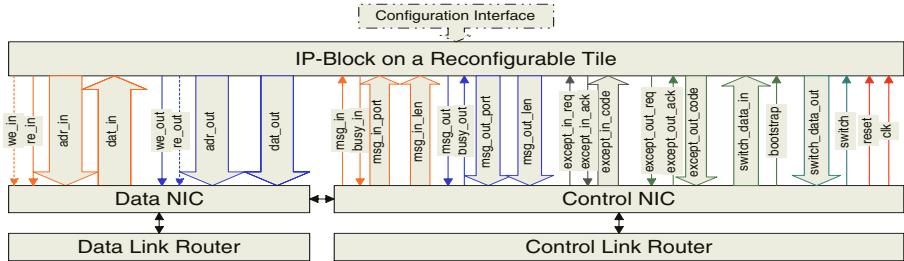


Fig. 4. Reconfigurable Tiles interface to all three NoCs through data and control NICs.

CN implements HW OS support to control IPs Another very important role of the CN is to allow control and monitoring of the IP running on a reconfigurable tile. To clearly understand the need for OS control here, let us consider the life-cycle of a reconfigurable IP block in our SoC platform.

Before instantiating the IP block in a tile by partial reconfiguration, we must isolate the tile from the communication resources, to ensure the IP does not do anything harmful on the DN before being initialized. To this end, the control NIC implements a reset signal and bit masks to disable IP communication (Fig. 4). After reconfiguration, the IP needs to be clocked. However, its maximum clock speed might be less than to that of our DN. Because we do not want to constrain the speed of our platform to the clock speed of the slowest IP (which can always change as new IP-blocks are modified at run-time), the OS can set a clock multiplexer to feed the IP with an appropriate clock rate.

The IP can now perform its computation task. At some stage it might generate an exception, to signal for instance a division by zero. The control NIC implements a mechanism to signal IP exceptions to the OS (Fig. 4). The OS can also send exceptions to an IP, as it can send signals to processes running on the ISP. One usage of these exceptions is to perform IP debugging.

Later on, the OS might decide to relocate the IP to another HW tile or as a process on the ISP [1–3]. The NIC implements a mechanism to signal task switching to the IP and to transmit its internal state information to the OS. The NIC also implements a mechanism to initiate an IP with a certain internal state, for instance when switching from SW to HW.

4.3 Reconfiguration Network

Our reconfigurable SoC targets a Xilinx VIRTEX-2TM PRO as an implementation platform. IPs are instantiated on tiles by partially reconfiguring the chip. In this case, the reconfiguration network is already present on the platform as the native reconfiguration bus of the VII-Pro. The reconfiguration bus is accessed through the internal reconfiguration access port (ICAP) and is based on the technology presented by Blodget in [4]. The main difference resides in the fact that our platform is driving the ICAP through the OS4RS, running on a PowerPC, instead of a dedicated soft core like the MicroBlazeTM.

5 Implementation Results

This section presents results about our enhanced HW support of an OS4RS, in terms of latencies induced by HW OS processing time and in terms of area overhead.

5.1 HW OS Reaction Time

The SW part of our OS4RS [1] is running on an ISP and controls the HW OS extensions located in the data and control NICs, through the control network (Sect. 4). Figure 5 shows the processing in SW and HW, when the OS4RS resets a reconfigurable IP block running on a HW tile. We assume that the control NIC is clocked at 22MHz and that the ISP can access the 16-bit wide control network at 50MHz. The SW part of the OS4RS sends the atomic *RST_IP* command to the control NIC of the IP in 120ns. A total of 12.8μs is spent in the control NIC to decode, process and acknowledge the commands issued from the SW part of the OS. Only 320ns are spent by the SW OS to send an atomic instruction and request the control NIC to clear the IRQ, acknowledging the command has been processed. The total processing time is under 13.2μs.

In the case of dynamic task relocation from SW to HW (Sect. 4.1), the reconfigurable IP needs to be initialized with the state information extracted from the SW version of the task [3]. Assuming we have 100 16-bits words of

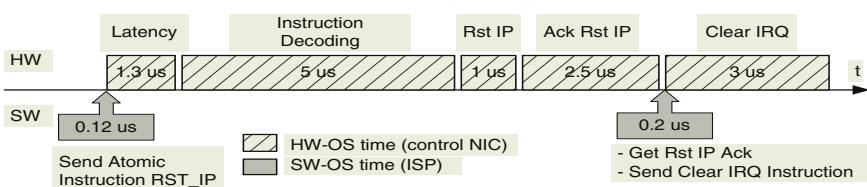


Fig. 5. OS4RS sends a Reset command to an IP. Most of the processing is performed in the control NIC, making it HW support for the OS4RS. Control NIC is clocked at 22MHz and control network is accessed by the ISP at 50MHz.

state information to transfer, the total transaction takes about $440\mu s$ (control NIC transmits a word to the IP in $4.3\mu s$).

In both cases the control NIC abstracts the access to the reconfigurable IP block from the SW part of the OS4RS. Because the NICs offload the ISP from low-level access to the reconfigurable IP blocks, they are considered as the HW part of the OS4RS.

5.2 HW OS Implementation Size

For research purposes, we implement the fixed NoCs together with the reconfigurable IPs on the same FPGA. We report therefore in table 1 the area usage of the NoCs in terms of FPGA logic and consider it as overhead to the reconfigurable IPs they support.

The old single NIC we presented in [2] performed both as control and data interface and is smaller than our new data and control NICs (Tab. 1), but it suffers from the limitations listed in section 3.2. The support of functions required by a full OS4RS such as state transfer, exception handling, HW debugging or communication load control come at the expense of a higher area overhead in the NIC. On our target platform, the Virtex-II Pro 20, this area overhead amounts to 611 slices, or 6.58 percent of the chip per reconfigurable tile instantiated. Nevertheless on a production reconfigurable SoC, the NoCs could be implemented as hard cores, reducing considerably the area overhead on the chip.

Table 1. HW overhead of Data and Control NICs, compared to the single NIC presented in [2].

Element	Virtex-II Slices	$XC2V6000(\%)$	$XC2VP20(\%)$
Control NIC and Router	250	0.74	2.7
Data NIC	361	1.07	3.89
Control+Data NIC	611	1.81	6.58
Old NIC from [2]	260	0.77	2.8

6 Conclusions

This paper presents how NoCs can be used as hardware components of an operating system managing reconfigurable SoCs. To support advanced features, such as dynamic task relocation with state transfer, HW debugging and security, an OS4RS requires specific HW support from the platform. We propose a novel architecture for reconfigurable SoCs composed of three NoCs interfaced to reconfigurable IPs. This approach gives a clean logical separation between the three types of communication: application data, OS control and reconfiguration bitstreams.

NoCs interfaced to reconfigurable IPs provide efficient HW support for an OS4RS. They open the way to future reconfigurable SoC platforms, managed by operating systems that relocate tasks between HW and SW to dynamically optimize resource usage.

References

1. V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins: Designing an Operating System for a Heterogeneous Reconfigurable SoC. Proc. RAW'03 workshop, Nice, April 2003.
2. T. Marescaux, A. Bartic, D. Verkest, S. Vernalde, R. Lauwereins: Interconnection Networks Enable Fine-Grain Dynamic Multi-Tasking on FPGAs. Proc. 12th Int. Conf. on Field-Programmable Logic and Applications, pages 795-805, Montpellier, September 2002.
3. J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, R. Lauwereins: Infrastructure for Design and Management of Relocatable Tasks in a Heterogeneous Reconfigurable System-on-Chip. Proc. DATE 2003, pages 986-992 , Munich, March 2003.
4. B. Blodget, S. McMillan, P. Lysaght: A Lightweight Approach for Embedded Reconfiguration of FPGAs. Proc. DATE 2003, pages 399-400, Munich, March 2003.
5. Rijpkema and al. E. Rijpkema et al.: Trade Offs in the Design of a Router with both Guaranteed and Best-Effort Services for Networks On Chip. Proc. DATE 2003, pages 350-355, Munich, March 2003.
6. H. Simmler, L. Levinson, R. Männer: Multitasking on FPGA Coprocessors. Proceedings 10th Int'l Conf. Field Programmable Logic and Applications, pages 121-130, Villach, August 2000.
7. W.J. Dally and B. Towles: Route Packets, Not Wires: On-Chip Interconnection Networks, Proc. 38th Design Automation Conference, June 2001.
8. <http://www.imec.be/ocapi>
9. <http://www.imec.be/reconfigurable>
10. J. Duato, S. Yalamanchili, L. Ni: Interconnection Networks, An Engineering Approach, September 1997. ISBN 0-8186-7800-3.

A Reconfigurable Platform for Real-Time Embedded Video Image Processing

N.P. Sedcole, P.Y.K. Cheung, G.A. Constantinides, and W. Luk

Imperial College, London SW7 2BT, UK.

pete.sedcole@imperial.ac.uk, p.cheung@imperial.ac.uk,
george.constantinides@ieee.org, w.luk@doc.ic.ac.uk

Abstract. The increasing ubiquity of embedded digital video capture creates demand for high-throughput, low-power, flexible and adaptable integrated image processing systems. An architecture for a system-on-a-chip solution is proposed, based on reconfigurable computing. The inherent system modularity and the communication infrastructure are targeted at enhancing design productivity and reuse. Power consumption is addressed by a combination of efficient streaming data transfer and reuse mechanisms. It is estimated that the proposed system would be capable of performing up to ten complex image manipulations simultaneously and in real-time on video resolutions up to XVGA.

1 Introduction

As advances are made in digital video technology, digital video capture sensors are becoming more prevalent, particularly in embedded systems. Although in scientific and industrial applications it is often acceptable to store the raw captured video data for later post-processing, this is not the case in embedded applications, where the storage or transmission medium may be limited in capacity (such as a remote sensor sending data over a wireless link) or where the data are used in real-time (in an intelligent, decision-making sensor for example). Real-time video processing is computationally demanding, making microprocessor-based processing infeasible. Moreover, microprocessor DSPs are energy inefficient, which can be a problem in power-limited embedded systems. On the other hand, ASIC-based solutions are not only expensive to develop, but inflexible, which limits the range of applicability of any single ASIC device.

Reconfigurable computing offers the potential to achieve high computational performance, at the same time remaining inexpensive to develop and adaptable to a wide range of applications within a domain. For this potential to become of practical use, integrated systems will need to be developed that have better power-performance ratios than currently available FPGAs, most likely by curtailing the general applicability of these devices such that optimisations for the particular application domain can be made. Such systems may be termed ‘domain specific integrated circuits’.

This paper outlines a proposed architecture for an integrated reconfigurable system-on-a-chip, targeted at embedded real-time video image processing. The

system is based on the Sonic architecture [1], a PCI-card based system capable of real-time video processing. Our objective is to integrate this system into a single device for embedded video processing applications.

In this paper we identify and discuss the unique issues arising from large scale integration of reconfigurable systems, including:

- How the complexities of designing such systems can be managed. Hardware modules and abstraction levels are proposed to simplify the design process.
- The implications of modularisation on the allocation and management of resources at a physical level.
- Effective connectivity and communication between modules.
- Minimising power consumed in data transmission and data reuse, the two most important factors in low-power design of custom computational platforms [2].

2 Related Work

Reconfigurable custom computing machines, implemented as arrays of FPGAs, have been successfully used to accelerate applications executing on PCs or workstations [3, 4]. Image processing algorithms are particularly suitable for implementation on such machines, due to the parallelisms that may be exploited [1, 5]. As mentioned in section 1, the system described in this paper is based on the Sonic architecture [1], a video image processing system comprising an array of FPGAs mounted on a PCI card.

Research has been conducted into embedded reconfigurable systems, and integrated arrays of processing elements [6–9]. Often these are conceived as accelerators of software-based tasks, and as such are closely coupled with a microprocessor, with the microprocessor forming an integral part of the data-path. As a consequence, these systems are usually adept at exploiting instruction-level parallelism; task-level parallelism is often ignored.

The proposed system has similarities to the DISC [10], which allows relocatable reconfigurable modules to vary in size in one dimension, and also includes the concept of position-independent connections to global signals. The modules in the DISC are very simple however. Our proposed system also incorporates ideas similar to the ‘dynamic hardware plugins’ proposed by Horta *et al.* [11] and virtual sockets described by Dyer *et al.* [12], although in both of these cases communication and interconnect structures are designed ad hoc on an application by application basis. Kalte *et al.* describe a system-on-a-programmable chip which does include a structured interconnection architecture for connecting dynamically reconfigurable modules [13]. Their system connects multiple masters to multiple slaves using either a multiplexed or crossbar-switch scheme. The interconnection scheme described in our paper allows any module to be a bus master, and is more suitable for streaming data transfers.

3 Managing Design Complexity

As device densities increase, circuit designers are presented with huge amounts of uncommitted logic, and large numbers of heterogeneous resources such as memories, multipliers and embedded hard microprocessor cores. Creating manageable designs for such devices is difficult; attempting to exploit dynamic reconfiguration as well only increases this complexity.

In order to make the design process tractable, we propose a modularised, hierarchical system framework, based on Sonic [1]. Modularisation partitions the system design into conceptually manageable pieces. In addition, high-level algorithmic parallelism can be exploited by operating two or more modules concurrently. To simplify the design, use and reuse of these modules, module interfaces need to be standardised, and abstractions are required in the transfer of data between modules.

3.1 System Architecture

As depicted in Figure 1, modularity is achieved by separating the data path into a variable number of processing elements connected via a global bus. Each processing element (PE) also has unidirectional *chain bus* connections to the adjacent PEs, for fast local data transfer. The left-most and right-most PEs are specialised input and output streaming elements respectively; data stream through the system in a general left-to-right direction.

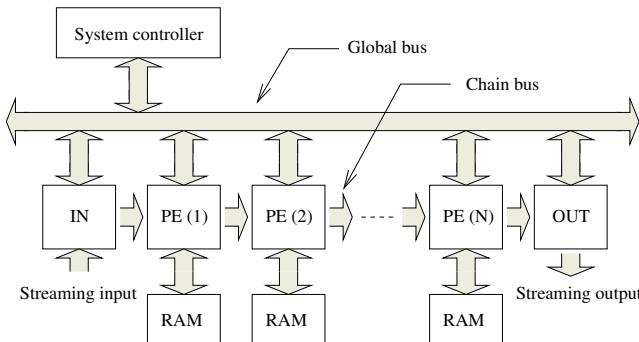


Fig. 1. The proposed system architecture.

Each element in the data path is designed to realise a complex task, such as a filter (2D convolution) or a rotation, implemented physically as a partial configuration instanced within a reconfigurable fabric. The processing applied to the data stream is determined by the type, number and logical sequence of the processing elements. The system designer controls this by programming a microprocessor within the system control module. PE module configurations (stored

as bitstreams) are loaded into the fabric and initialised by the control software, which also directs data-flow between PEs. As will be discussed below, the actual transfer of data is handled by a router within each PE. This scheme allows the implementation of a range of resource allocation and reconfiguration scheduling techniques, while realising the advantages of a data-path driven design.

Video processing algorithms can require a large amount of storage memory, depending on the frame size of the video stream. While available on-chip memory is constantly increasing, the size of storage space necessary, coupled with the bit-density (and therefore cost-per-bit) advantage of memory ICs, will ensure that external RAM will remain a necessary feature of video processing systems. The traditional memory bottleneck associated with external memory is avoided in our case by distributing the memory between the processing elements. The connection mechanisms between each PE and memory is discussed further in section 4.

3.2 Processing Elements

The logical composition of a processing element, as shown in Figure 2, comprises an engine, a router and input and output stream buffers. All of these are constructed from resources within the reconfigurable fabric. The PE has connections to the global bus (for communication between any two PEs), to the directly adjacent PEs to the left and right, and (optionally) to two external RAM banks. These interfaces are all standardised, which enables fully developed processing element configurations to be used and reused unchanged within a larger system design.

The core component of each PE is the processing engine; it is in this component that computation is performed on the image pixels. The engine is also the only part of the PE that is defined by the module designer. Serialised (raster-scan) data flow into and out of the engine by way of the buffers, the relevance of

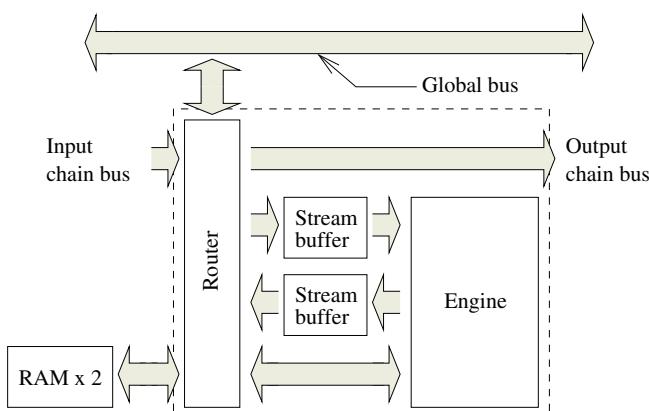


Fig. 2. The structure of a processing element.

which will be discussed in section 5. The key component for data movement abstraction is the router; this is directed by the system controller to transfer data between the buffers and the global bus, the chain bus or external memory. Since the data format is fixed, and all transfer protocols are handled by the router, the module designer is only concerned with the design of the computational logic within the engine.

4 Physical Structure

The mechanisms described in the previous section for controlling design complexity have ramifications for the physical design of the system. As mentioned above, the processing element modules are instanced within a reconfigurable FPGA fabric. The structure of this fabric needs to be able to support variable numbers and combinations of various sized modules. Since modules are fully placed and routed internally at design-time, the completed configurations must be relocatable within the fabric. The provision of mechanisms for connecting PE configurations to the buses and external RAM is required.

The physical system structure is illustrated in Figure 3. The processing elements are implemented as partial configurations within the FPGA fabric. The PEs occupy the full height of the fabric, but may vary in width by discrete steps. The structure and nature of the reconfigurable fabric is based on the Virtex-II Pro FPGA family from Xilinx, Inc. It is heterogeneous, incorporating not only CLBs but RAM block elements and other dedicated hardware elements such as multipliers. However, it exhibits translational symmetry in the horizontal dimension, such that the PEs are relocatable along the length of the fabric. The

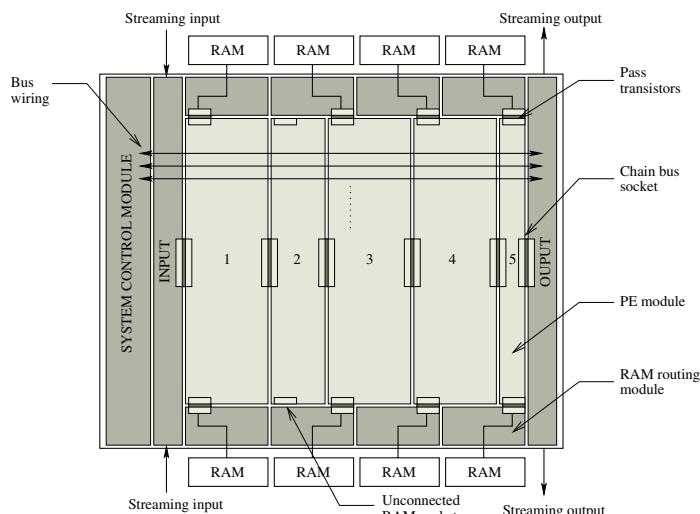


Fig. 3. A diagram representing the physical structure of the proposed system, with the reconfigurable fabric (shaded light grey) configured into five processing elements.

choice of a one-dimensional fabric simplifies module design, resource allocation and connectivity.

The global bus is not constructed from FPGA primitives; it has a dedicated wiring structure, with discrete connection points to the FPGA fabric. The advantage of this strategy is that the electrical characteristics of the global bus wiring can be optimised, leading to high speeds and low power [14, 15]. In addition, the wiring can be more dense than could otherwise be achieved.

Each processing element must have chain bus connections to the neighbouring PEs; this is accomplished through the use of ‘virtual sockets’, implemented as hard macros. The chain bus signals are routed as ‘antenna’ wires to specified locations along the left and right edges of the module. When two configurations are loaded adjacently into the array, these wires are aligned, and the signal paths may be completed by configuring the pass transistors (programmable interconnect points) separating the wires. Thus, each module provides ‘sockets’ into which other modules can plug into. Similar ideas have been proposed previously [11, 12], although the connection point chosen in previous work is a CLB programmed as a buffer.

A similar concept is used in connecting processing modules to the external RAM banks. Since the processing elements are variable-sized and relocatable, it is not possible to have direct-wired connections to the external RAM. The solution to this is to wire the external RAM to ‘routing modules’ which can then be configured to route the RAM signals to several possible socket points. This allows the registration between the RAM routing module and the processing element to be varied by discrete steps, within limits. If external RAM is not required by a particular processing element, such as PE 2 in Figure 3, the RAM resources may be assigned to an adjacent PE, depending on the relative placements.

5 Data Transfer and Storage

The transfer and storage of data are significant sources of power consumption in custom computations [2], so warrant specific attention. In the preceding system, Sonic, data transfer is systolic; each clock cycle one pixel value is clocked into the engine, and one is clocked out. This limits the pixel-level parallelism possible within the engine, and constrains algorithm design. In particular, data reuse must be explicitly handled within the engine itself, by storing pixel values in local registers. This becomes a significant issue for the engine design when several lines of image data must be stored, which can total tens of kilobytes.

In the proposed architecture the input stream buffer efficiently deals with data reuse. Being constructed from embedded RAM block elements (rather than from CLBs) a high bit density can be achieved. Image data is streamed into the buffer in a serial, FIFO-like manner, filling it with several lines of a frame. The engine may access any valid pixel entry in the buffer; addressing is relative to the pixel at the front of the queue. Buffer space is freed when the engine indicates it has finished with the data at the front of the queue. This system enables

greater design flexibility than a purely systolic data movement scheme while constraining the data access pattern sufficiently to achieve the full speed and power benefits of serial streaming data transfer. This is particularly beneficial when data are sourced from external RAM, where a sequential access pattern can take advantage of the burst mode transfer capability of standard RAM devices.

The input and output stream buffers are physically constructed from a number of smaller RAM elements for two reasons. Firstly, a wide data-path bit-width between the buffers and the engine can be achieved by connecting the RAM elements in parallel, enabling several pixels to be processed in parallel within the PE. The second important benefit is the ability to rearrange the input buffer RAM elements into two (or more) parallel stream buffers, when the engine requires more than one input data stream, such as in a merge operation. Likewise, the output buffer may be subdivided into several output streams, if the engine produces more than one output. We label each stream buffer input or output from an engine a ‘port’.

In addition to allowing efficient data reuse and fine-grained parallelism, the stream buffers create flexibility in the transfer of data over the global bus. Instead of a systolic, constant rate data-flow, data can be transferred from an output port buffer of one PE to the input port buffer of another PE in bursts, which allows the global bus to be shared between several logical communication channels. The arbitration between the various logically concurrent channels is handled by a reconfigurable arbitration unit within the system controller. This enables a range of arbitration strategies to be employed depending on the application, with the objective of preventing processing stalls from an input buffer under-run or output buffer overrun.

6 Performance Analysis

In the previous sections the proposed system was described in qualitative terms. We will now present a brief quantitative analysis to demonstrate how the system is expected to meet the desired performance requirements. To do this, it is necessary to make some assumptions about the sizes of various system elements; these assumptions will be based on the resources available in the Xilinx Virtex II Pro XC2VP125, and comparison with the latest incarnation of the Sonic architecture: UltraSONIC [16]. The numbers given here are speculative and do not represent any attempt at optimised sizing. Nevertheless, they are sufficient for the purpose of this analysis.

Based on the utilised logic cell count of modules in UltraSONIC and the cell count of the XC2VP125, between four and ten processing elements are possible in the current technology, depending on the complexity of each PE. Assuming the same I/O numbers as the Xilinx device (1200) and given the physical connectivity constraints of the system topology (see Figure 3) it is estimated that external RAM would be limited to eight banks, implying that not every PE would have access to external RAM. It is theoretically possible to implement 272 bus lines spanning the full width of the XC2VP125, however we will assume a bus width

of only 128 bits. The Xilinx device includes 18 columns of Block RAMs, each of which can be arranged as 1024 bits wide by 512 deep memory. Therefore assigning each processing element two 32KB stream buffers is not unreasonable, perhaps configured as 512 wide by 512 deep. A wide buffer facilitates pixel-level parallel computations, although for algorithms that do not exhibit much low-level parallelism a wide buffer would be a disadvantage.

External RAM sizes and system throughput is determined by the video format the system is applied to. Table 1 gives some figures for some representative video formats. In previous work, external RAM was sized such that one video frame would fit one RAM bank [16], which would imply between 330KB to over 3MB per bank in this case.

Using these figures, some calculations on the processing characteristics of the system can be made. With a 512-bit wide buffer, 16 pixels (at 32-bits/pixel) can be operated on in parallel. As Table 2 indicates, several complete lines of image data can be stored in the stream buffers for data reuse. An example engine clock rate is given, assuming each block of 16 pixels is accessed 10 times during the computation, a realistic value for a 2D convolution with a 5x5 kernel. This clock rate is at least an order of magnitude below the state-of-the-art for this technology, which is highly desirable as low clock frequencies correspond with lower operating voltages and very little pipelining, which all translate into lower power consumption. Table 2 also gives the required clock rate of the global bus, for five concurrent channels, all operating at the full throughput rates given in Table 1. Again, these speeds should be easily achievable, especially given that it is a custom structure and not constructed from FPGA primitives.

These calculations demonstrate that the proposed system is expected to be able to perform between four and ten complex image computations simultaneously and in real-time on video data of resolutions up to XVGA.

Table 1. A sample of video formats. Frame sizes are based on 32 bits per pixel, while throughput is calculated at 25 frames per second.

Format	Lines	Columns	Frame size	Throughput
PAL	288	352	330 KB	9.67 MB/s
DVD PAL	576	720	1620 KB	39.55 MB/s
XVGA	768	1024	3072 KB	75.00 MB/s

Table 2. System characteristics. The cycle time is the average time allowed to process 16 pixels in parallel. The transfer capacity of the bus is calculated assuming a 10% overhead for arbitration and control.

Format	Processing element			Global bus	
	Lines/buffer	Cycle time	Clock rate	Transfer capacity	Bus speed
PAL	23	6.3 μ s	1.6 MHz	53 MB/s	3.5 MHz
DVD PAL	11	1.5 μ s	6.5 MHz	218 MB/s	14.3 MHz
XVGA	8	0.8 μ s	12.3 MHz	413 MB/s	27.0 MHz

7 Conclusion and Future Work

A reconfigurable platform suitable for embedded video processing in real-time has been presented. The platform comprises a number of configurable complex processing elements operating within a structured communication framework, all controlled by a central system controller. The design aims to meet the demanding requirements of real-time video processing by exploiting fine-grained (pixel-level) parallelisms within the processing elements, as well as task-level algorithmic parallelisms by operating processing elements concurrently.

The customisation of processing elements enables the system to be adapted to a wide range of applications, and since these elements are dynamically reconfigurable, run-time adaptation is also possible. Moreover, the inherent modularity of the PEs, coupled with the communication infrastructure, facilitates design and reuse.

Finally, power efficiency is addressed by targeting data movement. Dedicated bus wiring can be optimised for low-power transmission, while data movement is minimised by reusing data stored in stream buffers and constraining processing element designs. Parallel processing results in lower average operating frequencies, which can be translated into lower power consumption by reducing the operating voltage.

Our next main objective is to translate the proposal as described in this paper into a prototype, so that we may assess its feasibility. Since the physical structure of the system is based heavily on the Virtex II Pro, it would be logical to implement the prototype in a device from this family. In order to investigate the operational performance of the system, it will be necessary to map one or more actual applications from Sonic to the new platform, which may require the development of tools and processes.

Acknowledgements

The authors would like to thank Simon Haynes, Henry Epsom and John Stone of Sony Broadcast and Professional Europe for their helpful comments. The support from Xilinx Inc., and Patrick Lysaght in particular, is appreciated. N.P. Sedcole gratefully acknowledges the financial assistance from the Commonwealth Scholarship Commission in the United Kingdom.

References

1. Haynes, S.D., Stone, J., Cheung, P.Y.K., Luk, W.: Video image processing with the Sonic architecture. *IEEE Computer* **33** (2000) 50–57
2. Soudris, D., Zervas, N.D., Argyriou, A., Dasygenis, M., Tatas, K., Goutis, C., Thanailakis, A.: Data-reuse and parallel embedded architectures for low-power, real-time multimedia applications. In: International Workshop - Power and Timing Modeling, Optimization and Simulation. (2000)
3. Arnold, J.M., Buell, D.A., Hoang, D.T., Pryor, D.V., Shirazi, N., Thistle, M.R.: The Splash 2 processor and applications. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors. (1993)

4. Vuillemin, J.E., Bertin, P., Roncin, D., Shand, M., Touati, H.H., Boucard, P.: Programmable active memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems* **4** (1996) 56–69
5. Athanas, P.M., Abbott, A.L.: Real-time image processing on a custom computing platform. *IEEE Computer* **28** (1995) 16–24
6. Callahan, T.J., Hauser, J.R., Wawrzynek, J.: The Garp architecture and C compiler. *IEEE Computer* **33** (2000) 62–69
7. Ebeling, C., Cronquist, D.C., Franklin, P.: RaPiD – Reconfigurable Pipelined Datapath. In: *Field-Programmable Logic and Applications*. (1996)
8. Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Taylor, R.R.: PipeRench: A reconfigurable architecture and compiler. *IEEE Computer* **33** (2000) 70–77
9. Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S., Agarwal, A.: Baring it all to software: RAW machines. *IEEE Computer* **30** (1997) 86–93
10. Wirthlin, M.J., Hutchings, B.L.: A dynamic instruction set computer. In: *IEEE Symposium on FPGAs for Custom Computing Machines*. (1995)
11. Horta, E.L., Lockwood, J.W., Taylor, D.E., Parlour, D.: Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In: *Design Automation Conference*. (2002)
12. Dyer, M., Plessl, C., Platzner, M.: Partially reconfigurable cores for Xilinx Virtex. In: *Field-Programmable Logic and Applications*. (2002)
13. Kalte, H., Langen, D., Vonnahme, E., Brinkmann, A., Rückert, U.: Dynamically reconfigurable system-on-programmable-chip. In: *Euromicro Workshop on Parallel, Distributed and Network-based Processing*. (2002)
14. Benini, L., De Micheli, G.: Networks on chips: A new SoC paradigm. *IEEE Computer* **35** (2002) 70–78
15. Dally, W.J., Towles, B.: Route packets, not wires: On-chip interconnection networks. In: *Design Automation Conference*. (2001)
16. Haynes, S.D., Epsom, H.G., Cooper, R.J., McAlpine, P.L.: UltraSONIC: a reconfigurable architecture for video image processing. In: *Field-Programmable Logic and Applications*. (2002)

Emulation-Based Analysis of Soft Errors in Deep Sub-micron Circuits*

M. Sonza Reorda and M. Violante

Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy
{matteo.sonzareorda, massimo.violante}@polito.it

Abstract. The continuous technology scaling makes soft errors a critical issue in deep sub-micron technologies, and techniques for assessing their impact are strongly required that combine efficiency and accuracy. FPGA-based emulation is a promising solution to tackle this problem when large circuits are considered, provided that suitable techniques are available to support time-accurate simulations via emulation. This paper presents a novel technique that embeds time-related information in the topology of the analyzed circuit, allowing evaluating the effects of the soft errors known as single event transients (SETs) in large circuits via FPGA-based emulation. The analysis of complex designs becomes thus possible at a very limited cost in terms of CPU time, as showed by the case study described in the paper.

1 Introduction

The adoption of new manufacturing deep sub-micron technologies is raising concerns about the effects of *single event transients* (SETs) [1] [2] [3], which correspond to erroneous transitions on the output of combinational gates, and several approaches to evaluate them have been proposed, which exploit fault injection [4]. During fault injection execution systems undergo to perturbations mimicking the effects of transient faults. The goal of these experiments is normally to identify portions of the system that are more sensible to faults, to gain statistical evidence of the robustness of hardened systems, or to verify the correctness of the implemented fault tolerance mechanisms.

Simulation-based fault injection [4] may be used to cope with the aforementioned purposes: it indeed allows early evaluating fault effects when only a model of the system is available. Moreover, it is very flexible: any fault model can be supported and faults can be injected in any module of the system. The major drawback of this approach is the high CPU time required for performing circuit simulation.

In the last years, several approaches to simulation-based fault injection have been proposed. Earlier works were based on switch-level simulation tools, such as the one described in [5], which can be adapted to the execution of fault injection experiments targeting the SET fault model. More recently, several authors proposed the use of HDL simulators to perform fault injection campaigns, and several approaches have been presented (e.g., [6]-[9]) for speeding-up the process.

* This work was partially supported by the Italian Ministry for University through the Center for Multimedia Radio Communications (CERCOM).

As soon as simulation-based fault injection gained popularity, researches faced the problem of reducing the huge amount of time needed for injecting faults in very complex designs, entailing thousands of gates. Two main approaches were presented. A first one addresses the problem from the simulation efficiency point of view, by minimizing the time spent for simulating each fault [10] [11]. Conversely, the second one focuses on the clever selection of the faults to be injected during simulations, thus reducing simulation time by reducing the total number of faults to be injected [12] [13]. The former approaches that exploit FPGA-based emulation seem to be well suited for analyzing very complex circuits. Indeed, they combine the versatility of simulation, which allows injecting faults in any element of circuits, while providing the performance typical of hardware prototypes.

When soft errors affecting deep sub-micron technologies become of interest, the already available emulation techniques fall short in supporting the analysis of SET effects. As better explained in section 2, SETs are transient modifications of the expected value on gate output, which have duration normally shorter than one clock cycle. As a result, SET effects can be analyzed only through circuit models or tools able to deal with timing information. This can be easily done by resorting to timed simulation tools, but at the cost of very high simulation time. Conversely, FPGA-based hardware emulation can hardly be used unless suitable techniques are available to take into account time-related information.

The main contribution of this paper is in proposing an FPGA-based fault injection environment suitable to assess the effects of SETs via emulation, thus allowing designers analyzing large circuits at low cost in terms of time needed for performing injection experiments. For this purpose we describe a novel technique able to introduce time-related information in the circuit topology. FPGA-based emulation of the time-enriched circuit can thus be exploited to asses SET effects without any loss of accuracy with respect to time-accurate simulations, provided that a suitable fault model is adopted. As a result, we are able to achieve the same accuracy of state-of-the-art approaches based on timed simulations while reducing fault injection execution time up to 5 orders of magnitude.

In order to assess the feasibility of this idea, we developed an emulation-based fault injection environment that exploits a rapid-prototyping board equipped with a Xilinx Virtex 1000E. The experimental results we gathered through this environment, although still preliminary, show the soundness of the proposed approach.

The paper is organized as follows: section 2 reports an overview of the considered fault model, while section 3 describes the circuit expansion algorithm used to embed time-related information in the circuit topology. Section 4 presents the fault injection environment we developed and section 5 reports some experimental results assessing the effectiveness of the proposed approach. Finally, section 6 draws some conclusions.

2 Single Event Transients

Today, the fault model that is normally used during fault injection experiments is the (single/multiple) bit-flip in the circuit storage elements, i.e., registers and embedded memories. However, with the adoption of deep sub-micron technologies, a new fault model is becoming of interest: the *single event transient*.

Single event transients are originated when highly energized particles strike sensible areas within combinational circuits. In deep sub-micron CMOS devices, the most sensible areas are depletion regions at transistor drains [14]. The particle strike produces there several hole-electron pairs that start drifting under the effect of the electric field. As a result, the injected charge tends to change the state of the struck node producing a short voltage pulse. As the depletion region is reformed, the charge-drift process decays, and the expected voltage level at the struck node is restored.

In deep sub-micron circuits the capacitance associated to circuit nodes is very small, therefore non-negligible disturbances can be originated even by small amounts of deposited charge, i.e., when energized particles strike the circuit. As reported in [14], in old 5 Volt CMOS technologies the magnitude of the voltage swing associated to SETs is about 14% greater than the normal voltage swing of the node and thus its impact is quite limited, in terms of both duration and magnitude. Conversely, if the technology is scaled to a 3.3 Volt one, the disturbance becomes 21% larger than a normal swing and thus the transistor that must restore the correct value of the struck node will employ more time to suppress the charge-drift process. As a result, the duration of the voltage pulse the striking particle originates increases with technology scaling. In very deep sub-micron technologies this effect may become a critical issue since the duration of the voltage pulse may become comparable to the gate propagation delay and thus the voltage pulse may spread throughout the circuit, possibly reaching its outputs.

As measurements reported in [14] show, a SET can be conveniently modeled at the gate level as an erroneous transition (either from 0 to 1 or from 1 to 0) on the output of combinational gates.

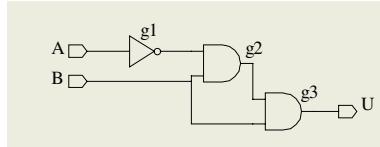
3 The Circuit Expansion Principle

This section describes the principles at the base of the fault injection technique we developed. Sub-section 3.1 presents the circuit expansion algorithm we exploited, sub-section 3.2 presents the fault list generation algorithm we devised; finally, sub-section 3.3 presents the fault model we defined, which can be fruitfully exploited for assessing the effects of SET in combinational circuits or in the combinational portion of sequential ones.

3.1 Expansion Algorithm

The algorithm we adopted for embedding time-related information in the circuit topology while preserving its original functionality was first introduced in [15] for evaluating power consumption of combinational circuits. Please note that although the original and the expanded circuits are functionally equivalent, i.e., they produce the same output responses to the same input stimuli, their topologies are different.

To illustrate the idea behind the algorithm, let us consider the circuit shown in figure 1, where input and output ports are driven by non-inverting buffers. We assume the variable-delay model, where all the gates in the circuit (including buffers) may have different propagation delays.

**Fig. 1.** Circuit example

We assume that the circuit is initially in steady state, i.e., an input vector was applied to inputs A and B and its effects have been already propagated through the circuit to the output U.

The algorithm we exploit modifies the given circuit C by building an expanded circuit C' such that C' is a superset of C . This task is performed according to the following expansion rules:

1. Initially, C' is set equal to C .
2. For each gate $g \in C$ whose output is evaluated at time T from the application of the input vector, a new gate g/T is added to C' . We refer to g/T as a *replica* of g , and to T as the *time frame* of g/T , i.e., g/T belongs to time frame T .
3. The fan-in of $g/T \in C'$ is composed of the replicas of the gates in the fan-in of $g \in C$ which belong to time frames $T' < T$.

Assuming that a new input vector is applied at time T_0 , and that each gate i has propagation delay equal to P_i , we compute the time frames of each replica according to the algorithm reported in figure 2.

```

TimeFrame[i] = {  $P_i$  }       $\forall i \in$  Circuit Inputs
TimeFrame[i] =  $\emptyset$            $\forall i \notin$  Circuit Inputs
foreach( $i \in$  Circuit Gates)
    foreach( $k \in$  Circuit Gate in the fan-in of  $i$ )
        TimeFrame[i] = TimeFrame[i]  $\cup$  ( $P_i +$ TimeFrame[k])

```

Fig. 2. Algorithm for computing time frames

Any gate i may be associated to a list of time frames ($TimeFrame[i]$), due to the existence of paths of different length connecting gate i with the circuit inputs.

The list of time frames associated to the circuit of figure 1 is reported in table 1; it was computed under the assumption that the propagation delay of each gate (including circuit inputs) is equal to the gate fan-out, while the propagation delay of circuit outputs is assumed to be 1 ns. For instance, in table 1 we have that the output of g_3 is first updated at time frame T_0+3 ns in correspondence to the new value of B, and then it is again updated at time frame T_0+4 ns following the update of g_2 .

The obtained expanded circuit embeds time-related information in a structural way: the gates belonging to time frame T are those whose outputs are evaluated T ns after the application of the new input vector.

The application of the expansion rules may lead to replicate the circuit outputs many times. In the considered example we have indeed that the output U is replicated twice. We refer to the replica of the output belonging to the last time frame (i.e., the highest time frame) as the circuit *actual output*. For the considered example U/5 is the

actual output of the circuit; conversely, U/4 store the intermediate value the circuit output assumes during the evaluation of the new input vector.

Table 1. Gate update times

Gate	A	B	g1	g2	g3	U
Time frame [ns]	1	2	2	3	3, 4	4, 5

3.2 Fault List Generation Algorithm

The computation of the list of faults to consider during fault injection experiments is a critical issue, since the number of faults in the fault list directly affects the injection time. As a result, the fault list generation algorithm should be able to select the minimum number of faults, i.e., it should select only the SETs (in terms of fault locations and fault injection times) that have chances of affecting the circuit outputs.

The algorithm we developed starts from the expanded version of the circuit whose sensitivity to SETs should be analyzed, and produces the list of faults that should be simulated. In the following we will refer to the gate where the SET originates as the *faulty gate*.

Figure 3 reports an example of SET: the circuit primary inputs are both set to 1, thus the expected output value is 0. Due to a charged particle, the output of g1 is switched to 1 for a period of time long enough for the erroneous transition to propagate through the circuit. As a result, we observe a transition on g3, whose output is set to 1. As soon as the SET effects disappear, the output switches back to the expected value.

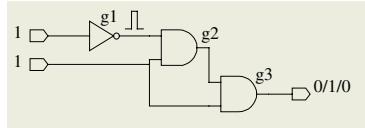
SET effects can spread through the fan-out cone of the faulty gate if, and only if, the duration of the erroneous transition is equal to or longer than the faulty gate propagation delay and if the magnitude of the transition is compatible with the device logic levels. In the following, we will concentrate our attention only on those particles that, when hitting the circuit, produce SETs that satisfy the above conditions.

Let T_h be the time when the SET originates, δ be the SET duration, T_s the time when the outputs of the circuit are sampled. Moreover, let Π be the set of propagation delays associated to sensitized paths stemming from the faulty gate to the circuit outputs, i.e., those paths that due to the input vector placed on the circuit inputs allow the SET to spread through the circuit.

Any SET is effect-less, i.e., its effects cannot be sampled on the circuit outputs, if the following condition is met:

$$T_h + \delta + t < T_s \quad \forall t \in \Pi \quad (1)$$

If eq. 1 holds, it means that once the SET effects disappear and the expected value is restored on the faulty gate, the correct value has enough time to reach the circuit outputs, and thus the expected output values are restored before they are sampled.

**Fig. 3.** SET effects

The values T_h and δ are known since they are used to characterize the SET in the fault list. Furthermore, T_s is known a-priori, and it is selected according to the circuit *critical path*, i.e., the maximum propagation delay between the circuit inputs and its outputs. Conversely, Π depends on the circuit topology and its accurate computation normally requires complex and time-consuming algorithms.

In our approach, eq. 1 is exploited in combination with the expanded version of the circuit, thus avoiding the computation of Π for each gate in the circuit. As a result, we can save a significant amount of CPU time that can be devoted to more detailed SET effects analysis.

Thanks to the properties of the expanded circuit, all the gates in the fan-in cone of the circuit *actual outputs* do not satisfy eq. 1. Indeed, the mentioned gates belongs to the circuit *critical path*, and thus any signal skew (like that introduced by SETs) is not acceptable: if allowed to spread through the circuit, the SET will indeed modify the outputs in such a way that an erroneous value will be sampled at time T_s . Moreover, let g/T be a gate in the fan-in cones of the circuit actual outputs. Then, only the effects of those SETs originated on g at time $T_h = T$ may reach circuit outputs.

Each fault in the list of possible SETs can be described as a couple (faulty gate, T_h). The list of faults that should be simulated can thus be easily obtained by traversing the expanded circuit, starting from its *actual outputs*, going toward its inputs and storing all the traversed gates.

3.3 Vector-Bounded Stuck-at

Once the list of possible faults has been computed, fault injection experiments are still required to assess SETs effects. Let (g, T_h) be the SET that we would inject, computed according to the algorithm described in section 3.2. Moreover, let T_v be the time when a new vector is applied to the circuit inputs and T_s be the time when circuit outputs are sampled.

The fault model we propose, called *vector-bounded stuck-at*, consists in approximating any SET with a stuck-at fault on the output of the faulty gate g/T_h in the expanded version of the circuit. During simulation, the vector-bounded stuck-at originates at time T_v and lasts up to T_s . As a result:

- The stuck-at affects the replica g/T_H of the faulty gate g belonging to time frame T_H only during the evaluation of one input vector.
- The stuck-at has no effects on the replicas of the faulty gate belonging to time frames other than T_H . Moreover, it does not affect g/T_H before and after T_v .

Thanks to the properties of the expanded circuit, although the stuck-at on g/T_h is injected at time T_v , it starts influencing the circuit only from T_h . Moreover, by resorting to the time-related information embedded in the expanded circuit, timed-simulation is no longer needed: the vector-bounded stuck-at can indeed be injected as

soon as the stimuli are applied to the circuit inputs and fault effects can last for exactly one vector. Zero-delay fault simulation can thus replace timed simulation without any loss of accuracy.

4 The Fault Injection Environment

A typical fault injection environment is usually composed of three modules:

- *Fault List Manager*: it generates the list of faults to be injected, according to a given fault model and the possible indications of designers (e.g., the most critical portions in the system, or a particular time interval).
- *Fault Injection Manager*: it selects a new fault from the fault list, injects it in the system, and then observes the results.
- *Result Analyzer*: it analyzes the data produced by the previous module, categorizes faults according to their effects, and produces statistical information.

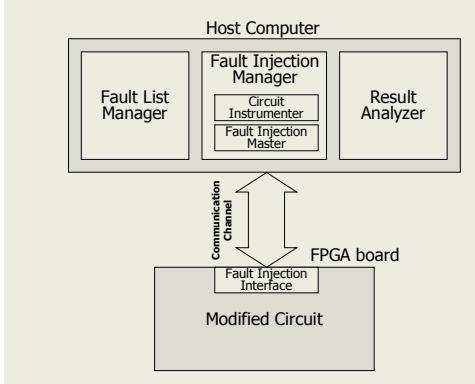
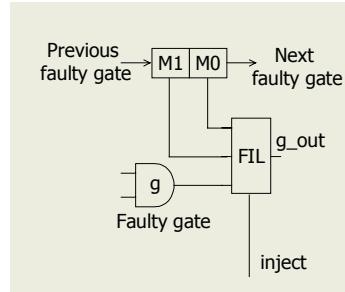
For the purpose of this paper we assume that the system under analysis is a VLSI circuit (or part of it). Therefore, we assume that a gate-level description of the system is available that was enriched with time-related information according to section 3.1. We also assume that the input stimuli used during fault injection experiments are already available, and we do not deal with their generation or evaluation. Moreover, the adopted fault model is the vector-bound stuck-at presented in sub-section 3.3.

We implemented the Fault List Manager as a software process that reads both the gate-level system description and the existing input stimuli, and that generates a fault list according to the adopted fault model. Similarly, we implemented the Result Analyzer as a software process.

In our approach the Fault Injection Manager module exploits a FPGA board that emulates an instrumented version of the circuit to support fault injection. The FPGA board is driven by a host computer where the other modules and the user interface run. As a result, the Fault Injection Manager is implemented as a hardware/software system where the software partition runs on the host, and the hardware partition is located on the FPGA board along with the emulated circuit. More in details, the Fault Injection Manager is composed of the following parts:

- *Circuit Instrumenter*: it is a software module running on the host computer that modifies the circuit for supporting fault injection;
- *Fault Injection Master*: it iteratively accesses to the fault list, selects a fault and orchestrates each fault injection experiment by sending to the board the input stimuli. When needed it triggers the fault injection. It finally receives from the board the system faulty behavior. The Fault Injection Master corresponds to a software module running on the host computer.
- *Fault Injection Interface*: it is implemented by the FPGA board circuitry, and it is in charge of interpreting and executing the commands the Fault Injection Master issues.

The architecture of the whole environment is summarized in Figure 4.

**Fig. 4.** FI environment architecture**Fig. 5.** Instrumented architecture for supporting fault injection

In order to support the injection of the vector-bound stuck-at fault model, we modify the circuit under analysis before mapping it on the FPGA board. For each faulty gate in the circuit the logic depicted in figure 5 is added.

Let suppose the gate g in figure 5 be the faulty gate. The following hardware modules are added for allowing fault injection on the output of g :

- *inject*: it is an input signal controlled by the Fault Injection Interface. When set to 1 it triggers the injection of the fault. This signal is set to 1 in correspondence of the clock cycle during which the fault should be injected.
- $M1$ and $M0$: it is a two-bit shift register, containing the fault the user intends to inject (01 corresponds to the vector-bound stuck-at-0, 10 corresponds to the vector-bound stuck-at-1, while 00 and 11 correspond to the normal gate behavior, i.e., the fault is not injected no matter the value of *inject*). The content of the shift register is loaded with the fault the user intends to inject before starting the application of the input stimuli.
- FIL : it is a combinational circuit devised to apply the vector-bound stuck-at to the gate g , depending on the value of *inject*, $M1$ and $M0$.

In order to simplify the set-up of fault injection experiments, the shift registers corresponding to the considered faulty gates are connect to form a scan chain, which can be the Fault Injection Interface can read/modify via three signals (scan-in, scan-out and scan enable). Since the fault that should be injected is defined by simply shifting in the scan chain the proper sequence of bits, the user can easily perform the analysis of single and multiple faults.

5 Experimental Results

We developed the proposed fault injection environment using the ADM-XRC development board [16], which is equipped with a Xilinx Virtex V1000E FPGA. The board hosts a PCI connector and it is inserted in a standard Pentium-class PC, which accesses the board via a memory-mapped protocol.

In order to insert the proposed hardware fault injection circuitry, we developed a tool that, starting from the original circuit, automatically synthesizes all the modules our architecture requires, according to section 4.

We developed the hardware/software Fault Injector Manager as follows:

- The Fault Injection Master is coded in C++ and it amounts to 500 lines of code.
- The Fault Injection Interface is implemented inside the Xilinx FPGA (along with the circuit under evaluation) and it works as a finite state machine that recognizes and executes the commands the Fault Injection Master issues. We used a memory-mapped protocol to control the operations of the FPGA board, which includes the following commands:
 - a. *Reset*: all the registers are set to the start state.
 - b. *Load scan chain*: the scan chain is loaded with the values specifying the fault to be injected.
 - c. *Apply vector*: the circuit Primary Inputs (PIs) are fed with a given input stimulus.
 - d. *Fault Inject*: the fault injection takes place.
 - e. *Output read*: the values on the circuit Primary Outputs (POs) are read.

The instrumented circuit, together with the Fault Injection Interface, is described in an automatically generated synthesizable VHDL code. We then used typical development tools to obtain the bitstream needed to program the Xilinx device. FPGA Compiler II is used to map the description to the Xilinx device, while Xilinx tools are employed to perform place, route and timing analysis of the implemented design.

We exploited the described fault injection environment to assess the effects of SETs in a circuit implementing a 27-channels interrupt controller. In particular, being interested in assessing the simulation performance the approach attains, we randomly generated several input stimuli of different lengths. We then applied the stimuli to the considered circuit with two different fault injection environments, while considering the same set of faults:

1. Faults were injected in the original circuit comprising 164 gates by exploiting the VHDL-based fault injection environment presented in [17].
2. Faults were injected according to the described emulation-based fault injection environment. For this purpose the considered circuit was first expanded to embed time-related information and then instrumented to support fault injection, obtaining an equivalent but enriched circuit comprising 2,232 gates.

For each fault injection experiment we recorded the CPU time needed for injecting the whole fault list that accounts for 418 faults. VHDL simulations were performed on a Sun Enterprise 250 running at 400 MHz and equipped with 2 GBytes of RAM. The same machine executed the synthesis and placement of the considered circuit in about 20 minutes.

From the results reported in table 2 the reader can observe the effectiveness of the proposed approach, which is able to reduce the time needed for performing fault injection experiments of about 5 orders of magnitude, while providing the same results in terms of fault effect classification of VHDL simulation-based fault injection. When the time for circuit synthesis and placements is also considered for evaluating the overall performance of our approach, we observe that the speed-up with respect to simulation is still very high: about 3 orders of magnitude.

Table 2. Simulation results

Number of input stimuli	VHDL simulation-based fault injection [s]	Emulation-based fault injection [s]
1,000	159,894.72	0.92
2,000	310,688.45	0.92
3,000	459,684.17	0.93
4,000	628,578.90	0.93

6 Conclusions

The paper presented an approach suitable to assess the effects of single event transients via FPGA-based emulation, making thus feasible the analysis of complex deep sub-micron circuits with a very limited cost in terms of time needed for performing fault injection experiments.

The approach we presented is based on a technique able to embed time-related information in the circuit topology, while preserving its original functionality. An FPGA-based fault injection environment is then exploited to perform injection experiments on the enriched circuit.

Experimental results are reported assessing the effectiveness of the proposed approach, which is able to reduce the time needed for performing fault injection by several orders of magnitude.

References

- [1] L. Anghel, M. Nicolaidis, "Cost Reduction of a Temporary Faults Detecting Technique", DATE'2000: ACM/IEEE Design, Automation and Test in Europe Conference, 2000, pp. 591-598
- [2] C. Constantinescu, "Impact of deep submicron technology on dependability of VLSI circuits", Proc. IEEE Int. Conference on Dependable Systems and Networks, 2002, pp. 205-209
- [3] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, L. Alvisi, "Modelling the effect of technology trends on the soft error rate of combinational logic", Proc. IEEE Int. Conference on Dependable Systems and Networks, 2002, pp. 389-398
- [4] Mei-Chen Hsueh, T.K Tsai, R.K Iyer, "Fault injection techniques and tools", IEEE Computer, Vol. 30, No. 4, 1997, pp. 75-82
- [5] P. Dahlgren, P. Liden, "A switch-level algorithm for simulation of transients in combination logic", Proc. Fault Tolerant Computing Symposium, 1995, pp. 207-216
- [6] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, "Fault Injection into VHDL Models: the MEFISTO Tool", Proc. Fault Tolerant Computing Symposium, 1994, pp. 66-75
- [7] T.A. Delong, B.W. Johnson, J.A. Profeta III, "A Fault Injection Technique for VHDL Behavioral-Level Models", IEEE Design & Test of Computers, Winter 1996, pp. 24-33
- [8] D. Gil, R. Martinez, J. V. Busquets, J. C. Baraza, P. J. Gil, "Fault Injection into VHDL Models: Experimental Validation of a Fault Tolerant Microcomputer System", European Conference of Dependable Computing (EDCC-3), 1999, pp. 191-208

- [9] J. Boué, P. Pétillon, Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Experimental Assessment of Fault Tolerance", Proc. Fault-Tolerant Computing Symposium, 1998, pp. 168-173
- [10] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, "Exploiting Circuit Emulation for Fast Hardness Evaluation", IEEE Transactions on Nuclear Science, Vol. 48, No. 6, December 2001, pp. 2210-2216
- [11] L. Antoni, R. Leveugle, B. Fehér, "Using run-time reconfiguration for fault injection in hardware prototypes", Proc. IEEE Int.l Symp. on Defect and Fault Tolerance in VLSI Systems, 2000, pp. 405-413
- [12] L. W. Massengill, A. E. Baranski, D. O. Van Nort, J. Meng, B. L. Bhuva, "Analysis of Single-Event Effects in Combinational Logic-Simulation of the AM2901 Bitslice Processor", IEEE Transactions on Nuclear Science, Vol. 47, No. 6, 2000, pp. 2609-2615
- [13] L. Berrojo, I. González, F. Corno, M. Sonza Reorda, G. Squillero, L. Entrena, C. Lopez, "New Techniques for Speeding-up Fault-injection Campaigns", Proc. IEEE Design, Automation and Test in Europe, 2002, pp. 847-852
- [14] K. J. Hass, J. W. Gambles, "Single event transients in deep submicron CMOS", IEEE 42nd Midwest Symposium on Circuits and Systems, 1999, pp. 122-125
- [15] S. Manich, J. Figueras, "Maximizing the weighted switching activity in combinational CMOS circuits under the variable delay model", Proc. IEEE European Design and Test Conference, 1997, pp. 597-602
- [16] ADM-XRC PCI Mezzanine card User Guide Version 1.2, <http://www.alphadata.co.uk/>
- [17] B. Parrotta, M. Rebaudengo, M. Sonza Reorda, M. Violante, "New Techniques for Accelerating Fault Injection in VHDL descriptions", IEEE Int.l On-Line Testing Workshop, 2000, pp. 61-66

HW-Driven Emulation with Automatic Interface Generation

M. Çakir¹, E. Grimpe², and W. Nebel¹

¹ Department of Computer Science

Carl von Ossietzky University Oldenburg, D-26111 Oldenburg, Germany
`{cakir,nebel}@informatik.uni-oldenburg.de`

² OFFIS Research Institute, D-26121 Oldenburg, Germany
`grimpe@offis.de`

Abstract. This paper presents an approach to automate the emulation of HW/SW-Systems on an FPGA-board attached to a host. The basic steps in design preparation for the emulation are the generation of the interconnection and the description of the synchronization mechanism between the HW and the SW. While some of the related work considers the generation of the interconnection with some manual interventions, the generation of the synchronization mechanism is left to the user as part of the effort to set up the emulation. We present an approach to generate the interconnection and the synchronization mechanism, which allows a HW-driven communication between the SW and the HW.

1 Introduction

During the emulation and prototyping of HW-designs with average complexity, universal FPGA-boards can be used instead of complex logic emulators. These are usually boards, that can be plugged into a host system. The communication between host and board typically uses a standard bus, e.g. via PCI [1]. In contrast to the stand-alone boards, these boards can be used more versatile and not only at the end of the design flow, because the usage of a host allows a flexible emulation of the environment.

Furthermore the combination of host and FPGA makes verification acceleration and validation of HW/SW-systems possible. To accelerate the verification of a HW design a part of the design is emulated on the FPGA and the rest is simulated on the host. For this usage the simulator and a part of the design on the FPGA have to be connected. In order to validate a HW/SW-system the SW communicates with the HW on the FPGA via a standard bus. In this case usually a model of the target processor is used to emulate the behaviour of the SW on the target processor (e.g. ISA models).

This paper concentrates on the interface generation as well as on synchronization mechanism for the emulation. In the next section the problems related with the emulation process are described shortly. In section 3 we give an overview about related work on co-simulation, co-emulation. Section 4 describes our approach to automate

the emulation of a HW design with PCI-based FPGA-boards. In section 5 the automatic interface generation is explained in detail. The last section draws a conclusion.

2 Problem Definition

FPGA-based emulation requires complex software flows and manual intervention to prepare the design for the emulation [2]. The preparation process for the emulation on an FPGA-board plugged into a host includes two major steps: i) the generation of the interconnection between the SW and the HW via a physical channel between the host and the FPGA, ii) the generation of the synchronization mechanism (Figure 1).

During the emulation of a HW/SW-system with a PCI-based FPGA-board, the communication between the FPGA and the host occurs via PCI. Although the PCI-core on the FPGA and the PCI-driver on the host are used to abstract the PCI-communication, the user has to design and implement the interconnection between the SW and PCI-driver as well as between the HW and PCI-core manually. This task must be treated differently for different designs, especially if the HW has several communication channels with the SW and if they can request concurrently data transfer.

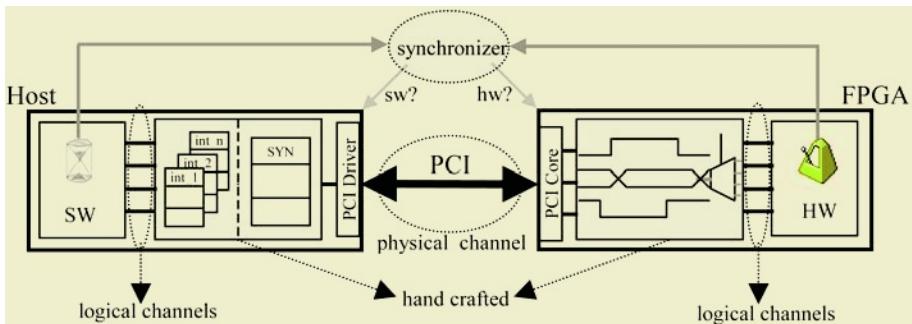


Fig. 1. Emulation on a PCI-board

On the other hand the SW on the host and the HW on the FPGA have to be synchronized with each other. An instance, we call it synchronizer in the following, has to determine the time of the communication between these parts, and this instance has to control the whole system. In general, the whole system would be controlled by the host or the FPGA. In case of the implementation of the synchronizer on the host there are two possibilities: causality based synchronization and timed synchronization. The first solution implicates in principle the sequential execution of the SW and the HW. For the timed synchronization the execution time of the SW on the target processor has to be determined (e.g. by means of instruction set simulator, simulation of the real time kernel). Furthermore the timed synchronization causes a high synchronization overhead. In case of the HW controlled synchronization the synchronizer is implemented as HW on the FPGA and runs in parallel to the HW-part of the design. In this

case the synchronizer offers more precise information regarding the timing, since the HW is at least cycle accurate. By means of using this feature and the causality between the HW and the SW it is possible to execute the HW and the SW in parallel, to improve the emulation performance without extra overhead.

In this paper we present a HW-driven approach to reduce the effort of the preparation process for the emulation as well as the synchronization overhead, and to generate (semi-) automatically the interconnection and the synchronization mechanism between the HW on the FPGA and the SW on the host.

3 Related Work

Research on validation/verification of HW/SW-systems with non-formal methods can be classified in two groups: co-simulation and co-emulation. While in the first group the whole system is simulated at several abstraction levels, the techniques in the second group use an emulation platform either for the whole or for part of the system. [3] is an example to emulate the whole system. However, most of the emulation techniques use a platform with FPGA to emulate the HW and a host to simulate the SW.

In [4] research on co-simulation is classified into three groups: i) co-simulation algorithms for various computation models, ii) automatic generation of co-simulation interface, and iii) co-simulation speed-up. This classification can be seen as valid for the emulation-based validation/verification. While the techniques to generate the interface for the co-simulation can be used for the emulation, the speed-up aspect should be handled differently due to the possibility of the parallelism at the emulation.

In simulation-based approaches HDL-simulators are used to simulate the HW and a host for the SW with processor models at several abstraction levels. To connect two designs with each other an interconnection topology has to be generated and the synchronization of the communication has to be described. In general the related work mostly concentrates on the generation of the interconnection, but the second aspect is usually disregarded and considered as a part of user effort.

[5, 6] describe a similar backplane-based approach. In this approach the connection is established via ports meaning that the user describes the ports which are connected with each other manually. This solution demands the definition of corresponding ports in SW and in HW. The problem in terms of the synchronization is solved partially in timed simulation by global time steps even if it means a certain degree of inflexibility. In case of functional untimed simulation the synchronization has to be described as handshake protocol by the user.

In [7] the connection topology is described with usage of the VCI specification. In a VCI specification the user has to describe the ports of the design, like in the entity of a VHDL-specification, in order to generate the connection topology. In addition to the port declaration a VCI-specification contains a sensitivity clause for every port which is similar to the sensitivity list of a VHDL-process. The sensitivity clause allows the protocol description of an interface (synchronization mechanism). If any

sensitivity clause becomes true, the data transfer between the VHDL-simulator and the SW is activated. This approach is one of the few approaches that considers the synchronization of the communication. However, this is suitable only for simple protocols and the user has to describe the VCI-specification and modify the design manually.

In emulation-based approaches similar methods are used to connect the HW and SW with each other. While [8, 9, 10, 11] present simulation-emulation platforms using an FPGA and an HDL-Simulator, in [12] a co-emulation platform is described. [13] describes a proposal for a standard C/C++ modeling interface for emulators and others verification platforms.

The solution of the synchronization problem during the co-emulation in above approaches is either i) a blocking communication mechanism, ii) synchronization with a global clock or iii) coprocessor-based synchronization with several limitations. While the approaches with blocking communication allow the execution of only one part of the system (either SW or HW) at any time [8], the global-clock-based approaches require a global clock and the SW as well as the HW have to include a time concept based on this clock [12]. For the co-processor based synchronization the user has to guarantee that the HW is able to process the last data from the SW before the next access from the SW to the HW [10, 11]. If this is not the case, sufficient waits have to be inserted manually in the SW to fulfill this assumption. In spite of this limitation and manual insertion of the waits into the SW, these approaches support only the combinational HW-designs, not the sequential designs. All of these approaches provide only the SW-controlled synchronization, which implicates the above-mentioned limitations and disadvantages.

In summary existing approaches have the following disadvantages: i) only some of them support the automatic generation of the interconnection, ii) the synchronization of the communication is mostly disregarded and left the user as part of the effort to set up the emulation, iii) only a few approaches consider the synchronization of the communication; however, the generation of the synchronization is in these approaches partial manual and they support only SW-driven synchronization iv) FPGA's are only used as fast simulation engines, but not to reduce the overhead of the synchronization.

4 Our Approach

In this section we shortly describe our approach and the tool ProtoEnvGen which was implemented to evaluate our approach. A detailed description can be found in [14].

The key objectives of our approach are to reduce the manual intervention, to prepare a design for the emulation and to increase the emulation speed, solving the problems described in section 2. To achieve this our approach i) allows semi-automatic generation of the interconnection, ii) reduces the overhead for the synchronization by means of HW-driven synchronization, iii) allows the mixed-level co-emulation without user intervention, iv) hides the implementation details of the emulation platform from the user.

Our emulation platform consists of a host (PC) and a universal FPGA-board plugged into the host. Our approach is verified on a PCI-based FPGA-board, but it can be adapted easily for any combination of host and FPGA-board. The design, which will be emulated, consists of a SW-part and a HW-part. The SW-part can be in principle the emulation of the environment of a HW-design or the SW-part of a HW/SW-system. However, in this section the SW refers to primarily the emulation of the environment of a HW-design, which will be emulated on the FPGA and called in the following DUT (Design Under Test).

In our approach the whole system is controlled primarily by the HW. However, we allow the controlling of the HW by the SW, too. The control of the system by the HW is based on the fulfillment of the communication requirements from the DUT with the SW. In order to fulfill the communication requirements from the DUT, the communication requests from the DUT are monitored on the FPGA. If any request is detected, the corresponding communication is initiated. If the communication request can not be satisfied, the DUT is stopped and a communication request is sent to the host via an interrupt. The stopping of the DUT means that the clock of the DUT (*DUT_clk*) stays stable and is not triggered. In case of an interrupt the SW initiates a data transfer to meet the communication request from the HW. The DUT is started again as soon as all requests from the DUT are satisfied.

On our evaluation platform we have two kinds of communication: the DUT-communication and the PCI-communication. The PCI-communication is only part of the emulation and will not be implemented in the real system. A PCI-driver on the host and a PCI-core on the FPGA are used to abstract this communication. The DUT-communication is the communication between the DUT and the SW and is described predominantly in the DUT by the user. Since the whole communication between the DUT and the SW occurs via PCI, the DUT-communication in our emulation platform has to be coupled with the PCI.

We assume that the DUT-communication can have several interfaces (DUT-interfaces) for data transmission to and from the SW on the host. Several DUT-interfaces mean that there are several communication channels between the DUT and the SW, which have to be mapped on the PCI and can request concurrent data transfer. It means that concurrent data transmission requests from the DUT-interfaces will be scheduled and the data transmission through every DUT-interface will be modeled on the PCI, namely on the host between the SW and the PCI-driver, as well as on the FPGA between the DUT and the PCI-core.

The design flow with ProtoEnvGen contains two steps: declaration of the DUT-interfaces and the generation of the emulation environment. While the first step is performed partially user defined, the last step is fully automated. Figure 2 illustrates the functionality of the ProtoEnvGen and its embedding in the emulation setup.

The inputs of the generation process are the VHDL-description of the DUT and the declaration of every DUT-interface. The outputs are a structural VHDL-description of the HW-part (ProtoEnv) and a set of C++-classes for the SW-part to communicate with the HW-part (ProtoEnv-API).

The ProtoEnv, which can be synthesized with a synthesis tool, consists of the DUT, the PCI-core, DUT-interfaces and a controller. In our experiments we have

used a commercial FPGA-board and a soft PCI-core provided by the board manufacturer [15] in order to validate our concept . The controller connects the DUT with the PCI-core via DUT-interfaces and coordinates the communication between the DUT and PCI-core, therefore synchronizes the communication between the HW and the SW. For the HW-driven communication the controller monitors the communication requests from the DUT-interfaces and manages the communication as well as the DUT according to these requests. In case of the SW-driven controlling of the DUT, the controller handles the DUT according to the order of the SW (e.g. stopping, starting of the DUT).

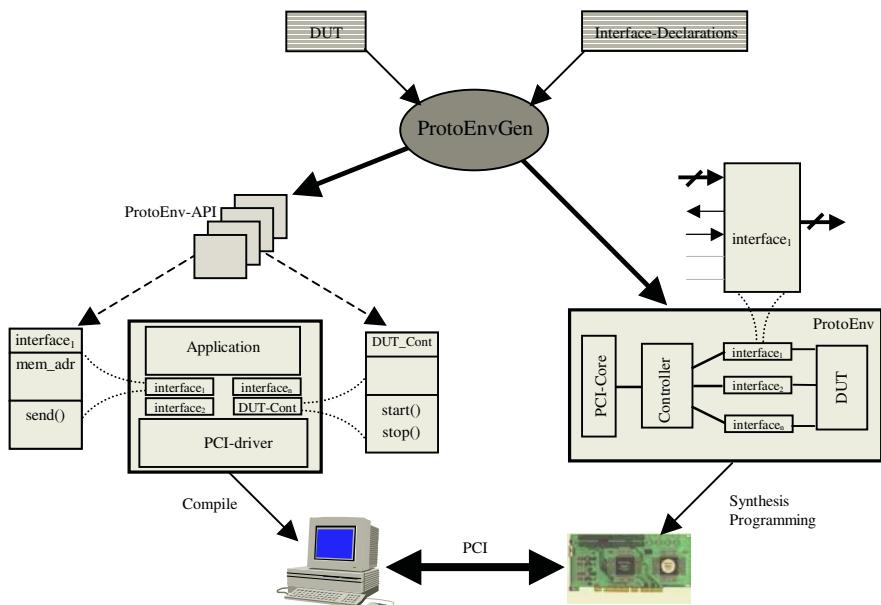


Fig. 2. Designflow with ProtoEnvGen:

The ProtoEnv-API provides access to any DUT-interface, control of the DUT and, primary, the fulfillment of the requests from the controller in case of the HW-driven communication. By means of this API the user can send or receive data to/from a DUT-interface. In order to access to DUT-interfaces for every kind of interface an interface class with an accessing method (send or receive) is generated and for every DUT-interface an object of the corresponding class should be instantiated in the SW (interface₁ in Figure 2). If the user wants to control the DUT from the SW, he or she can reset, start and stop the DUT. For HW-driven communication an interrupt handler is defined. If an interrupt is set by the controller, which means that a communication requirement from a DUT-interface has to be satisfied, the interrupt handler determines the reason for the interrupt and calls the corresponding method of the corresponding interface object. The ProtoEnv-API uses a PCI-driver in order to access the HW. Since the driver development is not the focus of our approach, we use a commercial tool provided by Jungo Ltd. to generate the PCI-driver [16].

5 Interface Generation for the Hardware-Driven Emulation

A DUT-interface is characterized by a port of the DUT and a protocol, and allows the DUT to communicate with the SW. Here, the protocol of a DUT-interface determines when the data transmission occurs, but not how the data transmission is realized physically (e.g. a protocol can characterize that a data transmission via corresponding interface takes place every x cycles, but it does not describe whether the communication is bus-based or point-to-point). A DUT-interface can describe data transmission without any reactivity between the SW and the DUT or a reactive data transmission. Both kinds of data transmission have different requirements regarding the timing of the data flow on the PCI and must be treated differently. To meet the timing constraints required by reactive communication between the DUT and the SW, the timing overhead caused by PCI-communication must be hidden. In contrast to a reactive communication a data transmission without any reactivity is not critical regarding the transfer time on the PCI and can be buffered. To increase the efficiency of the data transfer via PCI every non-reactive DUT-interface has a buffer in the on-board memory. In addition, interfaces are directed – either from HW to SW or vice versa.

In the first version of the ProtoEnvGen we used the Protocol Compiler provided by Synopsys to generate the VHDL-code of the DUT-interfaces with some manual interventions [17], [14]. In the rest of this section we present a method to generate automatically interfaces for the emulation on a PCI-based FPGA-board. In this approach the user intervention is limited only to the declaration of every DUT-interface. During the declaration of a DUT-interface the user should declare the name and whether it is reactive or non-reactive as well as the port, which should be involved in the communication via this interface. The rest of the information for the generation of the interface is extracted from the VHDL-description of the DUT.

The declaration of an interface is followed by the automatic generation of the DUT-interface. It includes the generation of the interconnection and the generation of the synchronization mechanism.

5.1 Generation of the Interconnection

In order to generate the interconnection the following steps are performed after the declaration of all interfaces:

i) Since we use a shared memory communication, the address of the shared-memory for every DUT-interface is determined. While for every non-reactive interface a memory area is assigned, a reactive interface needs only a register.

ii) For every DUT-interface a VHDL-instance is generated. As illustrated in Figure 3, an interface at structural level is an instance with some ports, which are either data ports for the data transfer or control ports for the synchronization, and which allows the connection of the DUT with the controller. The ports *interface_request* and *interface_grant* are intended for synchronization based on hand-shaking. When a data transfer via a DUT-interface is requested, the signal *interface_request* of the corresponding DUT-interface has to be set. The controller sets the signal *interface_grant*

as soon as the request of the interface is satisfied. Between the setting of the *interface_request* and the *interface_grant* the DUT is stopped by the controller.

To generate the interconnection in the HW-part the ports of the generated DUT-interface have to be connected to the corresponding ports of either the DUT or the controller.

iii) In the SW-part an interface-object of the corresponding interface-class is instantiated with the address of the shared memory of the DUT-interface as an attribute.

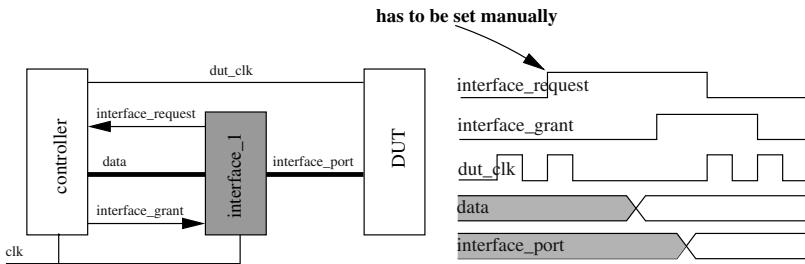


Fig. 3. Communication via a DUT-interface with manual synchronization

The generated interconnection grants access to any DUT-interface from the host by using the accessing methods of the corresponding interface object. However, the synchronization has to occur manually, since there exists no synchronization mechanism yet. This means that either the user has to modify the DUT-interface to set the *interface_request* (as illustrated in Figure 3) or only the SW-driven synchronization is possible.

5.2 Generation of the Synchronization Mechanism

In our approach a DUT-interface is activated when the DUT access to the port which is involved in and connected to the DUT-interface (*interface_port*). This means that every access to the *interface_port* in the DUT has to be detected and for every access the signal *interface_request* of the corresponding DUT-interface has to be set.

For the detection of a port access and the generation of the *interface_request* we search the corresponding statements (e.g. signal assignments) in the VHDL-description of the DUT, and modify them. This process is composed of the following for every DUT-interface: i) A new port, called *interface_enable*, is inserted into the entity of the VHDL-code. ii) On the *interface_enable* an event is generated when the *port_interface* is accessed. Therefore an event statement is inserted before every access statements to the *interface_port*. iii) The port *interface_enable* is connected to the port *interface_request*. Herewith the DUT-interface is activated when it is necessary and waits for the *interface_grant* from the controller.

This approach is problematic if the *interface_port* is a buffered input port of the DUT. For such a port after synthesis a register which is synchronized with the clock of the DUT (*DUT_clk*) is generated. When the *interface_enable* is set, the register of

the input port is already loaded. But the register has to be loaded after the generation of the *interface_grant* by the controller, since only after the setting of the *interface_grant* the controller provides the valid data for the actual request. On the other hand the reloading of the register after *interface_grant* causes the changing of the actual state of the DUT. To solve this problem we synchronize the register for the buffered input ports with the master clock of the HW-part instead of the *DUT_clk* and reloaded it as soon as the *interface_grant* is set. The master clock refers to the clock that triggers the PCI-core, the controller and the DUT-interfaces.

Figure 4 illustrates the automatic generation of the synchronization mechanism. Figure 4-a and 4-b show schematically this process on the basis of two block diagrams, while Figure 4-c concretely shows the realization of this concept on the basis of a VHDL-code in a simple example. The original DUT in 4-a has two data ports, and we assume that two interfaces are defined. Both; the ports *in_port* and *out_port* involve in the communication via an interface. The DUT accesses to *in_port* when the condition C_1 is true and to *out_port* when the condition C_2 is satisfied. In modified DUT (4-b) where the modifications are illustrated bold, TFFs are inserted to generate the *interface_enable* events. For example TFF₁ generates an event on the *int_1_en* if C_1 is true, when the *in_port* is accessed. The DFF₁, which is the buffer of the *in_port* and triggered by the *clk* instead of the *dut_clk*, is reloaded when the *int_1_grant* is set. To realize this concept in the example (4-c), the process *proc_new* and the lines marked with asterisk are inserted after the modification of the original DUT.

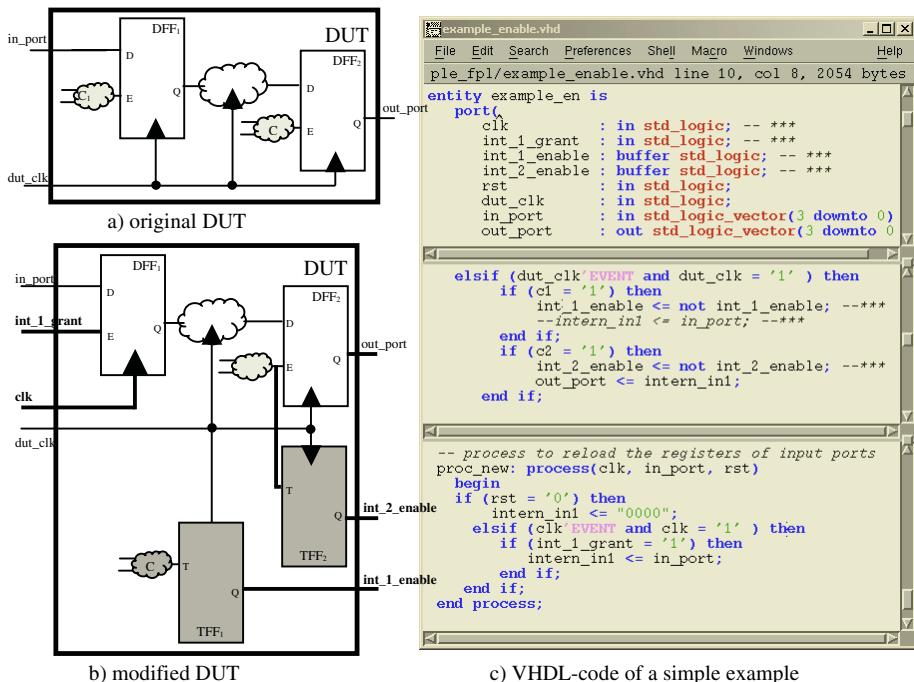


Fig. 4. Automatic generation of the synchronization mechanism

The synchronization mechanism presented above implicates the stopping of the DUT after the setting of every *interface_enable*. To avoid this problem the data of every non-reactive DUT-interface can be cached. It means that the controller provides every non-reactive DUT-interface with direction to the DUT valid data before these request new data and sets the signal *interface_grant* of the corresponding interface. Furthermore, this approach supports only the sequential designs, not the combinational designs yet.

6 Conclusion

We have presented an approach to automate the interface generation and the synchronization of the communication between the SW and the HW for the co-emulation. Our tool ProtoEnvGen, which has been implemented and tested with simple examples to evaluate our approach, is also presented shortly. Although in this paper the SW relates to the emulation of the environment of a HW-design, the presented HW-driven emulation approach is also suitable for HW/SW-systems. The HW-driven emulation allows mixed-level emulation regarding time, and speeds up the emulation through the parallel processing of the synchronizer with the DUT. Furthermore, it makes the generation of synchronization mechanism with minimum manual intervention possible.

Our current work is concentrated on the HW-driven emulation of HW/SW-systems, especially on the support of multi-tasking in the SW-part, and the improvement of the efficiency of our approach, e.g. through the caching of data during the communication and the support of the combinational designs, as well as on the verification of our approach with complex real world designs.

References

1. Tom Shanley, Don Anderson: PCI System Architecture, Addison-Wesley, 1999
2. Helena Krupnova, Gabriele Saucier: FPGA-Based Emulation: Industrial and Custom Prototyping Solutions, in Proc. FPL 2000
3. Spyder System, <http://www.x2e.de>
4. Sungjoo Yoo, Kiyoung Choi: Optimistic Timed HW-SW Cosimulation, in Proc. Asia-Pacific Conference on Hardware Description Language, 1997
5. Alexandre Amory, Fernando Moraes, Leandro Oliveira, Ney Calazans, Fabiano Hessel: A Heterogeneous and Distributed Co-Simulation Environment, in Proc. SBCCI 2002
6. Cosimate, <http://www.tni-valiosys.com>
7. C. A. Valderrama, F. Nacabal, P. Paulin, A. Jerraya: Automatic Generation for Distributed C-VHDL Cosimulation of Embedded Systems: an Industrial Experience, in Proc. RSP 1996
8. C. Fritsch, J. Haufe, T. Berndt: Speeding Up Simulation by Emulation – A Case Study, in Proc. DATE 1999
9. N. Canellas, J. M. Moreno: Speeding up HW prototyping by incremental Simulation/Emulation, in Proc. RSP 2000

10. Ramaswamy Ramaswamy, Russel Tessier: The Integration of SystemC and Hardware-Assisted Verification, in Proc. FPL 2002
11. Siavas Bayat Sarmadi, Seyed Ghassem Miremadi, Ghazanfar Asadi, Ali Reza Ejlali: Fast prototyping with Co-operation of Simulation and Emulation, in Proc. FPL 2002
12. W. Bishop, W. Loucks: A Heterogenous Environment for Hardware/Software Cosimulation, in Proc. 30th Annual Simulation Symposium, 1997
13. Functional Requirements Specification: Standard Co-Emulation Modeling Interface (SCE-MI) ver. 1.0 rev. 1.9 2002, <http://www.eda.org/itc/scemi19>
14. M. Çakir, E. Grime: ProtoEnvGen: Rapid ProtoTyping Environment Generator; in Proc. VLSI-SOC 2001
15. PLD Applications, <http://www.plda.com>
16. Jungo Ltd.: WinDriver User's Guide, <http://www.jungo.com>
17. Synopsys Inc.: Synopsys Protocol Compiler User Guide, 2000

Implementation of HW\$im

– A Real-Time Configurable Cache Simulator

Shih-Lien Lu and Konrad Lai

Microprocessor Research
Intel Labs
Hillsboro, OR 97124
Shih-Lien.l.Lu@intel.com

Abstract. In this paper, we describe a computer cache memory simulation environment based on a custom board with multiple FPGAs and DRAM DIMMs. This simulation environment is used for future memory hierarchy evaluation of either single or multiple processors systems. With this environment, we are able to perform real-time memory hierarchy studies running real applications. The board contains five Xilinx' VirtexTM II-1000 FPGAs and eight SDRAM DIMMs. One of the FPGA is used to interface with a microprocessor system bus. The other four FPGAs work in parallel to simulate different cache configurations. Each of these four FPGAs interfaces with two SDRAM DIMMs that are used to store the simulated cache. This simulation environment is operational and achieves a system frequency of 133MHz.

1 Introduction

Memory hierarchy design is becoming more important as the speed gap between processor and memory continues to grow. In order to achieve the highest performance with the most efficient design simulation is usually employed to assess design decisions. There are several ways to evaluate performance [1]. Backend simulation using address reference traces is one of the methods [2]. Traces can be collected either through hardware or software means. After the traces have been collected, they are stored in some forms. A backend (trace-driven) simulator is employed to evaluate the effectiveness of different cache organizations and memory hierarchy arrangements. This method is repeatable and efficient. However its usefulness is limited by the size of the traces. Another common method used to evaluate memory hierarchy is through complete system simulation. In this approach, every component of a complete system is modeled including the memory hierarchy. Full system simulation is flexible and provides direct performance of benchmarks [3][4]. However, full system simulation is less detail and slower in speed, which may limit the design space explored. It is also much more difficult to setup benchmarks on a simulated machine.

The ideal scenario is to emulate the memory hierarchy with hardware and perform a measurement of the actual system while executing real workloads. Yet the real time

TM Virtex is a trademark of Xilinx Corp.

emulation hardware is usually very costly and inflexible. It is difficult to explore all design space interested using emulation. We employed an alternative method called hardware cache simulation (HW\$im).

Hardware cache simulator is a real time hardware based simulation environment. It watches the front-side bus and collects traces of an actual system while the system is running real application software. However, instead of just storing the traces and dumping them out after the buffer is full, it processes the traces with hardware in real time and stores only the statistical information. That is, instead of collecting the traces and processing them with a software simulation program later on, the hardware cache simulator simulates the traces immediately. Since traces are processed directly and not stored, we are able to run for a much longer period of time and to observe long-term behavior of benchmarks. It also permits us to explore larger design spaces include large cache effects of multi-processor systems. It removes some artifacts from software simulation having limited trace length.

Hardware cache simulator provides a complementary tool to full system simulation with software. Since it is running at real-time, it allows us to scan different workloads with multiple configurations. When the hardware cache simulator indicates the possibility that a larger cache is significantly beneficial, we can zoom in with the current hardware tracing methodology or full system simulation. Moreover, since we are implementing this simulator with FPGAs we have the flexibility to modify coherency protocol and study migratory sharing behavior. If there are particular behaviors that can benefit from pre-fetching strategies we could also add these pre-fetching algorithms into the simulator to study their impact on performance.

The rest of the paper is organized as follows. Section 2 discusses related work in this area and provides a summary of the HW\$im. Section 3 describes the design of HW\$im. Section 4 describes our experience in developing the board. Section 5 presents some preliminary results collected using the HW\$im. We draw some conclusions in the last section.

2 Related Works and Summary

Much research has been done on using FPGAs for computing [5]. They fall into four general categories – 1) special or general purpose computing machines; 2) re-configurable computing machines; 3) rapid prototyping for verification or evaluation; and 4) emulation and simulation (or accelerators). There are several commercial products that offer boards that can be used for emulation purposes. Some examples are [6][7]. While these products offer flexibility, they are generally used for design verification and software/hardware co-development. Our requirement is a bit different. We need to be at speed with the system we are interfacing with. We are using this system to collect performance information of future systems instead of verifying designs.

A previously published work similar to this work is the IBM MemorIES [8]. The main differences are: (1) Our design uses DRAM instead of SRAM to store simulated cache directory information; (2) We use interleaving (parallel FPGAs) to bridge the bandwidth differences between system bus and SDRAM; (3) Our board is designed as a PCI board which can be installed on a host machine directly. Using DRAM allows larger simulated caches be studied. Interleaving of multiple FPGAs in parallel enable

us to perform parallel simulation of different configurations at the same time, thus removing the artifact of any non-deterministic behavior of running applications repeated sequentially. With PCI interface we can download simulation results at a much finer time unit.

Figure 1 illustrates the system setup of HW\$im. Two main structures are in the setup. The first is the system-under-test which is the system that runs the application whose behavior we are investigating. The second system is the host system that houses the custom built board with FPGAs. A logic analyzer interposer probe is used to capture the signals on the front side bus of the system under test and send them to the HW\$im

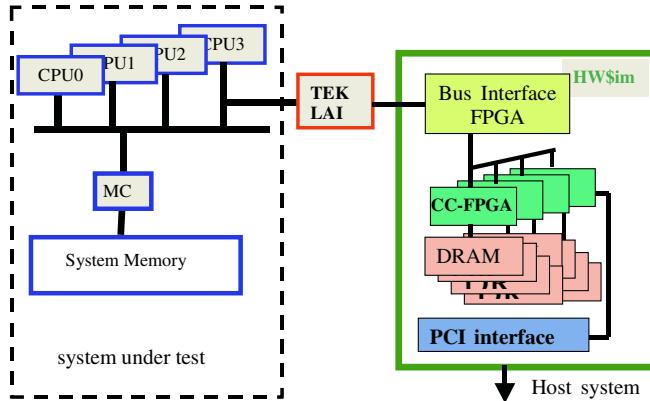


Fig. 1. HW\$im Systems Setup

Picture to the right of figure 2 shows the HW\$im board plugged into the host system collecting data. The host system is a dual processor machine with Intel Pentium® III Xeon™ Processors. Picture on the left of figure 2 illustrates the system under-test with the Tektronix LAI probe and cables.

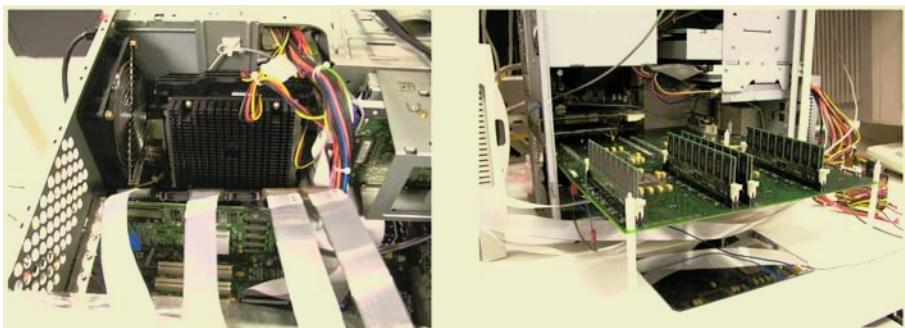


Fig. 2. Systems with HW\$im

[®] Pentium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[™] Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

3 Design Description

The key to hardware cache simulation is the rapid development of the capabilities of field programmable gate array devices. The amount of gates has grown substantially. They also contain many features that simplify system level designs. To accommodate the need for I/O pins of this design, we adopt the XC2V1000 chips with the FF896 package. This chip provides 432 user I/O pins. We also employ SDRAM DIMMs to store cache directories being simulated. It provides high capacity with a wealth of reference designs available.

3.1 Functional Description of Hardware Cache Simulator

There is total of four main functional blocks in the hardware cache simulator (HW\$im) as illustrated on the right hand side of Figure 1. They are 1) Bus Interface (BI); 2) Cache Controller (CC); 3) DRAM (simulated cache directories); and 4) PCI Interface. We will describe the functionality of each block.

3.2 Bus Interface (BI)

This block is responsible for interfacing the front-side bus of the processor and for filtering out transaction requests not emulated such as interrupts and I/O transactions. Transactions are identified and the proper information is queued in the BI according to the bus protocol. Once all of the information for a bus transaction has been collected the transaction is forward to the next block. Since requests on the bus may come in bursts we use an asynchronous queue to decouple the input and output of this block. If the system-under-test is a Pentium® III, the maximum request rate is every three bus clock (BCLK) cycles per request. For Pentium® 4 based systems, the maximum request rate is one request per 2 BCLK.

Since both Pentium® III and Pentium® 4 front-side bus are split transaction buses, the request and response phases can be separated in time to allow overlap transactions. There are distinct phases for front side bus - Arbitration, Request, Error, Snoop, Response and Data. There can be up to 8 total outstanding transactions across

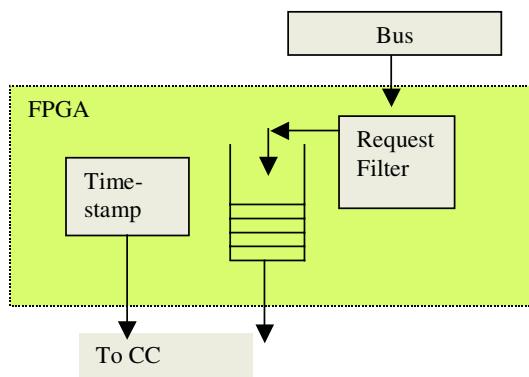


Fig. 3. Bus Interface

all agents at one time. Figure 3 depicts the major sub-blocks within the BI. The Request Filter sub-block in this figure is used to buffer the bus activities. Transaction may be completed out-of-order if they are deferred. This sub-block is also used to filter out I/O requests. Another block in the BI is a FIFO queue. It is used to average out the transaction rate and to buffer the front-side bus frequency from the internal transfer frequency. There is a time stamp unit. It provides the bus cycle field of the request information to the cache controller. With bus cycle time we are able to count the bus utilization rate. The output of the BI is a logical request with the following information:

BusCycle:ReqType:AgentID:Address

Transactions are removed from the queue at the request rate which has a maximum frequency of half the speed of the fastest front-side-bus (FSB) frequency. Many new FPGAs contain frequency synthesizers making the design easy and robust.

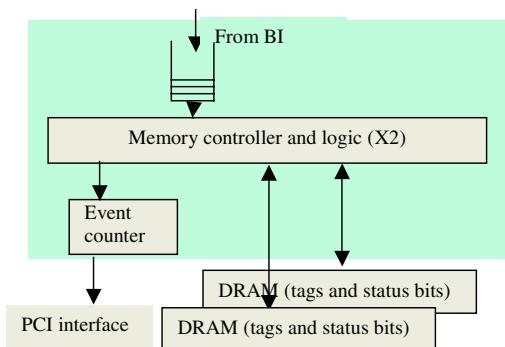


Fig. 4. Cache Controller (X4)

3.3 Cache Controllers and DRAM Interface

The second block is the cache controller. It consists of memory (DIMM) interface logic that performs 1) reading from DRAM; 2) comparing tags, state, etc. in parallel; 3) computing new LRU, state, etc.; 4) sending control signals to update statistic counters; and 5) writing modified tags back to DRAM. It also needs to prepare DRAM for the next access. Since most cache references do not alter the state of the cache, the last operation will only be performed when needed.

We estimated the number of cycles used for comparing tags and update states to be 5 cycles (each cycle is 7.5 ns). Since the width of the DIMM is 72 bits we will need to use multiple cycles to read or write the tags. Width of the tags is 8x4x4B. It will take 2 extra cycles to read and write the desired width. The total Read-Modify-Write access with Synchronous DRAM with these 5 cycles delay after read data is 18 cycles. This translates into a total of 135 ns. For P6 bus we may be able to live with 4-way lower bits address interleaving but for Pentium® 4 we must interleave 8-way. We will put two cache controllers on one FPGA since it is mostly I/O limited. Each DIMM interface will require about 100 pins. Putting 4 controllers on a single FPGA

will push the limit of its I/O pins. Instead we put 2 controllers on a FPGA and each controller will interface with a 256MB DIMM. It will be easier to design if we put the shared L3 on a separate memory controller. It will interface to event counters separately to collect statistics for shared L3 configuration. To simulate up to 1GB of L2 with different configuration we need to have 256MB of DRAM for each interleaved way. This setup will simulate from a 512MB direct map up to 4GB 8-way set-associative L2 cache for a quad-processor setup. If we use two 256MB DIMMs for shared L3 we will be able to simulate from 512MB direct map up to 16GB 8-way set-associative L3 cache. Figure 4 depicts the cache controller block diagram.

As mentioned, we collect cache simulation data in real time. Currently the HW\$im collects the following information: 1) Total References and total time; 2) Read/Write ratio; 3) Hit/miss rates; 4) Hit/HitM; 5) Deferred/retired; 6) Bus utilization rate; 6) Write backs and Replaced lines; 7) Address range histogram. More counters may be added if desired.

3.4 PCI Interface (PI)

The final block is the PCI (or console) interface. It is used to collect statistics collected during the simulation. Currently the PI is a passive device. Event counters are shadowed and read directly periodically. The period of the read can be adjusted.

4 Design Experiences

4.1 I/O and DCI

Xilinx Virtex™ II parts feature on-chip digitally controlled impedance (DCI) I/O. It enables designers to improve signal integrity without using external resistors for impedance matching on printed circuit boards. We employed DCI as part of the design allowing a local multi-drop bus to operate at speed. The signal level used is LVTTL through out the board since we need to interface with SDRAM as well as a PCI bridge chip. There is a translation from GTL+ to LVTTL at the LAI end. We use a part from Philips (GTL16612) requiring only a 3.3 V supply. The output of the GTL+/LVTTL translator is serial terminated to match the impedance of the cable. The bus connecting four parallel FPGAs is also serial terminated.

4.2 DCM and Clock Domains

It is important for a high performance designs to have efficient clocks. On the HW\$im board there are three clocking frequencies. First, there is the front side bus frequency of the system-under-test. All information on the front-side-bus is sampled with this frequency. Since the maximum request rate for Pentium® III is every three bus cycles we divide the bus clock by three. This is the second clock domain. Virtex™ II provides on-chip Digital Clock Manager (DCM) which make this process easy. Due to the length of the cable, signals arriving from the front-side-bus are shifted in phase from the on-chip clock signal generated with the DCM. Again the DCM used

provides means to phase shift the clock signal with easy interface. This divided clock is used to transfer data from the BI FPGA to cache controller (CC) FPGAs. On the CC FPGAs there are DCMs used to generate the frequency used to communicate with SDRAM. The flexible clock multiplier as part of the DCM does this. Finally there is the last clock domain used by PCI interface. We use the PCI clock generated by the PCI bus master on the host system to drive the bus between the PCI bridge part and CC FPGAs.

4.3 Timing Constraints

In order to operate at real-time several approaches are employed. First we included many timing constraints in the design User Constraints File (UCF) to ensure that the placed and routed design will meet timing requirements. Some examples are:

```
TIMESPEC "tsinputsspeed" = FROM "PADS" TO "FFS" 7.5 ns;
TIMESPEC "tsoutputspeed" = FROM "FFS" TO "PADS" 7.5 ns;
NET "BCLK" PERIOD = 7.5 ns;
```

Time ignores (TIGs) constraints are also used to remove violations that do not affect the design. When timing still cannot be met, extra pipeline stages are inserted to keep up the frequency. For the BI design we added two extra stages. For the CC FPGAs we added one extra pipeline stages.

4.4 Design Validation

Besides extensive simulation at the functional and timing (after place and route) level, we employed many design-for-test features to validate the correctness of the design. First extra pins were reserved for probing internal nodes. Since FPGA is re-programmable we are able to map internal nodes to these reserved probe pins for detail observation. Second, test-bench codes were programmed into the FPGA for on-line testing. These test-bench codes generate necessary input patterns and check the output for assertions. Many times simple pass or fail signaling is used to give quick feedbacks to the designer. For example we have a internal memory references generator that read from a file and feed the cache controller with known input pattern allowing us to debug the design. Third, a completely re-written set of test codes is used to check the hardware connectivity distinguishing completely hardware bugs from software bugs.

5 Cache Simulator

Just like any trace-driven cache simulator there are several challenges when we are observing behaviors at the front-side-bus level. These challenges are the result of mismatches between the system-under-test and the simulated system. First, a write to an E-state line by a processor cannot be seen on the bus by the simulated cache until a later time. Second, there are two coherence options when a read request hits a modified copy in a remote cache. With one of the option, after an implicit write-back,

both lines in the request CPU and the CPU with the modified line go into the shared state. Another option transfers both the data and the ownership from the current owner (with modified line) to the requester. Discrepancy may be created if the host cache and the simulated cache use different options.

5.1 Hidden Write-Shared: Due to Missing Write-Excl

Figure 5 illustrates the sequence of operations that may cause a discrepancy between simulated the cache and the real host system cache. Ci and Cj stand for CPU I and j. Initially both cache lines in Ci and Cj are invalid. A read by CPU Ci causes the line to go into the E state bringing the data into the cache. Another read by CPU Cj causes both CPUs to have the data in their respective caches with both lines set to the Share state.

Comd	Ci	Cj		Ci	Cj		Ci	Cj
Host	E	I	→	S	S	→	I	I
Simu	E	I		S	S		S	S
Comd	Ci	Cj		Ci	Cj		Ci	Cj
Host	→	I	Read	→	I	Write	→	E
Simu	S	S		S	S		recover E	I

Fig. 5. Example of a discrepancy caused by Missed Write to an Estate line

Since the simulated cache is usually larger in size, a line may be replaced in the real CPUs while still resides in the simulated machine. Later a new read follow by a write to the line causes the line to go from I to E to M states. During this sequence the simulated machine maintains the line in Ci and Cj at the S state. Only when a read by Ci later, which causes the real machine to issue a snoop hit on modify, will alert the simulated machine to recover its states. Before this happens all other read by any CPU to this line with the S state will not register the invalidation miss.

5.2 Hidden Read Miss and Write-Shared: When Host Uses S-S, but Simulated Cache Uses E-I

When the simulated machine uses the E-I coherency scheme while the real system uses the S-S also may cause discrepancy. When a read miss happens, the snoop result indicates that the line requested is at the M-state in another CPU's cache. There are two possible ways to change the state of these two lines. There is the E-I scheme that changes the line of the requested CPU to the E state and invalid the line holding the M-state. There is the S-S scheme that changes both lines to the S-state. When the system under test uses the S-S scheme and the simulated machine uses the E-I scheme there may be problems. This is similar to the case of missing write-excl and can be

handled in the same way. The difference is that the E-state is arrived when a read-miss hits a remote modified copy.

5.3 Recovery the Hidden Write-Shared by the Simulated Cache

As mentioned we can recover the hidden write-shared at a later time. Although such recovery can be very accurate, the delay of invalidation may affect the replacement in the simulated cache. In addition, the recovery scheme may add some complexity to the simulated cache. The recovery algorithm is described briefly here. When a read-miss hits multiple S-state lines in the simulated cache with HIT#/HITM# dis-asserted, change the state of the simulated cache to ES (a new state) and keep all other S-state copies. A hidden write-shared is recovered whenever a requested hit an ES-state in any of the simulated cache and a HITM# is asserted. In this case, all the S-state will be invalidated. The new line state in the requester's cache and the change of the ES state line will depend on the underline coherence protocol. A replaced dirty line that hits ES in the simulated cache will also cause a hidden write-shared recovery and an invalidation of all shared copies. This ES copy changes to M-state afterwards.

5.4 HW Cache Simulator Results

We have used the HW\$im to capture behaviors of programs. The following figures illustrates some of the capabilities of the HW\$im. Figure 6(a) shows the result of running a particular benchmark called SPECJbb [10]. First we can find out the cache miss ratio captured at a particular time unit for a level two cache of size 16MB organized as 8-way set associative cache with line size equals to 32B. We observe a change of behavior at about 60 seconds into the application run. This is due to garbage collection. Once the garbage collection is completed, access to the memory becomes more efficient in that the miss ratio decreased. Another interesting fact is that this application shows that the miss ratio changes dramatically from one time unit to the other. If our software simulation can only capture a small period of real time due to simulation speed then the results may be different depending on which time unit the data was collected.

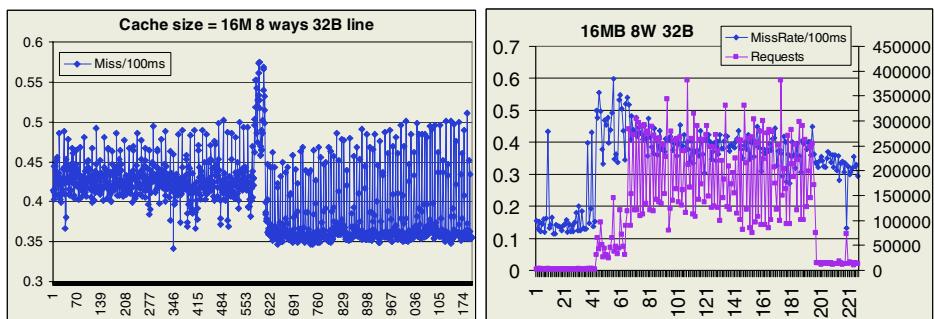


Fig. 6. Miss Ratio per 100 mille-second for - (a) SPECjbb; (b) MpegEnc

Figure 6(b) shows the HW\$im result of another application. This application is a multi-threaded media application. Besides the miss ratio we also show the number of requests send to the simulated cache. This program ran from the beginning to the end. Thus, we observe the change of request numbers during the run.

Figure 7 illustrates the effect of different cache size on miss ratio for a large vocabulary continuous speech recognition application software. This graph demonstrate that we are able to simulate very large cache size.

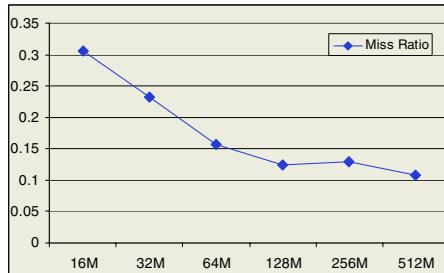


Fig. 7. Overall Miss Ratio of a Speech Recognition Application

6 Conclusion

A custom board with multiple FPGA used for future platform and memory hierarchy study is presented. This board operates at full system front-side-bus speed and can investigate real application behaviors on these possible future designs. This board is able running at full real-time speed due to the use of the state-of-the art FPGAs. It has been used to evaluate cache designs that are difficult to evaluate with current tracing technology.

References

- [1] M.A. Holiday, “Techniques for Cache and Memory Simulation Using Address Reference Traces,” International Journal in Computer Simulation, 1990.
- [2] Richard A. Uhlig and Trevor N. Mudge, “Trace-driven Memory Simulation: A Survey,” ACM Computing Surveys, 29 (2), 1997, pp. 128-170.
- [3] P. S. Magnusson et. al., “Simics: A Full System Simulation Platform,” IEEE Computer, February 2002, pp. 50-58. (<http://www.simics.com>)
- [4] S. A. Herrod, “Using Complete Machine Simulation to Understand Computer System Behavior
Ph.D. Thesis, Stanford University, February 1998. (<http://simos.stanford.edu/>)
- [5] Steve Guccione, “List of FPGA-based Computing Machines,” http://www.io.com/~guccione/HW_list.html
- [6] <http://www.quickturn.com/>
- [7] <http://www.aptix.com/>
- [8] Ashwini Nanda et. al., “MemorIES: a programmable, real-time hardware emulation tool for multiprocessor server design,” Proceedings of ASPLOS, 2000
- [9] http://www.xilinx.com/xlnx/xweb/xil_publications_index.jsp
- [10] <http://www.spec.org>

The Bank Nth Chance Replacement Policy for FPGA-Based CAMs

Paul Berube, Ashley Zinyk, José Nelson Amaral, and Mike MacGregor

Dept. of Computer Science, University of Alberta, Edmonton, Alberta, T6G 2E8, Canada,
`{berube, zinyk, macg, amaral}@cs.ualberta.ca`

Abstract. In this paper we describe a method to implement a large, high density, fully associative cache in the Xilinx VirtexE FPGA architecture. The cache is based on a content addressable memory (CAM), with an associated memory to store information for each entry, and a replacement policy for victim selection. This implementation method is motivated by the need to improve the speed of routing of IP packets through Internet routers. To test our methodology, we designed a prototype cache with a 32 bit cache tag for the IP address and 4 bits of associated data for the forwarding information. The number of cache entries and the sizes of the data fields are limited by the area available in the FPGA. However, these sizes are specified as high level design parameters, which makes modifying the design for different cache configurations or larger devices trivial.

Key words: field programmable gate array, cache memories, replacement policy, multizone cache, content addressable memories, Internet routing, digital design, memory systems

1 Introduction

The task of an Internet router is to determine which node in the network is the next hop of each incoming packet. Routers use a routing table to determine the next hop based on the destination address of the packet and on the routing policy. Thus a routing table is a collection of destination address/next hop pairs. When adaptive routing policies are employed, routers communicate with each other to update their routing table as the network changes.

Given a routing table configuration, the task of routing a packet consists on consulting the routing table to determine the next hop associated with the destination address. The delay incurred to perform this table lookup affects both the throughput of the network, and the latency to send a packet from a point A to a point B. Several techniques are available to implement the routing table lookup. Storing the table in memory and performing a software lookup may result in a prohibitively high routing delay and unacceptably low throughput. The lookup time can be improved by storing entries associated with frequently seen destinations in a fast hardware cache.

Such caches are standard in Internet routers. The table lookup in a router requires the implementation of the functionality of a content addressable memory (CAM) [2, 11]. While some network processors perform the CAM function using fairly conventional caching technology (NetVortex PowerPlant from Lexra [7], and Intel IXP1200 [6] for

instance), others implement fairly sophisticated CAMs. The IBM PowerNP processor provides an interface for an external implementation of a CAM [12]. The CISCO Toaster 3 network processor implements an on-chip ternary logic CAM [9]. The PMC-Sierra ClassiPI processor implements a very sophisticated CAM that stores not only the next hop associated with each destination address, but also a large set of rules that can affect the routing decisions [8].

Most network processors used in routers store all entries on a single CAM. The destination address in an incoming packet is formed by a prefix followed by wild cards or “don’t care” values that match any bit stored in the routing table. This paper is motivated by the idea of separating the entries in the routing table according to the length of the prefix. This separation is expected to be beneficial because incoming packet streams with different prefix length do not interfere with each other. Preliminary studies based on simulation indicate that we should obtain improved hit rates with a multi-zone CAM when compared with a monolithic implementation [3]. The idea of segregating the router table entries by prefix length is analogous to the idea of separating instruction and data streams into separate caches in general purpose processor architectures. In this paper we report the status of our design for a single zone CAM for IP lookup caching.

We developed a prototype for the content addressable memory (CAM) using the Xilinx Virtex-E family of FPGAs. In this paper we describe the implementation of this high-density ternary CAM. Section 2 introduces the overall architecture of our cache. Section 3 details how the Virtex-E FPGA features are used to implement the design. Implementation results are given in section 4. Section 5 describes some related work. Concluding remarks and future work follow in section 6.

2 Cache Architecture

A functional description of our design is presented in Figure 1. The prototype will be tested with a stream of router address lookups collected from network activities in actual routers. The router emulator sends destination addresses to the IP cache in the FPGA. The cache responds with the next-hop information, or indicates that a miss occurred. In the case of a miss, the router emulator sends the missing entry to the cache prototype. The router emulator tracks the number of hits and misses for an entire trace.

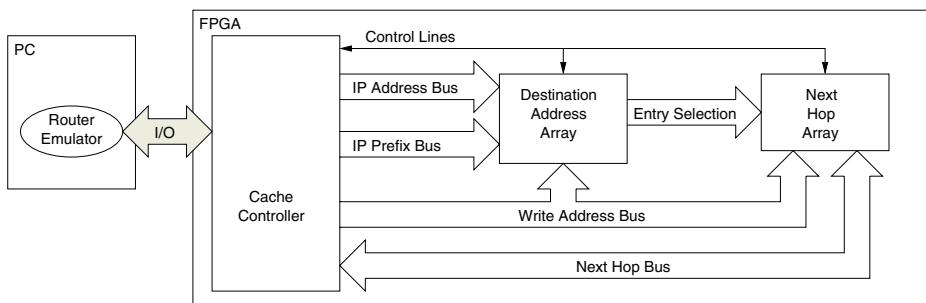


Fig. 1. A functional view of the cache design

2.1 Functional Description

Figure 1 presents a functional description of the IP caching system. The Cache Controller receives requests from the router emulator and returns lookup results. IP addresses are sent to the Destination Address Array (DAA) for matching through the IP Address Bus. The DAA stores IP address prefixes in which several of the least significant bits are set to don't care and thus match any value in the incoming IP address. The DAA is a fully associative ternary CAM. A match in the DAA produces an entry selection to fetch the corresponding next hop information from the Next Hop Array (NHA).¹ The next hop is sent through the Next Hop Bus to the Cache Controller that returns it to the Router Emulator.

If the IP address does not match any of the entries (IP address prefixes) in the DAA, a miss indication is sent to the cache controller, which returns a miss report to the router emulator. An update of the IP cache will be necessary. The emulator responds to the miss report with the length of the prefix that should be stored in the DAA and the next hop information. If there are empty (non-valid) entries in the CAM, one of them will be selected to contain the new address prefix/next hop information. If all the entries in the CAM are valid, the cache controller will select an entry for replacement according to the implemented replacement policy. The prefix of the IP address will be stored in the DAA and the next hop will be stored in the corresponding entry in the NHA.

2.2 Structural Description

Structurally the IP cache is divided into several blocks of entries. In our prototype each block stores 16 address prefix/next hop pairs. For a lookup, the cache controller sends the same IP address to all blocks and expects one of the blocks to reply with the value of the next hop. If a next hop is not found in any of the blocks, an entry will have to be selected for replacement. As shown in Figure 2 each block has a victim selection logic. This victim selection logic has a mechanism to keep track of recent references to CAM entries.

shown in Figure 3 we refer to all the entries that appear at position i on all blocks as the *bank i*. Each cache entry belongs to exactly one block and one bank. Our prototype design is modular at the block level, thus the number of blocks, n , can be varied to produce larger caches when more FPGA “real state” is available. However, to better use the space available in the VirtexE architecture, we use $k = 16$ entries per block.

3 Prototype Implementation

We will now present the implementation of our prototype design, showing how features of the VirtexE FPGA can be used to achieve a high cache density.

¹ In the current implementation of the prototype we do not handle the situation in which the address prefix matches multiple entries in the DAA.

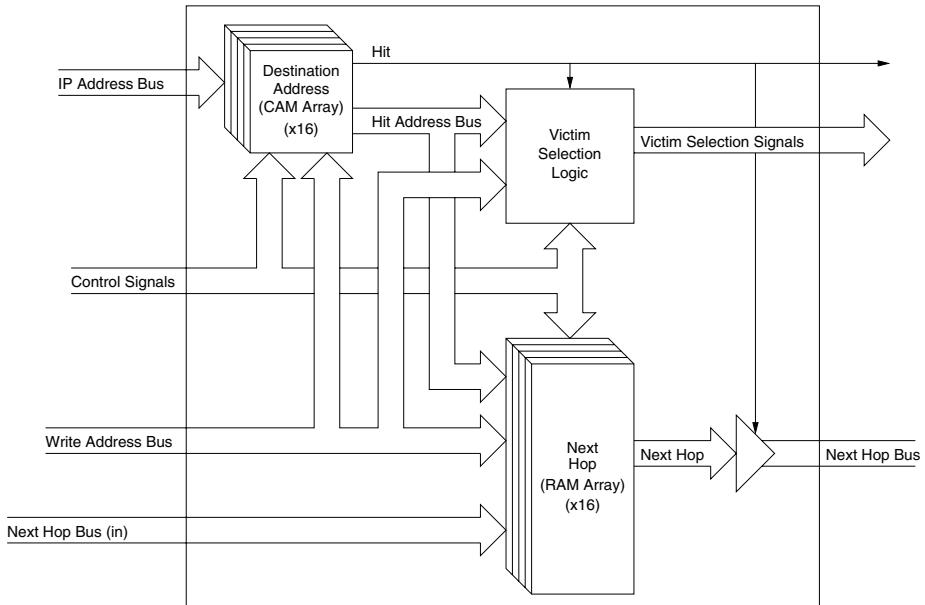


Fig. 2. The structure within a Block. All blocks share common input and output buses.

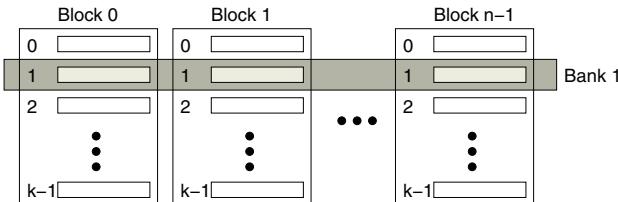


Fig. 3. The relationship between entries, blocks and banks.

3.1 Implementing CAM Cells with Shift Registers

In order to implement a 32-bit CAM cell to store an IP address in the DAA, we use 8 LUTs configured as 16-bit shift registers [1]. Each one of these LUTs will store the value of four of the IP address prefix bits as a one-hot encoding. For instance, if the four most significant bits of the address prefix are 1100, the LUT corresponding to these bits will store the value 0001000000000000. When a lookup occurs we use the incoming IP address to read the value stored in each shift register at the position specified. In this case the value read from the LUT will be 1 only when the incoming IP address has the value 1100 in its 4 most significant bits.

We want to be able to store address prefixes in the CAM cells, not only complete addresses. Let suppose that we want to store an address prefix that has in its 4 most significant bits the value 010X, where X indicate a *don't care* value, *i.e.* it should match either a 0 or a 1 value. In this case, the value stored in the LUT shift register corresponding

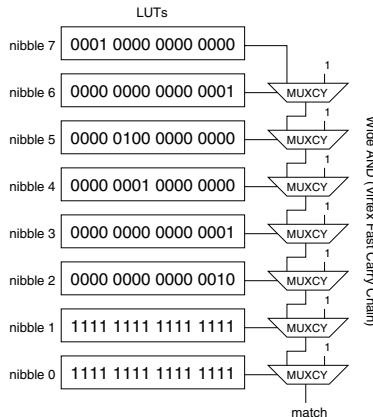


Fig. 4. The IP address prefix 192.168.1.0/24 is stored in the shift register LUTs. The most least significant bits, nibble 0 and nibble 1, correspond to the 8 don't care bits at the end of the prefix. The LUTs that store these don't care values are filled with 1s. A match is computed using a wide AND implemented using the Virtex carry chain MUXs (MUXCY).

to the four most significant bits is 000000000110000. Now an IP address with either 0101 or 0100 as its 4 most significant bits will read a 1 from that LUT.

A more concrete example is shown in Figure 4 where we illustrate the value stored in each of the 8 LUTs of a 32-bit CAM cell that contain the IP address prefix 192.168.1.0/24. This prefix has 24 bits and thus don't cares must be stored in the 8 least significant bits of the cell. In hexadecimal, the value stored in this cell is 0xC0A801XX. Notice that all the bits stored in the shift registers of LUTs 0 and 1 are 1s. Thus these two LUTs will return a 1 for any incoming address, producing the effect of a match with a don't care.

When a lookup is performed, the value read from each LUT shift register is fed into a wide AND gate implemented using Xilinx's carry chain building block as shown in Figure 4. If the output of the carry chain is a 1, the incoming IP address matches the prefix stored in the CAM cell.

The writing of an IP address prefix requires that the hardware compute the value of each bit to be stored in each LUT shift register. The circuit used to generate the input values for each shift register is shown in Figure 5. Values generated by a 4-bit counter are compared with the value in the IP address nibble corresponding to the LUT. If they match a value 1 is produced to be shifted into the shift register. In order to enable the storage of don't care values, a don't care mask is used to force a match for bits that are specified as don't care. This is done through the selection of either the counter value or the IP address nibble value to be sent to the comparator for each bit. If the IP address is sent, then it is compared with itself, ensuring a match. Since the don't care mask can be arbitrarily generated, the positioning and number of don't care bits is also arbitrary. However, in our application of IP routing, we generate don't care masks which correspond IP prefixes, where only the N least-significant bits contain don't care values.

Notice that a common single counter can be used for all the eight LUT shift registers that store a 32-bit IP address prefix. Moreover a single circuit for the computation of the

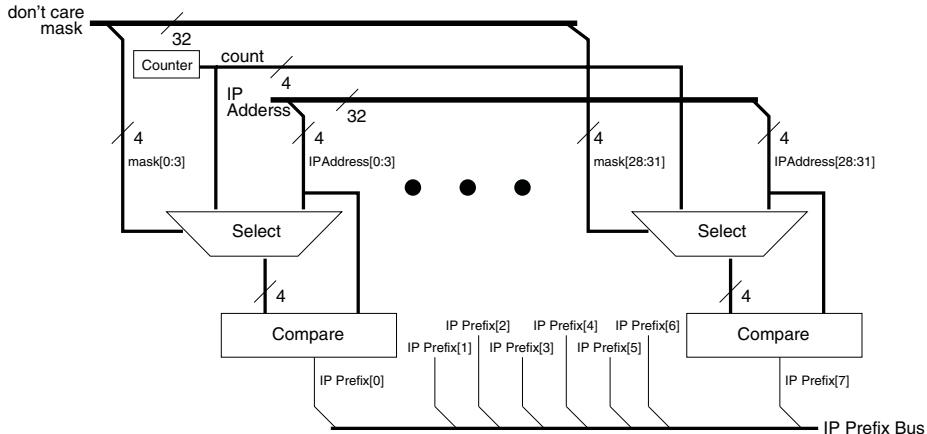


Fig. 5. Computation of the bits to store in the shift register for an IP address prefix nibble.

stored bits is implemented for the entire device. The bits generated by this circuit are written in the IP Prefix bus. The 8 CAM cells for the DAA entry that is performing a replacement will be the only ones selected to write the prefix bits generated.

A great advantage of our design is that we are able to implement a CAM without requiring a comparator for each cell. A simple reading of the 8 shift registers used for the cell will indicate if the IP destination address matched the prefix stored in the DAA. Moreover, each slice in the FPGA contains two flip-flops, thus if we were to use the flip-flops to store ternary values, we would need at least 32 slices for a 32 bit IP prefix. With our technique we can store the same prefix using only 8 LUTs, which requires only 4 slices. Furthermore, this design avoids the two levels of logic required to implement flip-flop based ternary comparisons, resulting in a much faster circuit.

3.2 Implementing the Next Hop Array Using SelectRAMs

The implementation of the NHA is composed of several 16x1 single port SelectRAMs, to achieve the desired data width for 16 entries. In our prototype implementation, we use a data width of 4 bits. When the IP cache is updated, the next hop value is written into the NHA concurrently with the update of the IP address prefix in the DAA. Since the NHA and DAA are co-indexed, they can share address and write-select lines. When a lookup is performed, the output from the NHA is through a high-impedance bus shared by all NHA blocks. Output to this bus is through a controlled buffer, with the control provided by the match signal from the corresponding entry in the DAA. In this design, we do not consider the possibility of multiple hits within a single zone of the cache. However, if multiple hits were possible, additional logic would be required to ensure that only one NHA entry supplied its output to the bus at any given time.

In a flip-flop based design, the NHA would require 4 flip-flops per entry (2 slices), or 32 slices per block. By using SelectRAM to store the next hop data, we require one LUT per bit per block, or 2 slices per block.

3.3 Replacement Policy

When a new entry must be stored in a cache memory and the cache is full, one of the existing entries must be selected for replacement. The algorithm used to select this *victim* entry is called a *replacement policy*. The goal of a replacement policy is to manage the contents of the cache, evicting cached values that are less likely to be referenced again in the near future to make room for a new value that was just referenced.

Standard replacement policies include first in, first out (FIFO), second chance and derivatives thereof. For very small caches a full implementation of least recently used (LRU) might be required. The second chance replacement policy associates a status bit with each entry. Second chance follows a modified FIFO discipline in which the status bit of an entry is checked when the entry is selected for replacement. If the status bit is 1, it is changed to 0 and the entry is not replaced. If the bit is 0, the entry is replaced [14].

Replacement policies can be implemented either as a parallel search, or as a serial search. Second chance is usually described as a serial search that loops through the cache entries until a valid victim is found. While a serial search reduces the amount of hardware components required to implement the cache, performance concerns dictate that a parallel search for a victim is preferable for a hardware IP address cache. However, a hardware implementation of the second chance policy with parallel victim selection is quite expensive. Thus we developed a new replacement policy that we called the *Bank Nth Chance* replacement policy.

The Bank Nth Chance Replacement Policy. The Bank Nth Chance replacement policy is an approximation of LRU and is similar to the second chance algorithm. As shown in Figure 3 the cache entries are organized in blocks and banks. Our replacement policy has a two level selection process. At the first level it uses a simple round-robin mechanism to select the bank that should produce a replacement *victim*. At the second level it uses an n -bit reference field to determine entries in the bank that are *valid victims* (in our prototype we use $n = 3$). A victim is valid if all its reference bits are equal to zero. The reference field of an entry is an n -bit shift register. Whenever the entry is referenced, a 1 is written into the most significant bit of the reference field. The entry is aged by shifting the field to the right by one position and writing a 0 into its most significant bit.

The cache controller maintains a Bank Pointer to select the bank from which the next victim will be chosen. If the bank selected by the bank pointer has no valid victims, all the entries of that bank are aged by shifting a 0 into their reference fields, and the Bank Pointer moves to the next bank. This process of aging the entries in a bank and moving the Bank Pointer will be referred to as an AGEANDADVANCE operation. After this operation is performed, one or more of the entries in the bank might become a valid victim. However, the bank will not be required to perform a victim selection until the next time it is selected by the bank pointer.

When a bank that has a valid victim is selected by the bank pointer, it is required to return the index of one of its entries. This entry will be used for the next replacement. The Bank Pointer is not moved, and thus the next attempt to select a victim will target the same bank. The cache controller selects a victim to be used for a new entry before a cache replacement is necessary. Whenever this pre-selected victim is referenced, the controller starts the selection of a new victim. This pre-selection of the entry reduces the average time required to perform replacements.

Implementation of Bank Nth Chance. The reference fields for the cache entries are stored in the FPGA LUTs configured as 16×1 SelectRAMs. Each block of 16 entries requires three SelectRAMs to store its 16 3-bit reference fields. The Block Reference Field Logic (BRFL), shown in Figure 6, performs victim selection and entry aging for each block of entries. The bank number is used to determine which entry within a block is to be examined, referenced, or aged. Each one of the SelectRAMs stores one bit of the reference field for 16 entries. All the entries of a bank are aged at the same time. Therefore the same bank number is input in all blocks, the Age signal is asserted and the Reference signal is negated causing the reference field to shift and writing a zero in the Field2 SelectRAM. When an entry is referenced, the entry's bank number is inputted in the victim selection logic of the entry's block, the Reference signal is asserted and the Age signal is negated, causing a 1 to be written in the entry's position in the Field2 SelectRAM. The organization of the cache entries in blocks and banks allows for the simultaneous access of all entries in a bank during the victim selection process. It also allows for the simultaneous aging of all entries in a bank. The victim selection logic is shown in Figure 7. A BRFL contains the Block Reference Field Logic shown in Figure 6. When several blocks contain valid victims for the selected bank, a priority encoder is used to select which one of these blocks will be used for the replacement. The priority encoder also outputs a signal to indicate whether a valid victim was found.

The motivation for the use of the round-robin discipline to visit banks should now be evident. Our design allows for a single bank to be visited at a time. Therefore it makes

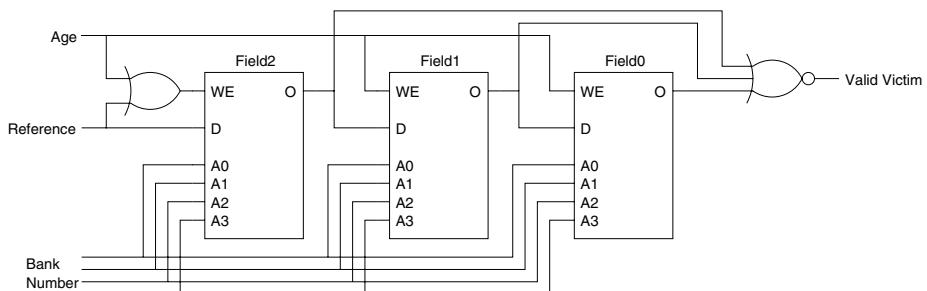


Fig. 6. Block Reference Field Logic (BRFL): Logic for victim selection and entry aging in the reference field of a block.

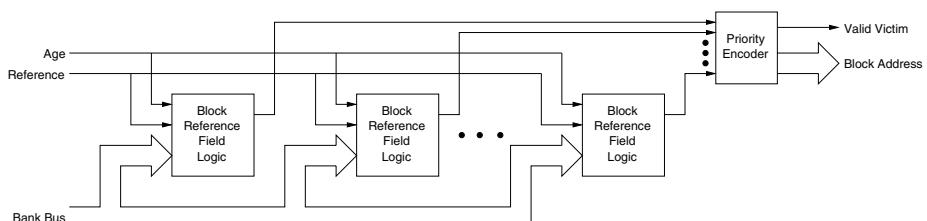


Fig. 7. Victim Selection Logic operating within a bank and across blocks.

sense to not attempt to select a victim from a bank that has recently reported that it had no more valid victims. Using our Bank Nth Chance replacement policy we maximize the inter-visit time for the same bank and thus avoid repeating searches for valid victims in the same set of entries often.

The use of three bits in the reference field builds a better approximation to the LRU policy than the 1-bit second chance algorithm. The round robin selection of banks and the repeated selection of a victim from a bank until the bank has no more victims is an implementation convenience that allows a good trade-off between an expensive fully parallel search for a valid victim and a slow sequential search.

Space Saving of Nth Chance. If flip-flops were used to store the reference fields, then 48 flip-flops would be required for one 16 entry block. Furthermore, the shift register control logic and valid victim check logic, which requires 3 LUTs, would need to be replicated for each entry, using 48 LUTs (24 slices) instead of 3 (2 slices). In total, each block in the flip-flop implementation requires 24 slices, compared to 3 slices when using SelectRAMs, resulting one eighth the space requirement.

4 Implementation Results

In this section we present design measurements obtained in the prototype for the Virtex 1000E device. These measurements can be summarized as follows:

Density: When compared with an implementation of a ternary memory based on flip-flops, our LUT/shift register based implementation is 8 times denser.

Cache Size: Our design allows for a 1.5K entry cache to be implemented in a Virtex 1000E FPGA.

Lookups/Second: The largest cache implemented in the Virtex 1000E can perform 16 million lookups per second. Smaller caches have better performance.

Lookup vs. Update Latency: Our cache architecture implements a 2 cycle lookup, and an 18 cycle write.

4.1 Design Density

We attain space saving in our design in the storage of the destination addresses through the use of LUTs configured as shift registers as opposed to flip-flops. With LUTs, each 32-bit entry requires 4 slices compared with 32 slices for a flip-flop based implementation.

Another source of space saving is the implementation of the victim selection logic in the cache controller. Instead of storing the reference bits in flip-flops, we used SelectRAMs for this storage. Consequently, our 3-bit fields require 1.5 slices per block, rather than 1.5 slices per entry, a 16 fold increase in density.

The simplicity of the Nth bank chance replacement policy allowed for a much simplified logic to update the reference field bits. Since the bits are stored in SelectRAMs, the same control logic is shared by 16 entries. In a flip-flop implementation, each field would require its own logic.

Combining the savings in all these aspects of the design, we achieved a bit storage density that is at least 8 times higher than could be achieved with a straightforward flip-flop based implementation in our Virtex 1000E device.

4.2 Cache Size

In this section we provide an estimate of the maximum cache sizes that could be implemented in several other devices in the Virtex family using our design. To obtain these estimates, we used simple linear regression on our actual implementation results for caches of up to 1024 entries. We found that device utilization could be broken up into 7.6 slices per entry for size-dependent storage and logic, and a fixed cost of approximately 175 slices for interface and control logic. Our estimates are presented in Table 1. We also present an upper bound on achievable cache size for a flip-flop based implementation. This upper bound is calculated based only on space requirements to implement the CAM cells in the DAA, store the contents of the NHA, and implement the reference fields for each entry. Additional logic that would be required for address encoding and decoding, and control, is ignored. Table 2 shows actual device utilization and speeds for various sizes of caches implemented in the Virtex 1000E. Our maximum cache size in the 1000E of 1568 entries closely matches our estimate of 1600 entries.

4.3 Lookup Throughput

Our implementation of a 1568 entry cache in the Virtex 1000E FPGA operates at 33 MHz. A lookup operation requires 2 cycles, so this cache can process 16 Million lookups per

Table 1. Estimates of the maximum cache sizes for each Xilinx VirtexE device. Flip-Flop estimates are based on bit storage space alone. LUT estimates are based on fixed and per-entry device utilization estimates from linear regression analysis of actual implementation results of complete caches.

Device		Maximum Cache Size	
Name	Slices	Flip-Flops	LUTs
50E	768	16	64
100E	1200	32	128
200E	2352	64	272
300E	3072	80	368
400E	4800	128	608
600E	6912	192	880
1000E	12288	336	1600
1600E	15552	432	2016
2000E	19200	528	2512

Table 2. Device usage and clock speed for various cache sizes in the Virtex 1000E.

Cache Entries	Slices Used	Speed (MHz)
64	659	70.2
128	1143	56.1
256	2121	50.5
521	4035	45.2
1024	7934	37.5
1568	21061	33.0

second. Therefore, even in its prototype incarnation, our cache should be able to handle current traffic in Internet edge routers, or the load of an OC-192 interface. When this design is used in a multizone cache design, each zone will be a smaller cache, and so will run at a higher frequency, as shown in Table 2.

4.4 Lookup and Update Latencies

A lookup in our cache requires reading from the Destination Address Array CAM cells. If a match occurs, data is output from the Next Hop Array. Although this lookup datapath is a simple combinatorial circuit, we break the datapath in two stages to allow for a higher clock frequency. This higher clock frequency benefits cache updates and victim selections. As a result, each lookup requires two clock cycles to complete. As our investigation of the design continues, and performance results become available, a single cycle lookup at a lower clock frequency may prove to increase overall performance.

Each write into the DAA requires 18 cycles. One cycle is required for setup, 16 to calculate and shift each of the 16 bits into the LUT shift registers, and a final cycle to complete the operation. The longer delay for updates is justified by the higher storage density, faster lookups due to the lack of comparison logic, and our very inexpensive implementation of ternary logic.

Furthermore, after the initial cycles of the write operation, the cache controller can begin searching for a new victim. In total, three cycles are required to search for a new victim. One cycle is required to age the entries in the current bank, another to advance the bank pointer and a final cycle to check the new bank for a valid victim. Four banks can be searched while the write completes. Initial software simulation indicates that, on average, between 1 and 2 banks will need to be searched to find the next victim. Consequently, victim selection should have a minimal impact on the performance of the cache.

5 Related Work

Our work is based on the prior work of MacGregor and Chvets [3, 13], who introduced the concept of a multizone cache for IP routing. Their work involved developing the statistical foundation for the multizone concept, and supporting simulations. We are building on this foundation by implementing the hardware proof-of-concept.

In the design of their Host Address Range Caches (HARC), Chiueh and Pradha show how the routing table can be modified to ensure that each IP address matches with a single IP prefix [4, 16]. They propose to “cull” the address ranges in the IP address space to ensure that each range is covered by exactly one routing table entry. The use of a similar technique would circumvent the single match restriction of our current design.

Gopalan and Chiueh use programmable logic to implement Variable Cache Set Mapping to reduce conflict misses in set-associative IP caches [10]. Their trace-driven simulation studies confirms that IP caches benefit from higher degrees of associativity thus lending support to our decision of designing a fully-associative cache that eliminates all conflict misses.

Ditmars implemented a system for IP packet filtering in firewalls and routers [5]. This system uses a CAM in an FPGA to match IP packets to filtering rules. However, this

system has fewer words in the CAM, which changed comparatively slowly over time. Therefore, this CAM could be implemented using software to partially reprogram the device to “hard code” the CAM contents according to a software-based replacement policy, while our cache must be fully implemented in hardware.

Commercial ternary content addressable memories (TCAM) is available from SiberCore Technologies [15]. Their TCAMs are implement with ASIC technology. The fastest device available can perform 100 million lookup/second. The largest CAM available can store 9 Million Trits.²

6 Conclusion

We have nearly completed the implementation of a high density cache in an FPGA. In the Xilinx Virtex 1000E FPGA we can store 1568 32-bit entries, each with 4 bits of associated data and another 3 bits used by the replacement policy. This design runs on a 33MHz clock, and requires only 2 cycles to perform a lookup. Furthermore, we have introduced the new Bank Nth Chance replacement policy, which is well suited to the properties of a LUT-based FPGA. Bank Nth Chance carefully compromises between the speed of a parallel search circuit and the compact size of a sequential search circuit, while maintaining a hit rate similar to that of true LRU.

References

1. Jean-Louis Brelet and Bernie New. *Designing Flexible, Fast CAMs with Virtex Family FPGAs*. Xilinx,
<http://www.xilinx.com/xapp/xapp203.pdf>, version 1.1 edition, September 1999.
2. L. Chivin and R. Duckworth. Content-addressable and associative memory: Alternatives to the ubiquitous ram. *IEEE Computer Magazine*, 22(7):51–64, July 1989.
3. Ivan Chvets. Multi-zone caching for ip address lookup. Master’s thesis, University of Alberta, Edmonton, AB, 2002. Computing Science.
4. Tzicker Chiueh and Prashant Pradhan. High performance IP routing table lookup using CPU caching. In *IEEE INFOCOMM (3)*, pages 1421–1428, 1999.
5. Johan M. Ditmar. A dynamically reconfigurable fpga-based content addressable memory for ip characterization. Master’s thesis, Royal Institute of Technology, Stockholm, 2000.
6. P. N. Glaskowsky. Intel’s new approach to networking: Follow-on to ixp1200 features new cores, new organization. *Microprocessor Report - www.MPRonline.com*, October 2001.
7. P. N. Glaskowsky. Lexra readies networking chip: new netvortex powerplant augments lexra’s ip business. *Microprocessor Report - www.MPRonline.com*, June 2001.
8. P. N. Glaskowsky. Reinventing the router engine: Pmc-sierra’s classipi defies easy classification. *Microprocessor Report - www.MPRonline.com*, March 2001.
9. P. N. Glaskowsky. Toaster3 pops up at mpf: Cisco details world’s first 10 gb/s network processor. *Microprocessor Report - www.MPRonline.com*, October 2002.
10. Kartik Gopalan and Tzicker Chiueh. Improving route lookup performance using network. In *Proc. of SC2002 High Performance Networking and Computing*, Baltimore, MD, November 2002.
11. T. Kohonen. *Content-addressable memories*. Springer-Verlag, New York, 2nd edition, 1987.

² A *trit* is a ternary bit as defined by Herrmann in the late 1980s [17].

12. K. Krewell. Rainier leads powernp family: Ibm's chip handles oc48 today with clear channel to oc192. *Microprocessor Report* - www.MPRonline.com, January 2001.
13. M. H. MacGregor. Design algorithm for multi-zone ip address caches. In *IEEE Workshop on High Performance Switching and Routing*, Torino, Italy, June 2003.
14. A. Silberschatz and P. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, MA, fifth edition, 1998.
15. SiberCore Technologies. How a tcam co-processor can benefit a network processor. <http://www.sibercore.com>.
16. Prashant Pradhan Tzi-Cker Chiueh. Cache memory design for network processors. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 409–419, Toulouse, France, January 2000.
17. Joh Patrick Wade. *An Integral Content Addressable Memory System*. PhD thesis, Massachusetts Intitute of Technology, May 1988.

Variable Precision Multipliers for FPGA-Based Reconfigurable Computing Systems

Pasquale Corsonello¹, Stefania Perri², Maria Antonia Iachino¹
and Giuseppe Cocorullo²

¹ Department of Informatics, Mathematics, Electronics and Transportation
University of Reggio Calabria, Loc. Vito de Feo, 88100 Reggio Calabria.
corsonello@ing.unirc.it

² Department of Electronics, Computer Science and Systems
University of Calabria, Arcavacata di Rende - 87036 - Rende (CS)
perri@deis.unical.it
g.cocorullo@unical.it
Italy

Abstract. This paper describes a new efficient multiplier for FPGA-based variable precision processors. The circuit here proposed can adapt itself at run-time to different data wordlengths avoiding time and power consuming reconfiguration. This is made possible thanks to the introduction of on purpose designed auxiliary logic, which enables the new circuit to operate in SIMD fashion and allows high parallelism levels to be guaranteed when operations on lower precisions are executed. The proposed circuit has been characterised using VIRTEX XILINX devices, but it can be efficiently used also in others FPGA families.

1 Introduction

The rapid increase in transistor count gives processor hardware the ability to perform operations that previously were only supported in software. The new computing paradigm that promises to satisfy the simultaneous demand for computing performance and flexibility is *Reconfigurable Computing* [1].

Reconfigurable computing utilizes hardware that can be adapted at run-time. The ability to customise the hardware on-demand to match the computation and the data flow of a required specific application has demonstrated significant performance benefits with respect to general-purpose architectures. As an example, thanks to the adaptability, reconfigurable architectures can exploit parallelism available in the matched application. This provides a significant performance advantage compared to conventional microprocessors.

Reconfigurable architectures are mainly based on Field Programmable Gate Arrays (FPGAs). An FPGA is a configurable logic device consisting of a matrix of programmable computational elements and a network of programmable interconnects. However, the development of Systems-on-Chips (SoCs) has been

recently demonstrated in which configurable logic and ASIC components are integrated [2, 3]. Among the existing families of FPGA the SRAM-based ones are the obvious candidates to the realization of evolvable hardware systems. In fact, SRAM-based configurable logic devices are able to adapt their circuit function to the computational demands downloading configuration data (i.e. bit-stream) from an external memory. Changing the bit-stream (i.e. reconfiguring the chip) allows both the functionality of the computational elements and the connections to be rearranged for a different circuit function.

Unfortunately, the flexibility achieved by reconfiguration is not for free. In fact, downloading a new bit-stream onto the chip needs a relatively long time. Moreover, during reconfiguration the logic has to stop computation. To partially solve these problems, FPGA architectures have been developed that support run-time reconfiguration [4, 5, 6]. The latter guarantees two advantages to be obtained: the time required for reconfiguring the chip is reduced; the portion of the chip that is not being reconfigured retains its functionality. Nevertheless, the reconfiguration time still could constitute a bottleneck in the design of an entire reconfigurable system.

Significant improvements can be obtained using an on-chip reconfiguration memory cache [7, 8], in which a few configurations can be stored. In this way, the time required to switch from one configuration to the next one is reduced with respect to the case in which bit-streams are loaded by an external memory.

Multimedia applications are becoming one of the dominating workloads for reconfigurable systems. Many examples of multimedia accelerators based on FPGAs are nowadays available. To achieve real time processing of media signals appropriate architectures supporting parallel operations (SIMD parallelism) on multiple data of 8-, 16- and 32-bit are needed [9, 10]. They must be able to run-time adapt data-size and precision to the computational requirements, often at instruction level. This means that, as an example, a 32-bit native precision circuit should be run-time substituted by two 16-bit or by four 8-bit circuits able to perform the same computation, previously executed using the 32-bit precision. In order to do this the hardware should be reconfigured every time the precision changes.

All the above-mentioned solutions do not allow reconfiguring the hardware at instruction level. For these reasons, architectures able to fast run-time adapt themselves to different precisions without loading new bit-streams from memories are highly desired inside FPGA-based designs. The chip could be actually reconfigured just when the running function is not required for a properly long time.

In this paper a new circuit is presented for realizing variable precision multiplication with high sub-word parallelism levels for use within high-density SRAM-based FPGAs. The proposed circuit run-time adapts its structure to different precisions at instruction level avoiding power and time-consuming reconfiguration.

The new multiplier has been characterised using VIRTEX XILINX devices, but it can be efficiently used also in others FPGA families. Moreover, the method demonstrated here can be successfully applied for the realization of others variable precision arithmetic and logic functions.

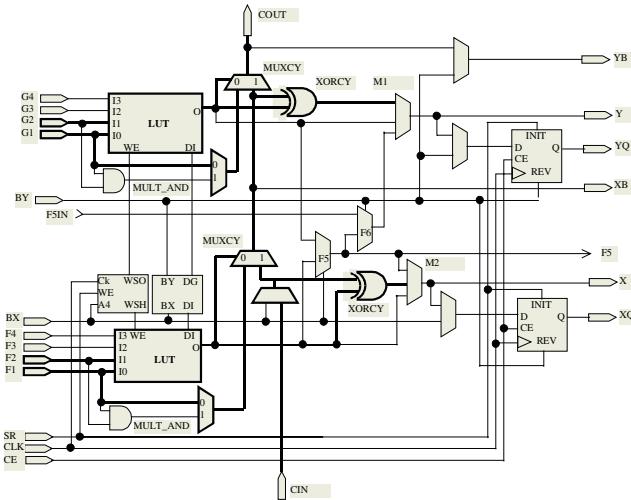


Fig. 1. Schematic diagram of the VIRTEX slice.

2 XILINX Virtex FPGA

VIRTEX, VIRTEX II and VIRTEX II PRO FPGA families developed by XILINX [5] are among the most used architectures in the design of systems for reconfigurable computing. They are SRAM-based FPGAs consisting of configurable logic blocks (CLBs) and programmable interconnects. The former provide the functional elements for constructing the user's logic, whereas the latter provide routing paths to connect the inputs and outputs of the CLBs exploiting programmable interconnection points (i.e. pass-transistor switches).

VIRTEX devices have a certain number of CLBs, ranging from 384 to 6144, each one consisting of two slices (S0 and S1) organized as shown in Fig.1. Each slice contains two look-up tables (LUTs), usable as logic function generators, two flip-flops and additional AND components (MULT_AND), multiplexers (MUXCY) and exclusive OR (XORCY) gates for high-speed arithmetic circuits. MULT_AND gates are used for building fast and small multipliers. MUXCY multiplexers are used to implement 1-bit high-speed carry propagate functions. XORCY gates are special XOR gates usable for generating 1-bit high-speed sum.

In the following, attention will be concentrated on these auxiliary elements to explain how they are used in the realization of high-speed carry chains, which are the basic elements of any arithmetic circuit.

When 1-bit full-adders are implemented, each LUT in the slice receives two operand bits (e.g. a_k and b_k) as input and generates the corresponding propagate signal $p_k = a_k \text{ XOR } b_k$. The LUT output drives the selection input of MUXCY and one input of XORCY, which generate the carry-out and the sum bit of the full-adder, respectively. Using this full-adder scheme the length of the carry chain is two bits for

slice and the fast carry propagation occurs in a vertical direction through the special routing resources on the CIN and COUT carry lines of the slice.

By means of an appropriate manual relative placement, a simple n -bit ripple-carry adder can be easily made able to optimally exploit these special resources placing its full-adders into one column of $\frac{n}{2}$ adjacent slices [5]. It is worth pointing out that carry lines do not contain programmable interconnection points. Therefore, these dedicated routing resources are much faster than the global routing ones.

For designing efficient arithmetic circuits on the FPGA platform special expertise is required. In fact, the optimisation phase in FPGAs based designs needs very different efforts compared to the ASIC designs. For example, as is well known, for ASIC designs, the ripple-carry scheme leads with the smallest but also the slowest addition circuit [11]. Thus, it is not appropriate when achieving high-speed performance is the target. On the contrary, a simple rippling-carry chain is the best choice when the adder has to be realized in FPGA devices. There, conventional alternative accelerated schemes, such as carry-select and carry-look-ahead, are usually meaningless due to the fact that they require the use of low-speed general routing resources.

3 The Proposed Implementation

Multiplier architectures optimised for ASIC designs are typically inadequate to gain advantages from dedicated carry-propagate logic and fast routing resources available in FPGA devices. As an example, schemes proposed in [12-14] based on the Booth coding result unsuitable for high-performance designs on FPGA platforms. On the contrary, it is well known that the Baugh-Wooley algorithm [15] allows the special resources available into FPGAs to be efficiently exploited. Moreover, no efficient architectures for realising variable precision SIMD multipliers have been yet proposed and optimised for FPGA based systems.

Typically, a NxN variable precision SIMD multiplier is able to execute either one NxN multiplication or two (N/2)x(N/2) or four (N/4)x(N/4) multiplications. The obtained results are usually stored into a 2N-bit output register.

The design of the new variable precision multipliers with N=32 is based on the following considerations. $A_{[31:0]}$ and $B_{[31:0]}$ being two 32-bit operands, the following identity is easily verified:

$$\begin{aligned} A_{[31:0]} \times B_{[31:0]} = & SH16L(SH8L(A_{[31:0]} \times B_{[31:24]}) + S48E(A_{[31:0]} \times B_{[23:16]})) + \\ & + S64E(SH8L(A_{[31:0]} \times B_{[15:8]}) + S48E(A_{[31:0]} \times B_{[7:0]})) \end{aligned} \quad (1)$$

where SHyL and SzE indicate a left shift by y bits and an extension to z bits, respectively.

It is useful to recall that an n -bit number can be easily extended to $(n+h)$ bits considering that: i) an n -bit unsigned number can be represented as a positive $(n+h)$ -bit signed number having all the h additional bits set to 0; ii) an n -bit signed number

can be represented as a signed ($n+h$)-bit number having all the h additional bits equal to the n^{th} .

The proposed architecture uses four 32×8 multipliers and it can execute one 32×32 or one 32×16 multiplication or two 16×16 or four 8×8 operations.

In order to manipulate both signed and unsigned operands also for the lower precisions, the 32×8 multipliers have to be properly organized. For the target FPGA devices family an extremely efficient kind of automatically core generated macros exists which is useful to this purpose: the controlled by pin x signed multiplier (CBPxS). An $n \times m$ CBPxS macro is able to perform signed-signed, unsigned-unsigned or signed-unsigned multiplications extending the operands to $(n+1)$ and to $(m+1)$ bits, respectively. The extension of the n -bit operand is internally executed by the macro itself on the basis of a control signal, which indicates whether that operand is a signed or an unsigned number. On the contrary, the m -bit operand has to be externally extended. This leads with a $n \times (m+1)$ multiplier which generates an $(n+m)$ -bit output.

In Fig.2 the block diagram of the new variable precision SIMD multiplier is depicted. It can be seen that its main building modules are:

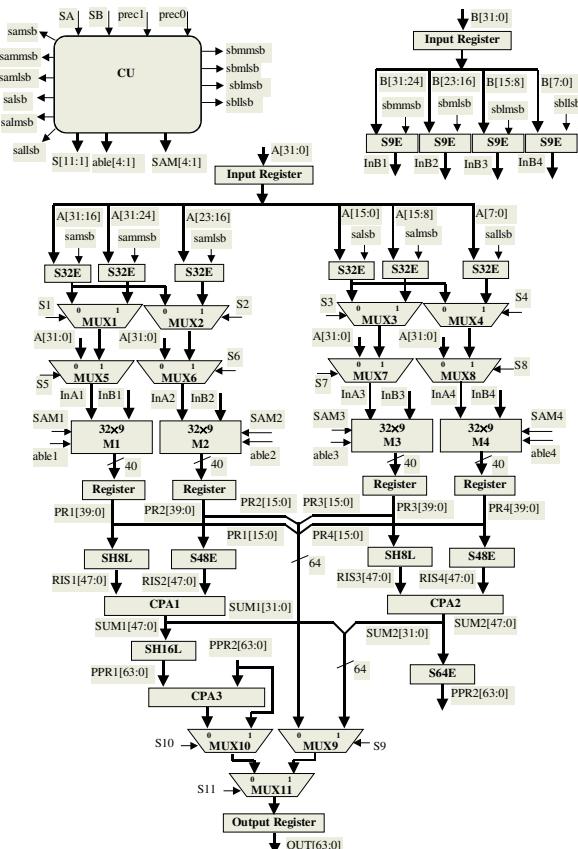


Fig. 2. Architecture of the proposed variable-precision multiplier

- A control unit (CU) needed to properly set the computational flow on the basis of the required precision;
- Four 32×9 CBPxS multipliers (M1, M2, M3 and M4) each generating 40-bit output;
- Two 48-bit and one 64-bit carry propagate adders (CPA1, CPA2 and CPA3).

The CU receives the signals prec1, prec0, SA, and SB as input and generates all the signals needed to orchestrate extensions and data flow. The signal SA indicates if the operand A is a signed ($SA=1$) or an unsigned ($SA=0$) number. The same occurs for the operand B on the basis of SB. Conversely, the signals prec1 and prec0 serve to establish the required precision. It is worth underlining that we suppose the above control signals available at runtime. Thus, also the required precision of different operations can be determined at runtime.

From Fig.2, it can be seen that to correctly form the data InAi and InBi input to the multipliers Mi (for $i=1,\dots,4$), the 16-bit and the 8-bit subwords of operand A are extended to 32 bits, whereas the 8-bit subwords of the operand B are extended to 9 bits. For any required precision the extended subwords of operand B have to be input to the four multipliers. Conversely, for operand A different conditions occur depending on the required precision. When operations on lower precision data have to be executed the multiplexing blocks MUXj, with $j=1,\dots,8$, allow the extended subwords of operand A to be input to the multipliers. On the contrary, when the highest precision multiplication has to be executed those multiplexing blocks select the entire operand A as input for all the multipliers.

In order to orchestrate the extension of the operands subwords and the data flow through the blocks MUXj, the CU generates appropriate control signals. SAmsb, SAmmsb, SAmlsb, SAlsb, SALmsb and SALlsb indicate if the modules for extension S32E have to treat the subwords A[31:16], A[31:24], A[23:16], A[15:0], A[15:8] and A[7:0], respectively, as signed or unsigned numbers. Analogously, SBmmsb, SBmlsb, SBllmsb and SBlllsb indicate if the modules S9E must consider the subwords B[31:24], B[23:16], B[15:8], and B[7:0], respectively, as signed or unsigned numbers. The CU also generates the signals S_r ($r=1,\dots,11$) and SAM_r . The former are used as the select input of the multiplexing blocks MUX_r, whereas the latter indicate if the data InA_i input to the multipliers Mi are signed or unsigned numbers. The above control signals allow the required operation to be matched and the appropriate data flow to be set.

When 32×32 multiplications are executed, the multipliers M1, M2, M3 and M4 calculate in parallel the products between the operand A and the subwords B[31:24], B[23:16], B[15:8] and B[7:0] all extended to 9 bits. Then, the 40-bit results PR1 and PR3 are left shifted by 8-bit positions, whereas the 40-bit products PR2 and PR4 are extended to 48 bits. The two 48-bit words Ris1 and Ris2 obtained in this way are added by means of the carry-propagate adder CPA1, which generates the 48-bit word SUM1. Contemporaneously, Ris3 and Ris4 are added by CPA2 forming the word SUM2. Then, SUM1 is left shifted by 16 bit positions thus forming the 64-bit partial result PPR1, whereas SUM2 is extended to 64-bit thus forming the partial result

PPR2. To generate the final 64-bit result OUT[63:0], PPR1 and PPR2 are added by CPA3.

It can be noted that the hardware used could support the execution of two 32x16 multiplications. However, a 64-bit output register is used, thus only one 32x16 multiplication result can be accommodated in this register. When this operation is required just two multipliers must operate, whereas the others can be stopped. We have chosen to make M3 and M4 able to go into action and to stop M1 and M2. To control this particular case the signals able1, able2, able3 and able4 are used. When able1 and able2 are low all the 40 bits of PR1 and PR2 (and consequently also the 48 bits of Ris1 and Ris2) are forced to zero. Therefore the whole result Out[63:0] is equal to PPR2, which is directly outputted by means of MUX10 and MUX11, without passing through CPA3.

When operations on 16-bit data are executed, the multiplexing blocks MUX9 and MUX11 allow the less significant 32-bit of the independent results SUM1 and SUM2 to be directly outputted without passing through CPA3. Analogously, when multiplications on 8-bit data are requested, MUX9 and MUX11 allow the less significant 16-bit of the independent results PR1, PR2, PR3 and PR4 to be directly outputted without passing through the addition modules CPA1, CPA2 and CPA3.

It is worth underlining that in order to optimally exploit logic and routing resources available in the target FPGA also for the new multiplier appropriate placement constraints have been used. In this case the RLOC attributes available at the schematic level have been combined with the PROHIBIT directives imposed by means of the .ucf and .pcf files [5]. This allowed the extremely compact layout shown in Fig. 3 to be obtained.

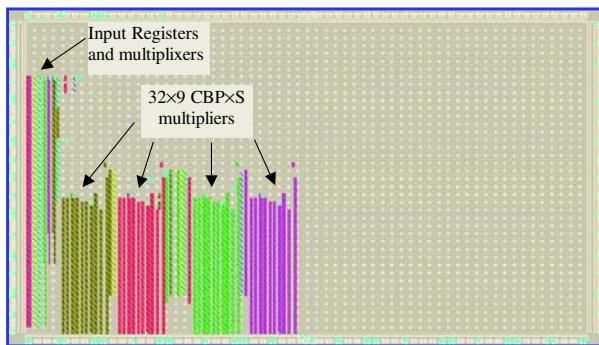


Fig. 3. Layout of the proposed variable-precision multiplier

4 Results

The new variable precision multiplier has been implemented and characterized using a FPGA XILINX VIRTEX XCV400 device. For purposes of comparison, we realized a 32x32 conventional fixed-precision core-generated multiplier. It is worth noting that core-generation tool does not allow the number of registers used in the multiplier

pipeline to be arbitrarily chosen. Just minimum and maximum pipelining are allowed. To make an effective comparison minimum pipelining has been chosen. Moreover, two further variable precision schemes have been realized and compared using the design criterion described in the above Section. The first one uses two 32x17 core multipliers and the second one utilizes two 16x17 multipliers. Their characteristics are summarized in Table 1.

Table 1. Performance comparison.

	Delay [ns]	Power [mW/MHz]	Slices	Latency [cycles]	MOPS 8x8	MOPS 16x16	MOPS 32x16	MOPS 32x32
New Mult. 32x9	23	19.5	1108	2	174	87	43.5	43.5
New Mult. 32x17	30.9	15.8	919	2	64.7	64.7	32.4	32.4
New Mult. 16x17	28.7	13.4	729	2	69.7	69.7	34.9	17.5
Reference Mult.	30.7	12.4	680	1	-	-	-	32.6

It can be easily observed that the new multiplier using 32x9 core multipliers assures the fully parallelism on low precision data to be obtained. Therefore, it reaches the maximum attainable computational performance with an acceptable power overhead. It is worth pointing out that it doesn't require latency cycles during precision variations. Thus, the two latency cycles reported in Table 1 only occur when pipeline is filled for the first time.

It is also worth pointing out that the power usage in the new variable precision multiplier varies with the precision settings. For this reason, average power dissipation has been reported in Table 1.

5 Conclusions

SRAM-based FPGAs seem the obvious candidates for the realization of systems for Reconfigurable computing. Reconfiguration provides such FPGAs with an extremely high flexibility, but it needs a long time and does not allow reconfiguring the hardware at the instruction level.

In this paper, we discussed efficient multipliers for FPGA-based variable precision processors. The new circuit operate in SIMD fashion and are able to match computational demands for several precisions avoiding time and power consuming reconfiguration. Thanks to this, they can be very fast run-time adapted to different data-size and precisions at instruction level.

References

- [1] K. Bondalapati, V.K. Prosanna, “Reconfigurable Computing Systems”, *Proceedings of IEEE*, Vol.90, n°7, 2002 pp. 1201-1217.
- [2] “Core embeds reconfigurable logic into SoCs”,
<http://www.electronicstalk.com/news/lsi/lsi102.html>.

- [3] S. Davis, C. Reynolds, P. Zuchowski, "IBM licenses embedded FPGA cores from Xilinx for use in SoC ASICs", <http://www.xilinx.com/publications/whitepapers/wp164.pdf>
- [4] XC6200 Field Programmable Gate Arrays, 1996
- [5] Virtex Series FPGAs, <http://www.xilinx.com>
- [6] M.J. Wirthlin, B.L. Hutchings, "Improving Functional Density Using Run-Time Circuit Reconfiguration", *IEEE Transactions on VLSI Systems*, Vol. 6, n°2, 1998, pp. 247-256.
- [7] Chameleon System, <http://www.chameleonsystems.com>
- [8] E. Cantò, J.M. Moreno, J. Cabestany, I. Lacadena, J.M. Insenser, "A Temporal Bipartitioning Algorithm for Dynamically Reconfigurable FPGAs", *IEEE Transactions of VLSI systems*, Vol. 9, n°1, 2001, pp. 210-218.
- [9] A. Peleg, U. Weiser, "MMX Technology", *IEEE Micro*, August 1996, pp. 42-50.
- [10] R.B. Lee, "Subword Parallelism with MAX-2", *IEEE Micro*, August 1996, pp. 51-59.
- [11] V. Kantabutra, S. Perri, P. Corsonello, "Tradeoffs in Digital Binary Adders", in Layout Optimizations in VLSI Design, Kluwer Academic Publisher, 2001, pp. 261-288.
- [12] A. Farooqui, V. G. Oklobdzija, "A Programmable Data-Path for MPEG-4 and Natural Hybrid Video Coding", *Proceedings of 34th Annual Asilomar Conference on signals, Systems and Computers*, Pacific Grove, California, October 29 - November 1, 2000.
- [13] M. Margala, J.X. Xu, H. Wang, "Design Verification and DFT for an Embedded Reconfigurable Low-Power Multiplier in System-on-Chip Applications", *Proceedings of 14th IEEE ASIC*, Arlington, September 2001.
- [14] S. Kim, M.C. Papaefthymiou, "Reconfigurable Low Energy Multiplier for Multimedia System Design", *Proceedings of 2000 IEEE Computer Society Annual Workshop on VLSI*, April 2000.
- [15] K. Hwang, *Computer Arithmetic, Principles, Architecture, and Design*, Wiley, 1979.
- [16] J. Fritts, W. Wolf, and B. Liu, "Understanding multimedia application characteristics for designing programmable media processors", SPIE Photonics West, Media Processors '99, San Jose, CA, pp. 2-13, Jan. 1999.

A New Arithmetic Unit in GF(2^m) for Reconfigurable Hardware Implementation

Chang Hoon Kim¹, Soonhak Kwon², Jong Jin Kim¹, and Chun Pyo Hong¹

¹ Dept. of Computer and Information Engineering, Daegu University,
Jinryang, Kyungsan, 712-714, Korea
chkim@dsp.taegeu.ac.kr

² Dept. of Mathematics and Inst. of Basic Science, Sungkyunkwan University,
Suwon, 440-746, Korea
shkwon@math.skku.ac.kr

Abstract. This paper proposes a new arithmetic unit (AU) in GF(2^m) for reconfigurable hardware implementation such as FPGAs, which overcomes the well-known drawback of reduced flexibility that is associated with traditional ASIC solutions. The proposed AU performs both division and multiplication in GF(2^m). These operations are at the heart of elliptic curve cryptosystems (ECC). Analysis shows that the proposed AU has significantly less area complexity and has roughly the same or lower latency compared with some related circuits. In addition, we show that the proposed architecture preserves a high clock rate for large m (up to 571), when it is implemented on Altera's EP2A70F1508C-7 FPGA device. Furthermore, the new architecture provides a high flexibility and scalability with respect to the field size m , since it does not restrict the choice of irreducible polynomials and has the features of regularity, modularity, and unidirectional data flow. Therefore, the proposed architecture is well suited for both division and multiplication unit of ECC implemented on FPGAs.

Keywords: Finite Field Division, Finite Field Multiplication, ECC, VLSI

1 Introduction

Information security has recently become an important subject due to the explosive growth of the Internet, the mobile computing, and the migration of commerce practices to the electronic medium. The deployment of information security procedures requires the implementation of cryptosystems.

Among these cryptosystems, ECC have recently gained a lot of attention in industry and academia. The main reason for the attractiveness of ECC is the fact that there is no sub-exponential algorithm known to solve the discrete logarithm problem on a properly chosen elliptic curve. This means that significantly smaller parameters can be used in ECC than in other competitive systems such as RSA and ElGamal with equivalent levels of security [1]. Some benefits of having smaller key sizes include

faster computations, reductions in processing power, storage space, and bandwidth. Another advantage to be gained by using ECC is that each user may select a different elliptic curve, even though all users use the same underlying finite field. Consequently, all users require the same hardware for performing the field arithmetic, and the elliptic curve can be changed periodically for extra security [1].

For performance as well as for physical security reasons it is often required to realize cryptographic algorithms in hardware. Traditional ASIC solutions, however, have the well-known drawback of reduced flexibility compared to software solutions. Since modern security protocols are increasingly defined to be algorithm independent, a high degree of flexibility with respect to the cryptographic algorithms is desirable. A promising solution which combines high flexibility with the speed and physical security of traditional hardware is the implementation of cryptographic algorithms on reconfigurable devices such as FPGAs [2-3], [10].

One of the most important design rule in FPGAs implementation is the elimination of global signals broadcasting. This is because the global signals in FPGAs are not simple wires but buses, i.e., they are connected by routing resources (switching matrices) having propagation delay [13], [16]. In general, since ECC require large field size (at least 163) to support sufficient security, when the global signals are used, the critical path delay increases significantly [11]. Due to this problem, systolic array based designs, where each basic cell is connected with its neighboring cells through pipelining, are desirable to provide a higher clock rate and maximum throughput performance on fine grained FPGAs [10-12].

In this paper, we propose a new AU, which performs both division and multiplication in GF(2^m) and has systolic architecture, for FPGAs implementation of ECC. The new design is achieved by using substructure sharing between the binary extended GCD algorithm [17] and the most significant bit (MSB)-first multiplication scheme [9]. When input data come in continuously, the proposed architecture produces division results at a rate of one per m clock cycles after an initial delay of $5m-2$ in division mode and multiplication results at a rate of one per m clock cycles after an initial delay of $3m$ in multiplication mode respectively.

Analysis shows that the proposed AU has significantly less area complexity and has roughly the same or lower latency compared with some related systolic arrays for GF(2^m). In addition, we show that the proposed architecture preserves a high clock rate for large m (up to 571), when it is implemented on Altera's EP2A70F1508C-7 FPGA device. Furthermore, the new architecture provides a high flexibility and scalability with respect to the field size m , since it does not restrict the choice of irreducible polynomials and has the features of regularity, modularity, and unidirectional data flow. Therefore, the proposed architecture is well suited for both division and multiplication unit of ECC implemented on fine grained FPGAs.

2 A New Dependence Graph for Both Division and Multiplication in GF(2^m)

2.1 Dependence Graph for Division in GF(2^m)

Let $A(x)$ and $B(x)$ be two elements in $\text{GF}(2^m)$, $G(x)$ be the irreducible polynomial used to generate the field $\text{GF}(2^m) \cong \text{GF}(2)[x]/G(x)$, and $P(x)$ be the result of the division $A(x)/B(x) \bmod G(x)$. We can perform the division using the following Algorithm I [17].

[Algorithm I] The Binary Extended GCD for Division in GF(2^m) [17]

Input: $G(x), A(x), B(x)$

Output: V has $P(x) = A(x)/B(x) \bmod G(x)$

Initialize: $R = B(x), S = G = G(x), U = A(x), V = 0, count = 0, state = 0$

1. **for** $i = 1$ to $2m - 1$ **do**

2. **if** $state == 0$ **then**

3. $count = count + 1;$

4. **if** $r_0 == 1$ **then**

5. $(R, S) = (R + S, R); (U, V) = (U + V, U);$

6. $state = 1;$

7. **end if**

8. **else**

9. $count = count - 1;$

10. **if** $r_0 == 1$ **then**

11. $(R, S) = (R + S, S); (U, V) = (U + V, V);$

12. **end if**

13. **if** $count == 0$ **then**

14. $state = 0;$

15. **end if**

16. **end if**

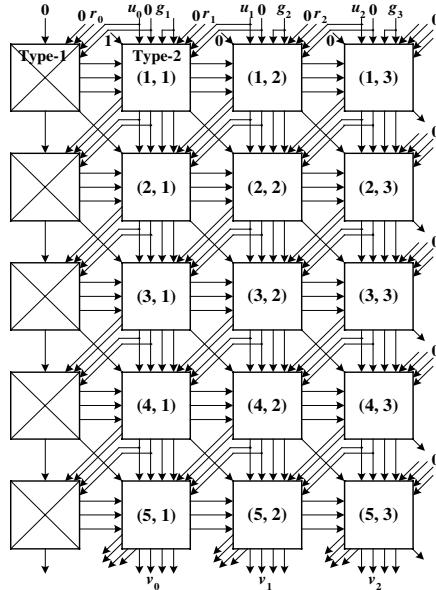
17. $R = R/x;$

18. $U = U/x;$

19. **end for**

Based on the Algorithm I, we can derive a new dependence graph (DG) for division in $\text{GF}(2^m)$ as shown in Fig. 1.

The DG corresponding to the Algorithm I consists of $(2m-1)$ Type-1 cells and $(2m-1) \times m$ Type-2 cells. In particular, we assumed $m=3$ in the DG of Fig. 1, where the functions of two basic cells are depicted in Fig. 2. Note that, since r_m is always 0 and s_0 is always 1 in all the iterations of the Algorithm I, we do not need to process them. The input polynomials $R(x), U(x)$, and $G(x)$ enter the DG from the top in parallel form. The i -th row of the array realizes the i -th iteration of the Algorithm I and the division result $V(x)$ emerge from the bottom row of the DG in parallel form after $2m-1$ iterations. Before describing the functions of Type-1 and Type-2 cell, we consider the implementation of $count$. From the Algorithm I, since $count$ increases to m , we can trace the value of $count$ by using m -bits bi-directional shift register (BSR) instead of $\log_2(m+1)$ -bits adder (subtractor). The BSR is also used for multiplication, as will be explained in section 2.3.

Fig. 1. DG for division in GF(2³)

In what follows, we add one 2-to-1 multiplixer, and *up* and *down* signals to each Type-2 cell. As shown in Fig. 1, in the first row, only the (1, 1)-th Type-2 cell receive 1, while the others receive 0 for *up*, and all the cells receive 0 for *down*. In the *i*-th iteration, *m*-bits BSR is shifted to the left or to the right according to *state*. When *cnt_n* in Fig. 2 is 1, it indicates that *count* becomes *n* ($1 \leq n \leq m$). In addition, when *count* reduces to 0, *down* in Fig. 2 becomes 1. As a result, all the *cnt_n* of Type-2 cells in the same row become 0, and *c-zero* and *state* in Fig. 2 are updated to 1 and 0 respectively. This is the same condition of the first iteration.

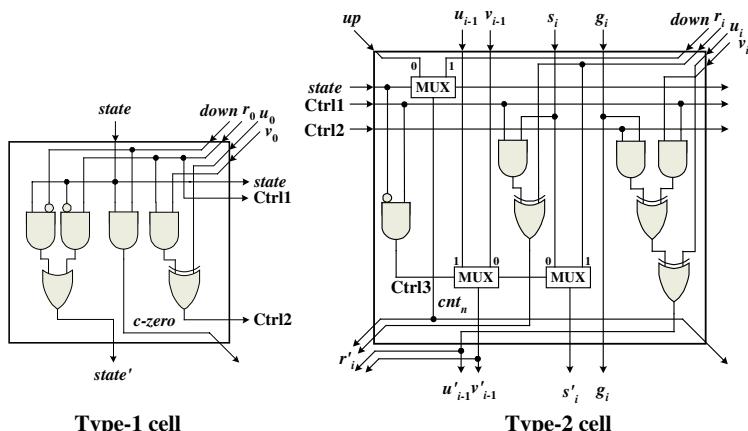


Fig. 2. The circuit of Type-1 and Type-2 cell in Fig. 1.

With *count* implementation result, we summarize the functions of Type-1 and Type-2 cell as follows:

(1) Type-1 cell: As depicted in Fig. 2, the Type-1 cell generates the control signals Ctrl1 and Ctrl2 for the present iteration, and updates *state* for the next iteration.

(2) Type-2 cell: the Type-2 cells in the i -th row generate the control signal Ctrl3 and perform the main operations of Algorithm I for the present iteration, and update *count* for the next iteration.

2.2 DG for MSB-First Multiplication in $\text{GF}(2^m)$

Let $A(x)$ and $B(x)$ be two elements in $\text{GF}(2^m)$, $G(x)$ be the irreducible polynomial, and $P(x)$ be the result of the multiplication $A(x)B(x) \bmod G(x)$. We can perform the multiplication using the following Algorithm II [9].

[Algorithm II] The MSB-first Multiplication Algorithm in $\text{GF}(2^m)$ [9]

Input: $G(x), A(x), B(x)$

Output: $P(x) = A(x)B(x) \bmod G(x)$

1. $p_k^{(0)} = 0$, for $0 \leq k \leq m-1$
2. $p_{-1}^{(i)} = 0$, for $1 \leq i \leq m$
3. for $i=1$ to m do
4. for $k=m-1$ down to 0 do
5. $p_k^{(i)} = p_{m-1}^{(i-1)} g_k + b_{m-i} a_k + p_{k-1}^{(i-1)}$
6. end
7. end
8. $P(x) = p^{(m)}(x)$

Based on the MSB-first multiplication algorithm in $\text{GF}(2^m)$, a DG can be derived as shown in the left figure of Fig. 3 [9]. The DG corresponding to the multiplication algorithm consists of $m \times m$ basic cells. In particular, $m=3$ in the DG of Fig. 3, and the right figure of Fig. 3 represents the architecture of basic cell. The cells in the i -th row of the array perform the i -th iteration of the multiplication algorithm. The coefficients of the result $P(x)$ emerge from the bottom row of the array after m iterations.

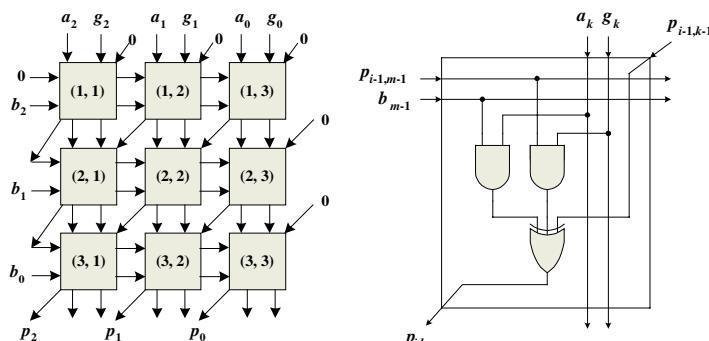


Fig. 3. DG and basic cell for multiplication in $\text{GF}(2^3)$ [9].

2.3 A New DG for Both Division and Multiplication in GF(2^m)

By observing the DG in Fig. 1 and the DG in Fig. 3, we can find that the U operation of the division is identical with the P operation of the multiplication except for the input values. In addition, there are two differences between the DG for division and the DG for multiplication. First, in the DG for multiplication, the input polynomial $B(x)$ enters from the left, while, in the DG for division, all the input polynomials enter from the top. Second, the positions of the coefficients of each input polynomial are changed in two DG. In this case, by modifying the circuit of each basic cell, both division and multiplication can be performed using the same hardware.

For the first case, by putting the m -bits BSR in each row of the DG for division, we can make the DG perform both division and multiplication depending on whether $B(x)$ enters from the left or it enters from the top. In other words, after feeding $B(x)$ into m -bits BSR, it is enough to shift it to the left until the multiplication is finished. For the second case, we can use the same hardware by making permutation of the coefficients of each input polynomial. In summary, the DG in Fig. 1 with appropriate modification can perform both division and multiplication. The resulting DG is shown in Fig. 4, and the corresponding Modified Type-1 and Type-2 cell in Fig. 2 are shown in Fig. 5.

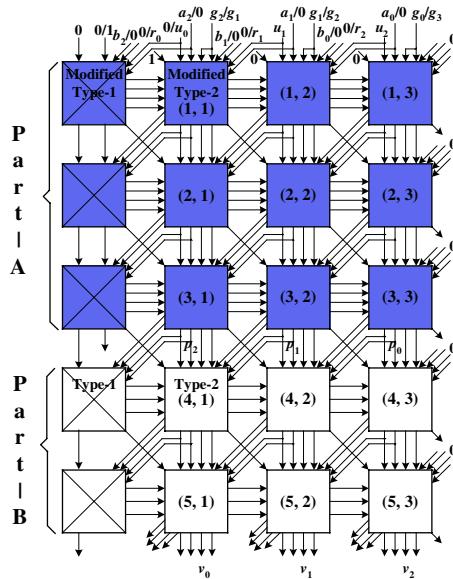


Fig. 4. New DG for both division and multiplication in GF(2³)

As described in Fig. 4, the DG consists of m Modified Type-1, $m \times m$ Modified Type-2, $m-1$ Type-1, and $(m-1) \times m$ Type-2 cells. The polynomials $A(x)$, $B(x)$, and $G(x)$ enter the DG for multiplication, and the polynomials $R(x)$, $U(x)$, and $G(x)$ enter the DG for division. The multiplication result $P(x)$ emerge from the bottom row of Part-A after m iterations, and the division result $V(x)$ emerge from the bottom row of

Part-B after $2m-1$ iterations. In Fig. 4, Part-A is used for both division and multiplication, and Part-B is only used for division. The modification procedures from Type-1, 2 cell to Modified Type-1, 2 cell are summarized as follows:

(a) Modification of Type-1 cell: We add *mult/div* signal, 2-to-1 AND gate (numbered by 1) and 2-to-1 multiplexer comparing to Type-1 cell. For multiplication mode, *mult/div* is set to 0, and for division mode, *mult/div* is set to 1. As a result, for multiplication, $b_{m-i}/\text{Ctrl1}$ has b_{m-i} , and $p_{m-1}/\text{Ctrl2}$ has p_{m-1} respectively. For division, it has the same function as the Type-1 cell in Fig. 2.

(b) Modification of Type-2 cell: We add *mult/div* signal, 2-to-1 AND gate (numbered by 1), 2-to-1 OR gate (numbered by 2) and 2-to-1 multiplexer comparing to Type-2 cell. In Modified Type-2 cell, since *Ctrl3* generates 0 for multiplication mode, a_{m-i}/v_{i-1} is always selected. For multiplication mode, the m -bits BSR is only shifted to the left direction due to the OR gate numbered by 2. In addition, the multiplexer numbered by 4 in Fig. 5 is also added due to the fact that, for multiplication mode, the AND gate number by 3 must receive a_{m-i}/v_{m-1} instead of a_{m-i-1}/v_i . As a result, when *mult/div* is set to 0, the Modified Type-2 cell in Fig. 5 performs as the basic cell in Fig. 3. In addition, when *mult/div* is set to 1, it performs as the Type-2 cell in Fig. 2.

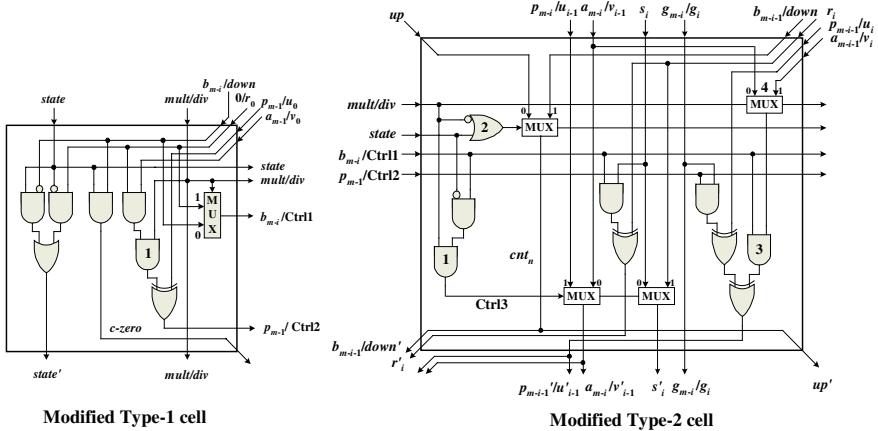


Fig. 5. The circuit of Modified Type-1 and Type-2 cell in Fig. 4

3 A New AU for Both Division and Multiplication in GF(2^m)

By projecting the DG in Fig. 4 along the east direction according to the projection procedure [14], we derive a one-dimensional SFG array as shown in Fig. 6, where the circuit of each processing element (PE-A and PE-B) is described in Fig. 7. In Fig. 6, “•” denotes a 1-bit 1-cycle delay element. The SFG array of Fig. 6 is controlled by a sequence 011...11 of length m . As described in Fig. 6, two different data set are feed into the array depending on division or multiplication mode.

As described in Fig. 7, the PE-A contains the circuitry of Modified Type-1 and Type-2 cell of Fig. 5. In addition, the PE-B contains the circuitry of Type-1 and

Type-2 cell of Fig. 2. Since two control signals Ctrl1 and Ctrl2, and *state* of the *i*-th iteration must be broadcast to all the Modified Type-1 and Type-2 cells in the *i*-th row of the DG, three 2-to-1 multiplexers and three 1-bit latches are added to each PE-A and PE-B. Four 2-input AND gates are also added to each PE-A and PE-B due to the fact that four signals (*i.e.*, $b_{m-i-1}/down$, r_i , p_{m-i-1}/u_i , and a_{m-i-1}/v_i in Fig. 5) must be fed to each row of the DG from the rightmost cell. When the control signal is in logic 1, these AND gates generate four zeros.

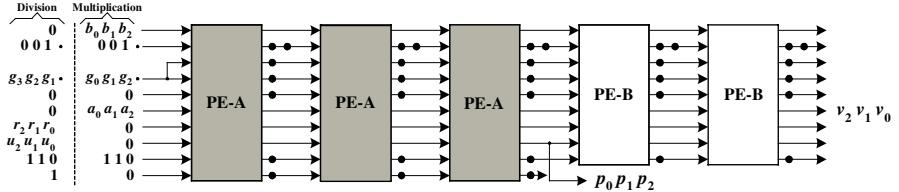


Fig. 6. A one-dimensional SFG array for both division and multiplication in GF(2³).

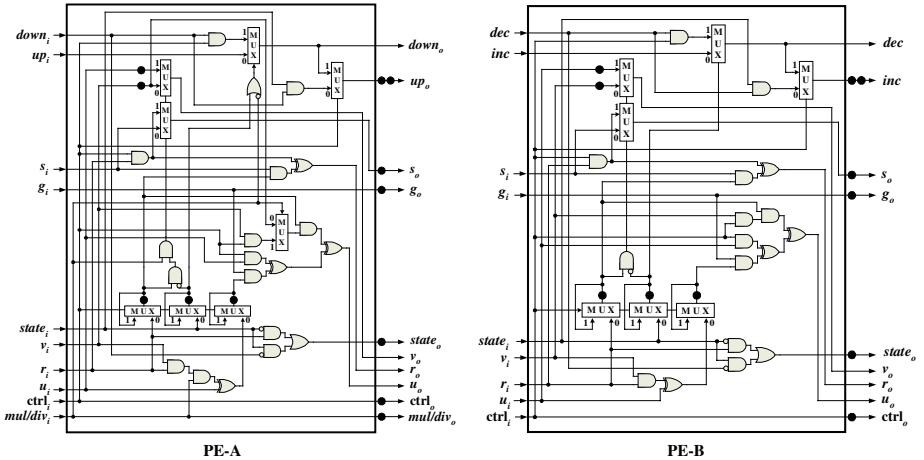
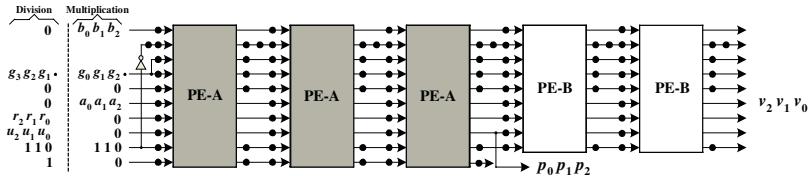


Fig. 7. The circuit of PE-A and PE-B in Fig. 6

The SFG array in Fig. 6 can be easily retimed by using the cut-set systolization techniques [14] to derive a serial-in serial-out systolic array, and the resulting structure is shown in Fig. 8. When the input data come in continuously, this array can produce division results at a rate of one per *m* clock cycles after an initial delay of $5m-2$ with the least significant coefficient first in division mode and can produce multiplication results at a rate of one per *m* clock cycles after an initial delay of $3m$ with the most significant coefficient first in multiplication mode.

Fig. 8. A new bit-serial systolic array for division and multiplication in GF(2^3)

4 Results and Conclusions

To verify the functionality of the proposed array in Fig 8, it was developed in VHDL and was synthesized using the Synopsis' FPGA-Express (version 2000.11-FE3.5), in which Altera's EP2A70F1508C-7 was used as the target device. The placement and route process and timing analysis of the synthesized designs were accomplished using the Altera's Quartus II (version 2.0).

We summarize theoretical comparison results in Table 1 with some related systolic arrays having the same I/O format. It should be mentioned that there are no circuits for both division and multiplication in GF(2^m) using the same hardware at this moment to the authors' knowledge. In addition, FPGA implementation results of our architecture are given in Table 2. As described in Table 1, all the bit-serial approaches including the proposed array achieve the same time complexity of $O(m)$. However,

Table 1. Comparison with bit-serial systolic arrays

Item \ Circuit	[4]	[5]	[6]	Proposed AU
Throughput (1 /cycles)	$1/(2m - 1)$	$1/m$	$1/m$	Division : $1/m$ Mult. : $1/m$
Latency (cycles)	$7m-3$	$5m-1$	$8m-1$	Division : $5m-2$ Mult. : $3m$
Maximum cell delay	$T_{AND_2} + T_{XOR_2}$ $+ T_{MUX_2}$	$T_{AND_2} + T_{XOR_2}$ $+ T_{MUX_2}$	$2T_{AND_2} + 2T_{XOR_2}$ $+ 2T_{MUX_2}$	$2T_{AND_2} + T_{XOR_2}$ $+ T_{MUX_2}$
Basic components and their numbers	AND ₂ : $3m^2+3m-2$ XOR ₂ : $1.5m^2+1.5m-1$ MUX ₂ : $3m^2+m-2$ Latch : $6m^2+8m-4$	AND ₂ : $2m-1$ OR ₂ : $3m$ XOR ₂ : $0.5m^2+1.5m-1$ MUX ₂ : $1.5m^2+4.5m-2$ Latch : $2.5m^2+14.5m-6$	Inverter : 2m AND ₂ : $26m$ XOR ₂ : $11m$ MUX ₂ : $35m+2$ FILO(m-bit) : 4 Latch : $46m+4m\log_2(m+1)$ zero-check ($\log_2(m+1)$ -bit) : $2m$ adder/subtractor ($\log_2(m+1)$ -bit) : $2m$	Inverter : 7m-2 AND ₂ : $26m-12$ OR ₂ : $3m-1$ XOR ₂ : $8m-4$ MUX ₂ : $15m-7$ Latch : $44m-24$
Operation	Division	Division	Division	Division and Multiplication
Area complexity	$O(m^2)$	$O(m^2)$	$O(m \cdot \log_2 m)$	$O(m)$

the proposed systolic array reduces the area complexity from $O(m^2)$ or $O(m \cdot \log_2 m)$ to $O(m)$, and has lower maximum cell delay and latency than the architecture in [6]. Although the circuits in [4-5] have lower maximum cell delay than the proposed array, they can not be applicable to ECC due to their high area complexity of $O(m^2)$. In addition, we do not need additional hardware components described in Table 3 to achieve multiplication, since the proposed array performs both division and multiplication.

Table 2. FPGA Implementation results of the proposed AU

Item \ m	163	233	409	571
# of LE	9920	14209	24926	34950
Clock (MHz)	138.87	127.99	141.8	121.11
Chip Utilization (%)	14.76	21.21	37.09	52.01

LE consists of one 4-to-1 LUT, one flip-flop, fast carry logic, and programmable multiplexers [16]

Table 3. Area-time complexity of the bit-serial systolic array for multiplication in GF(2^m) [9]

Throughput	Latency	Maximum cell delay	Basic components and their numbers
$1/m$	$3m$	$T_{\text{AND}_2} + 2T_{\text{XOR}_2}$	$\text{AND}_2 : 3m, \text{XOR}_2 : 2m,$ $\text{MUX}_2 : 2m, \text{Latch} : 10m$

From Table 2, it is noted that the proposed architecture preserves a high clock rate for large field size m because there are no global signals broadcasting. As a reference, 571 in Table 2 is the largest field size recommended by NIST [15]. Furthermore, since the proposed architecture does not restrict the choice of irreducible polynomials and has the features of regularity, modularity, and unidirectional data flow, it provides a high flexibility and scalability with respect to the field size m . All these advantages of our design lead that, if ECC is implemented on FPGAs to overcome the well-known drawback of ASIC, we can obtain maximum throughput performance with minimum hardware requirement by using the proposed AU.

Acknowledgement

This work was supported by grant No. R05-2003-000-11573-0 from the Basic Research Program of the Korea Science & Engineering Foundation

References

- [1] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999.
- [2] G. Orlando and C. Parr, “A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m)”, *CHES 2000*, LNCS 1965, Springer-Verlag, 2000.

- [3] M. Bednara, M. Daldrup, J. von zur Gathen, J. Shokrollahi, and J. Teich, "Reconfigurable Implementation of Elliptic Curve Crypto Algorithms," *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pp. 157-164, 2002.
- [4] C.-L. Wang and J. -L. Lin, "A Systolic Architecture for Computing Inverses and Divisions in Finite Fields $GF(2^m)$," *IEEE Trans. Computers.*, vol. 42, no. 9, pp. 1141- 1146, Sep. 1993.
- [5] M.A. Hasan and V.K. Bhargava, "Bit-Level Systolic Divider and Multiplier for Finite Fields $GF(2^m)$," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 972-980, Aug. 1992.
- [6] J.-H. Guo and C.-L. Wang, "Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in $GF(2^m)$," *IEEE Trans. Computers.*, vol. 47, no. 10, pp. 1161- 1167, Oct. 1998.
- [7] J.R. Goodman, "Energy Scalable Reconfigurable Cryptographic Hardware for Portable Applications," PhD thesis, MIT, 2000.
- [8] J.-H. Guo and C.-L. Wang, "Bit-serial Systolic Array Implementation of Euclid's Algorithm for Inversion and Division in $GF(2^m)$," *Proc. 1997 Int. Symp. VLSI Tech., Systems and Applications*, pp. 113-117, 1997.
- [9] C. L. Wang and J. L. Lin, "Systolic Array Implementation of Multipliers for Finite Field $GF(2^m)$," *IEEE Trans. Circuits and Syst.*, vol. 38, no. 7, pp. 796-800, July 1991.
- [10] T. Blum and C. Paar, "High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware", *IEEE Trans. Computers.*, vol. 50, no. 7, pp. 759-764, July 2001.
- [11] S.D. Han, C.H. Kim, and C.P. Hong, "Characteristic Analysis of Modular Multiplier for $GF(2^m)$," *Proc. of IEEK Summer Conference 2002*, vol. 25, no. 1, pp. 277-280, 2002.
- [12] R. Tessier and W. Burleson, "Reconfigurable Computing for Digital Signal Processing: A Survey", *J. VLSI Signal Processing*, vol. 28, no. 1, pp. 7–27, May 1998.
- [13] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, vol. 34, no. 2, pp. 171-210, June 2002.
- [14] S. Y. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ: Prentice Hall, 1988.
- [15] NIST, Recommended elliptic curves for federal government use, May 1999. <http://csrc.nist.gov/> encryption.
- [16] Altera, APEX^{TMII} Programable Logic Device Family Data Sheet, Aug. 2000. <http://www.altera.com/literature/lit-ap2.html>.
- [17] C.H. Kim and C.P. Hong, "High Speed Division Architecture for $GF(2^m)$," *Electronics Letters*, vol. 38, no. 15, pp. 835-836, July 2002.

A Dynamic Routing Algorithm for a Bio-inspired Reconfigurable Circuit

Yann Thoma^{1,3}, Eduardo Sanchez¹,
Juan-Manuel Moreno Arostegui², and Gianluca Tempesti¹

¹ Swiss Federal Institute of Technology at Lausanne (EPFL), Lausanne, Switzerland

² Technical University of Catalunya (UPC), Barcelona, Spain

³ Corresponding author. yann.thoma@epfl.ch

Abstract. In this paper we present a new dynamic routing algorithm specially implemented for a new electronic tissue called POEtic. This reconfigurable circuit is designed to ease the implementation of bio-inspired systems that bring cellular applications into play. Specifically designed for implementing cellular applications, such as neural networks, this circuit is composed of two main parts: a two-dimensional array of basic elements similar to those found in common commercial FPGAs, and a two-dimensional array of routing units that implement a dynamic routing algorithm which allows the creation of data paths between cells at runtime.

1 Introduction

Life is amazing in terms of complexity and of adaptability. After the fertilization of an ovule by a spermatozoid, a simple cell is capable of recursively dividing itself to create an entire living being. During its lifetime, an organism is also capable of self-repair in case of external or internal aggressions. Living beings possessing a neural network can learn tasks which allow them to adapt to their environment. And finally, at the population level, evolution allows a population to evolve in order to survive in an ever-changing environment. The aim of the POEtic project [7][8][10] is to design a new electronic circuit, drawing inspiration from these three life axes: Phylogenesis (evolution) [6], Ontogenesis (development) [9], and Epigenesis (learning) [4].

Ontogenetic methods, which are used to develop a self-repair circuit, need to change the functionality of the circuit at runtime. Epigenetic mechanisms, using neural networks, could also need to create new neurons, and therefore new connections between neurons at runtime. As commercially FPGAs usually don't have any dynamic self-reconfiguration capabilities, a new circuit capable of self-configuration is essential.

In the next section, we present the general architecture of the POEtic chip, decomposed into two subsystems. In section 3, we describe the basic elements of the circuit: the molecules. Section 4 fully explains the dynamic routing algorithm implemented in order to ease the creation of long distance paths into the chip.

2 Structural Architecture

The POEtic circuit is composed of two parts (figure 1): the organic subsystem, which is the functional part of the circuit, and the environmental subsystem. Cells, and thus organisms, are implemented in the organic subsystem. It is composed of a grid of small molecules and of a cellular routing layer. Molecules are the smallest unit of programmable hardware which can be configured by software, while dedicated routing units are responsible for the inter-cellular communication. The main role of the environmental subsystem is to configure the molecules. It is also responsible for the evolution process, and can therefore access and change every molecule's state in order to evaluate the fitness of an organism.

Each cell of an organism is a collection of molecules, which are the basic blocks of our circuit. The size and contents of the cells depend on the application. Therefore, for each application, a developer will have to design cells fitting into the molecules.

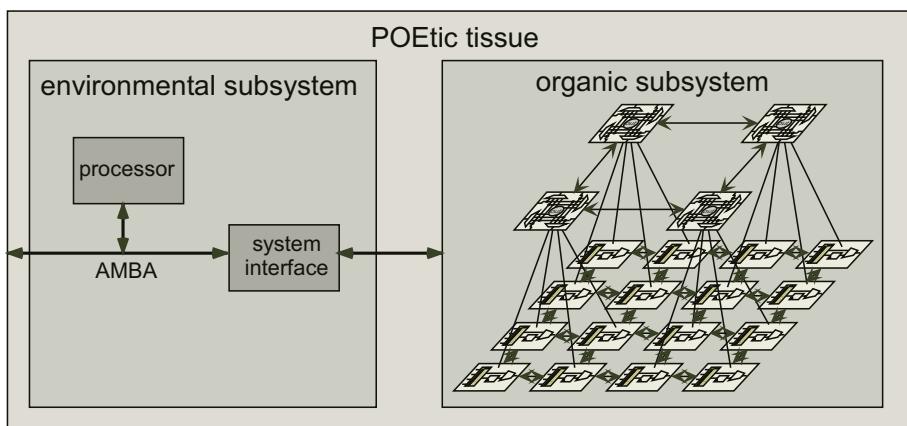


Fig. 1. The POEtic chip, composed of an environmental and an organic subsystems.

2.1 Environmental Subsystem

The environmental subsystem is primarily composed of a micro-controller: a 32-bit RISC-like processor. Its function is to configure the molecules, to run the evolutionary mechanisms, and to manage chip input/output. In order to speed up evolution processes, a random number generator has been added directly in the hardware. An AMBA bus [1] is used to connect the processor to a system interface that takes care of the communication between the processor and the organic subsystem. This bus is also connected to external pins in order to allow multi-chip communication, as well as the use of an external RAM.

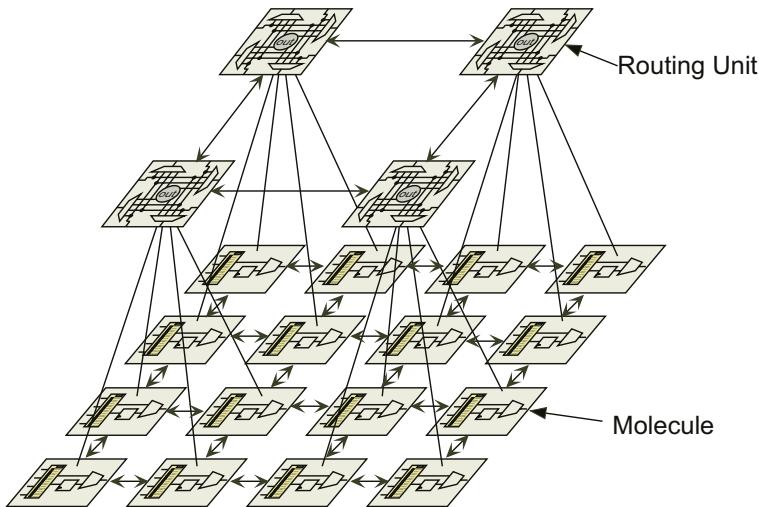


Fig. 2. The organic subsystem is composed of 2 layers: the molecules and the routing units.

2.2 Organic Subsystem

The organic subsystem is made up of 2 layers (cf. figure 2): a two-dimensional array of basic logic elements, called molecules, and a two-dimensional array of routing units. Each molecule has the capability of accessing the routing layer that is used for inter-cellular communication. This second layer implements a dynamic routing algorithm allowing the creation of data paths between cells at runtime.

3 Molecular Structure

As briefly presented above, the first layer of the POEtic tissue is a two-dimensional array of small programmable units, called molecules. Each molecule is connected to its 4 neighbors and to a routing unit (4 molecules for 1 routing unit), and contains a 16-bit look-up table (LUT) and a flip-flop (DFF). This structure, while seemingly very similar to standard FPGAs [2], is however specially designed for POEtic applications: different running modes let the molecule act like a memory, a serial address comparator, a cell input, a cell output, or others. With a total of 8 modes, these molecules allow a developer to build cells that are capable of communicating with each other, of storing a genome, of healing, and so on.

The 8 modes of operation of the molecule are the following:

- *Normal*: the LUT is a simple 16-bit look-up table.
- *Arithmetic*: the LUT is split into two 8-bit look-up tables: one for the molecule output, and one for a carry. A carry-chain physically sends the carry to the south neighbor, allowing rapid arithmetic operations.

- *Communication*: the LUT is split into one 8-bit shift register and one 8-bit look-up table. This mode can be used to implement a packet routing algorithm that will not be presented in this paper.
- *Shift memory*: the LUT is considered as a 16-bit shift register. This mode is very useful to efficiently store the genome in every cell. Shift memories can be chained in order to create memories of depth 32, 48, etc.
- *Configure*: the molecule has the capability of reconfiguring its neighbor. Combined with shift memory molecules, this mode can be used to differentiate the cells. A selected part of the genome, stored in the memory molecules, can be shifted to configure the LUT of other molecules (for instance to assign weights to neural synapses).
- *Input address*: the LUT is a 16-bit shift register and is connected to the routing unit. The 16 bits represent the address of the cell from where the information arrives. The molecule's output is the value coming from the inter-cellular routing layer (this mechanism will be detailed in the next section).
- *Output address*: the LUT is a 16-bit shift register and is connected to the routing unit. The 16 bits represent the address of the cell, and the molecule sends the value of one of its inputs to the routing unit (this mechanism will be detailed in the next section).
- *Trigger*: the LUT is a 16-bit shift register, and is connected to the routing unit. Its task is to supply a trigger every n clock cycles (where n is the number of bits of the addresses), needed by the routing algorithm for synchronization.

To be capable of self-repair and growth, an organism needs to be able to create new cells and to configure them. The configuration system of the molecules can be seen as a shift register of 80 bits split into 5 blocks: the LUT, the selection of the LUT's input, the switch box, the mode of operation, and an extra block for all other configuration bits. Each block contains, as shown in figure 3, together with its configuration, one bit indicating whether the block has to be bypassed in the case of a reconfiguration coming from a neighbor. This bit can only be loaded from the micro-processor, and remains stable during the entire lifetime of the organism.

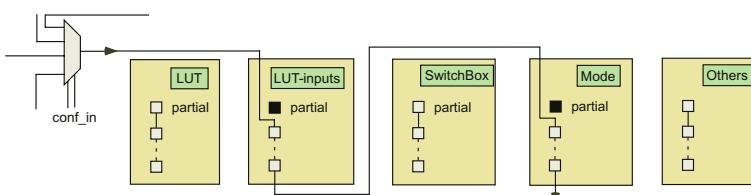


Fig. 3. All configuration bits of a molecule, split up into 5 blocks. The partial configuration bits of blocks 2 and 4 are set, enabling the reconfiguration of the LUT inputs and of the mode of operation by a neighboring molecule.

The special configure mode allows a molecule to partially reconfigure its neighborhood. It sends bits coming from another molecule to the configuration of one of its neighbors. By chaining the configurations of neighboring molecules, it is possible to modify multiple molecules at the same time, allowing, for example, the synaptic weights in a neuron to be changed.

4 Dynamic Routing

As presented above, our circuit is composed of a grid of molecules, in which cells are implemented. In a multi-cellular system, cells need to communicate with each other: a neural network, for example, often shows a very high density of connections between neurons. Commercial FPGAs have trouble dealing with these kinds of applications, because of their poor or nonexistent dynamic routing capacity. Given the purpose of the POEtic tissue, special attention was payed to this problem. Therefore, a second layer was added on top of the molecules, implementing a distributed dynamic routing algorithm. This algorithm uses an optimized version of the dynamic routing presented by Moreno in [5], to which we supplied a distributed control to where there is no global control of the routing process.

4.1 From Software to Hardware

Our dynamic routing algorithm finds the shortest path between two points in the routing layer. In 1959, Dijkstra proposed a software algorithm to find the shortest path between two nodes in a graph in which every branch has a positive length [3]. If we fix all branches to have a weight of 1, we can dramatically simplify the algorithm. It then becomes a breadth-first search algorithm, as follow:

```

1: paint all vertices white;
2: paint the source grey and enqueue it;
3: repeat
4:   dequeue vertex v;
5:   if v is the target, we found a path - exit the algorithm;
6:   paint v black;
7:   for each white neighbor w of v
8:     paint w grey;
9:     set parent w to v;
10:    enqueue w
11: until the queue is empty
12: if we haven't yet exited, we didn't find the target

```

This algorithm acts like a gas expanding in a labyrinth, but in a sequential manner, one node being expanded at a time, with a complexity of $O(V+E)$ where V is the number of vertices and E is the number of edges. Taking advantage of the hardwares' intrinsic parallelism, it is possible, based on the same principle as the breadth-first search algorithm, to expand all grey nodes at the same time.

This dramatically decreases the time needed to find the shortest path between two points, the complexity becoming $O(N+M)$, for a $N \times M$ array.

Finding the shortest path is not enough for the POEtic tissue, since we don't have a God telling us which routing unit is the source and which one is the target. In order to have a standalone circuit capable of self-configuration, we need a mechanism to start routings. Molecules, as explained in the previous section, have special modes to access the routing layer. Therefore, input or output molecules have the capability of initiating a dynamic routing.

4.2 Routing Algorithm

The dynamic routing system is designed to automatically connect the cells' inputs and outputs. Each output of a cell has a unique identifier at the organism level. For each of its inputs, the cell stores the identifier of the source from which it needs information. A non-connected input (target) or output (source) can initiate the creation of a path by broadcasting its identifier in the case of an output, or the identifier of its source in the case of an input. The path is then created using a parallel implementation of the breadth-first search algorithm presented above. When all paths have been created, the organism can start operation and execute its task until a new routing is launched, for example after a cell addition or a cellular self-repair.

Our approach has many advantages compared to a static routing process. First of all, a software implementation of a shortest path algorithm, such as Dijkstra's, is very time-consuming for a processor, while our parallel implementation requires a very small number of clock cycles to finalize a path. Secondly, when a new cell is created it can start a routing process without the need of recalculating all paths already created. Thirdly, a cell has the possibility of restarting the routing process of the entire organism if needed (for instance after a self-repair). Finally, our approach is totally distributed without any global control over the routing process, so that the algorithm can work without the need of the central micro-processor.

The routing algorithm is executed in three phases:

Phase 1: Finding a Master

In this phase, every target or source that is not connected to its correspondent partner tries to become master of the routing process. A simple priority mechanism chooses the most bottom-left routing unit to be the master, as shown in figure 4. Note that there is no global control for this priority, every routing unit knows whether or not it is the master. This phase is over in one clock cycle, as the propagation of signals is combinational.

Phase 2: Broadcasting the Address

Once a master has been selected, it sends its address in the case of a source, or the address of the needed source in the case of a target. As shown in section 3, the address is stored in a molecule connected to the routing unit. It is sent

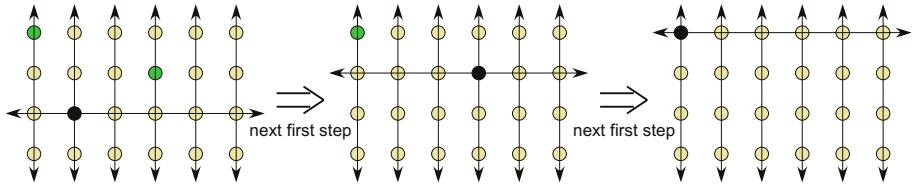


Fig. 4. Three consecutive first steps of the algorithm. The black routing unit will be the master, and therefore perform its routing.

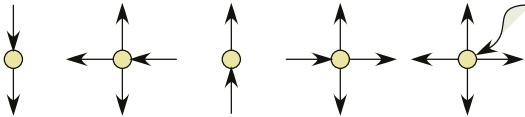


Fig. 5. The propagation direction of the address: north → south || east → south, west, and north || south → north || west → north, east, and south || routing unit → north, east, south, and west.

serially, in n clock cycles, where n is the size of the address. The same path as in the first phase is used to broadcast the address, as shown in figure 5.

Every routing unit, except the one that sends the address, compares the incoming value with its own address (stored in the molecule underneath). At the end of this phase, that is, after n clock cycles, each routing unit knows if it is involved in this path. In practice, there has to be one and only one source, and at least one target.

Phase 3: Building the Shortest Path

The last phase, largely inspired by [5], creates a shortest path between the selected source and the selected targets. An example involving 8 sources and 8 targets is shown in figure 6, for a densely connected network.

A parallel implementation of the breadth-first search algorithm allows the routing units to find the shortest path between a source and many targets. Starting from the source, an expansion process tries to find targets. When one is reached, the path is fixed, and all the routing resources used for the path will not be available for the next successive iterations of the algorithm.

Figure 7 shows the development of the algorithm, building a path between a source placed in column 1, row 2 and a target cell placed in column 3, row 3. After 3 clock cycles of expansion, the target is reached, and the path is fixed, prohibiting the use of the same path for a successive routing.

5 Conclusion

In this paper we presented a new electronic circuit dedicated to the implementation of bio-inspired cellular applications. It is composed of a RISC-like microprocessor and of two planes of functional and routing units. The first one, a

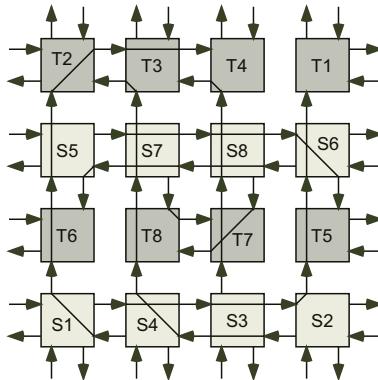


Fig. 6. Test case with a densely connected network.

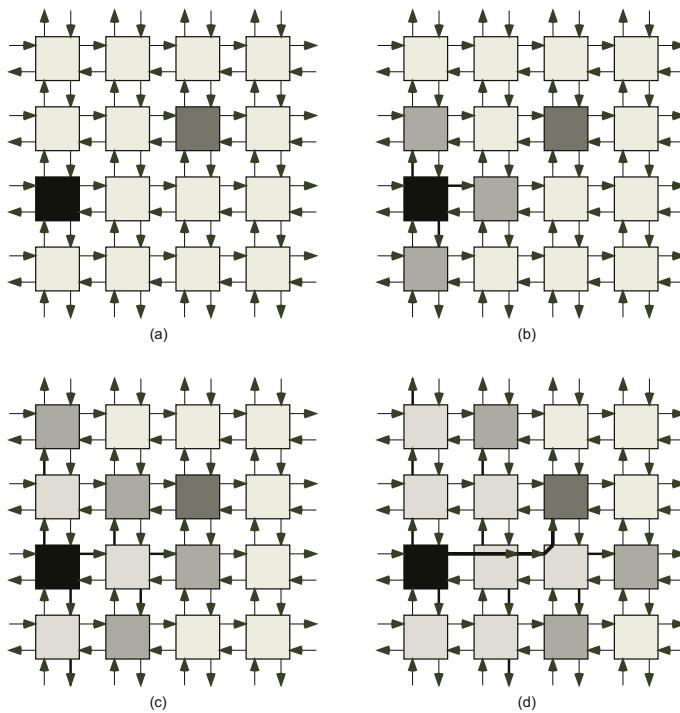


Fig. 7. Step (a) one, (b) two, (c) three and (d) four of the path construction process between the source placed in column 1, row 2 and target cell placed in column 3, row 3.

two-dimensional array of molecules, is similar to standard FPGAs and makes the circuit general enough to implement any digital circuit. However, molecules have self-configuration capabilities that are not present in commercial FPGAs and that are important for the growth of an organism and for self-repair at

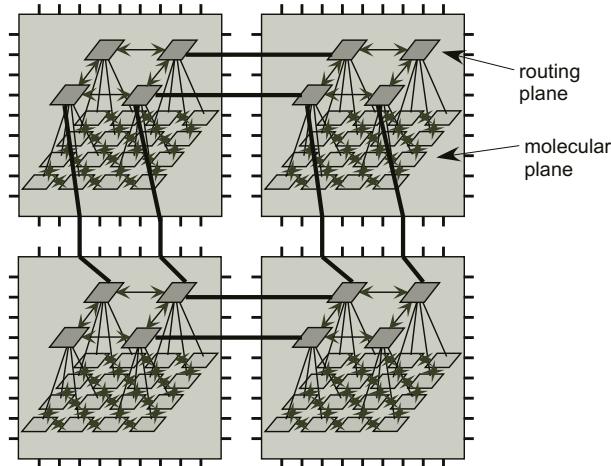


Fig. 8. A multi-chip organism shows the inter-cellular connections.

the cellular level. The second plane is a two-dimensional array of routing units that implement a dynamic routing algorithm. It is used for the inter-cellular communication, letting the tissue dynamically create paths between cells. This algorithm is totally distributed, and hence does not need the control of a microprocessor. Moreover, its scalability allows the creation of cellular networks of any size.

This circuit has been tested with a simulation based on the VHDL files describing the entire system. The next step of the project, which is currently under way and which should be completed by the time the conference will take place, is to develop, from the VHDL files, the VLSI layout and to realize a testchip to validate the design of our circuit.

Due to financial considerations, the first prototype of the POETIC chip will only contain approximately 500'000 equivalent gates. This size will not have enough molecules in one chip for complex designs. It will only be possible to implement a very simple organism on such a small number of molecules. Therefore, we included in the design the possibility of implementing a multi-chip organism by seamlessly joining together any number of chips (figure 8).

Acknowledgements

This project is funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under grant IST-2000-28027 (POETIC). The information provided is the sole responsibility of the authors and does not reflect the Community's opinion. The Community is not responsible for any use that might be made of data appearing in this publication. The Swiss participants to this project are supported under grant 00.0529-1 by the Swiss government.

References

- [1] ARM: Amba Specification, rev 2.0. Advanced RISC Machines Ltd (arm). http://www.arm.com/armtech/amba_spec, 1999.
- [2] Brown, S., Francis, R., Rose, J., Vranesic, Z.: *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.
- [3] Dijkstra, E.W.: A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [4] Eriksson, J., Torres, O., Villa, A. E. P.: Spiking Neural Networks for Reconfigurable POETic Tissue. In A.M. Tyrrell, P.C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES '03)*, volume 2606 of *LCNS*, pages 165–173, Berlin, 2003, Springer-Verlag.
- [5] Moreno Aróstegui, J. M., Sanchez, E., Cabestany, J.: An In-system Routing Strategy for Evolvable Hardware Programmable Platforms. In *Proc. 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 157–166. IEEE Computer Society Press, 2001.
- [6] Roggen, D., Floreano, D., Mattiussi, C.: A Morphogenetic System as the Phylogenetic Mechanism of the POETic Tissue. In A.M. Tyrrell, P.C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES '03)*, volume 2606 of *LCNS*, pages 153–164, Berlin, 2003, Springer-Verlag.
- [7] Sanchez, E., Mange, D., Sipper, M., Tomassini, M., Perez-Uribe, A., Stauffer, A.: Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware. In T. Higuchi, M. Iwata, and W. Liu, editors, *Evolvable Systems: From Biology to Hardware*, volume 1259 of *LCNS*, pages 33–54, Berlin, 1997. Springer-Verlag.
- [8] Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Perez-Uribe, A.: A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-inspired Hardware Systems. *IEEE Transactions on Evolutionary Computation*, 1:1:83–97, 1997.
- [9] Tempesti, G., Roggen, D., Sanchez, E., Thoma, Y., Canham, R., Tyrrell, A.M.: Ontogenetic Development and Fault Tolerance in the POETic Tissue. In A.M. Tyrrell, P.C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES '03)*, volume 2606 of *LCNS*, pages 141–152, Berlin, 2003, Springer-Verlag.
- [10] Tyrrell, A.M., Sanchez, E., Floreano, D., Tempesti, G., Mange, D., Moreno Aróstegui, J.-M., Rosenberg, J., Villa, A.E.P.: Poetic Tissue: An Integrated Architecture for Bio-inspired Hardware. In A.M. Tyrrell, P.C. Haddow, and J. Torresen, editors, *Evolvable Systems: From Biology to Hardware. Proc. 5th Int. Conf. on Evolvable Hardware (ICES '03)*, volume 2606 of *LCNS*, pages 129–140, Berlin, 2003, Springer-Verlag.

An FPL Bioinspired Visual Encoding System to Stimulate Cortical Neurons in Real-Time

Leonel Sousa¹, Pedro Tomás¹, Francisco Pelayo²,
Antonio Martínez², Christian A. Morillas², and Samuel Romero²

¹ Dept. of Electrical and Computer Engineering, IST/INESC-ID, Portugal
las@inesc-id.pt, pfzt@sips.inesc-id.pt

² Dept. of Computer Architecture and Technology, University of Granada, Spain
fpelayo@ugr.es, {amartinez, cmorillas, sromero}@atc.ugr.es

Abstract. This paper proposes a real-time bioinspired visual encoding system for multielectrodes' stimulation of the visual cortex supported on Field Programmable Logic. This system includes the spatio-temporal preprocessing stage and the generation of varying in time spike patterns to stimulate an array of microelectrodes and can be applied to build a portable visual neuroprosthesis. It only requires a small amount of hardware which is achieved by taking advantage of the high operating frequency of the FPGAs to share circuits in time. Experimental results show that with the proposed architecture a real-time visual encoding system can be implemented in FPGAs with modest capacity.

1 Introduction

Nowadays, the design and the development of visual neuroprostheses interfaced with the visual cortex is being tried to provide a limited but useful visual sense to profoundly blind people. The work presented in this paper has been carried out within the EC project "Cortical Visual Neuroprosthesis for the Blind" (CORTIVIS), which is one of the research initiatives for developing a visual neuroprosthesis for the blind [1, 2].

A block diagram of the cortical visual neuroprosthesis is presented in fig. 1. It includes a programmable artificial retina, which computes a predefined retina model, to process the input visual stimulus and to produce output patterns to approximate the spatial and temporal spike distributions required for effective cortical stimulation. These output patterns are represented by pulses that are mapped on the primary visual cortex and are coded by using Address Event Representation [3]. The corresponding signals are modulated and sent through a Radio Frequency (RF) link, which also carries power, to the electrode stimulator. This project uses the Utah microelectrode array [4], which consists on an array of 10×10 silicon microelectrodes separated by about $400 \mu\text{m}$ in each orthogonal direction (arrays of 25×25 microelectrodes are also considered). From experimental measures on biological systems, it can be established a time of 1 ms to "refresh" all the spiking neurons, which means an average time slot of $10 \mu\text{s}$ dedicated to each microelectrode. The RF link bandwidth allows communication

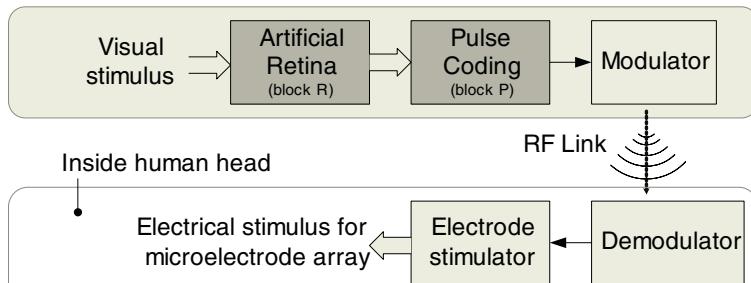


Fig. 1. Cortical visual neuroprosthesis

at a bit-rate of about 1 Mbps, which means an average value of 10 kbps for each microelectrode in a small size array of 10×10 electrodes or about 1.6 kbps for the 25×25 microelectrode array.

This paper addresses the design of digital processors for implementing the shaded blocks in fig. 1 in Field Programmable Logic (FPL) technology. The model of the retina adopted is a complete approximation of the spatio-temporal receptive fields' characteristic response of the retina ganglion cells (block R). The neuromorphic pulse coding is based on a leaky integrate-and-fire model of spiking neurons and on the Address Event Representation (AER), that communicates information about the characteristics of spikes and addresses of target microelectrodes without timestamps (block P). The architecture of the system has been designed having in mind the specifications of the problem referred above and the technical characteristics of nowadays Field Programmable Gate Array (FPGA) devices. Experimental results show that a complete artificial model that generates neuromorphic pulse-coded signals can be implemented in real-time even in FPGAs with low-capacity.

This paper is organized as follows. The architecture for modeling the retina and for coding the event lists that will be carried out to the visual cortex is presented in section 2. Section 3 reports the computational architectures designed for implementing the retina model in FPL, discussing their characteristics and suitability to the technology. Section 4 presents experimental results obtained by implementing R and P blocks on a FPGA and section 5 concludes the paper.

2 Model Architecture

The neuron layers of the human retina perform a set of different tasks, which culminate in the spiking of ganglion cells at the output layer [5]. These cells have different transient responses, receptive fields and chromatic sensibilities. The system developed in this paper implements the full model of the retina, which includes all the processing layers, plus the protocol for carrying the spikes to the visual cortex in a serial way through a RF link.

The two main blocks of the system in fig. 1 perform the following tasks: block **R**, the spatiotemporal filtering of the stimulus visual input, with contrast

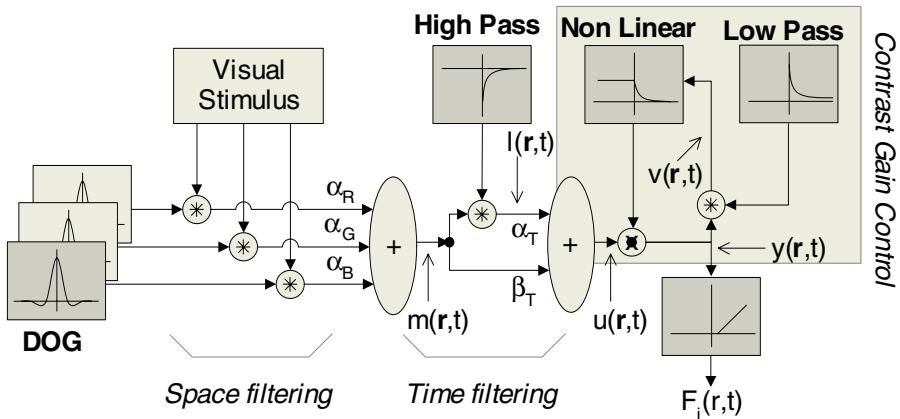


Fig. 2. Retina early layers

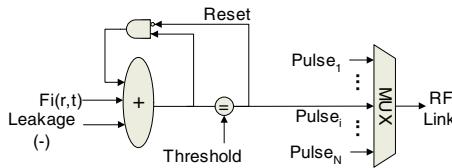


Fig. 3. Neuromorphic pulse coding

gain control and a rectifier circuit for computing the firing rate (fig. 2); block **P**, a neuromorphic pulse coding block which also implements the protocol used to communicate event lists without timestamps (fig. 3).

2.1 Retina Early Layers

The retina computational model used is based on the research published in [6], but it has been extended to the chromatic domain by considering independent filters for the basic colors. Fig. 2 presents the model architecture. It includes a spatial filter for contrast enhancement in the various color components which output is combined with programmable weights to reproduce receptive fields stimulated by specific color channels.

A Difference of Gaussians (DoG) is used to filter the stimulus in space:

$$DOG(\mathbf{r}) = \frac{\mathbf{a}_+}{2\pi\sigma^2} e^{-\frac{\mathbf{r}^2}{2\sigma^2}} - \frac{\mathbf{a}_-}{2\pi\beta\sigma^2} e^{-\frac{\mathbf{r}^2}{2\beta^2\sigma^2}} \quad (1)$$

where a_+ and a_- represent the relative weights of center and surround, respectively, and σ and $\beta\sigma$ ($\beta > 1$) are their diameters.

A high-pass temporal filter with the following impulse response is applied to the input signal already filtered in space:

$$h_{HP}(t) = \delta(t) - \alpha H(t) e^{-\alpha t} \quad (2)$$

where $H(t)$ represents the Heaviside step function and α^{-1} is the decay time constant of the response. In this paper, the bilinear approximation was applied to derive a digital version of the filter represented in the Laplace domain [7]. It leads to a first order Infinite Impulse Response (IIR) digital filter which can be represented by equation 3.

$$l[n] = b_{HP} \times l[n - 1] + c_{HP} \times (m[n] - m[n - 1]) \quad (3)$$

The relevance (weight) of the time response of a particular receptive field is also programmable in the model presented in fig. 2. The resulting activation $u(\mathbf{r}, t)$ is multiplied by a Contrast Gain Control (CGC) modulation factor $g(\mathbf{r}, t)$ and rectified to yield the ganglion cells firing rate response to the input stimulus.

The CGC models the strong modularity effect exerted by stimulus contrast. The CGC non-linear approach is also used in order to model the “motion anticipation” effect observed on experiments with a continuous moving bar [8]. The CGC feedback loop involves a low-pass temporal filter with the following impulse response:

$$h_{LP}(t) = Be^{-\frac{t}{\tau}} \quad (4)$$

where B and τ define the strength and constant time of the CGC. The filter is computed in the digital domain, by applying the same approximation as for the high-pass filter, by equation 5.

$$v[n] = b_{LP} \times v[n - 1] + c_{HP} \times (y[n] + y[n - 1]) \quad (5)$$

and the output of the filter is transformed into a local modulation factor (g) via the non-linear function:

$$g(t) = \frac{1}{1 + [v(t) \cdot H(v(t))]^4} \quad (6)$$

The output is rectified by using the function expressed in eq. 7:

$$F_i(\mathbf{r}, t) = \psi H(y(\mathbf{r}, t) + \theta)[y(\mathbf{r}, t) + \theta] \quad (7)$$

where ψ and θ define the scale and baseline value of the firing rate $f_i(\mathbf{r}, t)$.

The system to be developed is fully programmable and typical values for all parameters of the model are found in [6]. The model has been completely simulated in MATLAB, by using the *Retiner* environment for testing retina models [9].

2.2 Neuromorphic Pulse Coding

The neuromorphic pulse coding block converts the continuous-varying time representation of the signals produced in the early layers of the retina into a neural pulse representation. In this new representation the signal provides new information only from the moment a new pulse begins. The adopted model is a simplified version of an integrate-and-fire spiking neuron [10]. As represented in fig. 3, the

neuron accumulates input values from the respective receptive field (output firing rate determined by retina early layers) until it reaches a given threshold. Then it fires and discharges the accumulated value. A leakage term is included to force the accumulated value to diminish for low or null input values. The amplitude and duration of pulses are then coded by using AER, which is represented in a simplified way in fig. 3 by a multiplexer, to be sent to the addressed microelectrodes via the RF link. An event consists on an asynchronous bus transaction that carries the address of the corresponding microelectrode and is sent at the moment of pulse onset (no timestamp information is communicated). An arbitration circuit is required in the sender side and the receiver has to be listening to the data link with a constant latency.

3 FPL Implementation

This section discusses the implementation in FPL technology of the visual encoding system presented in the previous section. The usage of FPL technology will allow changing the model parameters without the need of projecting a second circuit. This has special importance since different patients have different sight parameters therefore requiring adjustments to the artificial retina. FPL may also allow changing model blocks if new information on how visual stimulus are processed reveals the need to introduce new filters.

For the design of the system, different computational architectures were considered both for the retina early layers and to the neuromorphic coding of the pulses. These architectures lead to implementations with different characteristics, in terms of hardware requirements and speed. The scalability and programmability of the system are also relevant aspects that are taken into account for FPL implementations.

3.1 The Retina Early Layers

To implement the retina early layers there can be multiple approaches which involve different hardware requirements, so the first step would be to analyze the necessary hardware to build the desired processor. Assuming a typical convolutional kernel of 7×7 elements for the DoG spatial filter and that high-pass and low-pass temporal filters are computed by using the difference equations 3 and 5, respectively, then 53 multiplications and 52 additions per image cell are required for just one spatial channel. For a matrix of 100 microelectrodes, the hardware required by a fully parallel architecture makes it not implementable in FPL technology. Therefore, the usage of the hardware has to be multiplexed in time, but the temporal restriction of processing the whole matrix with a frequency up to 100Hz must also be fulfilled. This restriction is however wide enough to consider the processing of cells with a full multiplexing schema inside each main block of the architecture model presented in section 2: DoG spatial filter, High-Pass temporal filter and Contrast Gain Control.

The architectures of these blocks can then be described as shown in figure 4, where each main block includes one or more RAM blocks to save processed

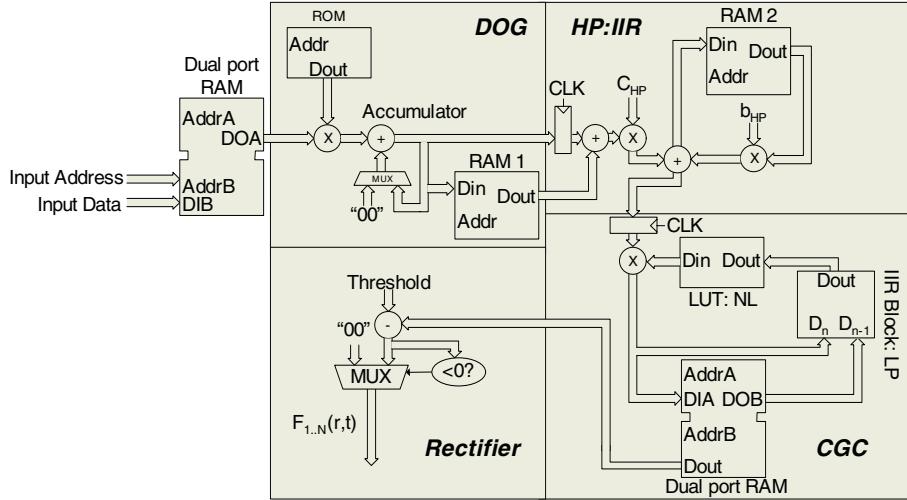


Fig. 4. Detailed diagram for the retina computational architecture.

frames in intermediate points of the system in order to compute the recursive time filters. At the input, dual-port RAM for the three color channel is used, while ROM is used to store the three coefficient tables for the spatial DoG filters. The operation of the processing modules is locally controlled and the overall operation of the system is performed by integrating the local control in a global control circuit. The control circuit is synthesized as a Moore state machine, because there is no need of modifying the output asynchronously with variations of the inputs. All RAM address signals, which are unconnected in the figure, are generated by the control circuit.

The DoG filter module calculates the 2D convolution between the input image and the predefined matrix coefficient stored in ROM. It operates in a sequential method, where pixels are processed one at a time in a row major order and an accumulator is used to store the intermediate and final results of the convolution. The local control circuit stores the initial pixel address, row and column, and then successively addresses all the data, and respective filtered coefficients stored in the ROM, required for the calculus of the convolution. After multiplying each filter coefficient the resulting value is successively accumulated. When the calculus is finished for a given pixel, the value is sent both to the internal RAM1 and to the next high-pass (HP) filter module. This module receives as operands the space filter results for the actual and the previous images ($m(\mathbf{r}, n)$ and $m(\mathbf{r}, n - 1)$) and the previous filter output $l(\mathbf{r}, n - 1)$, which is stored in the RAM2. The filter is computed by first adding both $m[n]$ and $m[n - 1]$ inputs, then multiplying the sum by the coefficient c_{HP} and the product is added with the previous filter output result $l[n - 1]$ multiplied by b_{HP} . The result is both stored in the RAM2 and communicated to the next processing module that corresponds to the CGC. This module consists on a low pass filter (eq. 5) and a

non-linear function (eq. 6) which is computed by using a lookup table stored in a RAM block. The low-pass filter circuit is not detailed in the figure because it is similar to the high-pass one. The last processing module is a rectifier which cuts the signal whenever its value decreases below a predefined threshold. It is implemented by a simple binary comparator whose output is used to control a multiplexer.

The overall circuit operates in a 3 stage pipeline corresponding to each one of the main processing blocks. Only the first pipeline stage require a variable number of clock cycles depending on the dimension of the filter kernel—49 cycles for a 7×7 kernel. Each of the two other pipeline stages is solved in a single clock cycle.

3.2 Neuromorphic Pulse Coding

Fig. 5 shows two processing modules *i*) for pulse generation and its representation and *ii*) to arbitrate the access to the serial bus (the input to the RF modulator in figure 1). This block is connected to the retina early layers through a dual port RAM (CGC block in fig. 4) where one writes data onto one port and the others reads it from the other.

The pulse generation circuit, which converts from firing rate to pulses, can be seen as a simple Digital Voltage Controlled Oscillator (DVCO) working in a two clock cycle stage pipeline. In the first stage the input firing rate is added to the accumulated value. In the second stage a leakage value is subtracted and, if the result is bigger than the threshold, a pulse is fired and the accumulator returns to the zero value (see fig. 5). Note that in this architecture the accumulator circuit is made with a RAM, since it corresponds to a single adder and multiple accumulator registers for the different microelectrodes.

AER was used to represent the pulses while the information is serialized. In a first approach, the architecture consists of an arbitration tree to multiplex the onset of events (spikes) onto a single bus. In this tree, we check if one or more inputs had a pulse to be sent and arbitrate the access to the bus trough

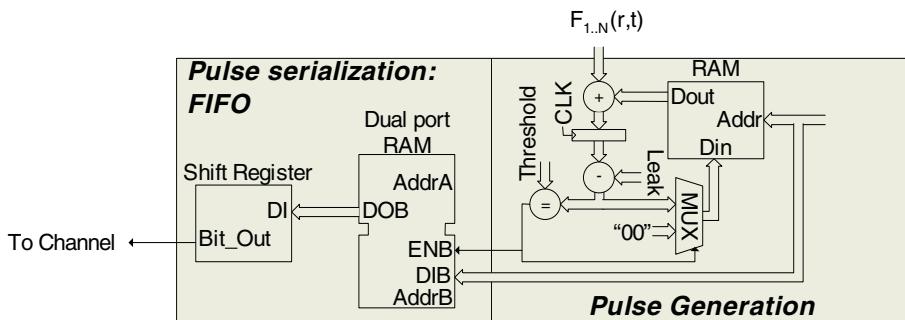


Fig. 5. Block diagram of the circuit for coding the pulses and to implement the address event protocol.

the request/acknowledge signals in the tree [3]. This tree consists of multiple subtrees, where registers can be used for buffering the signals between them. This architecture is not scalable, since it requires a great amount of hardware and is not a solution when arrays with a great number of microelectrodes are used (see section 4). To overcome this problem, and since the circuit for pulse generation is intrinsically sequential, a First In First Out (FIFO) memory is used to register the generated spikes for the microelectrodes. Only one pulse is generated in a clock cycle and information about it is stored in the FIFO memory because requests may find the communication link busy. This implementation has the advantage of not increasing the hardware resources required with the increase in the number of microelectrodes, but its performance is more dependent of the channel characteristics since it increases the maximum wait time to send a pulse.

The FIFO is implemented by using a dual port data RAM, where input is stored in one port and, at the other port, the pulse request is sent to the channel. Also to avoid overwriting when reading data from the FIFO a shift register is used to interface the data output of the RAM with the communication channel.

4 Experimental Results

The visual encoding system was described in VHDL and exhaustively tested on a DIGILAB II board supported on a XILINX SPARTAN XC2S200 FPGA [11] with modest capacity. The synthesis tool used was the one supplied by the FPGA manufacturer, the ISE WebPACK 5. This FPGA has modest capacities, with 56 kbit of block-RAM and 2352 slices, corresponding to a total of 200,000 system-gates. The functional validation of the circuits was carried out by comparing the results obtained with MATLAB Retiner [9].

For the test and experimental results presented in this section, only one space filter (DoG) is considered and the input signal is represented in 8-bit grayscale. However, internally 12-bit signals are used in order to reduce discretization errors. The considered dimension of the microelectrode array is 100.

Analyzing the results in table 1 it is clearly seen that the retina early layers do not require many FPGA slices but occupies a significant amount of memory. The synthesis of the circuit for an array of 1024 microelectrodes shows a similar percentage of FPGA slice occupation but it occupies all the 14 RAM blocks supplied by the SPARTAN XC2S200. In terms of time restrictions, the solutions works very well since it can process the array of input pixels in a short time, about 0.1ms for 100 pixels, which is much lower than the specified maximum of 10ms. In fact this architecture allows to process movies at frame rate of 100 Hz and a resolution of 10000 pixels without increasing the number of slices used.

The analysis of the AER translation method as however different aspects. If considered a typical matrix of 100 microelectrodes the tree solution (with or without intermediate registers) is a valid one as the resource occupation is low. However, the increase of the number of microelectrodes implies the usage of a great amount of hardware and this solution becomes impracticable for FPL technology. In that case, the FIFO based solution proposed in this paper has the

Table 1. Retina encoding system implemented on a SPARTAN XC2S200 FPGA

Block	Number of microelectrodes	Slice Occupation	Maximum clock frequency	Number of RAM blocks used
Retina early layers	100	18%	47MHz (49 cc*)	6
Pulse Generation	100	2%	51MHz	5
<i>AER</i>	100	9%	99MHz	0
<i>Registered Tree</i>	256	17%	98MHz	0
	512	37%	93MHz	0
<i>AER</i>	100	10%	64MHz	0
<i>Unregistered Tree</i>	256	21%	63MHz	0
	512	45%	57MHz	0
<i>FIFO AER</i>	**	5%	75MHz	1

* 49 clock cycles are required for processing a pixel

** not dependent of the number of micro-electrodes

great advantage of accommodating a great number of electrodes with a small amount of hardware and just one RAM block. With a channel bandwidth of about 1 Mbps, the FIFO based AER circuit does not introduce any temporal restrictions in the firing rate. Even operating at a lower clock frequency than the admitted by the retina early layers circuit, e.g. 40 MHz, the FIFO based circuit is able to attend 100 requests in about $2.5 \mu s$ which is much less than the value of 1 ms initially specified and also much less than the time required to send the information through the channel.

In table 1 results are individually presented for the retina early layers and for the neuromorphic pulse coding circuits. The overall system was also synthesized and implemented in a SPARTAN XC2S200 FPGA. A clock frequency greater than 40 MHz is achieved by using about 25% of the slices and 12 of the total of 14 RAM-blocks, for 100 microelectrodes.

5 Conclusions

This paper proposes a computational architecture for implementing a complete model of an artificial retina in FPL technology. The system is devised to stimulate a number of intra-cortical implanted microelectrodes for a visual neuroprosthesis. It performs a bio-inspired processing and encoding of the input visual information. The proposed processing architecture is designed to respect the constraints imposed by the telemetry system to be used, and with the requirement of building a compact and portable hardware solution.

The synthesis results demonstrate how the adopted pipelined and time multiplexed approach makes the whole system fit well on a relatively low complexity FPL circuit, while real-time processing is achieved. The use of FPL is also very convenient to easily customize (re-configure) the visual pre-processing and encoding system for each implanted patient.

Acknowledgements

This work has been supported by the European Commission under the project CORTIVIS (“Cortical Visual Neuroprosthesis for the Blind”, QLK6-CT-2001-00279).

References

1. Cortica visual neuro-prosthesis for the blind (cortivis): <http://cortivis.umh.es>.
2. Ahnelt P., Ammermller J., Pelayo F., Bongard M., Palomar D., Piedade M., Ferrandez J., Borg-Graham L., and Fernandez E. Neuroscientific basis for the design and development of a bioinspired visual processing front-end. In *Proc. of IFMBE*, pages 1692–1693, Vienna, 2002.
3. Lazzaro J. and Wawrzynek J. A multi-sender asynchronous extension to the address event protocol. In *Proc. of 16th Conference on Advanced Research in VLSI*, pages 158–169, 1995.
4. Maynard E. The utah intracortical electrode array: a recording structure for potential brain-computer interfaces. *Elec. Clin. Neurophysiol.*, 102:228–239, 1997.
5. Wandell Brian. *Foundations of Vision: Behavior, Neuroscience and Computation*. Sinauer Associates, 1995.
6. Wilke S., Thiel A., Eurich C., Greschner M., Bongard M., Ammermuler J., and Schwegler H. Population coding of motion patterns in the early visual system. *J. Comp Physiol A*, 187:549–558, 2001.
7. Oppenheim A. and Willsky A. *Signal and Systems*. Prentice Hall, 1983.
8. Berry M., Brivanlou I., Jordan T., and Meister M. Anticipation of moving stimuli by the retina. *Nature (Lond)*, 398:334–338, 1999.
9. Pelayo F., Martinez A., Romero S., Morillas Ch., Ros E., and Fernández E. Cortical visual neuroprosthesis for the blind: Retina-like software/hardware preprocessor. In *Proc. of IEEE-EMBS International Conference on Neural Engineering*, Capri, 2003.
10. Gerstner W. and Kistler W. *Spiking Neuron Models*. Cambridge University Press, 2002.
11. XILINX. *Spartan-II 2.5V FPGA Family: Functional Description*, 2001. Product Specification.

Power Analysis of FPGAs: How Practical Is the Attack ?

François-Xavier Standaert, Loïc van Oldeneel tot Oldenzeel, David Samyde,
and Jean-Jacques Quisquater

UCL Crypto Group
Laboratoire de Microélectronique
Université Catholique de Louvain
Place du Levant, 3, B-1348 Louvain-La-Neuve, Belgium
standaert,vanolden,samyde,quisquater@dice.ucl.ac.be

Abstract. Recent developments in information technologies made the secure transmission of digital data a critical design point. Large data flows have to be exchanged securely and involve encryption rates that sometimes may require hardware implementations. Reprogrammable devices such as Field Programmable Gate Arrays are highly attractive solutions for hardware implementations of encryption algorithms and several papers underline their growing performances and flexibility for any digital processing application. Although cryptosystem designers frequently assume that secret parameters will be manipulated in closed reliable computing environments, Kocher et al. stressed in 1998 that actual computers and microchips leak information correlated with the data handled. Side-channel attacks based on time, power and electromagnetic measurements were successfully applied to the smart card technology, but we have no knowledge of any attempt to implement them against FPGAs. This paper examines how monitoring power consumption signals might breach FPGA-security. We propose first experimental results against FPGA-implementations of cryptographic algorithms in order to confirm that power analysis has to be considered as a serious threat for FPGA security. We also highlight certain features of FPGAs that increase their resistance against side-channel attacks.

1 Introduction

Digital signal processing has traditionally been done using enhanced microprocessors but recent increases in Field Programmable Gate Arrays performance and size offer a new hardware acceleration opportunity. The last years brought cryptographic implementations into the field of FPGA designers as several conference and journal publications can witness [10, 11]. These cryptosystem designers frequently assume that secret parameters will be manipulated in closed reliable computing environments. However, the realities of physical implementations can be extremely difficult to control and may result in the unintended leakage of side-channel information. This leaked information is often correlated to the secret keys, thus adversaries monitoring this information may be able to recover the secret key and breach the security of the cryptosystem.

Side-channel attacks based on time, power and electromagnetic measurements were successfully applied to the smart card technology as witnessed by [1–5]. However, we have no knowledge of any attempt to implement them against FPGAs. Moreover, most major FPGA manufacturers provide no information about the actual security of their devices. This paper presents first experimental results in order to fill that gap. Based on various examples, we discuss the practicability of power analysis attacks against an application-oriented FPGA board but also highlight certain physical features of FPGAs and application boards that make the practical implementation of power analysis significantly harder than in the smart card context.

The paper is structured as follows. Section 2 presents the hardware used to carry out the experiments. Section 3 gives a short description of two cryptographic algorithms: DES and RSA. Section 4 introduces power analysis. We study Simple Power Analysis and Differential Power Analysis in sections 5 and 6. Finally, topics for further researches are in section 7 and conclusions in section 8.

2 Hardware Description

All our experiments were carried out on a VIRTEX-ARM board developed by DICE¹ (Figure 1). This board was developed in 2000 for a multi-purposes use. The board is composed of a control FPGA (Altera® FLEX®10K) and a Xilinx®Virtex®1000 FPGA associated with μ -controllers (Microchip®PIC®, ARM®) and fast access memories. It has multiple compatible PC interfaces (USB, PCI). Practical details about the Virtex1000BG560-4 FPGA that we investigated can be found in [8].

The voltages needed for the board are:

1. 5 volts for the PCI bridge.
2. 2.5 volts for the Virtex core and the ARM μ -processor.
3. 3.3 volts for other devices, including the Virtex I/O blocks.

The usual way to use this board has always been to plug it into a PCI port but to perform a power analysis against a chip, one must have access to its power supply in order to acquire power consumption traces. For this purpose, we insert a small resistance in the supply circuit. As the board has a single ground circuit and only the Virtex chip has to be analyzed (other devices add noise to the measurements) we decided to insert the resistance next to the source supplying the Virtex. We undersupplied certain unnecessary devices and we un-soldered the DC-DC 2.5V convertor (of which internal oscillations generate noise) before carrying out the experiments. Figure 1 illustrates the final test bed where the FPGA is programmed via the JTAG chain².

Finally, we used the following hardware to perform our tests :

1. Voltage sources to supply the 2.5 volts path and 3.3 volts path.
2. A waveform generator or a crystal oscillator to generate the clock signal.

¹ Microelectronics Laboratory at Université Catholique de Louvain, Belgium.

² Boundary-Scan Standard IEEE 1149.1, developed by the Joint Test Action Group.

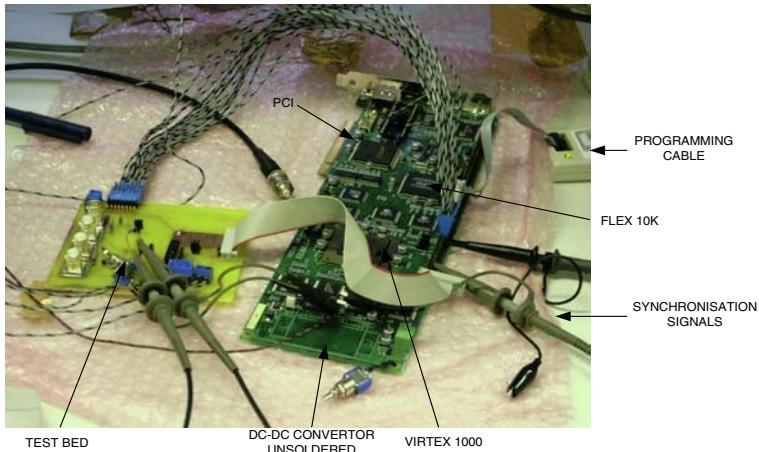


Fig. 1. The FPGA board

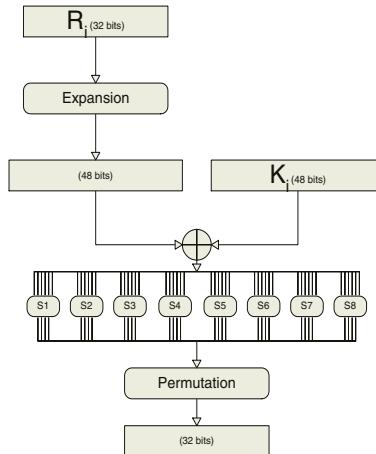
3. An oscilloscope to observe the power traces. We used the Tektronix 7140 with a 1 GHz bandwidth.
4. Computer softwares to generate the FPGA programming files and process the data after acquisition.

3 DES and RSA

In 1977, the Data Encryption Standard (DES) algorithm was adopted as a Federal Information Processing Standard for unclassified government communication. Although a new Advanced Encryption Standard was selected in October 2000, DES is still largely in use. DES [6] encrypts 64-bit blocks with a 56-bit key and processes data with permutations, substitutions and XOR operations. It is an iterative block cipher that applies a number of key-dependent transformations called rounds to the plaintext. This structure allows very efficient hardware implementations.

Basically, the plaintext is first permuted by a fixed permutation IP . The result is next split into the 32 left bits and the 32 right bits, respectively L and R that are sent to 16 applications of a round function. The ciphertext is calculated by applying the inverse of the initial permutation IP to the result of the 16-th round.

The secret key is expanded by the key schedule to 16×48 -bit subkeys K_i and in each round, a 48-bit subkey is XORed to the text. The key expansion consists of known bit permutations and shift operations. As a consequence, finding any subkey bit directly involves that the secret key is corrupted.

**Fig. 2.** The function f .

4 Introduction to Power Analysis

Integrated circuits are built out of individual transistors that act as voltage-controlled switches. Current flows across the transistor substrate when charge is applied to (or removed from) the gate. This current then delivers charges to the gates of other transistors, interconnect wires, and other circuit loads. The motion of electric charge consumes power and produces electromagnetic radiations, both of which are externally detectable. Therefore, individual transistors produce externally observable electrical behavior. Because microprocessor logic units exhibit regular transistor switching patterns, it is possible to easily identify macro-characteristics (such as microprocessor activity) by the simple monitoring of power consumption.

In Simple Power Analysis attacks, an attacker directly observes a system's power consumption. The amount of power consumed varies depending on the microprocessor instruction performed. Large features such as DES rounds may be identified, since the operations performed by the microprocessor vary significantly during different parts of these operations.

Differential Power Analysis is a much more powerful attack than SPA, and is much more difficult to prevent. While SPA attacks use primarily visual inspection to identify relevant power fluctuations, DPA attacks use statistical analysis to extract information correlated to the secret key.

Because it was not obvious that power analysis could detect some features of a running design, we performed a simple preliminary tests: we investigated the power consumption of NOT gates applied to bit vectors (all 0s or all 1s) and stored in registers. We clearly observed that the power consumption is correlated

Algorithm 1 Computation of $x^k \bmod n$

1. $z = 1;$
2. For $i = l - 1$ to 0 loop :

$$z = z^2 \bmod n;$$
 If $k_i = 1$ then $z = z \times x \bmod n$

to the Hamming weight³ of these bit vectors (see Figure 3). However, this test gave no indication about a possible dilution of the bit effect when a large design (like DES) is running. Moreover, the power consumption was made clearer because the bit vectors appeared at the outputs of the FPGA, while power analysis usually looks for internal bit switches.

5 Simple Power Analysis of FPGAs

Traditional controllers process the data sequentially and apply a set of instructions to the intermediate states of the computation. They can be viewed as control-oriented designs. As a consequence, an attacker may expect to detect two types of information from their side-channel leakages:

1. The instructions processed.
2. The data processed.

SPA typically tries to take advantages of the sequence of instructions processed. For example, distinguishing the square operation from the multiply operation would allow us to directly recover the key bits in an implementation of RSA. There are numerous examples of programs running on smart cards that allows to distinguish different instructions. However, simple countermeasures usually allows avoiding SPA by masking the instructions.

On the contrary, in most applications, FPGAs are used in order to perform parallel tasks. Cryptographic applications like DES or RSA can be implemented as data-oriented pipeline architectures with several operations running concurrently. Moreover, operations that are spread over several clock edges in smart cards may be reduced to only one clock period in FPGA implementations, which makes distinguishing them unlikely. As a consequence, in these cases, SPA becomes somewhat unpractical and an attacker is limited to information about the data processed as we have in Figure 3.

Exceptions obviously exist. For example applications where enable signals of registers are managed by a control part. Then the activity (or not) of registers may help to distinguish instructions.

We can illustrate these assumptions with an example. We investigated the power consumption of a DES running with weak keys that we can explain as follows. Because of the way the initial key is modified to get a subkey for each round of the algorithm, certain initial keys present special properties. Practically,

³ Hamming weight: number of one in the bit vector.

if the subkey used for any round of the algorithm is the same, the initial key is weak. DES has 4 weak keys and we used the following ones (in hexadecimal representation):

Weak Key Value	Actual Subkey
0101010101010101	0000000000000000
FEFEFEFEFEFEFEFE	FFFFFFFFFFFFFFFFFF

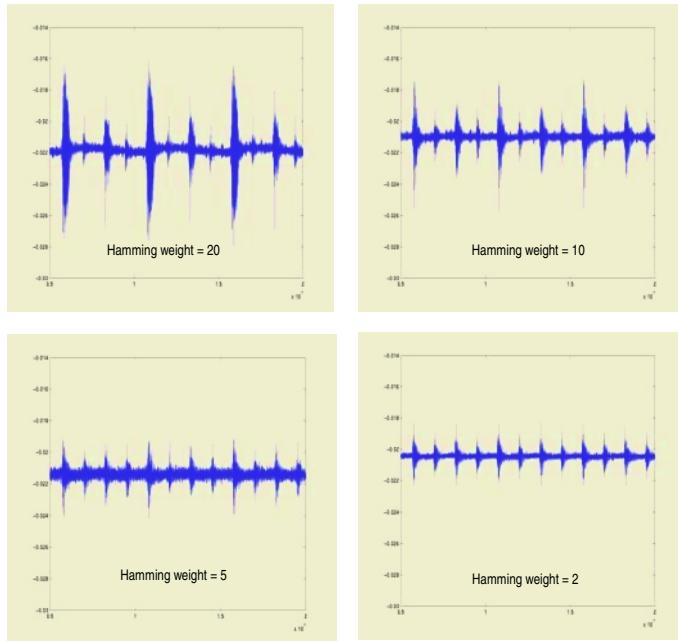


Fig. 3. Hamming weight (5000 traces averaged).

Figure 4 illustrates the power consumption of DES running with weak keys. We observe that:

1. We can clearly identify the rounds of our running DES.
2. The mean power consumed slightly differs between the two cases. One reason could be that the architectures slightly differ because a different key is stored in the VHDL code.
3. The patterns of the power consumed are clearly different. The second is fatter which corresponds to the expected behavior of the device.

This test confirms that the consumed power is strongly correlated with the internal bit switches of FPGAs. It also underlines that SPA-type attacks, where the attacker recovers secret parameters observing the shape of the traces are

made difficult by parallel computing (as all components are running concurrently). Moreover, FPGAs offer great opportunities to implement countermeasures against SPA.

6 Differential Power Analysis of FPGAs

As previous section confirmed that the power consumed by FPGAs is correlated with the internal bit switches, Differential Power Analysis is theoretically applicable. This section is devoted to experimental results of DPA implemented against RSA and DES running on a FPGA. We first studied modular implementation.

The basic premise of this attack is that by comparing the power signal of an exponentiation using a known exponent to a power signal using an unknown exponent, the adversary can learn where the two exponents differ, thus learn the secret exponent. The DPA technique begins by using the secret exponent to exponentiate L random values and collect their associated power signals $S_i[j]$ (j is a sample point). Likewise, L power signals $P_i[j]$ are collected using the known exponent. The average signals are then calculated and subtracted to form $D[j]$, the DPA bias signal.

$$D[j] = \frac{1}{L} \sum_{i=1}^L S_i[j] - \frac{1}{L} \sum_{i=1}^L P_i[j] = \bar{S}[j] - \bar{P}[j] \quad (4)$$

The portions of the signals $\bar{S}[j]$ and $\bar{P}[j]$ that are dependent on the intermediate data will average out to the same constant as long as the data produced by the RSA computation is equal. We have $D[j] = 0$ if the exponentiation operations are the same and $D[j] \neq 0$ if different.

There are several ways to perform the attack, depending on the assumptions made about the attacker. The simplest one is a "Multiple-Exponent, Single-Data" mode. Then, the attacker guesses the exponent bits (starting from the MSB), decides if the guess was correct by computing $D[j]$ and modifies the exponent bits one by one in order to get $D[j] = 0$ everywhere. Figure ?? shows our practical implementation of the attack. The left picture is a single power consumption trace where we observe the 13 clock edges corresponding to 13 "square and multiply" operations. The right picture shows peaks amplitudes for two keys that are equal until bit number 7. We observe that the consumption traces clearly diverge when exponents differ. Note that the attack depends on how different are the intermediate texts. As a consequence, repeating it with different texts improves its efficiency. Another critical point is that our RSA design was a toy-design with 12-bit vectors. As a consequence the difference between correct and wrong vectors is not large and it was difficult to underline their different power consumption. We increased the power consumption by repeating the RSA computation 20 times on the FPGA. Then we could clearly distinguish the secret exponent.

In the case of DES, the Differential Power Analysis requires a selection function $D(C, b, K_{Sb,16})$ that we define as computing the value of a bit b which is a

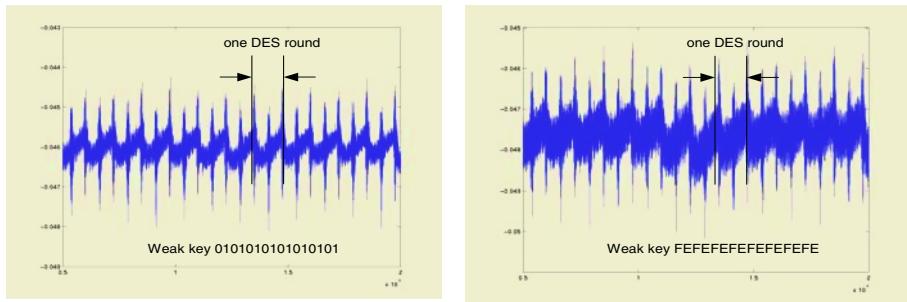


Fig. 4. Weak keys (5000 traces averaged).

part of intermediate vector L_{15} . As b results of a partial decryption through the last round of the algorithm, it can be derived from the ciphertext C and the 6 key bits entering in the same s-box as bit b .

To implement the DPA attack, an attacker first observes m encryptions and captures m power traces T_i and their associated ciphertexts C_i . No knowledge of the plaintext is required. With a guessed key K , the function D can be computed for each i and we can obtain two sets of traces: one corresponding with $D_i = 0$ and the other with $D_i = 1$. Each set is then averaged to obtain two average traces A_0 and A_1 and we can compute the difference $\Delta = A_0 - A_1$.

If $K_{Sb,16}$ is correct, the computed value for D will equal the actual value of target bit b with probability 1. As the power consumption is correlated to the data, the plot of Δ will be flat, with spikes in regions where D is correlated to the values being processes. If $K_{Sb,16}$ is incorrect, Δ will be flat everywhere.

The main difference between attacking DES and RSA is that while we have to distinguish the difference between two intermediate vectors in the RSA case, we have to observe the effect of a single bit in the case of DES, what we could not achieve with our low cost equipment. The following practical features of our FPGA board make the implementation of the DPA against DES a challenging task:

1. It should be noted that the application boards usually include several components of which the grounds are connected together. This makes the isolation of the FPGA consumption critical if the power measurements are carried out on the ground pin.
2. The manipulation of the selection bit that is spread over several clock edges in smart cards is reduced to one clock period in our FPGA implementation.
3. FPGAs are running at high work frequencies. Optimal implementations of the DES on the old VIRTEX technology run up to 170 MHz. Recent devices like VIRTEX-2 are much faster. This involve very high sampling rates to catch the consumption details.
4. Contrary to smart cards where the data is managed by 8-bit registers, FPGAs deal with all the bits (64 for DES) at once. This cause a dilution of the desired effect. This is even more critical when the key schedule or other tasks

are computed in parallel. As a result, the quantization of power traces may become the bottleneck of the attack, i.e. if the effect of a single bit is out of scale (less than one bit of quantization), the attack becomes unfeasible. Figure 3 illustrates this assessment with a comparison between 20-bit spikes and 2-bit spikes.

7 Further Research

A practical implementation of the DPA against DES is still matter of further research and there are plenty of potential sources for improvements. Nevertheless, there are many others scopes for further research. We propose the following list:

1. Reducing the noise during measurements by isolating the FPGA, using multiple-bit attacks, cooling the devices with nitrogen, ...
2. Applying intrusive attacks to FPGAs: depackaging, layer recovering,...
3. FPGAs usually consists in regular structure. As a consequence, Electro-Magnetic Analysis could be applied in order to focus the acquisition of information leaking to some relevant logic blocks.
4. FPGAs have multiple power sources. Analysis of their distribution inside the logic blocks could help to isolate some components of FPGAs.
5. Studying the security questions raised by the reconfigurability.

8 Conclusions

This work confirmed that power analysis has to be considered as a serious threat for FPGA security. Although certain features of our FPGA board made the practical implementation of power attacks significantly harder than in the smart card context, we have conducted relevant experimental tests. We analyzed the power of a DES running with weak keys and could clearly distinguish both keys. We also implemented a Differential Power Analysis attack against a toy-implementation of RSA. Many solutions would allow to improve our measurements, for example isolating the FPGA from its application board, and a lot of questions concerning the physical security of FPGAs remain open. As a future technological trend seems to be the combination of processors and reconfigurable hardware, there is a field for various research in the coming years.

References

1. P.Kocher, *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, In the proceedings of CRYPTO 96, Lecture Notes in Computer Science Volume 1109. Springer-Verlag, August 1996.
2. P.Kocher, J.Jaffe, B.Jun, *Differential Power Analysis*, in the proceedings of CRYPTO 99, Lecture Notes in Computer Science 1666, pp 398-412, Springer-Verlag.

3. T.S.Messerges, E.A.Dabbish, R.H.Sloan, *Examining Smart-Card Security under the Threat of Power Analysis Attacks*, IEEE transactions on computers, Vol.51, N5, May 2002.
4. P.Kocher, J.Jaffe, and B.Jun, *Introduction to Differential Power Analysis and Related Attacks*, Cryptography Research 607 Market Street, 5th Floor San Francisco, CA 94102, www.cryptography.com.
5. J.J.Quisquater, D.Samyde, *Electromagnetic Analysis (EMA): Measurements and Countermeasures for Smart Cards*, in Smart Card Programming and Security, Lecture Notes in Computer Science Volume 2140, pp.200-210, Springer-Verlag 2001.
6. National Bureau of Standards. *FIPS PUB 46*, The Data Encryption Standard. U.S. Departement of Commerce, Jan 1977.
7. R.Rivest, A.Shamir, L.Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, 21, pp 120-126, 1978.
8. Xilinx: *Virtex 2.5V Field Programmable Gate Arrays Data Sheet*,
<http://www.xilinx.com>.
9. Altera: *Flex 10K Field Programmable Gate Arrays Data Sheet*,
<http://www.altera.com>.
10. Proceedings of CHES 1999-2002 : Workshop on Cryptographic Hardware and Embedded System, Springer Verlag.
11. Proceedings of FPL 1999-2002 : The Field Programmable Logic Conference, Springer-Verlag.
12. D.Stinson, *Cryptography: Theory and Practice*, CRC Press 2000.

A Power-Scalable Motion Estimation Architecture for Energy Constrained Applications

Maurizio Martina, Andrea Molino, Federico Quaglio, and Fabrizio Vacca

Dipartimento di Elettronica – Politecnico di Torino
C.so Duca degli Abruzzi, 24 – 10129 TORINO (ITALY)
{maurizio.martina, andrea.molino, federico.quaglio,
fabrizio.vacca}@polito.it

Abstract. In the research community wireless devices are fostering many design and development activities. The augmented transmission bandwidth supplied by 3G transmission schemes will soon enable an ubiquitous fruition of multimedia content. This paper proposes a reconfigurable, power-scalable architecture for hybrid video coding, suitable for the mobile environment. The complete FPGA design flow shows very interesting performances both in terms of throughput, and power consumption.

1 Introduction

In the last few years, reconfigurable power-aware systems have gained a growing interest in the scientific community. The current global coverage wireless networks are designed specifically to transmit circuit-based voice; however, market analyses foresee that in 2005 50% of wireless traffic, and 80% in 2010 will be non voice, e.g. data transfer and multimedia applications. The major technological challenge will be then to guarantee wireless video access with an acceptable quality, real time response and interactivity. However, behind all these attractive motivations, there are many critical factors that could seriously tackle the design and implementation of multimedia-aware mobile terminals. The design of wireless terminals, in fact, has to deal with limited power budgets and reduced resources availability. Field Programmable Gate Arrays (FPGAs) emerged as the leading technology for their capability of being easily reconfigurable with a quite low effort, allowing, at the same time, to implement even large designs. They became not only powerful prototyping tools for complex designs, but also attracting alternatives to Application Specific Integrated Circuits (ASICs). Thus, FPGAs can be used instead of ASIC where both complex elaborations and certain reconfigurability are the main goals.

Different working profiles could be provided in order to obtain scalable performance (e.g. video quality, frame rate ...) with different levels of power dissipation. Reconfigurable blocks have to be designed to achieve several energy consumption profiles providing, at the same time, the necessary elaboration tasks.

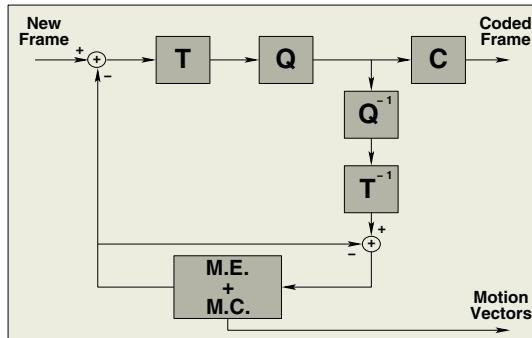


Fig. 1. A block scheme of an hybrid video coder

Good choices for this paradigm can be both DSPs and FPGAs. While the former are able to grant a quite large set of reconfigurable options, they usually suffer from high dynamic power consumption. This figure can seriously tackle their employment in strong power constrained environments, especially when computationally intensive algorithms will be used. On the other hand, FPGAs resemble the proper candidates to obtain a certain degree of reconfiguration and the needed elaboration capabilities with acceptable power consumption, provided that technology improvements can reduce static power consumption.

Power-Scalable solutions must be carefully designed in order to reduce the overall power dissipation. Shut-down policies of the most consuming blocks have to be exploited to reach effective energy savings without compromising the efficiency and the functionality of the entire system. In order to achieve different energy profiles, a Power-Scheduler has to be added to the terminal architecture either as an hardware block or as an Operating System's feature. In this work we investigate the feasibility of a new Motion Estimation IP for hybrid video coding: the main novelty behind the proposed architecture lays in the availability of different power consumption profiles. The paper is organized as follow. In Section 2 hybrid video coder principles and Motion Estimation techniques, for low-power operation, are discussed. In Section 3 the proposed IP for Motion Estimation, based on the previous discussion, is analysed. In Section 4 the IP implementation and its performances are presented; while in Section 5 some conclusions are drawn.

2 Hybrid Video Coders

In figure 1 a simplified description of an actual implementation of an hybrid video coder is sketched. Analysing the coder structure, a *direct path* and a *feedback path* can be identified. The direct path exploits the *spatial correlation* among near pixels in a frame; it is highly probable, in fact, that they can have similar values, thus low-frequencies components are the most relevant in the frame spectrum. To identify them, every new frame produced by the image sensor,

Table 1. Percentage of video coder blocks usage

Building Block	% Usage
Transformation	39.3 %
ME block	47.2 %
Quantiser	5.4 %
Entropy Coder	0.8 %
Other	7.3 %

passes through a *transformation* stage (**T**). The transformed image, then, passes through a *quantiser* (**Q**), that reduces the amount of bits needed for the frame representation. Then, the remaining bits are coded via an *entropy coder* (**C**) to adapt the information to communication medium's capacity. All these processes, typical also for still pictures, produce the so called Intra Frame (**I**).

The feedback path, on the other hand, tries to detect the *temporal correlation* among frames, i.e. successive frames have, with high probability, little scene variations with respect to the formerly coded ones. The objective is to *predict* the new frame, based on the knowledge of the previous ones, and then to transmit only the differences between the predicted and the real frame. Such this *error frame* passes through the **T**, **Q** and **C** stages obtaining then a so called Inter Frame (**P**) that requires few bits to be represented. The main hint in this step is that the feedback path has to "simulate" the receiver's steps before trying to predict the next frame. This must be done to avoid differences between the transmitter and the receiver, the prediction stage must work on the same quality frames than the receiver does. The image has to pass through a *de-quantiser* (\mathbf{Q}^{-1}) and an *anti-transformation* stage (\mathbf{T}^{-1}) to produce a frame similar to the one produced by the image sensor. This reconstructed frame is then stored in a local buffer. After, it is compared with a previously stored image to identify the differences, due to moving objects. This step, called *Motion Estimation* (**ME**), detects the motion occurred between the images in terms of *Motion Vectors* (**MVs**). A subset of these **MVs** are used to predict the next frame just applying them to the current frame during the *Motion Compensation* (**MC**) step. This prediction is compared with the actual image to obtain the aforementioned error frame. Understanding the computational effort of each block aids in designing a power efficient coder. Thus, we perform a profiling analysis on a software model of an H.263 video coder¹. As it can be inferred analysing table 1, Motion Estimation is the most demanding task; therefore a proper hardware implementation can reduce the overall consumption achieving better power figures.

Block-based Matching Algorithms,[2], divide the whole frame in $L \times W$ "mosaic" of *macroblocks* (**MBs**) each made of $N \times N$ pixels. Then each **MB** of the current frame is compared with a set of macroblocks of the past stored picture, obtaining the most correlated block, with the one of the current frame. Knowing the coordinates of both these **MBs**, the **MV**, is evaluated. A suitable metric is the *Sum of Absolute Differences* (**SAD**) function that reaches a minimum value

¹ The C code was developed at the University of British Columbia, [1].

when the correlation function has its maximum, [2], [5]. The SAD function requires, therefore, $N \times N$ operations to evaluate each macroblock, exhibiting a $O(N^2)$ computational complexity. Reducing this function's evaluation implies fewer power-consumption to perform ME. Depending on the search strategy employed, two separated classes can be identified: *Exhaustive Search Class* and *Heuristic Search Class*.

The former class is basically formed by the well-known Full Search (FS) algorithm, [2], [3]. As it can be inferred from the name, in this search strategy each macroblock in the current frame is compared with all MBs of the past image. Thus the best correlated block is straightforwardly identified but a great energy effort is required. In fact, considering a frame divided into $W \times W$ macroblocks the search complexity is $O(W^2)$; each search step has complexity $O(N^2)$ therefore the FS algorithm can require $O(W^2 N^2)$ operations.

To reduce the computational needs of the Full Search other sub-optimal search strategies have been proposed, all based on an assumption on the SAD function properties. Thus a reduced number of steps is needed to identify the best correlated frame. One of these sub-optimal algorithms was the so-called *Three-Step Search* (TSS) algorithm, [2]. In this approach, a constellation of 9 MBs, centred in the past frame, are evaluated to identify the block that exhibits the minimum SAD value. This MB becomes the central macroblock of a new constellation where the relative distance among the blocks is halved. The search is stopped when the macroblocks are one-MB-distance one from the others. Considering a frame of $W \times W$ macroblocks where $W = 15$, to perform a complete search requires 25 steps therefore $O(25N^2)$ operations; thus, there is considerable reduction with respect to the Full Search. One of the major drawbacks of TSS is that it is not *centre-biased*, i.e. in typical video sequences the main amount of motion is in the central area of the frame, [5]. Higher video quality can be achieved if the search process is performed first in this region. This can be done, with low computational complexity, resorting to the *Four-Step Search* (FSS) algorithm. This algorithm relies on the 9-MBs constellation used by TSS, but, in this approach, the macroblocks are 2-MB-distance long, one from the others, in all the search steps. This strategy is therefore centre-biased, and the algorithm stops when the minimum SAD value block is the one in the centre of the constellation. Then it is performed the so-called *Shrinking* step, i.e. the SAD function is evaluated also in the 8 MBs around the central one and 1-MBs-away from it. The shrinking step can be performed anytime during the search process so an analytical evaluation of the computational complexity is not possible. From simulations' results, otherwise, it has been found that FSS has a mean complexity of $O(19N^2)$ operations, [5]. Another algorithm which is an improvement of TSS, is the so-called *Simple and Efficient Search* (SES) algorithm, [4]. Based on the aforementioned assumption on SAD function, it is possible to identify the direction and the verse of the motion vectors without analysing all of the 9 macroblocks of the TSS. The SES algorithm starts to evaluate the SAD function in the central macroblock and in other two blocks that are in two orthogonal directions. The basic idea is to divide the frame

into four quadrants; depending on the value assumed by the SAD functions in these three MBs one of these quadrants is selected. This allow to evaluate only other few blocks of the 9-MBs constellation used by TSS. This approach allows a further computational reduction, approximately $O(12N^2)$ operations.

3 The Proposed Architecture

To obtain proper video-quality with reasonable power consumption we designed a Motion Estimation IP based on sub-optimal search strategies. Furthermore, it has to achieve different energy profiles that can be implemented on FPGAs. In this section we present the chosen algorithm and we validate its estimation properties. Heuristic search is needed in order to avoid energy waste due to a full search. Moreover, in video-surveillance applications an high video quality is not required since the main task is to properly detect abnormal situations. This means that video quality, and also the stream frame-rate, can be adapted depending on the environment conditions.

To obtain good video quality with low computational efforts, we have chosen the so-called *Majority Voting Algorithm* (MVA), [6]. This algorithm relays on the aforementioned FSS and TSS ones choosing, depending on a particular merit criterion, which of them has to be used. This is done because near macroblocks tend to exhibit similar motion, so chosen the proper search strategy for a MB is better performed by knowing the motion occurred by its neighbour. This approach can further reduce the overall search without compromising the quality. In fact, FSS, that is centre-based, can properly detect the motion in the centre of the frame, while TSS works well in all the other cases. The choice between TSS and FSS is based on the motion vector values of some of the MBs near the one under elaboration and called “Predictors”. These values are compared with a *threshold* that varies depending on the frame format (CIF, QCIF or others).

To achieve further computational reduction, we chose to implement the SES algorithm rather than the original TSS one; in fact SES can be considered like an improvement of the TSS scheme. Moreover, other reductions can be obtained selecting, when it is possible, a sub-sampling strategies called *Pixel Decimation*, [7]. In this approach, the SAD function is evaluated only for $N^2/4$ of the $N \times N$ MB pixels. This means some video quality losses, but great improvements in the power savings. Because of these losses, this strategy can be selected depending on the environmental conditions.

To validate our choices, we wrote a “software model” in C and we compared the estimated motion vectors of our approach with the ones obtained with the Full Search algorithm. In fact, FS has better estimation properties and can be used as *benchmark*. We analysed the differences in term of displacements between the MVs’ values obtained by FS and our MVA implementation. The statistical frequencies of these displacements, representing the estimation errors of our model are depicted in figures 2(a) and 2(b). These values have been obtained comparing the estimations of several frames from some well known

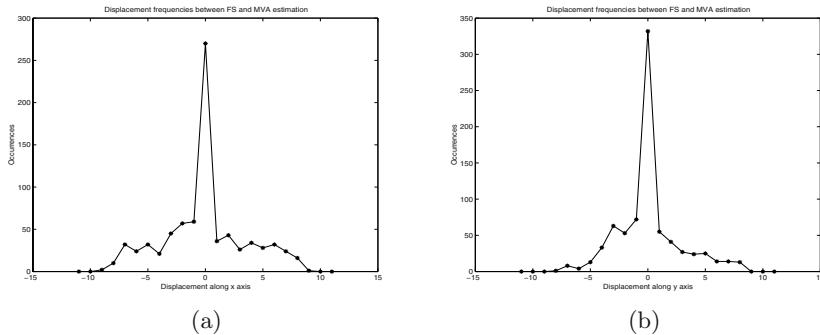


Fig. 2. Displacement between FS and MVA MV. (a) Along x-axis, (b) Along y-axis.

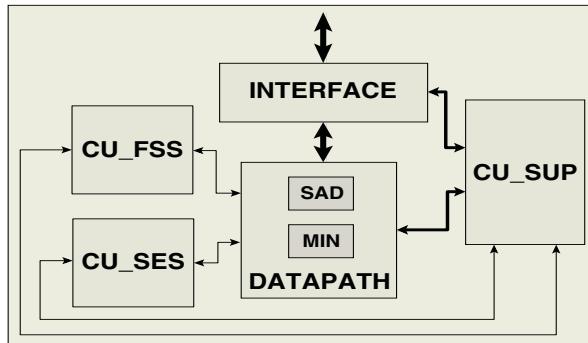


Fig. 3. Block-scheme of the proposed IP

test sequences like *Foreman* and *Coast-guard*. These sequences are all in QCIF format (176×144 pixels) with 16×16 pixels MBs and without Pixel Decimation.

Analysing these results, we can see that our MVA commits few errors with respect to the Full Search, in fact the statistical distributions, that are similar the well-known Laplace's Distribution, have zero mean and little variance. This has given us good confidence on our algorithm estimation properties; thus it is a good choice for an hardware implementation.

The block-scheme of the proposed IP is depicted in figure 3. As it can be seen this IP is made of several building blocks. The first one is the *Interface* block. As it was observed previously, this architecture is intended to be able to work with different profiles; this means that it has to accept different frame formats in particular CIF (352×288 pixels) and QCIF (176×144 pixels) sequences. So, an external CPU can select the proper format sending to our IP the frame “dimension”, in terms of pixel. Moreover the MBs’ format, like 16×16 or 8×8 pixel, can be chosen during the elaboration time. Also Pixel Decimation can be programmed by the control microprocessor (μ P). This one has to provide to our IP the corresponding thresholds, to be used in the MVA algorithm, for the chosen frame format. All these features make our architecture able to be

reconfigured “on-line” during the operation time. Moreover the interface block is used to access the external frame buffer in which the current and the past frame are stored.

The *CU_SUP* block, controls the information interchange with the CPU when this one programs the IP. Moreover, it selects the proper MB, of the current frame, on which evaluating the motion estimation. This block is also devoted to select, based on the knowledge of the predictor block MV values and of the threshold values, the proper algorithm, FSS or SES, for the estimation. This is done alternately enabling and disabling the control units devoted to the operation flow of these two search strategies.

In particular, the *CU_SES* block is the control unit that implements the Simple and Efficient Search algorithm. On the other hand, the *CU_FSS* block controls the FSS algorithm’s operation flow. In the *Data-path* block, otherwise, all the required analytical elaborations are performed. In particular, the SAD is evaluated for the chosen macroblocks of the search constellations by the *SAD block*. Then, the MB that exhibits the minimum SAD value is chosen by the *MIN* block.

4 Implementation and Results

All of these blocks were described in VHDL code and then synthesised on a Xilinx XCV300E FPGA. To improve the reconfigurability, the code was written by means of **generic** parameters. In this way, our IP can be adapted to the particular operative conditions just varying these parameters and re-synthesising the code. This provides the so-called “off-line” reconfigurability.

The FPGA *speed-grade* for the implementation was set to 8 and 352-pin Ball-Grid Array (BGA) package was chosen. The synthesis step was performed by means of Synplify_Pro tool from Simplicity. The synthesised structure was, then, Placed & Routed on the selected FPGA. This step was accomplished by mean of the WebPack tool provided by Xilinx. From these previous steps, we obtained some preliminary information on the architecture performances, addressed in table 2. The LUT (look-up table) occupation means the percentage of the total FPGA look-up tables used by our design; while the REG occupation is related with the percentage of non-Input/Output registers used. From table 2 it can be seen that there is a reduction for the maximum frequency achievable by the IP, between the post synthesis and the post Place & Route structures. This reduction is mainly due to some internal *critical paths* within

Table 2. Area & frequency IP performances

	Synthesis	Place & Route
LUT occupation	55 %	55 %
REG occupation	16 %	16 %
Maximum Frequency	38 MHz	23 MHz

the obtained IP. Further studies have to be exploited in order to eliminate them and to achieve better frequency performances.

To be of practical use, our architecture has to achieve very small energy requirements. Thus, we need to have some estimations of the dissipated power in order to have some energy profiles. This task was accomplished using the XPower Analyzer provided by Xilinx. First of all, we evaluated a statistical power consumption value setting the *activity ratios* of all the involved signal to 12.5 %. To obtain a more realistic estimation, we produced a test-bench to validate the overall IP. By means of Modeltech tool from Mentor Graphics, we were able to extract all the actual activity ratios for the IP signal and to store them in a so-called VCD file. This file became an entry for XPower that could perform a more accurate power dissipation estimation. We performed this evaluations doing the motion estimation on frames in QCIF format without applying Pixel Decimation and operating at the maximum frequency. All these results are depicted in table 3. Total P_D represents the overall FPGA power consumption due both to static and dynamic contributions. As it can be seen from table 3, the static power overcame the dynamic one acting as the limiting factor for the use of commercial FPGA in mobile devices. In fact, Static Power is due to internal implementation of the FPGA, therefore is a characteristic of each device family and the designer has no control on it. The dynamic P_D , on the other hand, can be reduced by a proper choice of the architecture that has to be implemented.

Table 3. Power Consumption IP performances

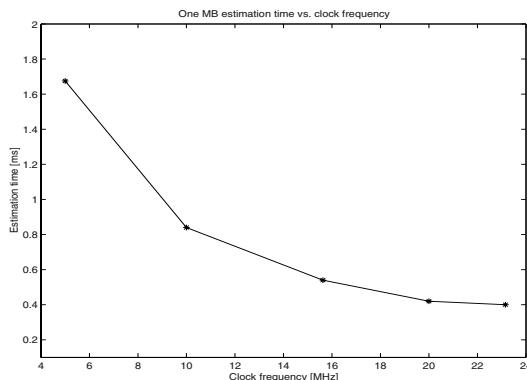
	Statistical estimation	Actual estimation
Total P_D	637 mW	579 mW
Worst Case		
Static P_D	540 mW	540 mW
Dynamic P_D	97 mW	39 mW

To analyse the power-saving properties of our architecture, we compared its energy consumption with the ones achievable running our C-code “software model” over some *general-purpose* μ Ps. We also compared the elaboration time needed to perform motion estimation for a QCIF frame for all the considered processors. The results of the comparisons are presented in table 4. As it can be seen, our IP achieves the lowest dissipation and one of the lower elaboration time. StrongArm SA1100 obtains a quite low energy consumption, but needs a long computation time². On the other hand, a reasonable elaboration time is achieved by Mobile Celeron processor, but it has an higher power dissipation, i.e. 23.8 W, and an higher cost that makes its use impractical in an most wireless

² This results were obtained with the JouleTrack tool, available at: <http://www-mtl.mit.edu/jouletrack/JouleTrack/index.html>

Table 4. Performances Comparisons

Selected processor	Elaboration time	Dissipated energy
Proposed IP	39,6 ms	0.023 J
StrongArm SA1100 206 MHz	1.1 s	0.39 J
Mobile Celeron 1,1 GHz	0,02 s	0.48 J
UltraSparc III	0,11 s	8.8 J
UltraSparc I	0,40 s	11.2 J

**Fig. 4.** One MB evaluation time vs. clock frequency

devices. The other processors have been addressed only for the purpose of having some evaluation metrics for the comparisons.

The proposed architecture is able to evaluate a 16×16 pixel macroblock (of the current frame) in less than $400 \mu\text{s}$ operating at maximum frequency; therefore, our IP performs the motion estimation on a 99 MBs' QCIF frames in less than 40 ms. Thus, frame-rates of 25 frames/s are achievable. ME for CIF frames, is exploited with rates of 6 frames/s, but high-rates are not the main target in video-surveillance operations. Furthermore we can applying Pixel-Decimation, at a cost of little losses in video-quality, obtaining higher rates even for CIF sequences³. Further power savings can be obtained reducing the operating frequency as was sketched in[8]; this comes at the cost of longer elaboration periods. We exploited this feature, evaluating our IP performances reducing the clock frequency; the obtained results are depicted in figure 4. These values can be used as evaluation metrics for selecting the proper power dissipation/elaboration time trade-off while designing a energy constrained application.

³ The higher resolution of CIF format can compensate the losses due to Pixel Decimation.

5 Conclusions

In this paper we presented a novel Power-Scalable Motion Estimation IP suitable for nomadic use on wireless terminals. The proposed architecture can achieve low dynamic power, good video quality and reasonable frame-rates. Moreover it can operate on different video format and can be reconfigured both off-line and on-line. Further researches have to be accomplished in order to reduce the internal critical path, achieving better maximum frequencies performances. Moreover, new FPGA architectures must be exploited searching low-static-power devices suitable for an actual implementation of our IP on a demonstrator platform.

References

1. M. Gallant, A. Joch, G. Cote and B. Erol, H.263+ Library software codec simulator—Version 0.2, University of British Columbia—Signal Processing and Multimedia Group
2. C. Stiller and J. Konrad, Estimating motion in image sequences – A tutorial on modeling and computation of 2D motion, *IEEE Signal Processing Magazine* **16**, 4 (1999) 70–91.
3. V. L. Do and K. Y. Yun, A low-power VLSI architecture for Full-Search block-matching motion estimation, *IEEE Transaction on Circuits and Systems for Video Technology* **8**, 4 (1998) 393–398.
4. J. Lu and M. L. Liou, A Simple and Efficient Search algorithm for block-matching motion estimation, *IEEE Transaction on Circuits and Systems for Video Technology* **7**, 7 (1997) 429–433.
5. L. Po and W. Ma, A novel Four-Step Search algorithm for fast block motion estimation, *IEEE Transaction on Circuits and Systems for Video Technology* **6**, 3 (1996) 313–317.
6. D. S. Turaga and T. Chen, Estimation and mode decision for spatially correlated motion sequences, *IEEE Transaction on Circuits and Systems for Video Technology* **11**, 10 (2001) 1098–1107.
7. A. Zaccarin and B. Liu, Fast algorithms for block motion estimation, *Proceedings of the 1992 IEEE International Conference on Acoustic, Speech and Signal Processing* 3 (1992) 449–452.
8. A. Sinha and A. Chandrakasan, Dynamic power management in Wireless Sensor Networks, *IEEE Design & Test of Computer* **18**, 2 (2001).

A Novel Approach for Architectural Models Characterization. An Example through the Systolic Ring

P. Benoit¹, G. Sassatelli¹, L. Torres¹, M. Robert¹, G. Cambon¹, and D. Demigny²

¹ LIRMM, UMR UM2-CNRS C5506,
161 rue Ada, 34392 Montpellier Cedex 5, France
(33)(0)4-67-41-85-69
name@lirmm.fr

² ETIS, UMR-CNRS 8051,
6 avenue du Ponceau, 95014 Cergy Pontoise Cedex, France
(33)(0)1-30-73-66-10
demigny@ensea.fr

Abstract. In this article we present a model of coarse grained reconfigurable architecture, dedicated to accelerate data-flow oriented applications. The proliferation of new academic and industrial architectures implies a large variety of solutions for platform-based designers. Thus, efficient metrics to compare and qualify these architectures are more and more necessary. Several metrics, *Troughput Density*[3][12], *Remanence*[4] and *Operative Density* are then used to perform comparisons on different architectures. Architectures are often customisable and purpose several parameters. Therefore, it is crucial to characterize the architectural model according to these parameters. This paper proposes as a case study the Systolic Ring, and gives a set of metrics as functions of the architecture parameters. The methodology illustrated is generic and proved very efficient to highlight architectural properties such as the scalability.

1 Introduction

A System on Chip (SoC) allows the integration of a whole system on the same silicon die by combining different IP (Intellectual Property) cores. This technology provides significant benefits such as decreasing device's cost and power consumption and is therefore suitable for embedded communication products. Thanks to process geometries dropping, these systems are more and more complex bringing attractive functionalities such as multimedia abilities. However, design techniques must fit silicon technologies evolution in order to cope with the time to market constraints: it is obvious that it is more and more difficult to validate the whole functionality of an entire system. In order to reduce the designing times, SoC providers purpose customisable platform-based designs. Thus, the provider can adapt the platform to different customer needs (size of memory and buses, hardwired accelerators etc.) with a customisable architectural model. This methodology allows to significantly decrease the circuits' time to market.

In this context, customisable cores seem to be very attractive: by changing few parameters of the architectural model, different kind of optimisation could be

imagined: for example, the processing power/area trade-off. In a SoC context, this ability allows to cope with different customer needs at the core level; this means that the core architectural model must have been fully characterised according to the parameters of the architecture. From the SoC provider point of view, the most competitive and attractive platform will be the one made of the most evolutionary, customisable and scalable soft cores.

The lack of flexibility of specific hardware has motivated the integration of reconfigurable cores. These architectures provide hardware-like acceleration style while retaining most of the software flexibility : a simple bit-stream defines the functionality. Among the last couple of years lots of new approaches appeared [1]. Real innovations like coarse grain reconfigurable fabrics [2] or dynamical reconfiguration have brought numerous improvements, solving several weaknesses of traditional FPGA architectures. Besides this point, several recurrent issues remain, and the proliferation of architectures lays on an additional problem for platform-based designers: choose the right IP core for a given set of specifications. Some works have already proposed useful tools to directly compare reconfigurable computing architectures, like the Dehon criterion[3][12]. Nevertheless, none addresses the characterisation of architectural models. This characterisation supposes to evaluate metrics as a function of the architecture parameters.

Efficiency and comparison of architecture is always a major problem to address. In this paper, we distinguish two orientations. The first one considers the application implementation efficiency. The second one is based on intrinsic characterisation of the architectural model. The goal of this paper is to propose a method to characterize an architectural model: the approach proposed here is performed on a case study, but could be easily extended to any coarse grained reconfigurable architecture.

The next section presents a model of coarse-grained dynamically reconfigurable architecture, the Systolic Ring. This architecture already presented in [4][5], will be used as an example in our characterisation approach.. The third section presents metrics dedicated to compare different architectures. Finally, we propose a novel approach of using intrinsic metrics to characterise an architectural model.

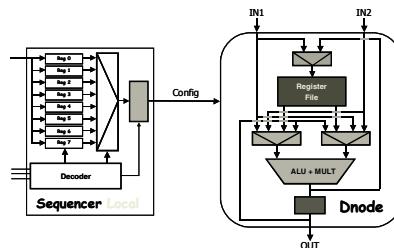


Fig. 1. The Dnode architecture

2 The Systolic Ring Architectural Model

The objective of this section is to briefly describe the Systolic Ring architecture, already presented in [4][5]. This architectural model will be used as an example in the fifth section to present the way to characterize a model of reconfigurable architecture.

The Systolic Ring architectural model features a highly optimised DSP-like coarse grain reconfigurable block, the Dnode (figure 1). The Dnode, a processing element, is the building block of the architectural model. This component is configured by a fixed-size microinstruction code. The configuration comes from a local sequencer. This sequencer can manage the Dnode configuration autonomously (up to eight instructions) or can be managed at a higher level by a global sequencer.

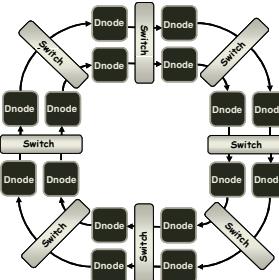


Fig. 2. . The operative layer

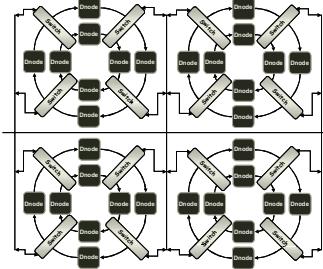


Fig. 3. Multi - Systolic Ring instances

In the Systolic Ring model, at a higher processing elements are brought together in clusters. A cluster is called “layer”. A layer is made of a number “N” of Dnodes; N is the first parameter of the architectural model. In order to interconnect these layers, a configurable switch component is used to establish a full connectivity between two layers and to inject data from the memory. The number “C” of layers used in the model is the second parameter. The global interconnect structure of the processing elements is depicted on figure 2 : this example of the Systolic Ring model is made of 8 layers and 2 Dnodes per layer. The last layer is connected to the first layer with a switch (each switch is connected to the data memory via dedicated FIFO inputs). Thus, the topology is circular, allowing an efficient implementation of pipelined datapaths of different sizes through the successive layers. The Systolic Ring also provides reversed datapaths through a feedback network made of pipelined registers. A reversed datapath is associated to each switch. In addition, a Systolic Ring bus connecting all switches in the architecture and the global sequencer is also available. An other way to increase the processing power in a Systolic Ring architecture consists in using multiple instances of the circular datapaths. Figure 3 depicts this third interconnection level. The Systolic Ring bus is then used to interconnect the number S ($S=4$ in the example) of Systolic Ring instances. S is the third parameter of the Systolic Ring architectural model.

3 Metrics for Architecture Comparisons

The main goal of metrics consists of evaluating quantitatively and relatively the performances of the architecture to implement an application (application oriented metrics) and of quantifying the structure intrinsic characteristics (intrinsic oriented metrics).

3.1 Application Oriented Approach

The most used metric to compare computing machines to implement an algorithm is the *Throughput Density* [3], also presented in [12]. This metric is computed as follows: $D = I / (Area * Time)$. The area is expressed in relative unit, λ^2 , where λ is the half size of the minimal width gate in a given technology (for more details, refer to [3]) and the time is expressed in absolute unit, *i.e.* seconds. This metric highlights the architecture which uses the most efficiently the silicon (smallest time and area) for closed technologies and a given application.

3.2 Intrinsic Approaches

3.2.1 Remanence

Remanence has been introduced in [6]. It characterises the architecture dynamism and is computed as follows: $R = (Na.Fe) / (Nc.Fc)$. Na is the number of the architecture building block (operators for coarse grain reconfigurable architectures and processors, CLBs for FPGA). Nc is the number of configurable building blocks per cycle. Fe is the building blocks operating frequency and Fc the configuration frequency. The architecture granularity and the technology do not influence the resulting *Remanence* due to the fact that *Remanence* is a ratio.

Remanence represents the number of cycles to (re)configure the whole architecture. For instance, a “very dynamic” architecture has a *Remanence* value equal to one (all the building blocks are configured in only one cycle) and on the opposite, “a highly static” architecture has a *Remanence* value very important.

3.2.2 Scalability and Operative Density

An architectural model is scalable when the number of building blocks increase implies a linear increasing of the architecture area. In this case, the area “A” can be expressed as a linear function of the number of the building blocks “N” as follows: $A(N)=k*N \forall N$. Consequently, the derived function of $A(N)$ is a constant : its value is equal to k , the increasing factor. The value of k can be then computed for any value of N and determined as follows: $k=A(N)/N \forall N$. The inverse of k , $1/k$, is an interesting characteristic of an architectural model, because it represents the *Operative Density* “ Do ”. Do is computed as follows: $Do(N) = N/A(N) = 1/k$. The *Operative Density* allows to characterise the number of building blocks per area unit.

In order to avoid the influence of the technology, we propose to use a relative unit, λ^2 , to express the area of the whole architecture as a function of the number of the building blocks. When the architecture is scalable, Do is a constant whatever the value of N . However, Do can also be used to illustrate a non-scalable architectural model, to characterise the evolution of the operative density as a function of the architectural model parameters. A complete illustration of this is given in the following section.

3.3 Comparisons

For these comparisons, we have selected three types of architecture dedicated to digital data processing: FGRA or Fine Grained Reconfigurable Architectures (Xilinx

FPGA Virtex E[7] and ARDOISE, a platform based on Atmel FPGA[8]), CGRA or Coarse Grained Reconfigurable Architecture (The Systolic Ring architecture, DART [9], MorphoSys [10]) and a VLIW DSP from Texas Instrument (Very Long Instruction Word Digital Signal Processor), the TMS320 C62 [11].

3.3.1 Throughput Density Comparisons

The results of throughput densities are summarized in table 1. The application considered is a classical algorithm of digital signal processing: the Discrete Cosine Transform (DCT). It is supposed that this algorithm is applied to a 64*64 image, on 8*8 blocks (two-dimensional DCT).

Table 1. Throughput densities comparisons

Name	Type	Techno.	Implement.	N	# cycles	f (MHz)	Time (μs)	Area (MΛ ²)	D
Virtex E	FGRA	0.18μm	Signal Flow	N/A	4171	80	52.14	8000	2.4.10⁻⁶
Systolic Ring	CGRA	0.18μm	Matrix	24	6826	200	34.13	500	5.8.10⁻⁶
DART	CGRa	0.18μm	N/A	24	9536	130	73.35	300	4.5.10⁻⁶
MorphoSys	CGRa	0.35μm	Signal Flow	128	1344	100	13.44	5500	1.3.10⁻⁶
TMS320C62	DSP VLIW	0.15μm	Matrix	8	10240	300	34.13	12300	2.4.10⁻⁶

Firstly, we can notice that different algorithm implementations are used. In order to compare the most objectively *Throughput Densities (D)*, the same implementation would have to be mapped on each architecture. Indeed, it is well known that a signal flow implementation allow better performances. An other point to consider is the CMOS technology. Time is given in absolute units (seconds): thus, only close feature sizes can be objectively compared. Consequently, we can suppose that MorphoSys would reach greater performances with similar silicon technologies as the other architectures presented in the table. Then, we observe that the DSP and the FPGA have a very close *D* value (around 2.4 .10⁻⁶). The two other examples of CGRA reach a better throughput density due to the fact of the parallel and pipelined topology of the operators allowing a better implementation.

Even if the results proposed here are interesting and useful to qualify these architectures, they are also limited because targeting only one algorithm with several implementations and different feature sizes. In order to compare efficiently the five architectural models, it would be necessary to produce results as a function of the architectural model parameters on several applications and closed feature sizes.

3.3.2 Remanence Comparisons

For the five architectures presented in the table 2, configuration and operating frequencies are equal. The building block granularity can differ from an architecture to an other one. Thus, for the ARDOISE architecture based on an partially dynamically reconfigurable AT40K FPGA [8], the building block grain is the CLB: consequently, 2304 CLBs are available. Seven cycles are needed to configure one CLB, therefore 0.14 CLB is reconfigured in one cycle. Concerning the four others architectures, the grain is coarse: for the Systolic Ring, DART and MorphoSys, the building blocks are made of one multiplier and one ALU. We will consider for each

one that the total number of operators is the product of the number of building blocks by 2.

These *Remanence* values show the dynamism of the architecture or the number of cycles needed to configure the whole structure. Hence, the DSP is the most dynamic architecture because it is able to configure the 8 operators in only one cycle. The CGRA architectures can also behave very dynamically by reconfiguring the whole structure in only one cycle thanks to SIMD control modes. They also propose more elevated *Remanence* allowing to configure more slowly the whole architecture but then can fix their configuration reducing consequently the consumed power by the configuration. The highest *Remanence* value is reached by the FPGA showing that if it is needed to reconfigure the architecture, 16457 cycles are necessary.

Table 2. Remanence comparisons

Name	Type	Na	Nc (min)	Nc (max)	F	R (min)	R (max)
ARDOISE	FGRA	2304	0.14	0.14	33	16457	16457
Systolic Ring	CGRA	24	0.5	24	200	1	48
DART	CGRA	24	1	24	130	1	24
MorphoSys	CGRA	128	1	128	100	1	128
TMS320C62	DSP VLIW	8	8	8	300	1	1

3.3.3 Operative Density Comparisons

As stated in the previous parts, multipliers and ALUs are separated to compute the total number of operators(N). Here, in order to compare the most objectively different architectures, we only consider coarse grained operators. By the way, for the ARDOISE architecture, the number of operators (26) is the number of synthesized operators on the whole FPGA architecture (2304 CLBs).

Table 3. Operative densities comparisons

Name	Type	N	Area($M\lambda^2$)	Do (N)
ARDOISE	FGRA	26	12300	$0.21 \cdot 10^2$
Systolic Ring (S=1, C=6, N=2)	CGRA	24	500	$4.80 \cdot 10^2$
Systolic Ring (S=4, C=8, N=2)	CGRA	128	2700	$4.74 \cdot 10^2$
DART	CGRA	24	300	$8.00 \cdot 10^2$
MorphoSys	CGRA	128	5500	$2.32 \cdot 10^2$
TMS320C62	DSP VLIW	8	12300	$0.06 \cdot 10^2$

The *Operative Densities* results show that CGRA use more efficiently the silicon area due to the fact of the hardwired operators. The area dedicated to control units in the DSP involves for this one the worst operative density value. The ARDOISE architecture operative density, because of the fine grained granularity, is worse than CGRAs. Concerning CGRA, DART has the greatest *Do* value, meaning that the number of operators per $M\lambda^2$ is the most elevated, conferring a greater processing power per $M\lambda^2$. The important interconnection resources available for the Systolic Ring and MorphoSys results in a worse *Do* value.

4 Characterize a Model: An Example through the Systolic Ring

The metrics previously described provide very useful results in order to compare different architectures. However, architectural models such as the Systolic Ring, proposing parameterization abilities, are not fully characterized. Therefore, it seems to be very important to propose a novel way of using previous metrics to reach this objective of characterization. The idea consists in computing the metrics as a function of the architectural parameters. By the way, a fully characterized core could be integrated in platform-based design and parameterized following the customer needs. For a platform-based design provider, this full characterization could be used as a tool to compare different architectural models in order to choose the best one according to his needs.

4.1 Remanence Analysis

The Systolic Ring architectural model is based on four management modes. Thus, we will provide four *Remanence* formalisations. Because the number of Systolic Ring instances is not an influencing factor, it will not appear in the *Remanence* results.

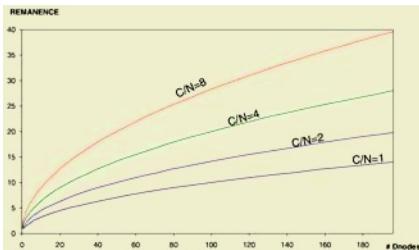


Fig. 4. Global Mode Remanence

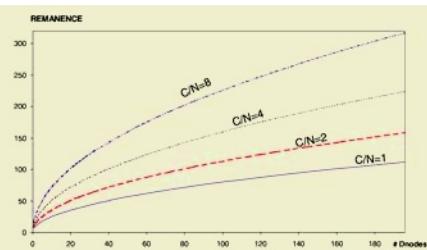


Fig. 5. Maximum of Local Mode Remanence

In global mode, one layer of the architecture is configured each cycle, therefore N Dnodes are configured each cycle ($N_c=N$). The total number of Dnodes being $C \cdot N$ ($N_a=C \cdot N$), we easily deduce that the *Remanence* R is equal to C . In order to represent the evolution of the *Remanence* value as a function of the number Dnodes, we take four examples of customized Systolic Ring. This customization is based on the C/N ratio. This ratio represents the trade-off made on the pipeline degree versus spatial parallelism. To increase the pipeline degree, the C/N ratio must be elevated. At the opposite, to increase the parallelism potential, the C/N ratio must be low. The four ratios chosen are 1, 2, 4 and 8 : we can observe on the figure 4 that the higher the pipeline, the higher the *Remanence*. We also remark that this *Remanence* is increasing quickly for low $C \cdot N$ values (few Dnodes), and more slowly for higher $C \cdot N$ values. We can also highlight that the most elevated value of R is less than 40 for more than 180 Dnodes, meaning that a 180 Dnodes Systolic Ring can be totally reconfigured in less than 40 cycles.

In local mode, local configuration registers of the Dnode sequencer are pre-programmed to execute a maximum of 8 instructions corresponding to 8 Dnodes configuration. One layer being addressed during the configuration phase, the number of Dnodes configured each cycle is N divided by the number of registers

preprogrammed. Consequently, supposing that all the Dnodes are preprogrammed with the same number of instructions (from two instructions to eight instructions), we easily deduce that the minimum of the *Remanence* will be $2C$ and the maximum, $8C$.

These results are shown on the figure 5. As a consequence, we can observe that local mode involves higher *Remanence* values meaning that the whole architecture will need more cycles to be reconfigured. However, the maximum reached is only 320 cycles, and is even lower than a dynamic FPGA *Remanence* (16457 for ARDOISE).

An other means to manage the Systolic Ring is the SIMD mode. In SIMD global mode, all the Dnodes receive the same configuration instructions. This is performed in only one cycle. Therefore, the *Remanence* value is one, meaning that the whole architecture is reconfigured in only one cycle. In SIMD local mode, each Dnode local sequencer register is pre-programmed with the same instruction. Therefore, the *Remanence* value in this mode is equal to the number of pre-programmed registers (from two to eight). With these low *Remanence* values SIMD mode is the most dynamic mode to manage the Systolic Ring.

4.2 Scalability Analysis

This analysis is based on the operative density evaluation. In order to compute this metric, we need to determine the area of the architectural model as a function of the number of processing elements. The total area is approximated by the sum of the 4 constituting elements of our model (PE : Processing Elements ; Config : Configuration memory):

$$A_{\text{TOTAL}} = S \cdot [A_{\text{Config}}(C,N) + A_{\text{Control}}(C,N) + A_{\text{Interconnect}}(C,N) + A_{\text{PE}}(C,N)] = f(S,C,N)$$

The total area A_{TOTAL} is a parameterised function depending on the three architecture parameters. The number of Dnode in the structure is simply the product of these three parameters: $\# \text{Dnodes} = S \cdot C \cdot N$. The following results of operative density evaluations are expressed as a function of the total number of Dnodes. The systolic ring architectural model has been fully placed and routed in a $C=4$, $N=2$ version. The 0.35μ CMOS AMS technology has been used and the resulting area has allowed us to calibrate this area evaluation.

The figure 6 illustrates the operative density limits for one Systolic Ring instance. Three customized C/N ratios, from 1/8 (low pipeline degree versus spatial parallelism) to 4 (high pipeline degree versus spatial parallelism), were selected. Firstly, we can observe that $Do(N)$ decreases quickly for the lowest C.N values and

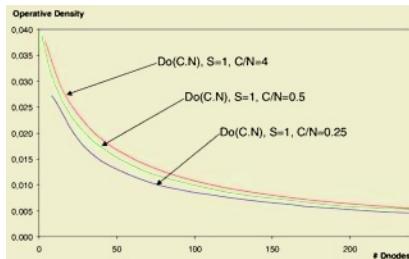


Fig. 6. Operative Density limits for $S=1$

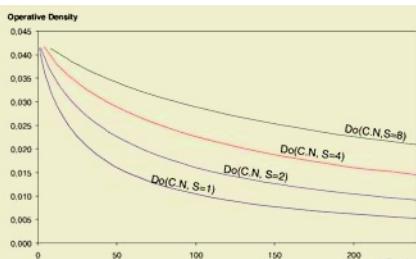


Fig. 7. Scaling Operative Density ($C=N$)

then decreases more slowly for the highest values. This figure shows that the Systolic Ring is not so scalable. The most influencing factor lays on the interconnection resources. Indeed, the full connectivity allowed between each architecture layer involves a quadratic increasing of the switches' area.

The third interconnection scheme, multi Systolic Ring instantiations (refer to the third paragraph) allows to improve the scalability. The figure 7 illustrates this other level of the Systolic Ring customisability. The operative densities results presented here were performed for C/N ratio equal to one (balanced pipeline degree and parallelism potential), and for one, two, four and eight Systolic Ring instances. The curves represented on the figure show that for a fixed number of Dnodes (proportional to a given processing power), choosing several Systolic Ring instances will significantly improve the scalability compared to one Systolic Ring instance. This means that the silicon area needed to integrate the same number of Dnodes is lower when using several Systolic Ring instances. Thus, this scheme allows to balance area versus interconnect resources.

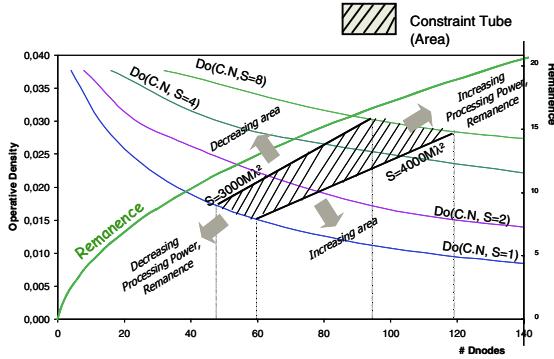


Fig. 8. Characterization and customisation illustration

4.3 Characterization and Customization of the Model

Let suppose that the silicon area available is between $3000\text{M}\lambda^2$ and $4000\text{M}\lambda^2$. This design space defines the constraint tube. As mentioned in the previous paragraph, the C/N ratio must be selected. This ratio will be fixed following the targeted applications. We choose here for example a C/N ratio equal to 4. Following the area evolution as a function of C and N for one instance of the Systolic Ring, corresponding curves are plotted for several instances of the Systolic Ring. The figure 8 represents the corresponding curves. The constraint tube is then plotted following the area constraints (the two lines were extrapolated from the Systolic Ring area formalisation). In order to show the architecture dynamism, we also plot the *Remanence* curve and add it to the graphic. The figure 8 shows how a Systolic Ring user (a platform-based designer for example) can tune his core with the architectural parameters. Indeed, in the constraint tube, many solutions are possible. For an area around $3000\text{M}\lambda^2$, the number of Dnodes can take the values from around 45 to 90 meaning that within the same silicon area, the processing power can be multiplied by a factor two. This characterizes an increased operative density. This is allowed by the

way of using eight instances of the Systolic Ring instead of only one. However, this multi-instantiation implies a reduced connectivity between the Dnodes of the architecture and an increased *Remanence*. For an area around $4000M\lambda^2$, the processing power can also be doubled by the same means. Between these two areas, many solutions are possible and the Systolic Ring user can easily compare different possible trade-offs.

5 Conclusion

After having compared different architectures and show the limitations of classical comparing approaches, we have presented a general methodology for the characterization of architectures dedicated to digital signal processing. This methodology is based on evaluating metrics, *Remanence* and *Operative Density*, as functions of the architecture parameters. This methodology helps the designer to choose between several architectural trade-offs, as shown for the Systolic Ring example. This architecture presented in the second section, was used as a case study for both *Remanence* and scalability analysis. These considerations helped to determine architecture feature trade-offs and also contributed to establish the limitations of the architecture considering a set of application-relative constraints (parallelism type, area, processing power). Future works take place in a similar analysis on other crucial factors in a SoC design context such as the power consumption.

References

- [1] W. H. Mangione-Smith *et al* : "Seeking Solutions in Configurable Computing," IEEE Computer, pp. 38-43, December 1997.
- [2] R. Hartenstein, H. Grünbacher: "The Roadmap to Reconfigurable Computing" Proc.FPL2000, Aug.27-30, 2000; LNCS, Springer-Verlag
- [3] André DeHon : "Comparing Computing Machines", Configurable Computing: Technology and Applications, Proc. SPIE 3526, 2-3 November 1998.
- [4] G. Sassatelli *et al* : "Highly Scalable Dynamically Reconfigurable Systolic Ring-Architecture for DSP applications", IEEE Design Automation and Test in Europe (DATE'02) , pp. 553-557, march 2002, Paris, France.
- [5] G. Sassatelli : "Architectures reconfigurables dynamiquement pour les systèmes sur puce", Ph.D. thesis, Université Montpellier II, France, April 2002.
- [6] D. Demigny *et al* : « La rémanence des architectures reconfigurables, un critère significatif des architectures », proc. of JFAAA, pp. 49-52, december 2002, Monastir, Tunisie.
- [7] Xilinx, the Programmable Logic Data Book, 2000.
- [8] D. Demigny *et al* : "Architecture à reconfiguration dynamique pour le traitement temps réel des images" Techniques et Science de l'Information Numéro Spécial Architectures Reconfigurables, 18(10) : pp. 1087-1112, december 1999.
- [9] R. David *et al* : "DART : A Dynamically Reconfigurable Architecture dealing with Next Generation Telecommunications Constraints", 9th IEEE Reconfigurable Architecture Workshop RAW, April 2002.

- [10] H. Singh *et al* : “MorphoSys: An Integrated Re-configurable Architecture”; Proc. of the NATO RTO Symposium on System Concepts and Integration, avril, 1998, Monterey, USA.
- [11] “TMS320C62X Image/Video Processing library Programmer’s Reference”, march 2000, www.ti.com
- [12] M. J. Wirthlin and B. L. Hutchings : “Improving Functional Density Using Run-Time Circuit Reconfiguration”, IEEE Transactions On Very Large Scale Integration (VLSI) Systems, Vol. 6, pp. 247-256, june 1998.

A Generic Architecture for Integrated Smart Transducers*

Martin Delvai, Ulrike Eisenmann, and Wilfried Elmenreich

University of Technology, Vienna, Austria,
Institut für Technische Informatik

`delvai@vlsivie.tuwien.ac.at, eisenmann@thechilli.net,`
`wil@vmars.tuwien.ac.at`

Abstract. A smart transducer network hosts various nodes with different functionality. Our approach offers the possibility to design different smart transducer nodes as a system-on-a-chip within the same platform. Key elements are a set of code compatible processor cores which can be equipped with several extension modules. Due to the fact that all processor cores are code compatible, programs developed for one node run on all other nodes without any modification. A well-defined interface between processor cores and extension modules ensures that all modules can be used with every processor type. The applicability of the proposed approach is shown by presenting our experiences with the implementation of a smart transducer featuring the processor core and a UART extension module on an FPGA.

1 Introduction

A smart transducer is a sensor or actuator element that is integrated with a processing unit and a communication interface [1]. The processing unit transforms the raw sensor signal to a digital representation, checks and calibrates the signal, and transmits the digital information to its users via a standardized communication interface. In case of an actuator, the smart transducer accepts standardized commands and transforms these into control signals. In many cases, the smart transducer is able to locally verify the control action and provide a feedback at the transducer interface.

A demonstration of a smart transducer network is presented in the DSoS project (*Dependable Systems of Systems* (IST-1999-11585)), where smart transducers are built with commercial embedded 8-bit microcontrollers featuring integrated standard UART communication interfaces.

The miniaturization process of sensors and actuators, however, offers completely new possibilities for system designers. More and more sensor elements are microelectronic mechanical systems (MEMS) that can be integrated on the same silicon die as the associated microcontroller and the communication controller. Such an *integrated smart transducer* promises a number of advantages

* This work was supported by the Hochschuljubiläumsstiftung der Stadt Wien via project MOSAIC (H-1147/2002).

over a discrete design: (i) non-linear and electrically weak sensor signals can be generated, conditioned, transformed into digital form, and calibrated without any noise pickup from long external signal transmission lines [2], (ii) power consumption of the smart transducer can be reduced significantly to a level where long-life battery-powered sensors or systems powered by solar cells are possible, (iii) the size of a smart transducer will decrease to a level where the size of the package containing sensor, processing unit, and communication interface is insignificant compared to the size of connectors and casing, (iv) production costs for large lot sizes are considerably lower than for solutions based on a discrete design.

At the same time, a developer faces several problems when designing a smart transducer. Selecting the right microcontroller for a smart transducer is difficult since a smart transducer network can host various different nodes, where the spectrum ranges from simple sensor nodes instrumenting a contact switch up to nodes with control functions or image processing capabilities. Thus, in the majority of cases it will not be possible to select a single processor that economically covers all expected applications [3]. Moreover, if the smart transducer should provide real-time functionalities, commercial processor architectures are often not appropriate, since they are optimized for average throughput. Worst-case analysis for such systems, which is essential for real-time applications, lead to unrealistically pessimistic estimations [4]. Finally, the smart transducers, although different in processing power and interaction capabilities, should support interoperability via a standard communication interface.

It is the objective of this paper to present an architecture for integrated smart transducers that meets the above described requirements on scalability and interoperability. The paper is structured as follows: Section 2 describes the architecture of a smart transducer node in general. Section 3 describes a modular approach to build integrated smart transducers using a microprocessor core with various extension modules. Section 4 provides information about implementations of the processor core hardware, an extension module, that protects processor resources against unauthorised accesses, and a communication extension module. Section 5 gives an overview of related work in the field, while the paper is concluded in Section 6.

2 Smart Transducer Architecture

The generic smart transducer architecture introduces a two-level design approach [5] that reduces the overall complexity of a smart transducer by separating transducer-specific implementation issues from interaction issues between different smart transducers.

Figure 1 depicts the two functionalities as protocol part and local application. The protocol part instruments the network interface. Each network interface of a smart transducer must provide some standard functionalities to transmit data in a temporally deterministic manner in a standard data format, provide means

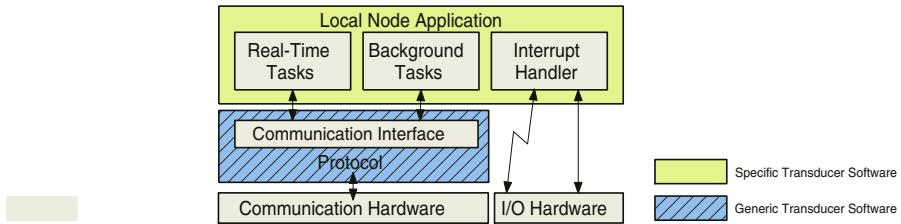


Fig. 1. Generic Smart Transducer Architecture

for fault tolerance, and enable a smooth integration into a transducer network and its application.

For example, the smart transducer interface (STI) [6] standard specifies some services for smart transducers. It comprises a time-triggered transport service within the distributed smart transducer subsystem and a well-defined interface to a CORBA (Common Object Request Broker Architecture) environment. The key feature of the STI is the concept of an Interface File System (IFS) that contains all relevant transducer data. This IFS allows different views of a system, namely a real-time service view, a diagnostic and management view, and a configuration and planning view. The interface concept encompasses a communication model for transparent time-triggered communication. A time-triggered sensor bus will perform a periodical time-triggered communication to copy data from the IFS to the fieldbus and write received data into the IFS. Thus, the IFS is the source and sink for all communication activities. Furthermore, the IFS acts as a temporal firewall [7] that decouples the local transducer application from the communication activities. Data is transmitted in a standard UART format.

The general architecture including the protocol part applies for every smart transducer, thus implying a generic implementation approach. The realization, however, faces the following problems: Due to the differing requirements on code size and processing power for the different local applications, it is not economic to select a single processor architecture. Moreover, each smart transducer type has its own I/O hardware configuration. An architecture should support, on the one hand, the configuration of many types of I/O extensions, and, on the other hand, provide a consistent interface to these modules in order to enable software reuse. Building smart transducers on commercial hardware is possible, but due to the various microcontroller types and I/O interfaces, programming and reuse is complex and error-prone.

3 Implementation Concept

As explained in the previous section a smart transducer network comprises various nodes with different requirements in terms of processing speed, size, interfaces and so on. Due to this fact different standard microcontrollers have to be used within the same network - the programmers have to know instruction

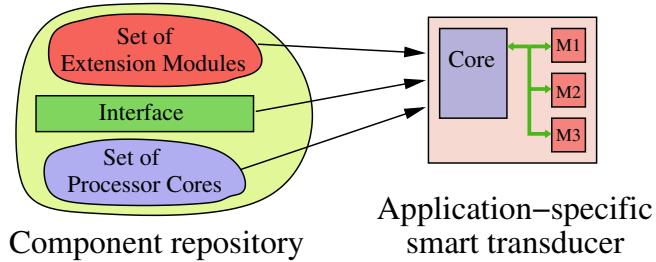


Fig. 2. Construction of an individual microcontroller

set and peculiarities of each microcontroller type in order to be able to implement, administrate, and maintain such a network. Furthermore, software pieces with the same functionality, e.g. the communication protocol, have to be implemented and tested for each microcontroller type separately. This proves to be especially difficult in the field of real time applications, where not only the correct functionality, but also the temporal behavior has to be considered. Usually, the employment of different hardware components requires a redesign of the software. To overcome these problems we designed a modular construction system, consisting of a set of processor cores and a set of different extension modules. The processor cores are all fully code compatible, but have different features in terms of computational power, required silicon area, memory-size and power consumption. Additionally, every activity within these processors is temporally predictable, which facilitates the design of real-time applications. [8]

All processor cores provide a generic interface to extension modules, which can be used to fit the microcontroller to different requirements. Figure 3 depicts the idea behind that approach.

It is difficult to verify the correct functionality of a smart transducer due to the fact, that not only the value domain, but also the temporal domain has to be considered. Especially the temporal aspect is critical. To evaluate the functional-

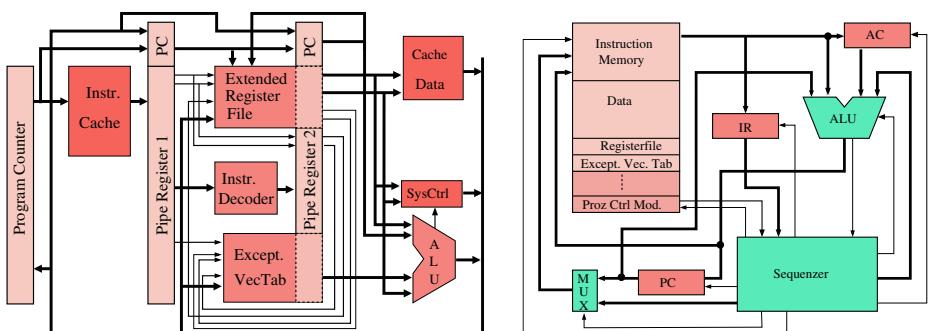


Fig. 3. Architecture of SPEAR and NEEDLE processor cores

ity of such a smart transducer in a network long simulation periods are necessary. Such a simulation would require an unacceptable amount of time. Thus, it is particularly important to get prototypes of smart transducers at early development stages. The use of FPGAs is ideal for this purpose. The smart transducer can be integrated into a real network and tested over longer periods of time. Therefore, we test prototypes of the assembled network nodes on FPGA test boards. Since all employed components of our architecture are described as VHDL models it is easily possible to implement a proved system in a silicon chip.

3.1 Processor Cores

Currently, two processor cores, named SPEAR and NEEDLE [3], have been implemented by our group. While the SPEAR core is designed to support moderate arithmetic performance, NEEDLE pays attention to a compact design. A further processor core LANCE supporting high computational power is planned. All processor cores are absolutely code compatible - this means that they not only use the same instruction set, but also the same exception handling, an identical memory architecture (from a logical point of view) and the same status and configuration registers. Due to this fact the programming code can be interchanged between different processor cores without any modification.

SPEAR Processor Core: SPEAR features a 16 bit processor that executes instructions over a three-stage pipeline. The processor core comprises a set of 32 registers. 26 registers are general purpose registers and 6 registers have a special function: three are coupled with dedicated instructions to efficiently implement stacks and the other three registers are used to save the return address in case of a subroutine call or an exception. Data and instruction memory are both 4 kB in size. It is also possible to add up to 128 kB external instruction memory and 127 kB external data memory. The upper 1 kB of the data memory is reserved for the memory mapped extension modules. In this way an extremely simple and efficient access to these modules is provided. The SPEAR processor supports 32 exceptions, of which 16 are hardware exceptions (= interrupts) and 16 can be activated by software (= trap). The exception vector table contains the pointers to the exception service routines.

NEEDLE Processor Core: NEEDLE is absolutely code compatible with the SPEAR processor, but the implementation is more compact. NEEDLE has no pipeline and executes instructions within several clock cycles. Due the fact that during each clock cycle at most two memory accesses are performed, it was possible to map the entire memory architecture (instruction memory, data memory, register file and exception vector table) into a single 4 kB dual-ported memory block. Moreover, the possibility to bind up to 64 different extension modules to the data memory area still exists. The block diagram of SPEAR and NEEDLE is shown in figure 3.

3.2 Extension Modules

To ensure scalability the processors are easily expandable by different extension modules, such as communication interfaces (sensors, actuators, PS/2, VGA, parallel port, USB, network interfaces, etc.) or application-specific extension modules such as a floating point unit or a protection unit. The extensions are mapped to the top address space of the data memory. For the processors the extension modules are only storage positions that can be accessed with simple load and store instructions. Therefore, from the processor's point of view it makes no difference, whether the extension is a simple sensor, actuator or a complex floating-point unit. Due to the well-defined generic interface [3] modules developed for one processor core can be used by all other cores. It is also possible to instantiate an extension module more than one time. For example, a smart transducer node can be equipped with several UART modules, having different mapping addresses in the data memory.

To illustrate the possibilities offered by the extension modules, two modules, the *protection control unit* and a customized *UART* will be described in the following. The first one extends the functionality of the processor core, the second one shows, how problems existing in commercial UART modules can be solved by a customized UART implementation.

Protection Control Module: The software that runs on a smart transducer node comprises a protocol code and an application part. As explained in section 2 the local application should not have direct access to the network bus. To guarantee that such an illegal access cannot happen, we have to ensure that only the protocol software can access the communication module. Such a protection mechanism is provided by the *protection control* extension module.

The protection control module allows assigning a *protection level* to individual data memory blocks, instruction memory blocks, registers and to extension modules. The module supports four protection levels: zero, low, high, and supervisor protection. As depicted in figure 4 the module comprises four look-up tables, which are mapped into the data memory of the processor. Via these look-up tables a protection level to each resource can be assigned. The addresses of instruction memory, data memory, register file and extension modules are directly connected to the input of the respective look-up table. The module controls the access by comparing the output of each look-up table (i.e., the assigned protection level) with the current protection level defined in the processor status register (i.e., protection level of the current task) and by evaluating the control signals associated to each address. When a process tries to access a resource with the current protection level being lower than the protection level of the resource, an access-violation exception is generated.

UART Extension Module: In [9], we have examined the applicability of common UARTs for real-time communication and identified the following problems: the arithmetic error, the error due to frequency-drift and the send jitter problem.

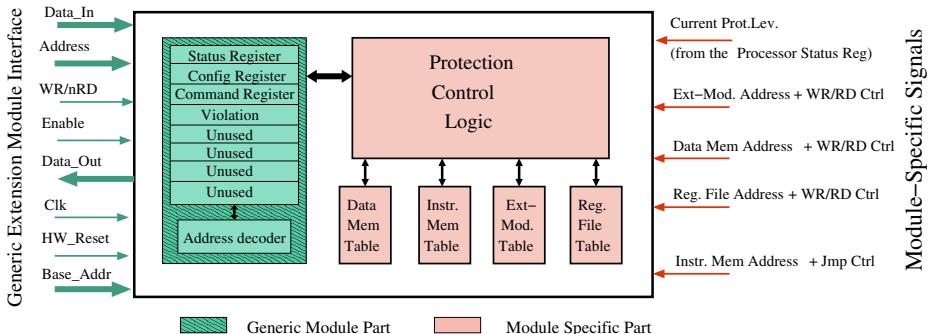


Fig. 4. Protection control module

Therefore, we have developed an alternative VHDL implementation of a UART unit that resolves the intrinsic UART problems and allows the implementation of more efficient protocols, respectively the employment of cheap on-chip oscillators with large drift rates.

The UART module is able to work with an RC-oscillator that provides a frequency of $1 \text{ MHz} \pm 50\%$ and a frequency-drift of $\pm 10\%$ per second. As the baud rate is influenced by this drift it has to be continuously adjusted. To deal with the baud rate drift rate problem the UART module offers a synchronization mode. In this mode the UART can be forced to synchronize periodically, like it is necessary in fieldbus protocols such as TTP/A [10] or LIN [11]. For synchronization the UART searches for a regular bit pattern as it is outlined in Figure 5. The synchronization pattern allows the UART to determine the used baud rate.

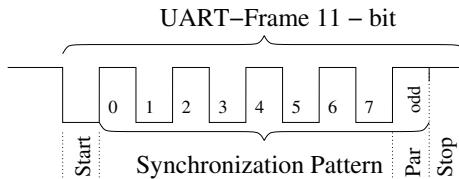


Fig. 5. Synchronization pattern

A substantial divergence from the desired transfer rate is caused by the arithmetic error in baud rate setting, i. e., the accuracy of a selected baud rate depends heavily on the selected clock source. Therefore, it is necessary to select special crystal frequencies (e. g. $1.8432 \cdot 10^6$) to be able to use standard communication rates [9]. Our UART module uses an extrapolation mechanism that minimizes this rounding errors.

As a further problem, many commercial UARTs exhibit intrinsic delays of the sending instant of a transmission because the UART baud rate generator period-

ically generates potential transmission points. Thus, the start of a transmission is delayed until the next transmission point which is perceived as a send jitter by an outside user. The send jitter can be a problem for real-time communication.

Our UART module overcomes this disadvantage by a different design approach. Thus, the UART module starts with the message transmission immediately after receiving the transmit signal transmission, which almost completely eliminates the send jitter.

The bus interface contains a hardware filter that preprocesses the input signal from the bus to achieve robustness against spike interferences. Additionally, oversampling has been implemented.

Figure 6 illustrates the block diagram of the UART extension module.

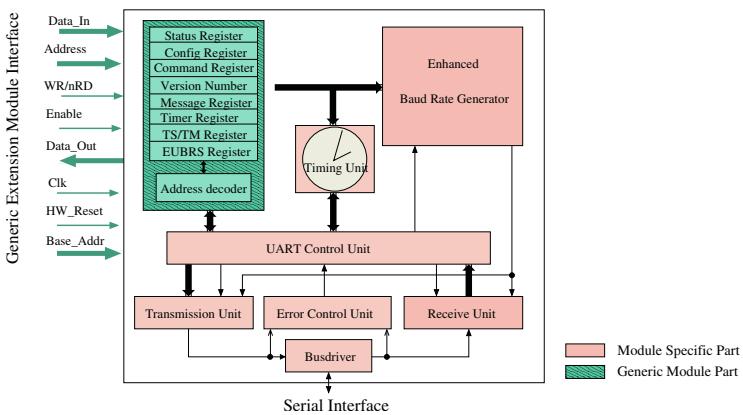


Fig. 6. Block diagram of the UART extension module

3.3 Development Tools

For software development an assembler is implemented. In the next step we will port the GNU C compiler to our processor architecture. Further a *system design tool* and a *debugger* are available to support assembling and testing of smart transducers. The system design tool provides software models of the processor cores and the extension modules. These components can be connected in order to form the desired smart transducer. The entire system could be simulated and verified. It is also possible to define own components in a high level language based on C++. In this way it is possible to evaluate a virtual hardware setup and to decide whether the real hardware should be build or not.

The debugger provides a high degree of flexibility, since it is easily adaptable to different purposes. For example it is possible to set breakpoints not only to a program counter value but also to sensor values or bus signals. Furthermore the debugger allows also the validation of real-time applications. Both, the system

design tool and the debugger are controlled via the same graphical user interface. A detailed description of the system design tool and the debugger can be found in [12, 13].

4 Experiences

We have used an Altera APEX 20KE300EQC240-1X FPGA on a Digilab prototyping board for implementing the described architecture. The modular approach has been successfully tested by composing the two processor cores with several extension modules.

The processor core properties are compared in Table 1. While SPEAR computes one instruction per clock cycle at a maximum clock frequency of 40 MHz yielding 40 MIPS (million instructions per second), NEEDLE performs one instruction within either two or three clock cycles leading to an average performance of 10 MIPS at a peak clock frequency of 25 MHz. The size of the processors is given by Logic Elements (these are the smallest hardware units in the APEX FPGA) and the instruction and data memory. Due to the different memory size of the two processor cores, the necessary silicon area for the NEEDLE core will be expected to be one half of that of the SPEAR core. The values for the power consumptions are estimates given by the Synopsys design analyzer tool.

Table 1. Comparison of the SPEAR and the NEEDLE processor

	SPEAR	NEEDLE
Maximum Speed	40 MHz/40 MIPS	25 MHz/10 MIPS
Logic Elements (LE)	1318	1010
Instr-/Data Mem	4/4 KB	2/1,92 KB
Reg File	26 GPR - 6 SPR	26 GPR - 6 SPR
Interrupts/Traps	16/16	16/16
Instructions	80	80
Power consumption	113 mW	62 mW

The size of the extension modules heavily depends on their functionality. For example, the protection control module needs 279 LE, while the UART module accounts for 699 LE.

As communication interface, we have implemented software for a time-triggered UART communication that conforms to the OMG Smart Transducer Interface standard. The implementation is supported by the UART module that, in contrast to most commercial UART units, provides temporal predictability over the send instant of a message and a minimization of the arithmetic error in baud rate setting. The improved UART module greatly reduces the implementation effort for the smart transducer protocol software.

5 Related Work

The core-based system design approach is discussed by Gupta and Zorian in [14]. Their work outlines the advantages of using predesigned, preverified, silicon circuit modules as building blocks of large and complex applications on a single silicon die. ALTERA¹ offers a set of processor cores for embedded applications called Nios® following a similar idea as our proposed approach. The main difference for both approaches lies in its real-time capabilities. WCET analysis for code segments running on the Nios processor version 2.0 or higher is difficult, since the execution time of a single instruction depends on the preceding and the following instruction, the operands used, how recently operands were modified and other additional factors. In contrast our processor cores provide a fully temporal predictable behavior, which make them more suitable for to be used in real-time smart transducer networks.

Related work on sensor integration with VLSI circuitry can be found throughout the literature. Main research focuses on wide bandgap semiconductor materials [15], thin film sensors [16], and MEMS devices [17]. These approaches are related to the work presented in this paper as they outline possibilities for further on-chip I/O modules. The examined literature on smart sensor technologies, however, usually neglects the integration of a transducer with an appropriate communication network interface.

Besides the OMG STI, there are some other communication standards for smart transducers. Many fieldbus protocols, such as Controller Area Network, Local Area Network, Local Interconnect Network, Profibus, Foundation Fieldbus, WorldFip, and Interbus also provide possible solutions for the communication interface. The main differences between these approaches are in real-time features and implementation complexity. We have chosen the OMG STI since it provides hard real-time capabilities while having a low implementation complexity. In general, our architecture supports the easy adaption to a different communication interface by exchanging or adding communication I/O modules.

Another related standard is the IEEE 1451 smart transducer standard, which is not another fieldbus protocol but can be treated in the same way in our case, since IEEE 1451 specifies a 10-wire transducer-independent interface [18], which could be implemented as a communication extension module in our architecture.

6 Conclusion

This paper presented an architecture for the implementation of integrated smart transducers, i. e., a sensor or actuator element, a microcontroller and a network interface on a single silicon die.

The key element of our architecture is a set of fully code compatible microprocessor cores, with a well-specified interface to hardware extension modules that are synthesized on the same semiconductor chip. Due to the modular approach

¹ <http://www.altera.com/products/devices/nios/nio-index.html>

of the proposed architecture, the system is open to various external extension modules such as physical sensors/actuators or network communication modules.

Currently we have implemented two different microprocessor cores and several extension modules. The microprocessor cores are compatible at register and machine language level, so that software can be easily reused. The two extension modules are designed to support a smart transducer implementation. The improved UART module reduces the implementation effort for the smart transducer protocol software, while the protection unit protects resources like communication interfaces from unauthorized access.

References

1. W. Elmenreich and S. Pitzek. Smart transducers – principles, communications, and configuration. In *Proceedings of the 7th IEEE International Conference on Intelligent Engineering Systems (INES'03)*, volume 2, pages 510–515, Assuit – Luxor, Egypt, March 2003.
2. P. Dierauer and B. Woolever. Understanding smart devices. *Industrial Computing*, pages 47–50, 1998.
3. M. Delvai, U. Eisenmann, and W. Huber. Modular construction system for embedded real-time applications. In *Tagungsband of Austrochip 2002*, Vienna, Austria, 2002.
4. P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, May 2000.
5. S. Poledna, H. Angelow, M. Glück, M. Pisecky, I. Smaili, G. Stöger, C. Tanzer, and G. Kroiss. TTP two level design approach: Tool support for composable fault-tolerant real-time systems. *SAE World Congress 2000, Detroit, MI, USA*, March 2000.
6. Object Management Group (OMG). *Smart Transducers Interface Final Adopted Specification*, August 2002. Available at <http://www.omg.org> as document ptc/2002-10-02.
7. H. Kopetz and R. Nossal. Temporal firewalls in large distributed real-time systems. *Proceedings of the 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97)*, pages 310–315, 1997.
8. M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - The SPEAR design example. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.
9. W. Elmenreich and M. Delvai. Time-triggered communication with UARTs. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, Västerås, Sweden, August 2002.
10. H. Kopetz et al. Specification of the TTP/A protocol. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, March 2000. Available at <http://www.ttpforum.org>.
11. Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc. Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification and LIN press announcement. SAE World Congress Detroit, <http://www.lin-subbus.org>, 1999.
12. M. Delvai, M. Jankela, and A. Steininger. Towards virtual prototyping of embedded computer systems. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'03)*, Orlando, FL, USA, July 2003.

13. M. Delvai, C. El Salloum, and A. Steininger. A generic real-time debugger architecture. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'03)*, Orlando, FL, USA, July 2003.
14. R. K. Gupta and Y. Zorian. Introducing core-based system design. *IEEE Design & Test of Computers*, 14(4):15–25, Oct.-Dec. 1997.
15. B. W. Licznerski, K. Nitsch, and H. Teterycz. Polycrystalline wide bandgap materials in sensor technology. In *Abstract Book of the 3rd International Conference on Novel Applications of Wide Bandgap Layers*, pages 32–35, Poland, 2001.
16. S. M. Vaezi-Nejad. Advanced sensors scene in europe. In *Proceedings of the IEEE Instrumentation and Measurement Technology Conference*, volume 2, pages 926–931, Ottawa, Canada, May 1997.
17. G. K. Fedder. Structured design of integrated MEMS. In *Proceedings of the 12th IEEE International Conference on Micro Electro Mechanical Systems*, pages 1–8, January 1999.
18. Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1451.2-1997, Standard for a Smart Transducer Interface for Sensors and Actuators - Transducer to Micro-processor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*, September 1997.

Customisable Core-Based Architectures for Real-Time Motion Estimation on FPGAs

Nuno Roma, Tiago Dias, and Leonel Sousa

Instituto Superior Técnico / INESC-ID
Dept. of Electrical and Computer Engineering
Rua Alves Redol, 9-1000-029 Lisboa – PORTUGAL
{Nuno.Roma, tdias, las}@inesc-id.pt
<http://sips.inesc-id.pt>

Abstract. This paper proposes new core-based architectures for motion estimation that are customisable for different coding parameters and hardware resources. These new cores are derived from an efficient and fully parameterisable 2-D single array systolic structure for full-search block-matching motion estimation and inherit its configurability properties in what concerns the macroblock dimension, the search area and parallelism level. The proposed architectures require significantly fewer hardware resources, by reducing the spatial and pixel resolutions rather than restricting the set of considered candidate motion vectors. Low-cost and low-power regular architectures suitable for field programmable logic implementation are obtained without compromising the quality of the coded video sequences. Experimental results show that despite the significant complexity level presented by motion estimation processors, it is still possible to implement fast and low-cost versions of the original core-based architecture using general purpose FPGA devices.

1 Introduction

Motion estimation is a fundamental operation in motion-compensated video coding [1], in order to efficiently exploit the temporal redundancy between successive frames. Among the several possible approaches, block-matching is the most used in practice. In this strategy, the current frame is divided into equal sized $N \times N$ pixel blocks that are displaced within a $(N + 2p - 1) \times (N + 2p - 1)$ search window defined in the previous frame. The motion vector is determined by looking for the best matched block of this search window. The Sum of the Absolute Differences (SAD) is the matching criteria that is usually used by most systems, due to its efficiency and simplicity.

Although the Full-Search Block-Matching (FSBM) method exhaustively considers all possible candidate blocks, which guarantees the optimal solution, it requires a lot of computational resources. In fact, FSBM motion estimation can consume up to 80% of the total computational power required by a video encoder. Hence, most of the fast block-matching motion estimation algorithms that have been proposed over the last years restrict the search space by a given search pattern, providing suboptimal solutions (e.g. [2, 3]). However most of these algorithms apply non-regular processing and require complex control schemes, which make their hardware implementation difficult and rather inefficient.

Hence, the main objective of this paper is to propose efficient core-based VLSI array architectures based on FSBM to be implemented in Field Programmable Logic (FPL) devices. These architectures are based on a highly efficient core that was recently developed by the authors of this work, that combines both pipelining and parallel processing techniques to design powerful motion estimators based on multiple array architectures and using Application Specific Integrated Circuits (ASIC) [4]. In this paper, this original core architecture is used to derive simpler structures with reduced hardware requirements. The complexity of these architectures is reduced by decreasing the precision of the pixel values or/and the spatial resolutions in the current frame, while maintaining the original resolution in the search space. The pixel precision is configured by defining the number of bits used to represent the input data and by masking or truncating the corresponding Least Significant Bits (LSBs). On the other hand, spacial resolution is adjusted by sub-sampling the blocks of the current frame. By doing so, while the best candidate block in the previous frame is exhaustively searched, the SAD of each candidate block is computed by using only sparse pixels. Considering a typical setup with 16×16 pixels block, by applying 2 : 1 or 4 : 1 alternate sub-sampling schemes the number of considered pixels decreases by 1/4 and 1/16, respectively.

The efficiency of the proposed structures were evaluated by implementing these customisable core-based architectures in Field Programmable Gate Arrays (FPGA). It is shown that the amount of hardware required by these architectures when sub-sampling and truncation techniques are applied is considerable reduced, which allows the usage of a common framework for designing a wide range of motion estimation processors with different characteristics. Moreover, experimental results obtained with benchmark video sequences show that the application of those techniques does not introduce a significant degradation in the quality of the coded video sequences. These reduced hardware requirements architectures fit well in actual FPGAs, being a real alternative to those fast motion estimation techniques that require non-regular processing.

This paper is organised as follows. The original FSBM core architecture is presented in section 2. Section 3 proposes reduced hardware architectures to implement motion estimators on FPL devices. Experimental results obtained with FPGAs are presented in section 4 and section 5 concludes the paper.

2 FSBM Core-Based Architectures

Most motion estimation architectures that have been proposed in the last few years for hardware implementation are based on the optimum FSBM algorithm. The main reason for this is not only related to the better performance levels that it generally provides, but it is mainly due to the regularity properties that it also offers. In fact, not only does it conduct to much more efficient hardware structures, but it also provides the usage of significantly simpler control units, which is always a fundamental factor towards a real-time operation based on hardware structures.

Several FSBM structures have been proposed over the last few years (e.g.: [4–6]). Very recently, it was presented in [4] a new class of parameterisable hardware architectures that is characterised by offering minimum latency, maximum throughput and a full and efficient utilisation of the hardware resources. This last characteristic is a fun-

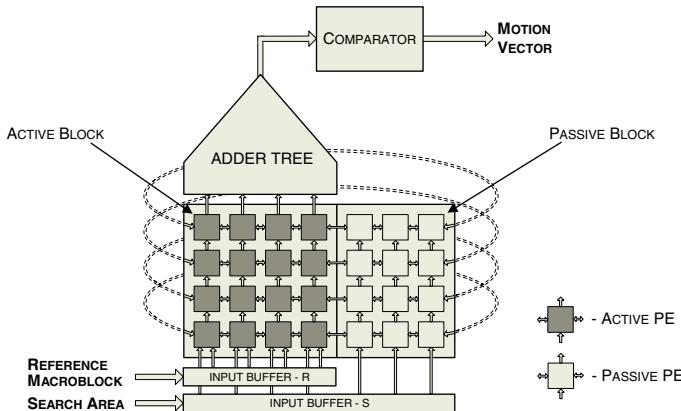


Fig. 1. Processor array proposed in [4] based on an innovative cylindrical structure and adopting the zig-zag processing scheme proposed by Vos [6] ($N = 4, p = 2$).

damental requisite in any FPL system, due to the limited amount of hardware resources. To achieve such performance levels, a peculiar and innovative processing scheme based on a cylindrical hardware structure and on the zig-zag processing sequence proposed by Vos [6] was adopted (see fig. 1). With such a scheme, not only was it possible to minimise the processing time, but it also provided the ability to prevent the usage of some hardware structures (the so called *passive processor elements (PEs)*) that do not carry useful information at some clock cycles.

Moreover, besides this set of performance and implementation characteristics, one other important feature of this class of processors is concerned with its scalable and configurable architecture, making it possible to easily adapt the processor configuration to fulfil the requisites of a given video coder. By adjusting a small set of implementation parameters, processing structures with distinct performance and hardware requirements are obtained, providing the ability to adjust the required hardware resources to the target implementation technology. As an example, while high performance processors requiring more resources are more suited for implementations using technologies such as ASIC or Sea-of-Gates, those low-cost processors that are meant to be implemented in FPL devices with limited hardware resources should use configurations requiring reduced amount of hardware.

3 Reduced Hardware Architectures

Despite the set of configurable properties offered by FSBM architectures and, in particular, by the class of processors proposed in [4], FPL devices often do not provide enough hardware resources to implement such processors. In those cases, the solution that is often adopted is the usage of processing structures based on sub-optimal motion estimation algorithms, that provide faster processing times and require reduced amounts of hardware. Three different categories of sub-optimal motion estimation algorithms have been proposed:

- *Reduction of the set of considered candidate motion vectors*, where the search procedure in the previous frame is restricted to a search pattern within the search window, by using hierarchical search strategies [2, 3];
- *Decimation at the pixel level*, where the considered similarity measure is computed by using only a subset of the $N \times N$ pixels of each reference macroblock [7–9];
- *Reduction of the precision of the pixel values*, where the similarity measure is computed by truncation the LSBs of the input values to reduce the hardware resources required by the used arithmetic units [9, 10].

The main drawback of these solutions is a corresponding increase of the prediction error that inevitable arises from using a less accurate estimation. This tradeoff usually leads to a difficult and non-trivial relationship between the final picture quality and the prediction accuracy that can not be assumed to be linear. In general, a larger prediction error will lead to higher bit rates, which will conduct to the usage of greater quantisation step sizes to compensate this increase, thus affecting the quality of the decoded images.

Up until now only a few VLSI architectures have been proposed to implement fast motion estimation algorithms, by restricting the search positions according to a given search pattern [9]. In general, they imply the usage of non-regular processing structures and require higher control overheads, which difficults their design using efficient systolic structures. Consequently, they have been extensively used in software applications, where such restrictions do not usually apply so strictly.

The set of architectures that are proposed in this paper try to combine the advantages offered by the regular and efficient FSBM structures proposed in [4] with the several strategies to reduce the amount of hardware resources that are offered by sub-optimal motion estimation algorithms. By doing so, it will be possible to implement processors based on this new class of fast motion estimation processors in any FPL device, even in FPGAs with limited resources. To achieve such objective, the original FSBM architecture will be adapted to apply two of the three decimation categories referred above: the decimation at the pixel level and the reduction of the precision of the pixel values.

3.1 Decimation at the Pixel Level

By applying the decimation at the pixel level, the image data of the current frame is sub-sampled in the orthogonal directions, by considering alternate pixels in each direction. In fact, this scheme corresponds to using a lower resolution version of the reference frame in the search procedure, that is carried out within the previous full-resolution frame. As an example, in a $2 : 1$ sub-sampling scheme just one in each pair of consecutive pixels in each direction is considered, giving rise to decimated images with $1/4$ of the area of the original image. In the general case, the SAD similarity measure for a configuration using a $2^S : 1$ sub-sampling in each direction is given by (considering N a power of 2):

$$SAD(l, c) = \sum_{i=0}^{\frac{N}{2^S}-1} \sum_{j=0}^{\frac{N}{2^S}-1} |x_t(i \cdot 2^S, j \cdot 2^S) - x_{t-1}(l + i \cdot 2^S, c + j \cdot 2^S)| \quad (1)$$

The FSBM circuit proposed in [4] can be easily adapted to carry out this type of sub-sampling. In fact, considering that the computation of the SAD similarity measure

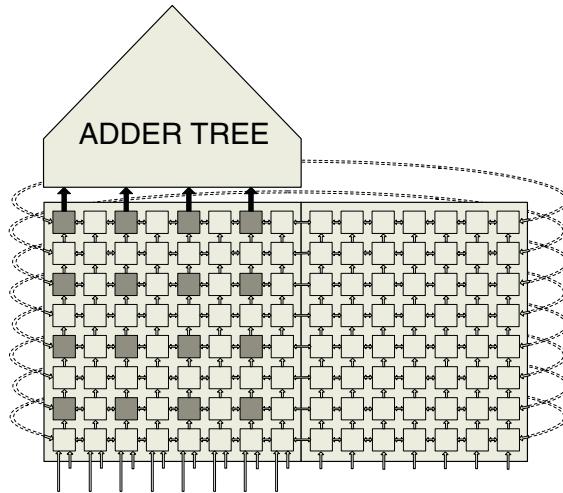


Fig. 2. Modified processing array to carry out a 2 : 1 decimation function in the computation of the SAD similarity measure using the architecture proposed in [4] ($N = 8, p = 4$).

is performed in the active block of the processor, the decimation can be implemented by replacing the corresponding set of active PEs by passive PEs. By doing so, only the pixels with coordinates $(i \cdot 2^S, j \cdot 2^S)$ will be considered. In fig. 2 it is presented the block diagram of the modified processing array.

Since the amount of hardware resources required by passive PEs is considerable lower than that required by the active PEs, significant amounts of hardware can be saved. Moreover, by adopting the sub-sampling pattern presented in fig. 2, extra amounts of resources can be saved, since the number of inputs of the adder tree block is also reduced by the same sub-sampling factor (S).

3.2 Reduction of the Precision of the Pixel Values

As it was previously referred, one other strategy to decrease the amount of hardware required by FSBM processors is to reduce the bit resolution of the pixel values considered in the computation of the SAD similarity function by truncating the LSBs. By adopting this strategy alone ($S = 0$) or in conjunction with the previously described sub-sampling method ($0 < S < \log_2 N$), the SAD similarity measure is given by:

$$SAD(l, c) = \sum_{i=0}^{2^S-1} \sum_{j=0}^{2^S-1} \left| \left\lfloor \frac{x_t(i \cdot 2^S, j \cdot 2^S)}{2^T} \right\rfloor - \left\lfloor \frac{x_{t-1}(l + i \cdot 2^S, c + j \cdot 2^S)}{2^T} \right\rfloor \right| \quad (2)$$

$$\equiv \sum_{i=0}^{2^S-1} \sum_{j=0}^{2^S-1} \left| x_t(i \cdot 2^S, j \cdot 2^S)_{7:T} - x_{t-1}(l + i \cdot 2^S, c + j \cdot 2^S)_{7:T} \right| \quad (3)$$

where T is the number of truncated bits and $x_t(i, j)_{7:T}$ are the $(8 - T)$ most significant bits of a pixel value of the t^{th} frame.

The adaptation of the original FSBM architecture proposed in [4] to apply this bit truncation scheme is straightforward. In fact, it is only necessary to reduce the operands width of the several arithmetic units implemented in the active PEs, in the adder tree block and in the comparator circuit. Such modification potentially increases the maximum frequency of the pipeline and will significantly reduce the amount of required hardware, thus providing the conditions that will make it possible to implement the motion estimation processors in FPL devices.

4 Implementation and Experimental Results

The proposed customisable core-based architectures for motion estimation were completely described using IEEE-VHDL language. Both behavioural and structural parameterisable descriptions were carried out by making extensive use of “generic” type configuration inputs. These descriptions were used to synthesise several different setups of the proposed core in a general purposed VIRTEX XCV3200E-7 FPGA using the Xilinx Synthesis Tool from ISE 5.2.1. The considered set of configurations assumed each macroblock composed by 16×16 pixels ($N = 16$) and a maximum displacement in each direction of the search area of $p = 16$ pixels. These configurations were thoroughly tested using sub-sampling factors (S) varying between 0 and 2 and a number of truncated bits (T) of 0, 2 and 4. The experimental results of these implementations are presented in tables 1 and 3.

The proposed core-based architectures can provide significant savings of the required hardware resources. From the set of configurations presented in table 1, one can observe that reduction factors of about 75% can be obtained by using a sub-sampling factor $S = 2$ and by truncating the 4 LSBs. However, this relation should not be assumed to be linear. By considering only the pixel level decimation mechanism ($T = 0$), it can be shown that a reduction of about 38% is obtained by using a 2 : 1 sub-sampling factor ($S = 1$), while a 4 : 1 decimation will provide a reduction of about 51%. The same observation can be done by considering only the reduction of the precision of the pixel values ($S = 0$). While using 6 representation bits ($T = 2$) a reduction of the number of CLB slices of about 31% is obtained (a reduction of about 23% of the number of Look-up Tables (LUTs)), if only 4 representation bits are considered ($T = 4$) it provides a reduction of about 51% of the CLB slices (a reduction of about 47% of the number of LUTs). In table 2 it is presented the set of FPGA devices that should be used by each configuration, in order to maximize the efficiency of the hardware resources used by each processor.

In table 3 it is presented the variation of the maximum operating frequency of the considered configurations with the number of truncated bits (T). Contrary to what could be expected, the reduction of the operands width of the several arithmetic units does not significantly influence the processors performance. This fact can be explained if one takes into account the synthesis mechanism that is used by this family of FPGAs to synthesise and map the logic circuits using built in fast carry logic and LUTs.

To assess and evaluate the efficiency of the synthesised processors in an implementation based on FPL devices, they were embedded as motion estimation co-processors in the H.263 video codec provided by Telenor R&D [11], by transferring the estimated mo-

Table 1. Percentage of the CLB slices and LUTs that are required to implement each configuration of the proposed core-based architecture for fast motion estimation in a VIRTEX XCV3200E-7 FPGA ($N = 16$; $p = 16$).

S	T					
	0		2		4	
	CLB Slices	LUTs	CLB Slices	LUTs	CLB Slices	LUTs
0	90.7%	30.7%	62.6%	23.5%	43.8%	16.3%
1	56.2%	19.0%	38.2%	14.6%	26.6%	10.7%
2	44.7%	14.8%	30.1%	11.4%	21.0%	7.8%

Table 2. Alternative FPGA devices to implement each of the considered configurations ($N = 16$; $p = 16$).

S	T		
	0	2	4
0	XCV3200	XCV2600	XCV1600
1	XCV2000	XCV1600	XCV1000
2	XCV1600	XCV1000	XCV600

Table 3. Variation of the maximum operating frequency with the number of truncated bits (T) ($N = 16$; $p = 16$).

T		
0	2	4
76.1 MHz	77.8 MHz	79.8 MHz

tion vectors to the video coder. Peak signal-to-noise ratio (PSNR) and bit-rate measures were used to evaluate the performance of each architecture. These results were also compared with those obtained with a sub-optimal 4-steps logarithmic search algorithm [1], implemented in software.

The first 300 frames of several QCIF benchmark video sequences with different spatial detail and amount of movement were coded in interframe mode, by considering a GOP length of 30 frames and an intermediate quantisation step size of $\Delta = 30$ to keep the quantisation error as constant as possible.

In fig. 3 it is presented the obtained PSNR values for the *carphone* and *mobile* video sequences, characterised by the presence of high amount of movement and high spatial detail, respectively. Several different setups in what concerns the number of truncated bits (T) and the sub-sampling factor (S) for the decimation at the pixel level were considered. For comparison purposes, it was also presented the PSNR values obtained with the 4-steps logarithmic search algorithm. As it can be seen, the PSNR value for the INTER type frames of the *mobile* sequence is almost constant (25 – 26 dB). Hence, one can conclude that the degradation introduced by using the reduced hardware architectures is negligible: less than 0.15 dB when the 4 LSBs are truncated and the sub-sampling factor is 2 : 1. Contrasting, the PSNR behaviour for the *carphone* sequence varies significantly along the time. The main reason for this fact is the amount of movement that is also varying. Even so, the reduced hardware architectures proposed in this paper only introduce a slightly degradation in the quality of the coded frames, when compared with the performances of both the reference FSBM architecture ($S = T = 0$) and of the 4-steps logarithmic search algorithm. A maximum reduction of about 0.5 dB is observed

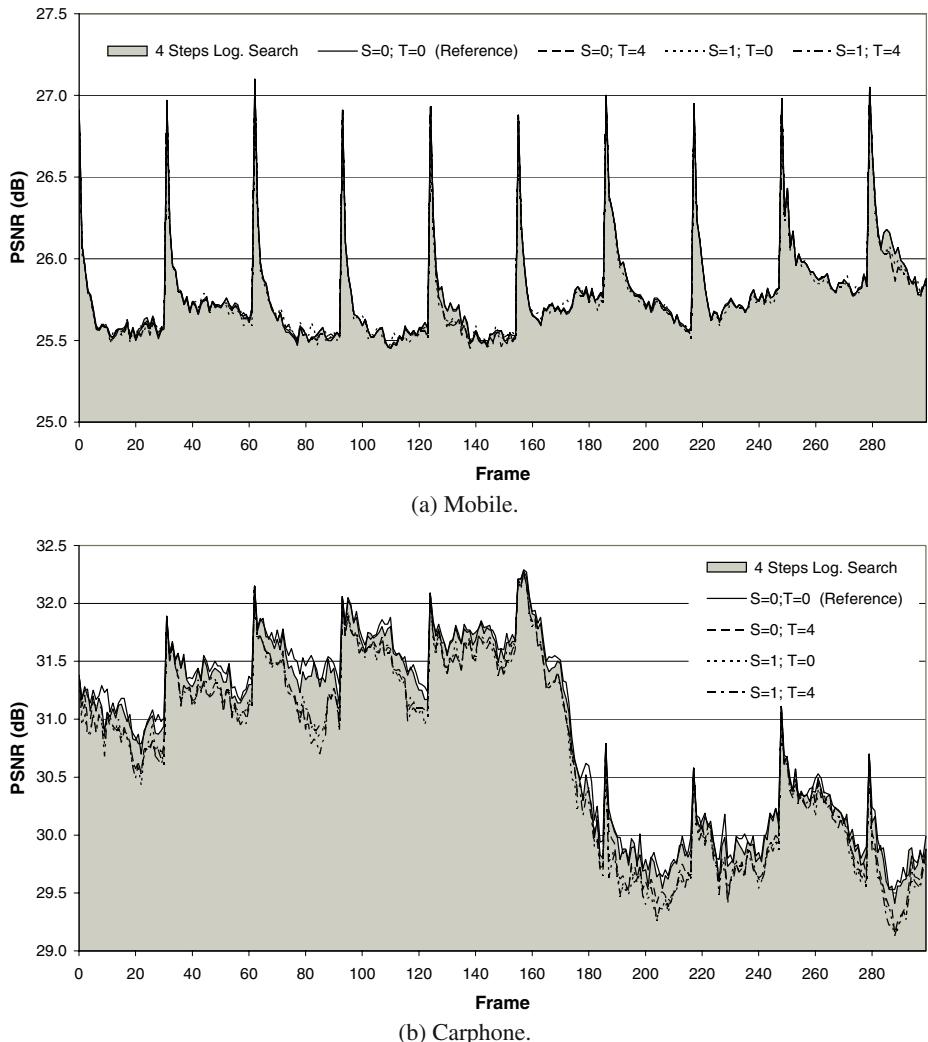


Fig. 3. Comparison between the PSNR measure obtained for three different setups of the proposed reduced hardware architecture, for the original reference configuration and for the 4-steps logarithmic search algorithm.

in a few frames when the PSNR value obtained with the original (reference) architecture is greater than 30 dB.

As it was previously referred, these video quality results were obtained with a constant quantisation step size. The observed small decrease of the PSNR can be explained by the slight increase of the quantisation error, as a consequence of the inherent increase of the prediction differences in the motion compensated block, obtained with these sub-optimal matching processors. The obtained bit-rate required to store or transfer each

Table 4. Variation of the output bit rate to encode the considered video sequences by using different setups of the proposed core-base architecture and the 4-steps logarithmic search algorithm.

T			T				
S	0	2	S	0	2		
0	31.9 kbps	-0.8%	+3.4%	0	49.9 kbps	+0.0%	+2.7%
1	+1.7%	+3.0%	+3.8%	1	+4.5%	+4.1%	+8.0%
2	+3.9%	+3.8%	+3.8%	2	+10.7%	+8.4%	+8.6%
Four-step logarithmic search			Four-step logarithmic search				
(a) Miss America.			(b) Silent.				

T			T				
S	0	2	S	0	2		
0	271.4 kbps	+0.0%	+0.3%	0	79.3 kbps	-0.8%	+4.8%
1	+1.1%	+0.3%	+0.5%	1	+7.3%	+5.8%	+11.6%
2	+6.7%	+0.6%	+0.5%	2	+14.5%	+11.6%	+12.0%
Four-step logarithmic search			Four-step logarithmic search				
(c) Mobile.			(d) Carphone.				

coded video sequence is presented in table 4 for the two sequences considered above and for two more sequences (*Miss America* and *Silent*). The relative values presented in table 4 reveal the increment of the average bit-rate when the number of truncated bits and the sub-sampling factor increase. This increment reaches its maximum value of 12% for the *carphone* sequence and for a processor setup with only 4 representation bits and a pixel decimation of 2 : 1, due to the presence of a lot of movement. Nevertheless, the obtained values for the other three considered video sequences with less amount of movement are quite smaller and are similar to those obtained with the 4-steps logarithmic search algorithm.

Consequently, one can conclude that the configuration using a 2 : 1 decimation ($S = 1$) and using 4 representation bits ($T = 4$) presents the best compromise between hardware cost (a reduction of about 70%) and video quality. Moreover, this configuration could be implemented by using the lower-cost XCV1000 FPGA, which only has about 40% of the total number of system gates provided by the XCV3200 FPGA.

5 Conclusion

In this paper, new customisable core-based architectures are proposed to implement real-time motion estimation processors on FPGAs. The base core of these architectures is a new 2-D array structure for FSBM motion estimation that leads to an efficient usage of the hardware resources, which is a fundamental requisite in FPL systems. The proposed core-based architectures consist on a wide range of processing structures based on FSBM with different hardware requirements. The reduction of the amount of required hardware is achieved by applying decimation at the pixel and quantisation levels, but still searching all candidate blocks of a given search area.

The proposed core-based architectures were implemented on FPGAs devices from Xilinx and their performance were evaluated by including the motion estimaton processors on a complete video encoding system. Experimental results were obtained by sub-sampling the block of the current frame with 2 : 1 and 4 : 1 decimation factors and by truncating 2 or 4 LSBs of the representation. From the obtained results it can be concluded that a significant reduction of the hardware resources is achieved with these architectures. Moreover, the quality of the coded video is not compromised and the corresponding bit-rate is not significantly increased. One can also conclude from the results that the configuration using a 2 : 1 decimation ($S = 1$) and using 4 representation bits ($T = 4$) presents the best compromise between hardware cost (a reduction of about 70%) and video quality.

Acknowledgements

This work has been supported by the POSI program and the *Portuguese Foundation for Science and for Technology* (FCT) under the research project *Configurable and Optimized Processing Structures for Motion Estimation* (COSME) POSI/CHS/40877/2001.

References

1. Bhaskaran, V., Konstantinides, K.: Image and Video Compression Standards: Algorithms and Architectures. 2nd edn. Kluwer Academic Publishers (1997)
2. Koga, T., Iinuma, K., Hirano, A., Iijima, Y., Ishiguro, T.: Motion-compensated interframe coding for video conferencing. In: Proc. Nat. Telecomm. Conference, New Orleans, LA (1981) G5.3.1–G5.3.5
3. Jain, J.R., Jain, A.K.: Displacement measurement and its application in interframe image coding. IEEE Transactions on Communications **COM-29** (1981) 1799–1808
4. Roma, N., Sousa, L.: Efficient and configurable full search block matching processors. IEEE Transactions on Circuits and Systems for Video Technology **12** (2002) 1160–1167
5. Ooi, Y.: Motion estimation system design. In Parhi, K.K., Nishitani, T., eds.: Digital Signal Processing for Multimedia Systems. Marcel Dekker, Inc (1999) 299–327
6. Vos, L., Stegherr, M.: Parameterizable VLSI architectures for the full-search block-matching algorithm. IEEE Transactions on Circuits and Systems **36** (1989) 1309–1316
7. Liu, B., Zaccarin, A.: New fast algorithms for the estimation of block matching vectors. IEEE Transactions on Circuits and Systems for Video Technology **3** (1993) 148–157
8. Ogura, E., Ikenaga, Y., Iida, Y., Hosoya, Y., Takashima, M., Yamash, K.: A cost effective motion estimation processor LSI using a simple and efficient algorithm. In: Proceedings of International Conference on Consumer Electronics - ICCE. (1995) 248–249
9. Lee, S., Kim, J.M., Chae, S.I.: New motion estimation algorithm using adaptively-quantized low bit resolution image and its vlsi architecture for MPEG2 video coding. IEEE Transactions on Circuits and Systems for Video Technology **8** (1998) 734–744
10. He, Z.L., Chan, K.K., Tsui, C.Y., Liou, M.L.: Low power motion estimation design using adaptative pixel truncation. In: Proceedings of the 1997 international symposium on Low power electronics and design, Monterey - USA (1997) 167–171
11. Telenor Research Norway: TMN (H.263) encoder/decoder, version 2.0. (1996)

A High Speed Computation System for 3D FCHC Lattice Gas Model with FPGA

Tomoyoshi Kobori and Tsutomu Maruyama

Institute of Engineering Mechanics and Systems, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 JAPAN
kobori@darwin.esys.tsukuba.ac.jp

Abstract. In this paper, we describe a new computation method for 3D FCHC lattice gas model with FPGA. FCHC lattice gas model is a class of 3D cellular automata and used for simulating fluid dynamics. Many approaches with FPGAs for cellular automata have been researched to date. However, practical three dimensional cellular automata such as an FCHC lattice gas model could not be processed efficiently because they required large size data for each cell and very complex update rules for computing cells. We implemented the new method on an FPGA board with one XC2V6000. The speed gain for FCHC lattice gas model with $128 \times 128 \times 128$ lattice is about 200 times compared with Athlon processor 1800 MHz.

1 Introduction

Lattice gas model is a class of cellular automata, and used for simulating fluid dynamics. Because of its simplicity and data independency, lattice gas model has been widely studied, and many approaches for its high performance computing with parallel systems and FPGAs have been proposed [4][5].

We have proposed a new computation method for two dimensional (2D) cellular automata, and showed that 2D lattice gas model can be accelerated on a small system with limited memory bandwidth[3]. This system for 2D cellular automata consists of only one desktop computer and one PCI board with one FPGA, and users are able to compute various kinds of 2D cellular automata by reconfiguring circuits in the FPGA according to C-like programs written by the users[2]. We have also proposed a computation method for three dimensional (3D) cellular automata which is an extension of the 2D computation method and runs efficiently on the same hardware system[1]. With this extended method, we showed that we could drastically accelerate the computation of several simple 3D models. This computation method, however, could not efficiently process practical 3D cellular automata such as a 3D FCHC lattice gas model, because the number of data input/output width of the FPGA is still limited, and can not read/write data of many cells at a time when the data width of the cells is large, which is a common case in practical 3D models.

With the recent improvement of the size and functions of FPGAs (especially the size of internal RAMs, the capability of dual-port RAMs and more number of

usable I/O pins), it becomes possible to compute practical 3D cellular automata models such as an FCHC lattice gas model with one latest FPGA.

In this paper, we propose an extended computation method for 3D FCHC lattice gas model based on our previous 3D computation method. This computation method is also based on the computation method for 2D lattice gas model we proposed before. In this computation method, status of L cells (decided by the number of I/O pins of the FPGA) are read from the external memory in every clock cycle, and two $x - y$ layers of the lattice and a small part of the next $x - y$ layer are stored in the FPGA. Then, status of L cells in the FPGA are updated for N times (thus, $N \times L$ cells are processed in parallel), and, the new status of the L cells are written back to another external memory. Using this computation method, cells with large data width on large size lattice can be efficiently processed in parallel and pipeline.

In this paper, we, first, introduce the FCHC lattice gas model and the computation method for the model. After that, we describe the implementation of the method and its results.

2 FCHC Lattice Gas Model

2.1 Overview

Many of 2D lattice gas models are based on FHP model with hexagonal cells. Extension to 3D lattice gas models from the FHP models, however, is not straightforward. In 3D lattice gas model, no regular cell with the required symmetries (for 3D lattice gas model) exists. Thus, the *Four dimensional Face centered Hyper-Cubic* (FCHC) cell is used to satisfy the required symmetries.

In FCHC model (Figure 1(a)), to project four dimensional (4D) grid onto 3D grid, the fourth dimension (w axis) is considered as the periodical boundary in three dimensions and takes only 2 values ((1, 0), (-1, 1) and (-1, 0)). The Figure 1(b) shows an example of coordinate system in FCHC lattice model. With the arrangement of cells shown in Figure 1(b), the other three axes are not influenced by w axis. Each cell in FCHC has 25 particles (on 24 directions to other cells and one position to stay in the cell), and only one particle is allowed to travel in

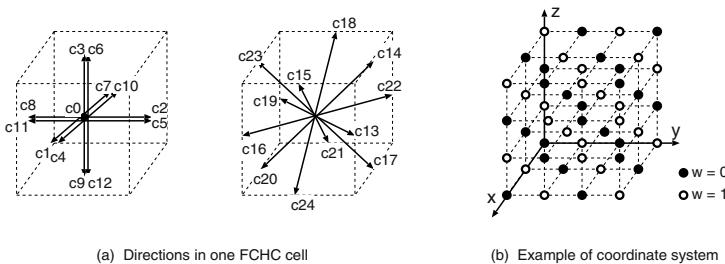


Fig. 1. FCHC Model

each direction. The vectors indicated by each particle on 24 directions are shown as follows:

$$\begin{aligned} C_1 &= (1, 0, 0, 1), & C_2 &= (0, 1, 0, 1), & C_3 &= (0, 0, 1, 1), & C_4 &= (1, 0, 0, -1), \\ C_5 &= (0, 1, 0, -1), & C_6 &= (0, 0, 1, -1), & C_7 &= (-1, 0, 0, -1), & C_8 &= (0, -1, 0, -1), \\ C_9 &= (0, 0, -1, -1), & C_{10} &= (-1, 0, 0, 1), & C_{11} &= (0, -1, 0, 1), & C_{12} &= (0, 0, -1, 1), \\ C_{13} &= (1, 1, 0, 0), & C_{14} &= (0, 1, 1, 0), & C_{15} &= (1, 0, 1, 0), & C_{16} &= (1, -1, 0, 0), \\ C_{17} &= (0, 1, -1, 0), & C_{18} &= (-1, 0, 1, 0), & C_{19} &= (-1, -1, 0, 0), & C_{20} &= (0, -1, -1, 0), \\ C_{21} &= (-1, 0, -1, 0), & C_{22} &= (-1, 1, 0, 0), & C_{23} &= (0, -1, 1, 0), & C_{24} &= (1, 0, -1, 0). \end{aligned}$$

In FCHC lattice gas model, each cell consists of 25 bits data (for 24 directions to neighbor cells and 1 bit for a staying particle), and 25 bits is necessary (24 bits for particles coming from neighbor cells and 1 bit for a staying particle) to compute new status of each cell. Owing to this large number of parameters to decide new status, the update rule becomes very complex, and even in softwares, tables (called collision table) are used to compute new status of each cell. The size of the table is, however, quite large (the number of entries is $2^{25}\text{bit} = 33554432$, and the size of table becomes 512 Mbits). Therefore, the reduction algorithm [7] is used to reduce the size of the table.

2.2 Reduction Algorithm

The reduction strategy is called *momentum normalization*. A basis of the strategy is to specify that the reduced table will contain only the states which have a *normalized momentum*. The coordinates of the momentum \mathbf{q} are defined by

$$q_\alpha = \sum_{i=1}^{24} [S_i C_{i\alpha}] \quad (\alpha = 1, \dots, 4)$$

where S_i is the boolean variable representing the presence or absence of a particle with velocity \mathbf{C}_i as shown in section 2.1. The momentum is said to be normalized if the coordinates satisfy the following

$$q_1 \gg q_2 \gg q_3 \gg q_4 \gg |q_1 - q_2 - q_3| \tag{1}$$

This definition corresponds to a simple sequence of 11 step reduction procedures below. It is not difficult to show that a given input state S is reduced to a state with a normalized momentum by the following sequence. In each step, an *optional isometry* which is applied if the given inequality is satisfied.

1. If $q_1 < 0$, S_1 is applied to \mathbf{q} .
2. If $q_2 < 0$, S_2 is applied to \mathbf{q} .
3. If $q_3 < 0$, S_3 is applied to \mathbf{q} .
4. If $q_4 < 0$, S_4 is applied to \mathbf{q} .
5. If $q_1 < q_2$, P_{12} is applied to \mathbf{q} .
6. If $q_3 < q_4$, P_{34} is applied to \mathbf{q} .
7. If $q_1 < q_3$, P_{13} is applied to \mathbf{q} .
8. If $q_2 < q_4$, P_{24} is applied to \mathbf{q} .

9. If $q_2 < q_3$, P_{23} is applied to \mathbf{q} .
10. If $q_1 + q_4 < q_2 + q_3$, apply \sum_1 .
11. If $q_1 < q_2 + q_3 + q_4$, apply \sum_2 .

In this sequence,

S_α : symmetry with respect to the plane x_α

$P_{\alpha\beta}$: symmetry with respect to the plane $x_\alpha = x_\beta$

\sum_1 : symmetry with respect to the plane $x_1 + x_4 = x_2 + x_3$

\sum_2 : symmetry with respect to the plane $x_1 = x_2 + x_3 + x_4$

If the collision table is invariant under duality, the additional sequence can be executed before the momentum normalization. In the additional sequence, if the number of particles exceeds 12, the input state is replaced by its duality (the input state is bit-inverted). This sequence of procedures brings the number of status down to 316273.

3 Computation Method for FCHC Lattice Gas Model

In this section, we would like to introduce a new computation method for FCHC lattice gas model.

3.1 Basic Idea of the Computation Method

Basic idea of the computation method is based on the computation method for 2D model that we proposed in [3]. The computation method for 2D model consists of two phases: parallel processing of L cells (L is decided by the number of I/O pins and bit-width of one cell), followed by pipeline processing for applying update rule N times continuously (Figure 2). This method can be classified into three strategies by the following relations.

1. I/O width of the FPGA $>=$ lattice width
2. I/O width of the FPGA $<$ lattice width \leq FPGA internal memory size
3. I/O width of the FPGA $<$ FPGA internal memory size $<$ lattice width

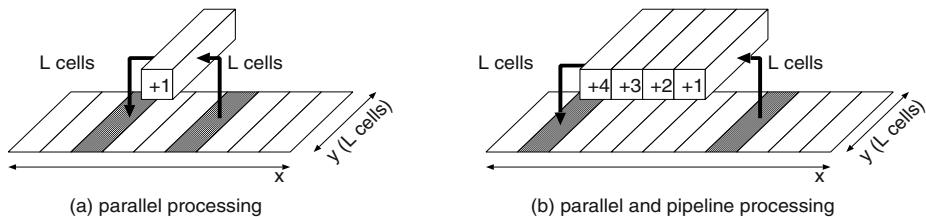


Fig. 2. Basic Strategy of the Computation Method for 2D Cellular Automata

The typical size of lattice in FCHC lattice gas model is $128 \times 128 \times 128$. Thus, the second strategy can be used for a basis of a new computation method for 3D FCHC model when we use the latest FPGAs (with large size internal RAMs). When the size of lattice is larger than FPGA internal memory size, the third strategy can be used though it causes some overhead as described in [3].

3.2 Outline of the Computation Method

Figure 3 shows the computation method for 3D FCHC model. In Figure 3, suppose that an FPGA can read data of L cells at once (along x direction; data on the lattice are arranged to read L cells along x direction in advance) and have three memory planes to store all data of three $x-y$ layers of the lattice (one layer has $x \times y$ cells, typically 128×128 cells).

As shown in Figure 3(1), first, the FPGA continues to read L cells in each clock cycle. When three layers ($3 \times x \times y$ cells) from the top of the lattice are stored in the three memory planes (Figure 3(2)), the update rule is applied to L cells (black parts of Figure 3(2)) in the second plane which stores the second layer. In subsequent clock cycles, new status of L cells are computed in every clock cycle by using the stored cells. During this computation, cells in the fourth layer of the lattice are read in continuously and stored in the first memory plane

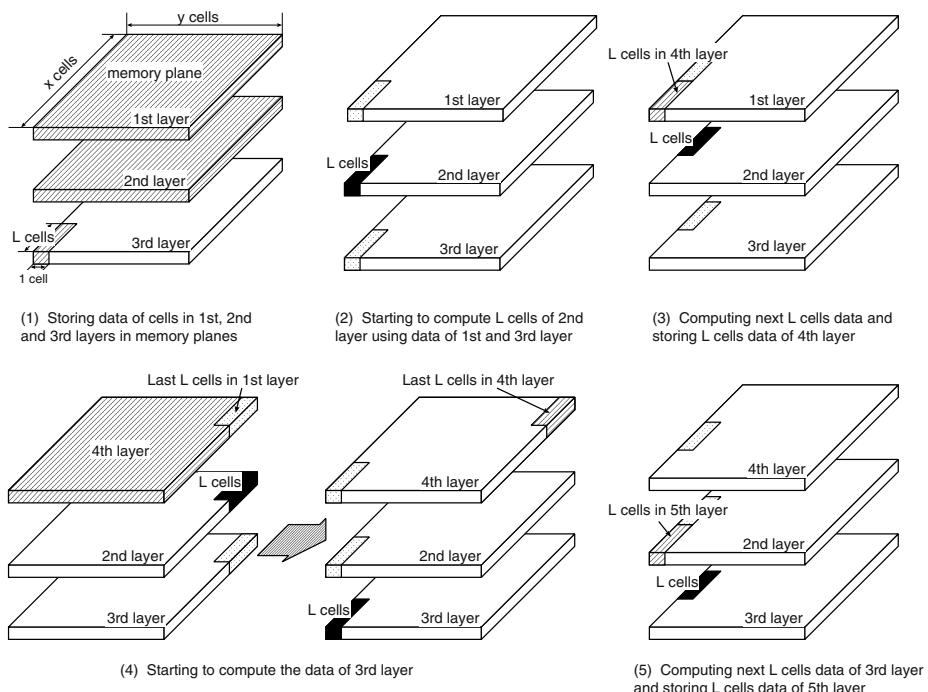


Fig. 3. Outline of the Computation Method for 3D Cellular Automata

because the cells in the first layer are not used again (Figure 3(3)). In this phase, dual-port access to the memory plane is very important because we can continue the computation which accesses the three memory planes while loading data in the next layer of the lattice into one of the three memory plane (if dual-port access is not supported, we need one more memory plane to process the computation and the data loading in parallel).

When status of all cells in the second layer (in the second memory plane) are updated, L cells in third layer (in the third memory plane) are started to be updated (Figure 3(4)), and, L cells in the fifth layer of the lattice are beginning to be read into the second memory plane (Figure 3(5)).

In these procedures, status of L cells are updated every clock cycle. The generation of the output by the FPGA is the generation of the inputs +1. By pipelining the circuit and storing data of cells in $N \times 3$ layers, we can process $L \times N$ cells at a time (N is the number for applying the update rule continuously). In this case, the generation of the output by the FPGA becomes the generation of the inputs + N , and the effective parallelism is $L \times N$.

3.3 An Improvement to Reduce the Memory Size

Figure 4 shows an improvement of the method to reduce the memory size. In Figure 4, two memory planes to store two x - y layers and a small size memory to store a part of the next layer are used. In the small size memory, two lines along x axis ($x \times 2$ cells), and $L \times 2$ cells in the next line are stored. In our circuit which is described in Section 4, L is 2, and typical sizes of x and y are 128 respectively, therefore, memory size reduction by this improvement is very effective.

As shown in Figure 4(1), first, an FPGA reads two layers from the top of the lattice in the two memory planes and two lines along x axis in the small size memory by reading L cells in every clock cycle. In practice, when the FPGA finishes to read in the first layer and the first line of the next layer, the FPGA becomes ready to compute new status of cells on boundaries. However, in order to simplify the figure and its explanation, we skip the computation of cells on boundaries. When $2 \times L$ cells of the third line are stored in the small size memory as shown in Figure 4(2), status of L cells (black part in Figure 4(2)) are updated (the update of the first $L - 1$ cell is skipped again because it includes a boundary computation). The two lines and $2 \times L$ cells are necessary to give correct neighbor cells in the bottom layer to the L cells which are updated. Then, FPGA reads next L cells on the third line, and status of next L cells (black part in Figure 4(3)) are updated. The first L cells in the first plane are not necessary any more, thus, the first L cells in the small size memory (gray part in Figure 4(3)) are moved to the part of the first memory plane in which the first L cells in the first layer is stored. In Figure 4(4), FPGA reads the next L cells on the third line, and status of next L cells (black part in Figure 4(4)) are updated, and the next L cells in the small size memory (gray part in Figure 4(4)) are moved to the first memory plane again.

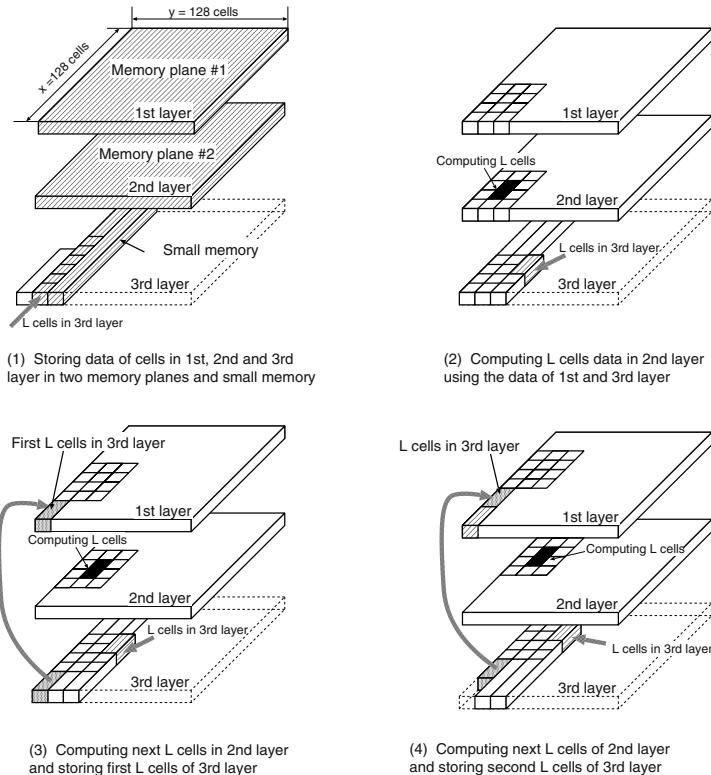


Fig. 4. An Improved Method for 3D Cellular Automata

By repeating this cycle of reading L cells into small size memory, updating L cells on the first/second memory plane, and moving L cells from the small size of memory to the second/first memory plane, we can update status of all cells of the whole lattice.

4 Details of the Implementation of the Computation Method

4.1 ADM-XRC-II PCI Mezzanine Board

Before describing the detail of the implementation of the method, we would like to introduce features of the FPGA and the PCI board on which we implemented the method.

The PCI board (ADM-XRC-II by Alpha Data) has one Virtex-II XC2V6000, six 4MB SSRAM banks with 32 bit width (24MB and 196 bit width in total), and flexible front panel I/O. In our implementation, the front panel I/O is used for an interface with two additional SSRAM banks (8MB each). Accordingly, we

can use eight SSRAM banks. Each memory bank can be accessed independently from the FPGA and the host computer. DMA data transfer is supported for the memory access by the host computer.

Virtex XC2V6000 has two kinds of memories; distributed RAMs consists of LUTs and block RAMs (dual-port). With the dual-port block RAMs, we can store enough data of cells required by the computation method.

4.2 The Overview of the Circuit on XC2V6000

The total I/O width of the FPGA is 32×8 bits because the FPGA board support eight memory banks. In our implementation, two memory banks are used to store input to the FPGA (data of the lattice of generation g), and another two memory banks are used to store the output of the FPGA (data of the lattice of generation $g + N$, where N is the pipeline depth). The data width of each cell in the lattice is 25 bits. Therefore, data of two cells are read in and written out in every clock cycle (L is 2). The other four memory banks are used to store the collision table. The width of the collision table is also 25 bits, which means that four cells on the lattice can be processed in parallel. Thus, the depth of the pipeline (N) becomes 2 (4/2).

In order to process four cells in parallel and pipeline, we need to store four layers of the lattice in the FPGA (two layers for the first pipeline stage and two layers for the next pipeline stage). Memory planes to store these four layers are implemented using dual-port Block RAMs of XC2V6000. The maximum layer size which can be stored in XC2V6000 is 128×128 , which is the typical lattice size in the FCHC lattice gas model. In our computation method, we can take any size for z axis. Thus, we can process any type of $128 \times 128 \times k$ size lattice by exchanging x , y and z axis. When the size of other two axes is larger than 128, we can extend our computation method as described in Section 3.1. The two small size memories (one for the first pipeline stage and another for the second pipeline stage) are implemented using shift registers called *Shift Register Look up table* of XC2V6000.

4.3 Details of the Circuit

Figure 5 shows the block diagram of the circuit. In the circuit, there are four (2×2) data computing units to compute next status of cells and two data control units for storing data of cells. With one set of one data control unit and two data computing units, two cells of the same generation are processed in parallel. By duplicating the set, one more generation of the two cells is computed continuously on the FPGA (pipeline processing). Thus, four cells are processed at a time.

Figure 6 (a) shows the structure of the data control unit. Each data control unit consists of block RAMs to store data of two layers ($128 \times 128 \times 2$ cells), and shift registers (*Shift Register Look up table*) to store two read-in lines (2×128 cells) and 2×2 cells. 50 block RAMs and 450 LUTs are used in one data control unit.

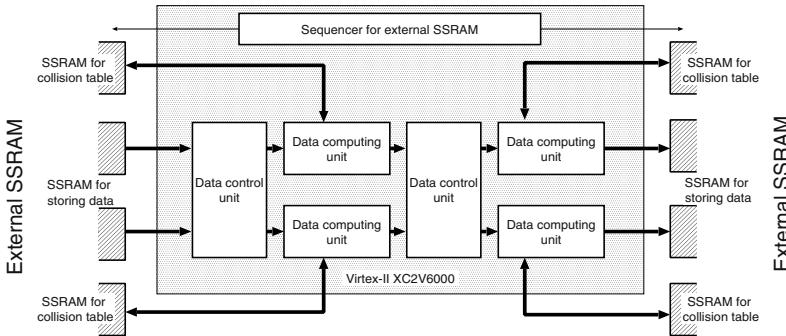


Fig. 5. The Block Diagram of the Circuit

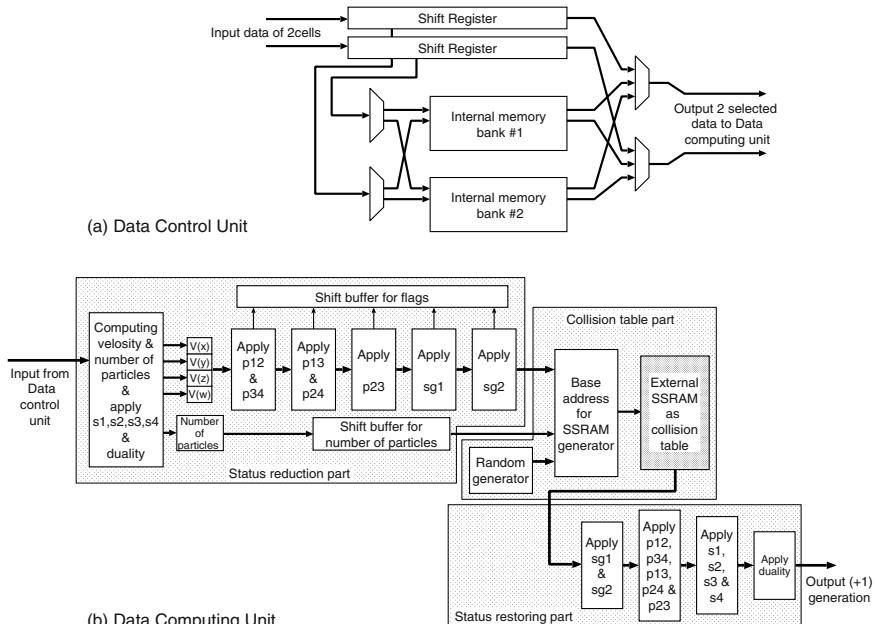


Fig. 6. The Block Diagrams of Sub-units

Figure 6 (b) shows the structure of the data computing unit. It consists of three parts; status reduction part, collision table part and status restoring part. In the status reduction part, input to the unit (25 bits) is reduced to 19 bits, and the collision table in the external SRAM is accessed using the reduced status as its address. Then, each bit in the status which is read out from the collision table is exchanged by re-executing reduction steps in reverse order to restore correct new status in the status restoring part. In status reduction and restoring parts, some rules are applied in parallel. This data computing unit is completely pipelined, and can process new status of one cell in each clock cycle.

5 Results

Table 1 shows the results of 3D FCHC lattice gas model with $128 \times 128 \times 128$ cells. The operation frequency is 52.7 MHz. The performance gain excluding data transfer time between the FPGA and the host computer is 2345 times compared with Athlon Processor 1800 MHz (a C program which is translated from a Fortran Program is used for this comparison).

Table 1. Speed Gain for 3D FCHC Lattice Gas Model

	Computation time (for 2 generation)	Speed gain
Software	$2 \times 23.8 \times 10^3$ (msec)	1
FCHC lattice gas system with DMA	242.3 (msec)	197
FCHC lattice gas system without DMA	20.3 (msec)	2345

This drastic speed gain comes from parallel and pipelined processing (4 cells are processed in parallel and pipeline, data in 24 neighbors are accessed and processed in parallel, the sequence of state reduction steps are executed in parallel and pipeline), and in 3D cellular automata, many data which spread in large address space are accessed for computing new status of one cell, which suppresses the effectiveness of cache memories in micro-processors.

When all outputs by the FPGA are transferred to the host computer (namely, status of all cells in every two generations are sent), the speed gain becomes 197 times. However, we need to transfer the results in every hundreds of generations in general. Thus, we can expect more than one thousand of speed gain.

6 Conclusions

In this paper, we proposed a new computation method of 3D FCHC lattice gas model for small systems with limited memory bandwidth. We implemented the method on a FPGA board (ADM-XRC-II with one Virtex-II XC2V6000), and the speed gain for a 3D FCHC lattice gas model with $128 \times 128 \times 128$ lattice is about 200 times compared with Athlon Processor 1800 MHz. With this method, we could achieve a high performance computing of FCHC lattice gas model which is equivalent to large parallel/distributed systems.

The circuit which we implemented consists of data control unit and data computing unit, which are completely separated. Therefore, the method can be applied for other 3D cellular automata which requires large data width for each cell and very complex update rules. We are going to develop hardware libraries for those 3D cellular automata. With the libraries, user can process those types of 3D cellular automata efficiently and easily.

References

1. Tomoyoshi Kobori, Tsutomu Maruyama. "High Speed Computation of Three Dimensional Cellular Automata with FPGA" FPL2002, PP.1126-1130, 2002.
2. Tomoyoshi Kobori, Tsutomu Maruyama and Tsutomu Hoshino."A Cellular Automata System with FPGA" Proc. FCCM '01, IEEE Computer Soc 2001.
3. Tomoyoshi Kobori, Tsutomu Maruyama and Tsutomu Hoshino. "High Speed Computation of Lattice Gas Automata with FPGA" FPL2000, PP.801-804, 2000.
4. Norman Margolus."An FPGA architecture for DRAM-based systolic computations" Proc. FCCM '97, pp. 2-11, IEEE Computer Soc 1997.
5. C. Adler, B. M. Boghosian, E. G. Flekkoy, N. Margolus and D. H. Rothman, "Simulation Three-Dimensional Hydrodynamics on a Cellular-Automata Machine", Journal of Statistical Physics, 1995
6. U. Frisch, D. d'Humières, B. Hasslacher, P. Lallemand, and Y. Pomeau."Lattice gas hydrodynamics in two and three dimensions" Complex Systems, 1 PP.649-707 1987.
7. M.Hènon, "Implementation of the FCHC Lattice Gas Model on the Connection Machine", J. Stat. Phys. Vol.68,353-377 1992

Implementation of ReCSiP: A ReConfigurable Cell SImulation Platform

Yasunori Osana, Tomonori Fukushima, and Hideharu Amano

Dept. of Computer Science, Keio University
3-14-1 Hiyoshi, Kouhoku-ku, Yokohama #223-8522, JAPAN

Abstract. A reconfigurable accelerator for cell simulators called “ReCSiP” is proposed. It consists of both reconfigurable hardware and software platform. For high performance simulation, numerical solution of kinetic formulas, which require a large amount of computation, are processed on the reconfigurable hardware. It also provides programming interface for developing cell simulators. In this paper, Michaelis-Menten solver is designed and implemented on ReCSiP. The result of preliminary evaluation shows that ReCSiP is 8 times faster than Intel PentiumIII 1.13GHz when simple metabolic simulations are executed.

1 Introduction

Simulating cellular process is a big challenge in both of biology and computer science. Whole-cell simulators which are currently under development[1][2] are much more complicated than traditional computational sciences like fluid dynamics or molecular dynamics because it has various kinds of chemical reaction in a system[3].

When a cellular system is simulated by computer software, the system is modeled as a network of chemical reactions. Each reaction is described in the form of differential equation, and optimal numerical solution is selected and applied on each equation. Since a lot of equations must be solved at every time step, simulations take long computation time.

However, traditional parallel systems (e.g. PC/WS clusters, shared memory machine and so on) are not suitable for performance enhancement because of the frequent synchronization as the result of tight dependencies between chemical reactions in the next (or previous) time-steps.

In this paper, we propose a reconfigurable accelerator “ReCSiP”, and Michaelis-Menten solvers are implemented on it. Chemical reactions are independent in itself, but across time-steps, they have dependencies to each other. This causes communication bottleneck on traditional parallel systems, but this bottleneck can be avoided by parallel processing of differential equations in a single reconfigurable platform.

ReCSiP consists of hardware platform called ReCSiP board and software executed in the host computer. Parallel processing of differential equations of cell simulation is executed on the board. Software layer provides the device driver and programming interface for easy use of the ReCSiP board.

2 Overview of ReCSiP

The purpose of ReCSiP is to accelerate metabolic simulation in private, desktop computing environment. Biologists can use high performance systems such as supercomputers or clusters in grid environment, but such resources are globally shared, and frequently busy and congested. Desktop super computing environment for personal use should be investigated as an alternative choice for biologists. ReCSiP is designed to achieve high throughput computation by the co-operation of the host CPU and reconfigurable platform consisting of FPGA.

As shown in Fig.1, ReCSiP consists of the hardware layer called ReCSiP board, and the software layer with the device driver and API. By using API codes, users can access ReCSiP board directly from their programs. Cell simulators and other applications can be easily accelerated with ReCSiP.

2.1 Structure of ReCSiP Board

ReCSiP board has an FPGA, a Xilinx Virtex-II (XC2V6000-4) as a core reconfigurable resource, and a QuickLogic QuickPCI (QL5064) for 64bit/66MHz PCI interface. 4 sets of 32bit \times 1Mwords (4Mbytes) synchronous SRAM and a

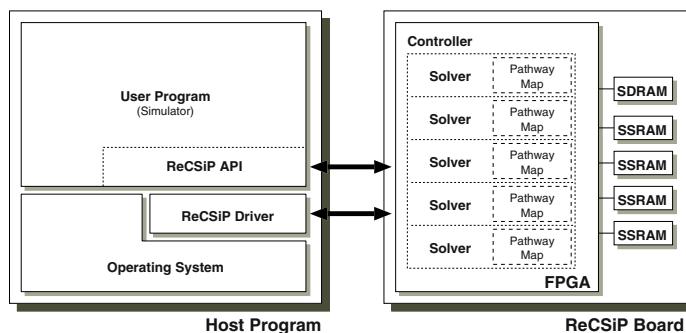


Fig. 1. Overview of ReCSiP

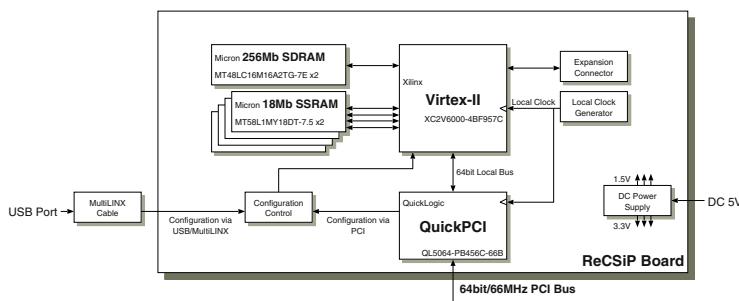


Fig. 2. Structure of ReCSiP Board

Table 1. Components on ReCSiP board

	Vendor	Series	Part No.
Core FPGA	Xilinx	Virtex-II	(XC2V6000-4BF957C)
PCI Interface	QuickLogic	QuickPCI	(QL5064-PB456C-66B)
Memory (SRAM)	Micron	SyncBurst SRAM	(MT58L1MY18DT-7.5)
Memory (DRAM)	Micron	Synchronous DRAM	(MT48LC16M16A2TG-7E)

set of 32bit \times 16Mwords (64Mbytes) SDRAM are connected to Virtex-II. The memory modules can be accessed from the FPGA simultaneously, and remove the bottleneck of memory access from the FPGA. The specification of the board is shown in Table 1.

PCI interface in QuickPCI manages both PIO access and DMA transfer at 64bit/66MHz. The local bus between QuickPCI and Virtex-II is also 64bit/66MHz, and the maximum frequency of SDRAM/SSRAM is 133MHz. The clock frequency of memory modules are controlled by the clock manager on Virtex-II.

For users who design the logic on FPGA, ReCSiP's standard modules are available including local bus interface, SDRAM interface, SSRAM interface and so on.

2.2 The ReCSiP Software

The software provides the programming interface to ReCSiP board. ReCSiP driver is the lowest layer, which has basic interface using `ioctl()`. ReCSiP API is an easy-to-use user level library, but it can access ReCSiP board without the driver's overhead by memory mapped I/O with the system call, `mmap()`.

To extend the API to fit the application, API extension mechanism is implemented. Registers or memories on the board can be accessed from user programs like as usual variables since this mechanism can automatically generate low level interface code from address map file using `mmap()` or other system calls. By this mechanism, users can build up their simulator on ReCSiP easily.

**Fig. 3.** The ReCSiP Board

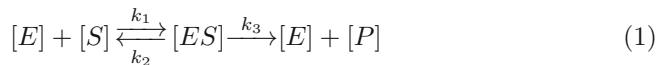
2.3 Michaelis-Menten Solver on ReCSiP

As the first example of cell simulation with ReCSiP, a solver of a basic enzyme reaction kinetic model was designed and evaluated.

2.4 Michaelis-Menten Reaction Model

There are various kinds of chemical reactions in a cellular system, and approximate models are used in kinetic simulations. Michaelis-Menten model is the most widely used model of substrate-enzyme reaction is represented as (1).

In this scheme, $[E]$ is concentration of the enzyme and $[S]$ is concentration of the substrate. The enzyme-substrate complex $[ES]$ is formed by collision of enzyme and substrate. Then $[ES]$ quickly changes to $[E]$ and the product, $[P]$. This reaction is catalyzed.



Formation/degradation velocity of each substance in scheme (1) can be approximated as (2), (3), (4) and (5).

$$\frac{d[S]}{dt} = -k_1[S][E] + k_2[ES] \quad (2)$$

$$\frac{d[P]}{dt} = k_3[ES] \quad (3)$$

$$\frac{d[E]}{dt} = -k_1[S][E] + (k_2 + k_3)[ES] \quad (4)$$

$$\frac{d[ES]}{dt} = k_1[S][E] - (k_2 + k_3)[ES] \quad (5)$$

Therefore, it is possible to calculate concentration of each substance in time series when all initial concentrations and k_s (kinetic constants) in the target system are given.

2.5 The First Prototype

The solver of Michaelis-Menten model using the 1st-order Euler's method was designed and implemented. The first prototype was designed to find problems in the design and implementation of metabolic simulation accelerator on the FPGA. It is only the specialized processing unit for Michaelis-Menten model, so it has no interface logic to PCI or any other systems.

Implementation. The first prototype of Michaelis-Menten solver has following components as shown in Fig.4:

- single precision, floating point adder (addsub)
- single precision, floating point multiplier (mult)
- exponent shifter for numerical integration (shift)

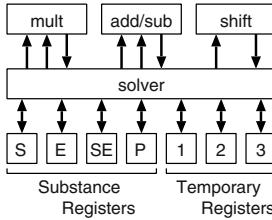


Fig. 4. Structure of the First Prototype

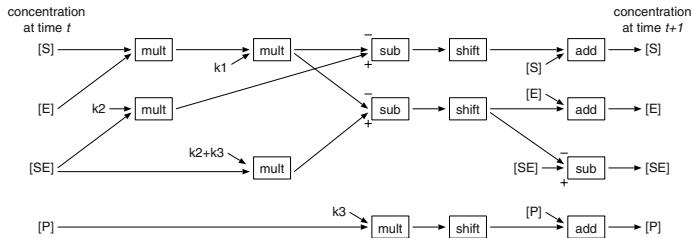


Fig. 5. Steps to Solve the Michaelis-Menten Model

- substance registers which hold concentration of each substances
- temporary registers which hold intermediate data

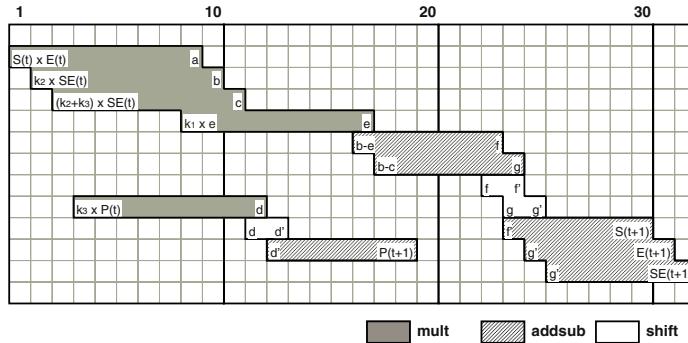
With these components, the kinetic equations ((2)~(5)) with the integration step can be solved as Fig.5. In the first step of this method, each derivative is calculated. Then, the exponent part of derivative is shifted to get d/dt^1 . At last, the difference is added to the value at time t , and the integration completes.

The method shown in Fig.5 was implemented with the modules addsub, mult and shift. The pipeline completes the process of 4 equations in 32 clocks, and it can start a new process every 10 clocks. The schedule of calculation is shown in 6.

Evaluation. The prototype was designed with Verilog-HDL. Synthesis, place and route were done with Synopsys FPGA Compiler-II and Xilinx ISE 4.1. The maximum operating frequency was calculated from the result of place and route, then the throughput was calculated. This evaluation was done with the place and route for multiple units on an FPGA. However, the evaluated system doesn't have any host interface. So it's just for preliminary evaluation.

The result is shown in Table 2. By comparison to throughput of microprocessors in Table 3, 2 units of this prototype have the same throughput as 1.13GHz version of PentiumIII.

¹ This method is very fast and easy to implement numerical integration because it does not need a divider. In our second prototype, d/dt is calculated by multiplying dt to k_n before the simulation starts.

**Fig. 6.** Pipeline Schedule of the First Prototype**Table 2.** The Size, Frequency and Throughput of the First Prototype

Units	Slices (Used %)	Frequency	Throughput
1	2,211 (6%)	62MHz	6.2
2	4,422 (13%)	64MHz	12.8
3	6,644 (19%)	58MHz	17.4
4	8,845 (25%)	56MHz	22.4
8	17,849 (52%)	54MHz	43.2

Table 3. Throughput of Microprocessors

Processor	Frequency (MHz)	OS/Compiler	Throughput (Mop/sec)
UltraSPARC-II	300	Solaris8 / gcc-2.95.2	5.21
Pentium III	800	FreeBSD-4.2 / gcc-2.95.2	6.25
Pentium III	1133	FreeBSD-4.2 / gcc-2.95.2	10.15

The Communication Bottleneck. This prototype receives concentration of 4 substances from the host CPU, then sends them back at each time step. The size of this transaction is $4 \times 2 \times 32(\text{bit}) = 256(\text{bit})$. Here, the maximum bandwidth of 64bit/66MHz PCI bus is $64 \times 66 = 4.2(\text{Gbps})$, so $4.2(\text{Gbps}) \div 256(\text{bit/timestep}) = 16.5(\text{timestep/sec})$ is the maximum processing ability. However, as shown in Table 3, desktop CPU will soon reach to 16Mops/sec. Some breakthrough is needed to solve this communication bottleneck, and exploit the power of the FPGA.

2.6 The Second Prototype

Concept. To avoid the communication bottleneck, the second prototype (Fig.7) was designed and implemented. This prototype has a mechanism to program how the solver receives, processes, stores and sends back data. The program for the solver is stored in “Code RAM”, while the data is stored in “Data RAM”. Data

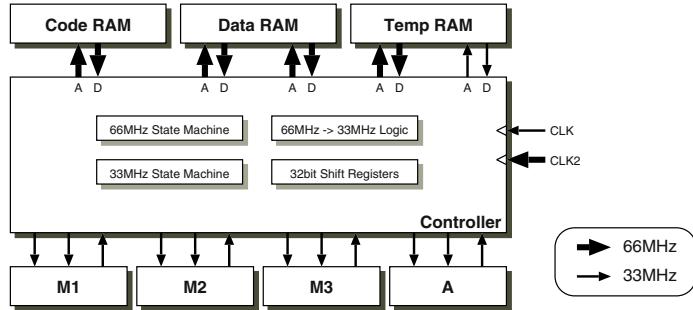


Fig. 7. The Structure of Second Prototype

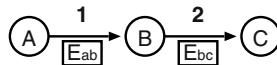


Fig. 8. Sample of Reaction Pathway

in the Data RAM can be used in the succeeding time step, and can be partially modified from the host. Host program does not need to get all data in every time step, since all data in simulation is stored in the Data RAM. By accessing to only essential data, transactions over PCI bus can be reduced drastically.

For example, to process the reaction pathway in Fig.8, data transfer between the solver and host is as follows:

- in the first prototype:
 - to process reaction 1, receive A, B, Eab and A.Eab at time t
 - as the result of reaction 1, send A, B, Eab and A.Eab at time $t + \Delta t$
 - to process reaction 2, receive B, C, Ebc and B.Ebc at time t
 - as the result of reaction 2, send B, C, Ebc and B.Ebc at time $t + \Delta t$
- in the second prototype:
 - receive the program
 - to process reaction 1 and 2, receive A, B, C, Eab, Ebc, A.Eab and B.Eab at time t *if necessary*
 - as the result of the reactions, A/dt , B/dt , C/dt , Eab/dt , Ebc/dt , $A.Eab/dt$ and $B.Eab/dt$ *if necessary*

In this way, transactions over PCI bus can be reduced by using the memory on the accelerator board.

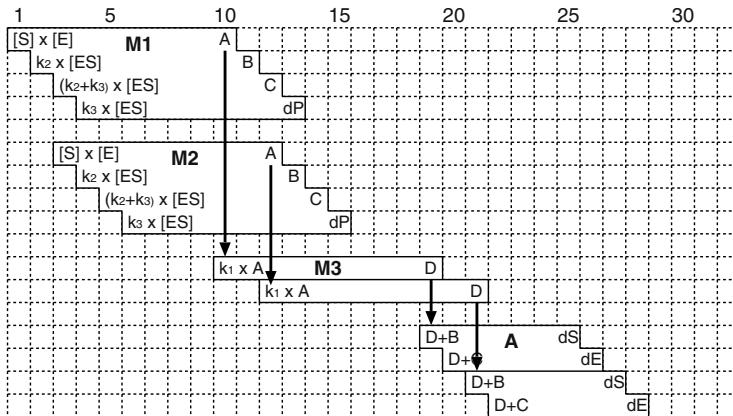
Implementation. As shown in Fig.7, the second prototype of the solver has 3 multipliers (M1, M2 and M3) and an adder (A). The solver itself is fully pipelined, and controlled with a simple finite-state machine as shown in Fig.9. The address for Code RAM is simply incremented by every clock cycle, then Data RAM is accessed as Code RAM points to.

Table 4. Data RAM for the Sample Pathway

Address	Data	Address	Data	Address	Data	Address	Data
00h	A	04h	k_{11}	08h	B	0ch	k_{21}
01h	Eab	05h	k_{12}	09h	Ebc	0dh	k_{22}
02h	A.Eab	06h	k_{13}	0ah	B.Ebc	0eh	k_{23}
03h	B	07h	$k_{12} + k_{13}$	0bh	C	0fh	$k_{22} + k_{23}$

Table 5. Format of Code RAM

Address	Pointer	
$4n \times 00h$	S	E
$4n \times 01h$	k_2	ES
$4n \times 02h$	$k_2 + k_3$	k_1
$4n \times 03h$	k_3	—

**Fig. 9.** Pipeline Schedule of the Second Prototype

The solver is fully pipelined, and it can start process each 2 clocks. Most of the logic operates at 33MHz, Data RAM and Code RAM run at 66MHz to ensure sufficient bandwidth. Code RAM is the array of pointer, which points to Data RAM in the format shown in Table 5. The width of Code RAM is 32 bit, and two 16 bit pointers are stored in each address. Some special values are reserved as control commands.

The second prototype is designed so as to be implemented on ReCSiP board. It provides a simple and easy software interface by mapping Code RAM, Data RAM and some control registers on the memory address space of the host processor, via PCI bus. Simulation will start by storing the code and data on each RAM, then kick the control register.

Evaluation. The size, maximum operating frequency and throughput are shown in Table 7. It is about 8 times faster than 1.13GHz version of Intel PentiumIII.

Table 6. Contents of Code RAM for the Sample Pathway

Address	Pointer		Address	Pointer	
00h	A	Eab	04h	B	Ebc
01h	k_{12}	A.Eab	05h	k_{22}	B.Ebc
02h	$k_{12} + k_{13}$	k_{11}	06h	$k_{22} + k_{23}$	k_{21}
03h	k_{13}	—	07h	k_{23}	—
			08h	END	—

Table 7. The Size, Frequency and Throughput of the Second Prototype

Units	Slices (Used %)	Frequency	Throughput
1	3,713 (10%)	48MHz	24
2	7,449 (22%)	44MHz	44
3	11,163 (33%)	43MHz	64
4	14,897 (44%)	42MHz	84

With Code RAM and Data RAM, transactions over the PCI bus can be reduced as possible, and so it is not a bottleneck now.

Controlling the solver by Code RAM and Data RAM is also a highly flexible way, since it can describe complicated reaction pathway. However, each unit on the FPGA cannot communicate each other in the current implementation. To exploit the performance of this accelerator, some communication facilities like shared registers or FIFOs should be implemented, to process a large reaction pathway in parallel.

3 Conclusion

A host-accelerator co-operation environment for bioinformatics, “ReCSiP” is proposed. As the preliminary evaluation, a solver for a basic metabolic simulation using Michaelis-Menten model is implemented. Its performance is 8 times as that of 1.13GHz PentiumIII by the parallel execution in the solver.

Current Michaelis-Menten solver needs further improvement to run various type of simulations with more realistic models. For example, some more stiff algorithms of numerical solution of differential equations should be implemented.

Now, implementation of other applications including microscopic image processing or DNA sequence matching is going on.

Now we’re planning to design ReCSiP board version 2. The following new facilities are added on the current version:

- 18Mbit QDR-SRAMs instead of current 18Mbit synchronous SRAM,
- a DDR-SDRAM SO-DIMM slot for running big simulations, and
- a direct board to board communication interface.

References

1. J. Schaff et al. The virtual cell. In *Proceedings of Pacific Symposium on Biocomputing*, volume 4, pages 228–239, Jan. 1999.
2. Masaru Tomita et al. E-cell: software environment for whole-cell simulation. *Bioinformatics*, 15(1):72–84, Jan. 1999.
3. Kouichi Takahashi et al. Computational challenges in cell simulation: A software engineering approach. *IEEE Intelligent Systems*, pages 64–71, Oct. 2002.

On the Implementation of a Margolus Neighborhood Cellular Automata on FPGA

Joaquín Cerdá¹, Rafael Gadea¹, Vicente Herrero¹, and Angel Sebastià¹

¹ Group of Digital Systems Design, Dept. Of Electronic Engineering,

Universidad Politécnica de Valencia,

46022 Valencia, Spain

{joacerbo, rgadea, viherbos, asebasti}@eln.upv.es

<http://dsd.upv.es>

Abstract. Margolus neighborhood is the easiest form of designing Cellular Automata Rules with features such as invertibility or particle conserving. In this paper we propose two different implementations of systems based on this neighborhood: The first one corresponds to a classical RAM-based implementation, while the second, based on concurrent cells, is useful for smaller systems in which time is a critical parameter. This implementation has the feature that the evolution of all the cells in the design is performed in the same clock cycle.

1 Introduction

Cellular Automata (CA) model massively parallel computation and physical phenomena [1]. They consist of a lattice of discrete identical sites called *cells*, each one taking a value from a finite set, usually a binary set. The value of the cells evolve in discrete time steps according to deterministic rules that specify the value of each site in terms of the values of the neighboring sites. This is a parallel, synchronous, local and uniform process [1, 2, 5, 10, 14].

CA are used as computing and modeling tools in biological organization, self-reproduction, image processing and chemical reactions. Also, CA have proved themselves to be useful tools for modeling physical systems such as gases, fluid dynamics, excitable media, magnetic domains and diffusion limited aggregation [13, 3, 4, 5, 8, 12]. CA have been also applied in VLSI design in areas such as generation of pseudo-random sequences and their use in built-in self test (BIST), error-correcting codes, private-key cryptosystem, design of associative memory and testing the finite state machine [9, 10, 11, 15, 16].

1.1 Invertible Cellular Automata and Margolus Neighborhood

A CA is invertible when its global function is a bijection, i.e., if every configuration – which, by definition, has exactly one successor – also has exactly one predecessor [6].

Invertibility is such a desiderable property, because in the context of dynamical systems coincides with microscopic reversibility. One of the most common ways of constructing invertible cellular automata is by using a partitioning schema [6].

Partitioning Cellular Automata (PCA) are based on a different kind of local map that takes as input the contents of a region and produces as output the new state of the whole region (rather than of a single cell). This way, the state space is completely divided into non-overlapping regions. In order to exchange information between regions, partitions must change at the next step. The partitioning format is specially good for many applications because it makes very easy to construct invertible rules.

The most important partitioning scheme is the Margolus Neighborhood, introduced in [3]. In this neighborhood each partition is 2x2 cells as shown in figure 1. We alternate between even partitions (solid lines) and odd partitions (dotted lines) in order to couple them all. Periodic boundary conditions are assumed.

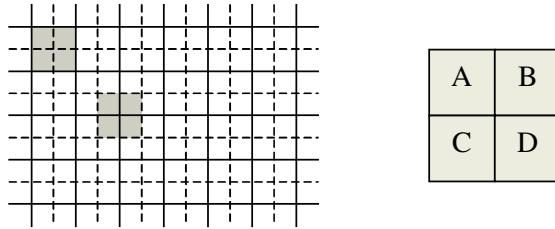


Fig. 1. Margolus Neighborhood (*left*): even (*solid lines*) and odd (*dotted lines*) partitions of a two-dimensional array into 2x2 blocks. The block on the right introduces a notation to refer to the cells into the partition

Several rules based on Margolus neighborhood have been introduced in different areas. Among them we can mention BBMCA introduced by Margolus itself and capable of universal computation [3], rules TM and HPP for modelling gases [8] and the rule DIAG_then_DOWN for simulating particles that fall down and pile up [7].

2 Sequential Implementation of Margolus neighborhood

A classical architecture for implementing Margolus neighborhood at VLSI is presented in figure 2. In this approach, the lattice of cells is stored in a RAM memory disposed as a 2D array of single bits. The control path has to generate the proper signals to read the four positions that form a block and present them to the process unit. The process unit applies the transformation codified by the local map and after it is finished, the control generates the signals to store the result back in the memory.

It is necessary for the control unit to have an input from a parity generator that ensures the correct alternancy between even and odd evolutions, so the directions in the memory to be accessed are different in each case.

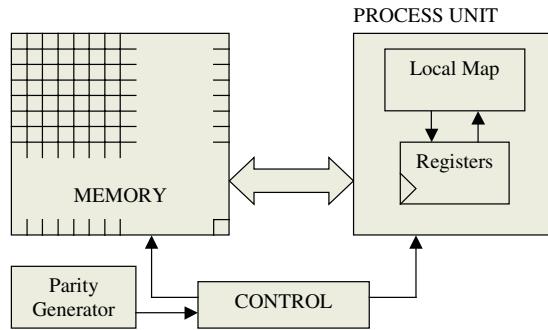


Fig. 2. Sequential implementation of Margolus neighborhood. The value of the cells is stored in a RAM array

The proposed architecture was implemented on several FPGA families to compare performances. For the logic synthesis we used LeonardoSpectrum Version 2002b.21 from Exemplar Logic. This allows us to synthesize on different families from several manufactures. The results shown here are those obtained for ALTERA and XILINX devices. For the ALTERA devices the implementation and simulation was performed with MAX+PLUSII v. 10.0 (FLEX10K family) and QUARTUS II v. 2.2 (APEX20K, APEXII and STRATIX families). Implementation and simulation on XILINX devices was performed using FOUNDATION SERIES 3.1i.

2.1 Sequential Control Implementation

As a first stage, we implemented only the control unit on the FPGA. This is thought to operate on a external RAM in which information about cell states is stored. Table 1 summarizes the results on Maximum frequency and Logic Cells (LCs) needed for ALTERA devices. Implementations are done for a 8x8 matrix of cells.

Similar results can be given for XILINX Devices. XILINX Slice is more complex than ALTERA LC. In fact, it contains 2 LUTs, so used resources seem to be lower in this case, but it is due to the granularity.

Table 1. Results of Sequential Control Implementation in ALTERA devices

Family	FLEX10K	APEX20K	APEXII	STRATIX
Device	EPF10K20 RC240-3	EP20K200 RC240-1	EP2A25 F672C7	EP1S25 F780C6
LCs	72	75	75	61
Maximum Frequency	82,64MHz	87,67MHz	268,24MHz	308,36MHz

Table 2. Results of Sequential Control Implementation in XILINX devices

Family	4000	VIRTEX	VIRTEXII
Device	4020XLPQ160-09	V200BG256-6	XC2V80FG256-4
Slices	25	38	38
Maximum Frequency	82,905MHz	158,806MHz	173,461MHz

2.2 Using Embedded Memory

In most cases we can use the embedded memory included in the devices for storing information about the matrix of cells. This limits the maximum size of the system: The maximum size of the matrix of cells depends on the total amount of RAM embedded on the chip. Table 3 gives the results of implementation of a 8x8 matrix on the listed ALTERA devices. Also we have included the maximum size of the array allowed for a device and for a family¹.

The same results for XILINX devices are given in Table 4. It is important to note that some families, such as 4000, lacks of embedded memory and emulate it using distributed memory (LUTs).

Table 3. Results of Sequential Implementation using embedded memory in ALTERA devices

Family	FLEX10K	APEX20K	APEXII	STRATIX
Device	EPF10K20	EP20K200	EP2A25	EP1S25
	RC240-3	RC240-1	F672C7	F780C6
LCs	71	75	76	61
Maximum Frequency	30,58MHz	54,24MHz	113,91MHz	187,21MHz
Maximum Size (Device)	64x64	256x256	512x512	1024x1024
Maximum Size (Family)	128x128	512x512	1024x1024	2048x2048

Table 4. Results of Sequential Implementation using embedded memory in XILINX devices

Family	4000	VIRTEX	VIRTEXII
Device	4020XLPQ160-09	V200BG256-6	XC2V80FG256-4
Slices	31	38	39
Maximum Frequency	55,460MHz	92,200MHz	120,337MHz
Maximum size (Device)	128x128	128x128	256x256
Maximum size (Family)	512x512	512x512	1024x1024

¹ Actually the total dimension, if we could use all the memory available, should be a little more, but the design has been thought to have size of $2^n \times 2^n$.

The main drawback of the design is the fact that the time required to perform an evolution increases linearly with the total size of the array. The control unit needs 4 clock cycles to read a 2x2 block from memory and 4 cycles to store them back. Also, the process unit needs one cycle to perform the evolution of the partition, so 9 clock cycles are necessary to evolve each 2x2 block from memory.

2.3 Dual-Port Memories

Evolution rates can be increased by relying on special features of the most powerful families. Concretely, we can get advance of the embedded dual-port RAM that allows the user to perform simultaneous read-and-write operations. This can reduce the number of cycles needed to perform a block evolution from 9 to 5, almost doubling the number of evolutions per second that the system can achieve. But this feature is not supported by all the families under study. Results on the suitable families are given in Table 5 for the 8x8 array.

Even if we rely on most powerful families and take advantage of dual-port embedded memories, sequential implementation leads us to a system that presents poor timing-characteristics as size increases. This effect is due to the fact that memory must be explored and updated sequentially. So, the evolutions per second that the system can achieve decreases with the total size. Figure 3 shows this evolution for the best devices studied.

Table 5. Results of Sequential Implementation using dual-port embedded memory in ALTERA and XILINX devices

Family	APEXII	STRATIX	VIRTEX	VIRTEXII
Device	EP2A25 F672C7	EP1S25 F780C6	V200BG256-6	XC2V80FG256-4
LCs / Slices	91	68	40	40
Maximum Frequency	111,02MHz	187,86MHz	85,690MHz	143,021MHz

3 Concurrent Implementation

In some applications, evolution rates like those presented in the previous paragraph are non acceptable. The main advance of a Cellular Automata Structure is precisely its high parallelisation degree, and the implementation previously proposed converts it into a serial scheme to perform the matrix actualisation. If we want to obtain a real concurrent Cellular Automata, new strategies need to be explored.

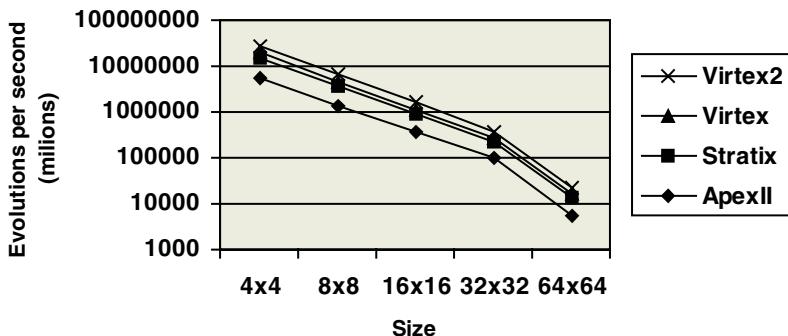


Fig. 3. Effects of the size in number of evolutions per second in sequential implementation

3.1 Proposed Architecture

If we carefully study the connection schema of a Margolus Neighborhood Cellular Automata, easily we can distinguish between four classes of cells, depending on its position into the global matrix. In figure 4 these four classes are shown. Class 1 Cells are updated in odd cycles as Type D cells (referred to the notation introduced in figure 1) from a dotted line block, and are updated in even cycles as Type A Cells from a solid line block. The rest of classes and their connectivities are easily inferred from the figure.

This connection scheme leads us to reduce all classes of Cells to a common structure that is depicted on figure 5. For each cell in the matrix we need to define two different functions: the even one and the odd one. A multiplexer selects between results on even or odd branches depending on the present cycle. Also, to easily supply initial data to the circuit, we have included a second multiplexer for data synchronous load. Finally, an Enable terminal has been added to hold and start the evolution.

If we fix the proposed architecture for all the classes of cells in the matrix, the only difference between classes is the way the neighbor cells are connected to the inputs of the even/odd functions.

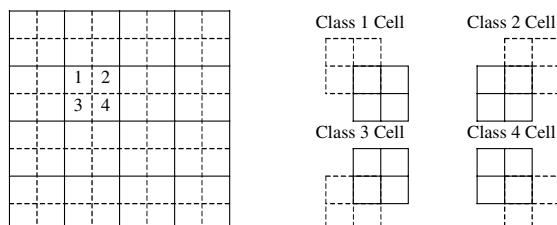


Fig. 4. Classes of cells in Margolus neighborhood. Each class of cells behaves as one type of cell of those introduced in figure 1 different for each even/odd evolution

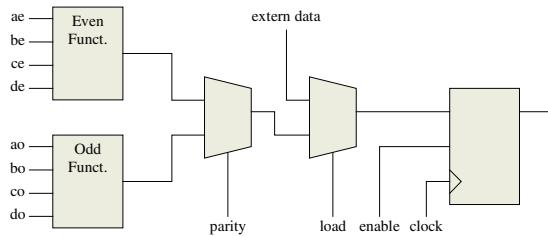


Fig. 5. Common structure of a cell. Each one supports two different functions: the even one and the odd one

3.2 Implementation and Results

This concurrent architecture was implemented on several FPGA families to compare performances. For the logic synthesis we used LeonardoSpectrum Version 2002b.21 from Exemplar Logic on different families from several manufacturers. The results shown here are those obtained for ALTERA and XILINX devices. For the ALTERA devices the implementation and simulation was performed with MAX+PLUSII v. 10.0 (FLEX10K family) and QUARTUS II v. 2.2 (APEX20K, APEXII and STRATIX families). Implementation and simulation on XILINX devices was performed using FOUNDATION SERIES 3.1i.

ALTERA Logic Cell contains a four-input look-up-table that is specially indicated for implementing one of the even/odd functions. So, in every device from ALTERA, the complete Cell showed in figure 5 needs 4 LCs to be implemented: two for the two functions and the rest for the multiplexers and register.

Table 6 summarizes the results of the implementations for a 8x8 matrix in different ALTERA devices. It is important to remark that even though in the synthesis step every device needs 4 LCs per Cell, implementation changes that number, due to the mapping process. In family STRATIX LC is a little more complex, so mapping can be optimized and the number of LCs per Cell is reduced to 3, allowing us to include more Cells in the device. Table 6 also lists the maximum size of the matrix, considering the number of LCs in the device and in the family². Finally, the maximum frequency is listed for the 8x8 design. This value is clearly greater than that obtained in the sequential implementation. But the main advantage accomplished is the way the circuit evolves: it updates the whole matrix in the same clock cycle, independently of the size. So, for any size of the array, the rate of evolutions per second is constant and equal to one clock period.

² In this case, design is not limited to sizes of $2^n \times 2^n$. With this implementation the only limitation, characteristic of Margolus neighborhood, is that size must be the square of an even number.

Table 6. Results of Concurrent Implementation in ALTERA devices

Family	FLEX10K	APEX20K	APEXII	STRATIX
Device	EPF10K20 RC240-3	EP20K200 RC240-1	EP2A25 F672C7	EP1S25 F780C6
LCs (Cell)	4	4	4	4
LCs (Full Design)	257	257	257	193
Maximum Frequency	81,96MHz	152,05MHz	152,07MHz	303,12MHz
Maximum size (de- vice)	16x16	40x40	68x68	92x92
Maximum size (fam- ily)	54x54	100x100	114x114	194x194

Table 7. Results of Concurrent Implementation in XILINX devices

Family	4000	VIRTEX	VIRTEXII
Device	4020XLPQ160-09	V200BG256-6	XC2V80FG256-4
Slices (Cell)	2	2	2
Slices (Full Design)	96	97	97
Maximum Frequency	40,596MHz	103,082MHz	165,618MHz
Maximum size (de- vice)	22x22	39x39	18x18
Maximum size (fam- ily)	74x74	90x90	176x176

Similar results can be obtained for XILINX devices. As we mentioned before, XILINX Slice contains 2 LUTs, so one single Cell requires only 2 Slices to be implemented. However, after the implementation the mapping can be optimized, reducing the number to approximately 1,5 Slices per Cell. Table 7 summarizes these results for XILINX devices. Again, given maximum frequency corresponds to the 8x8 design.

Seeing these results one questions how the total size affects maximum frequency. In practice, we obtain little variation as the total size increases, due to the locality connection of cells in the overall design. This effect can be seen in Figure 6, in which only the most powerful families were used.

3.3 Comparison between Implementations

The two main differences between both strategies are evident from the given results: sequential implementation permits large matrix sizes, thus having the drawback of that the time per evolution increases linearly with the size of the matrix. On the other hand, concurrent implementation is more adequate when time is a critical parameter, even though the sizes obtained are small. This could be indicated in some VLSI applications such as random number generation or BIST.

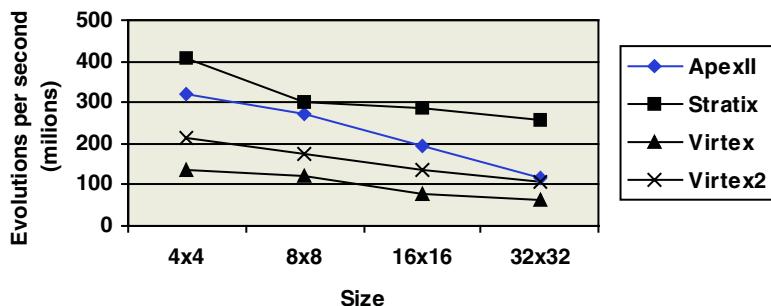


Fig. 6. Effects of the size in number of evolutions per second in the concurrent implementation

4 Conclusions

We have studied Margolus neighborhood Cellular Automata. We propose two implementations for Margolus neighborhood Cellular Automata, the first one RAM based that allows large sizes but offers poor timing characteristics as processing times increases linearly with memory size. This processing time can be reduced to the half using dual-port memories to speed-up data access. The other implementation, that we call concurrent, gives support to much small systems in which time is a critical factor, performing a complete evolution in only one clock cycle.

References

1. Wolfram, S.: Cellular Automata. Los Alamos Science, 9 (1983) 2–21
2. Wolfram, S.: Statistical mechanics of cellular automata. Reviews of Modern Physics, 55 (1983) 601–644
3. Margolus, N.: Physics-Like models of computation. Physica 10D (1984) 81-95
4. Toffoli, T.: Cellular Automata as an alternative to (rather than an approximation of) Differential Equations in Modeling Physics. Physica 10D (1984) 117-127
5. Toffoli, T.: Occam, Turing, von Neumann, Jaynes: How much can you get for how little? (A conceptual introduction to cellular automata). The Interjournal (October 1994)
6. Toffoli, T., Margolus, N.: Invertible cellular automata: a review. Physica D, Nonlinear Phenomena, 45 (1990) 1–3
7. Gruau, F. C., Tromp, J. T.: Cellular Gravity. Parallel Processing Letters, Vol. 10, No. 4 (2000) 383–393
8. Smith, M. A.: Cellular Automata Methods in Mathematical Physics. Ph.D. Thesis. MIT Department of Physics (May 1994).
9. Wolfram, S.: Cryptography with Cellular Automata. Advances in Cryptology: Crypto '85 Proceedings, Lecture Notes in Computer Science, 218 (Springer-Verlag, 1986) 429–432
10. Sarkar, P.: A brief history of cellular automata. ACM Computing Surveys, Vol. 32, Issue 1 (2000) 80–107

11. Shackleford, B., Tanaka, M., Carter, R.J., Snider, G.: FPGA Implementation of Neighborhood-of-Four Cellular Automata Random Number Generators. Proceedings of FPGA 2002 (2002) 106–112
12. Vichniac, G. Y.: Simulating Physics with Cellular Automata. *Physica* 10D (1984) 96–116
13. Popovici, A., Popovici, D.: Cellular Automata in Image Processing. Proceedings of MTNS 2002 (2002)
14. Wolfram, S.: *A New Kind of Science*. Wolfram media (2002)
15. Corno, F., Rebaudengo, M., Reorda, M.S., Squillero, G., and Violante, M.: Low power BIST via hybrid cellular automata. 18th IEEE VLSI Test Symposium, 2000.
16. Corno, F., Reorda, M.S., Squillero, G.: Exploiting the selfish gene algorithm for evolving hardware cellular automata. Proceedings of Congress of Evolutionary Computation (CEC2000).

Fast Modular Division for Application in ECC on Reconfigurable Logic

Alan Daly¹, William Marnane¹, Tim Kerins¹, and Emanuel Popovici²

¹ Dept. of Electrical & Electronic Engineering,
University College Cork, Ireland

{aland,liam,timk}@rennes.ucc.ie

² Dept. of Microelectronic Engineering,
University College Cork, Ireland
e.popovici@ucc.ie

Abstract. Elliptic Curve Public Key Cryptosystems are becoming increasingly popular for use in mobile devices and applications where bandwidth and chip area are limited. They provide much higher levels of security per key length than established public key systems such as RSA. The underlying operation of elliptic curve point multiplication requires modular multiplication, division/inversion and addition/subtraction. Division is by far the most costly operation in terms of speed. This paper proposes a new divider architecture and implementation on FPGA for use in an ECC processor.

1 Introduction

Elliptic Curve Cryptosystems (ECC) were independently proposed in the mid-eighties by Victor Miller [1] and Neil Koblitz [2] as an alternative to existing public key systems such as RSA and DSA. No sub-exponential algorithm is known to solve the discrete logarithm problem on a suitably chosen elliptic curve, meaning that smaller parameters can be used in ECC with equivalent security to other public key systems. It is estimated that an elliptic curve group with 160-bit length has security equivalent to RSA with a bit length of 1024-bit [3].

Two types of finite field are popular for use in elliptic curve public key cryptography: $GF(p)$ with p prime, and $GF(2^n)$ with n a positive integer. Many implementations focus on using the field $GF(2^n)$ due to the underlying arithmetic which is well suited to binary numbers [4]. Addition is simply bitwise XOR in $GF(2^n)$, whereas carry adders must be used when operating in $GF(p)$. However, most $GF(2^n)$ processors are limited to operation on specified curves and key sizes. An FPGA implementation of a $GF(2^n)$ processor which can operate on different key sizes without reconfiguration has previously been presented in [5]. ECC Standards define different elliptic curves and key sizes which ECC implementations must be capable of utilising [6][7]. With a $GF(p)$ processor, any curve or key length up to the maximum size, p , can be used without reconfiguration.

Few implementations of ECC processors over $GF(p)$ have been implemented on hardware to date due to the more complicated arithmetic required[8]. The

modular division operation has been implemented in the past by modular inversion followed by modular multiplication. No implementations to date have implemented a dedicated modular division component in an ECC application.

This paper proposes a modular divider for use in an ECC processor targeted to FPGA which avoids carry chain overflow routing. The design proposed has a bit length of 256-bits and thus would provide security well in excess of RSA-1024 when used in an ECC processor.

2 Elliptic Curve Cryptography over $GF(p)$

An elliptic curve over the finite field $GF(p)$ is defined as the set of points (x, y) , which satisfy the elliptic curve equation

$$y^2 = x^3 + ax + b$$

where x, y, a and b are elements of the field, and $4a^3 + 27b^2 \neq 0$.

To encrypt data, it is represented as a point on the chosen curve over the finite field. The fundamental encryption operation is point scalar multiplication, i.e. a point P_1 is added to itself k times.

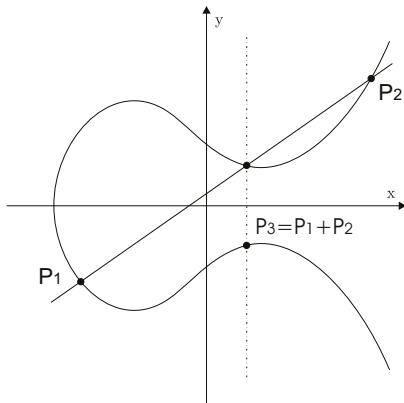
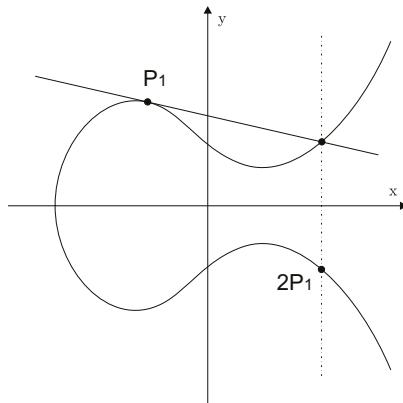
$$\begin{aligned} Q &= kP \\ &= \underbrace{P + P + \dots + P}_{k \text{ times}} \end{aligned}$$

In order to compute kP , a double and add method is used and k is represented in binary form and scanned right to left from LSB to MSB, performing a double at each step and an addition if k_i is 1. Therefore the multiplier will require $(m - 1)$ point doublings and an average of $(\frac{m-1}{2})$ point additions, where m is the bitlength of the field prime, p .

The operations of elliptic curve point addition and point doubling are best explained graphically as shown in Fig.1 and Fig.2. To add two distinct points, P_1 and P_2 , a chord is drawn between them. This chord will intersect the curve at exactly one other point, and the reflection of that point through the x -axis is defined to be the point $P_3 = P_1 + P_2$.

In the case of adding point P_1 to itself (doubling P_1), the tangent to the curve at P_1 is drawn and found to intersect the curve again at exactly one other point. The reflection of this point through the x -axis is defined to be the point $2P_1$. (Note: The *point at infinity*, \mathcal{O} is taken to exist infinitely far on the y -axis, and is the identity of the elliptic curve group.)

Point Multiplication operations on elliptic curves over $GF(p)$ are performed by modulo addition, subtraction, multiplication and inversion. Different coordinate systems can be used to represent elliptic curve points, and it is possible to reduce the number of inversions by representing the points in *projective coordinates*. However, this results in a large increase in the number of Multiplications required per point operation (13 for addition, 6 for doubling) [3][8].

**Fig. 1.** Point Addition**Fig. 2.** Point Doubling

The point addition/doubling formulae in *affine coordinates* are given below. Let $P_1=(x_1,y_1)$ and $P_2=(x_2,y_2)$, then $P_3=(x_3,y_3)=P_1 + P_2$ is given by:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) - y_1 \\ \lambda &= \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2 \end{cases} \end{aligned}$$

The performance of an ECC processor depends on the efficiency of the finite field computations required to perform the point addition and doubling operations. Point addition requires 2 multiplications, 1 division and 6 addition/subtraction operations. Point doubling requires 3 multiplications, 1 division and 7 additions/subtractions. Division is the critical operation and the speed of the overall processor will depend on a high-speed divider.

3 Modular Inversion

One method to perform the division operation is to perform an inversion followed by a multiplication.

The multiplicative inverse of an integer a ($\bmod M$) exists if and only if a and M are relatively prime. There are two methods to calculate this inverse. One is based on Fermat's Theorem which states that $a^{M-1}(\bmod M) = 1$ and therefore $a^{M-2}(\bmod M) = a^{-1} (\bmod M)$. Using this fact, the inverse may be calculated by modular exponentiation. However this is an expensive operation requiring on average $1.5 \log_2 m$ multiplications. Another method to calculate the inverse is by knowing that the greatest common divisor of any two integers may be expressed as a linear combination of the two.

Table 1. Area and speed results for the Inverter designs.

Design	Area (Slices)	% of XCV2000e	Equivalent Gates	Max Freq. (MHz)
64-bit	735	3.8	13,855	45
128-bit	1,259	6.6	24,628	38
256-bit	2,453	12.8	48,226	28

Since a and M are relatively prime, the following expression may be solved for r and s :

$$ar + Ms = 1 \pmod{M}$$

This linear equation implies that:

$$ar \equiv 1 \pmod{M}$$

Therefore, r is the inverse of a (\pmod{M}). The values of r and s may be derived by reversing the steps of the Euclidean algorithm. This procedure is known as the Extended Euclidean algorithm.

Kaliski proposed a variation of the extended Euclidean algorithm to compute the Montgomery Inverse, $a^{-1}2^m \pmod{M}$ of an integer a [9]. The output of this algorithm is in the Montgomery domain, which is useful when performing further Montgomery multiplications [10]. The Montgomery multiplier is an efficient algorithm for computing modular multiplication, however inputs must first be mapped into the Montgomery domain, and then re-mapped before output. Therefore, the Montgomery multiplier is useful only when repeated multiplications are necessary, such as in the RSA cryptosystem where modular exponentiation is the underlying operation. Efficient Montgomery multipliers for implementation on FPGA have previously been presented in [11]. The performance of ECC point addition and doubling could be improved by performing all operations in the Montgomery domain.

Montgomery inverters have previously been presented in [12][13][14][15][16]. Inverter speed and area results based on designs presented in [14] are given in table 1. The architecture requires a maximum of $(3m + 2)$ clock cycles to perform an m -bit Montgomery inversion, or $(4m + 2)$ cycles to perform a regular integer inversion.

4 Modular Division

A new binary add-and-shift algorithm to perform modular division was presented recently by S.C. Shantz in [17]. This provides an alternative to division by inversion followed by multiplication. The algorithm is given here in algorithm 1. It computes $U = \frac{y}{x} \pmod{M}$ in a maximum of $2(m - 1)$ clock cycles where m is the bitlength of the modulus, M .

Algorithm 1 : Modular Division

Input : $y, x \in [1, M - 1]$ and M
Output : U , where $y = U * x \pmod{M}$

```

 $A := x, B := M, U := y, V := 0$ 
while ( $A \neq B$ ) do
    if ( $A$  even) then
         $A := A/2;$ 
        if ( $U$  even) then  $U := U/2$  else  $U := (U + M)/2;$ 
    else if ( $B$  even) then
         $B := B/2;$ 
        if ( $V$  even) then  $V := V/2$  else  $V := (V + M)/2;$ 
    else if ( $A > B$ ) then
         $A := (A - B)/2;$ 
         $U := (U - V);$ 
        if ( $U < 0$ ) then  $U := (U + M);$ 
        if ( $U$  even) then  $U := U/2$  else  $U := (U + M)/2;$ 
    else
         $B := (B - A)/2;$ 
         $V := (V - U);$ 
        if ( $V < 0$ ) then  $V := (V + M);$ 
        if ( $V$  even) then  $V := V/2$  else  $V := (V + M)/2;$ 
    end while
    return  $U;$ 

```

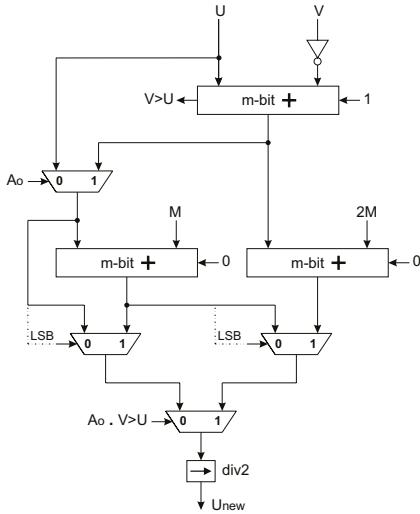
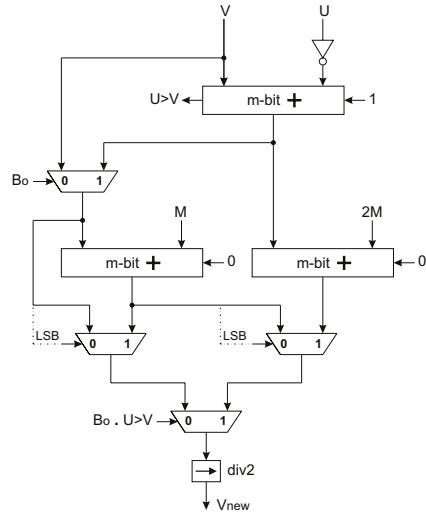
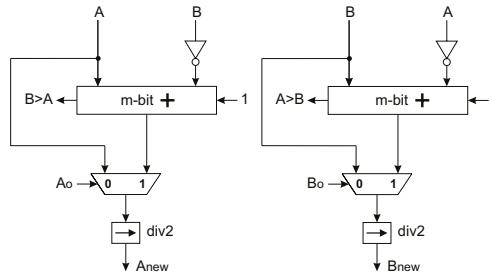
5 Basic Division Architecture

As can be seen from algorithm 1, the divider makes decisions based on the parity and magnitude comparisons of m -bit registers. To determine the parity of a number, only the LSB need be examined (0 implies *even*, 1 implies *odd*). However, true magnitude comparisons can only be achieved through full m -bit subtractions, and thus introduce delay before decisions can be made.

The first design presented here uses m -bit carry propagation adders to perform the additions/subtractions. At each iteration of the while loop in algorithm 1, U_{new} is assigned one of the following values, depending on the parity and relative magnitude of A and B : U , $\frac{U}{2}$, $\frac{(U+M)}{2}$, $\frac{(U-V)}{2}$, $\frac{(U-V+M)}{2}$ or $\frac{(U-V+2M)}{2}$.

The basis for the proposed designs is to calculate all 6 possible values concurrently, and use multiplexors to select the final value of U_{new} . This eliminates the time required to perform a full magnitude comparison of A and B before calculation of U and V can even commence. The same is true for the determination of V_{new} . These architectures are illustrated in Fig.3 and Fig.4.

The architecture for the determination of A_{new} and B_{new} is simpler as illustrated in Fig.5. The U , V , A and B registers are also controlled by the parity and relative magnitudes of A and B , and are not clocked on every cycle.

Fig. 3. Determination of U_{new} Fig. 4. Determination of V_{new} Fig. 5. Determination of A_{new} and B_{new}

6 Proposed (Carry-Select) Division Architecture

The clock speed is dependant on the bitlength of the divider since the carry chain of the adders contributes significantly to the overall critical path of the design. When the carry chain length exceeds the column height of the FPGA, the carry must be routed from the top of the column to the bottom of the next. This causes a significant decrease in the overall clock speed. The carry-select design proposed here halves this adder carry chain at the expense of extra adders and control, but in doing so improves the performance of the divider.

This architecture is similar to a carry-select inverter design proposed in [14]. The values of $(U - V)$ and $(A - B)$ are determined by splitting the m -bit registers U , V , A , and B into $(\frac{m}{2})$ -bit registers U_L , U_H , V_L , V_H , A_L , A_H , B_L and B_H . The values of $(U - V)_L$ and $(U - V)_H$ are then determined concurrently to produce $(U - V)$ as illustrated in Fig.6 and Fig.7.

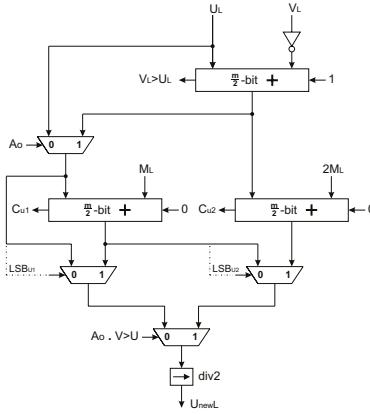


Fig. 6. Carry-Select Architecture: Determination of U_{newL}

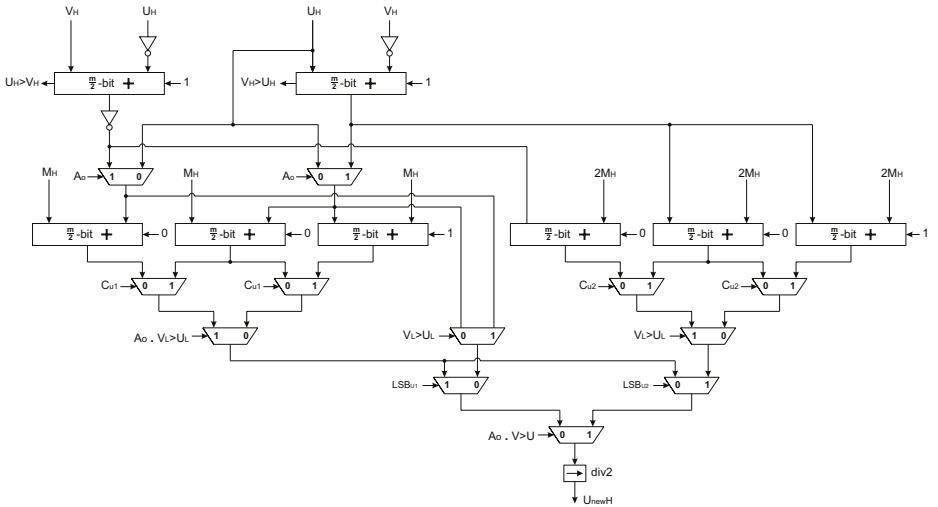


Fig. 7. Carry-Select Architecture: Determination of U_{newH}

When calculating $(U - V)$, if $(V_L > U_L)$, then one extra bit must be “borrowed” from U_H . (i.e. The value of $(U - V)_H$ will actually be $(U_H - V_H - 1)$) It is observed that $(U_H - V_H - 1)$ is actually equal to $(\bar{V}_H - \bar{U}_H)$, and therefore only a bitwise inverter is needed to produce this value as seen in Fig.7. However, it is also possible that a carry from the addition of M_L or $2M_L$ will affect the final value of U_{newH} . Therefore carries of -1, 0 and 1 must be accounted for in the determination of U_{newH} . To allow for this, an extra $(\frac{m}{2})$ -bit adder with a carry-in of 1 is required to calculate $(U_H - V_H + M_H + 1)$.

The determination of V_{new} is similar to that of U_{new} . The values of A_{new} and B_{new} are determined as illustrated in Fig.8.

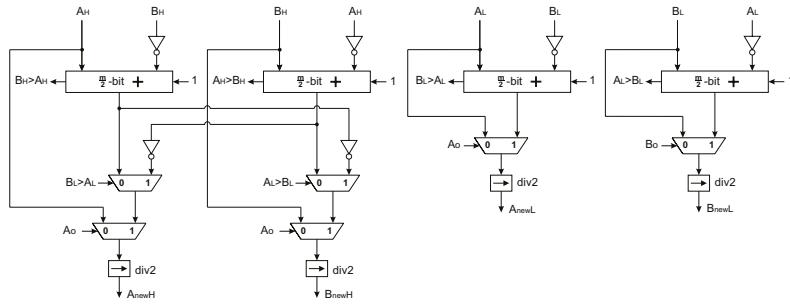


Fig. 8. Carry-Select Architecture: Determination of A_{new} and B_{new}

Table 2 compares the proposed carry-select divider and the basic divider in terms of adder and multiplexor area. The adders and multiplexors in the carry-select divider are half the size of those in the basic design, so overall there is a 50% increase in the size of adders required. The multiplexors are actually instantiated as 4-input Look-Up Tables (LUT's) on the FPGA, and so the 80% increase is not reflected in the actual mapped area results.

7 Results

Speed and area comparisons of 64, 128 and 256-bit dividers for both designs are given in Table 3. The percentage increase in speed and area for each design are given in Table 4. VHDL synthesis and place and route were performed on Xilinx ISE. The results are post place and route with a top level architecture to load the data in 32-bit words. The target FPGA device for this research is the Xilinx Virtex XCV2000e-6bg560 which has 80 CLB's per column. Therefore the maximum unbroken carry chain length is 160 bits.

Little improvement in performance is achieved for the 64 and 128-bit dividers due to increased control logic and multiplexing. However, once the carry chain exceeds the FPGA column height, the basic design suffers a considerable deterioration in clock speed. The proposed carry-select design performs over 50% faster for the 256-bit divider, which is a realistic bit-length for secure ECC communications. A 50% increase in area is also observed, thereby keeping the (time \times area) product similar for both designs.

Table 2. Number of adders/multiplexors required for both designs.

Divider	Adders	2:1 Multiplexors
Basic	$8 \times n\text{-bit}$	$10 \times n\text{-bit}$
Carry-Select	$24 \times \frac{n}{2}\text{-bit}$	$36 \times \frac{n}{2}\text{-bit}$
Area Increase	50 %	80 %

Table 3. Area and speed results for the two designs.

Design	Area (Slices)	% of XCV2000e	Equivalent Gates	Max Freq. (MHz)
64-bit	1,212	6.3	20,858	45
32-bit x 2	1,472	7.7	26,566	45
128-bit	2,215	11.5	39,622	31
64-bit x 2	3,217	16.8	56,236	38
256-bit	3,872	20.2	72,610	17
128-bit x 2	5,849	30.5	104,918	27

Table 4. Percentage increase in area/speed of carry-select design over basic design.

Divider Bitlength	Increase in Area (%)	Increase in Speed (%)
64-bit	21.5 %	0 %
128-bit	45.2 %	22.5 %
256-bit	51 %	58.8 %

Comparing these results to the inversion architecture results presented in Section 3, it is observed that both inverter and divider have almost identical maximum operating clock frequencies.

The divider requires half the number of clock cycles to perform the operation, however it requires significantly more area. It is estimated that using this new architecture, division can be performed twice as fast as by the alternative invert and multiply architecture.

8 Conclusions

Modular division is an important operation in elliptic curve cryptography. In this paper, two new FPGA architectures, based on a recently published division algorithm [17] have been presented and implemented. The basic design computes all possible outcomes from each iteration and uses multiplexors to select the correct answer. This avoids the necessity to await the outcome of a full m -bit magnitude comparison before computation can begin. The second, carry-select divider design splits the critical carry chain into two, and again performs all calculations before the magnitude comparison has been completed. For a 256-bit divider, an improvement in speed of over 50% was achieved with the proposed carry-select divider at a similar area cost over the basic design. The operation speed of the proposed divider is almost identical to that of an inverter, and needs only half the number of clock cycles. Since an additional modular

multiplication is required when using inversion in ECC point multiplication, this division architecture is better suited for implementation in an ECC processor.

Acknowledgement

This work is funded by a research innovation project from Enterprise Ireland.

References

1. V. S. Miller. "Use of Elliptic Curves in Cryptography". *Advances in Cryptography Crypto'85*, (218):417–426, 1985.
2. N. Koblitz. "Elliptic Curve Cryptosystems". *Math Comp*, 48:203–209, 1987.
3. I. Blake, G. Seroussi, and N. Smart. "Elliptic Curves in Cryptography". London Mathematical Society Lecture Note Series 265. Cambridge University Press, 2000.
4. M. Ernst, M. Jung, F. Madlener, S. Huss, and R. Blümel. "A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over $GF(2^n)$ ". *Cryptographic Hardware and Embedded Systems - CHES 2002*, (LNCS 2523):381–398, Aug 2002.
5. T. Kerins, E. Popovici, W. Marnane, and P. Fitzpatrick. "Fully Parameterizable Elliptic Curve Cryptography Processor over $GF(2^m)$ ". *12th Intl Conference on Field-Programmable Logic and Applications FPL2002*, pages 750–759, Sept 2002.
6. IEEE Standards Department. IEEE 1363/D13 Standard Specifications for Public Key Cryptography, 2000.
7. ANSI X9.62. Public Key Cryptography for the Financial Services Industry. The Elliptic Curve Digital Signature Algorithm (ECDSA), 1999.
8. G. Orlando and C. Paar. "A Scalable $GF(p)$ Elliptic Curve Processor Architecture for Programmable Hardware". *Cryptographic Hardware and Embedded Systems - CHES 2001*, (LNCS 2162):348–363, May 2001.
9. B. S. Kaliski Jr. "The Montgomery Inverse and it's applications". *IEEE Trans. on Computers*, 44(8):1064–1065, Aug 1995.
10. P. L. Montgomery. "Modular Multiplication without Trial Division". *Math Computation*, 44:519–521, 1985.
11. A. Daly and W. Marnane. "Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic". *10th Intl Symposium on FPGA (FPGA 2002)*, pages 40–49, Feb 2002.
12. A. Gutub, A. F. Tenca, and C. K. Koc. "Scalable VLSI Architecture for $GF(p)$ Montgomery Modular Inverse Computation". *IEEE Computer Society Annual Symposium on VLSI*, pages 53–58, April 2002.
13. A. Gutub, A. F. Tenca, E. Savas, and C. K. Koc. "Scalable and unified hardware to compute Montgomery inverse in $GF(p)$ and $GF(2^n)$ ". *Cryptographic Hardware and Embedded Systems - CHES 2002*, (LNCS 2523):484–499, Aug 2002.
14. A. Daly, W. Marnane, and E. Popovici. "Fast Modular Inversion in the Montgomery Domain on Reconfigurable Logic". *Irish Signals and Systems Conference ISSC2003*, To Appear : July 2003.
15. E. Savas and C. K. Koc. "The Montgomery Modular Inverse - Revisited". *IEEE Trans. on Computers*, 49(7):763–766, July 2000.
16. T. Kobayashi and H. Morita. "Fast Modular Inversion Algorithm to Match any Operation Unit". *IEICE Trans. Fundamentals*, E82-A(5):733–740, May 1999.
17. S. C. Shantz. "From Euclid's GCD to Montgomery Multiplication to the Great Divide". Technical Report TR-2001-95, Sun Microsystems Laboratories, 2001.

Non-uniform Segmentation for Hardware Function Evaluation

Dong-U Lee¹, Wayne Luk¹, John Villasenor², and Peter Y.K. Cheung³

¹ Department of Computing, Imperial College, London, UK
`{dong.lee,wl}@ic.ac.uk`

² Electrical Engineering Department, University of California, Los Angeles, USA
`villa@icsl.ucla.edu`

³ Department of EEE, Imperial College, London, UK
`p.cheung@ic.ac.uk`

Abstract. This paper presents a method for evaluating functions in hardware based on polynomial approximation with non-uniform segments. The novel use of non-uniform segments enables us to approximate non-linear regions of a function particularly well. The appropriate segment address for a given function can be rapidly calculated in run time by a simple combinational circuit. Scaling factors are used to deal with large polynomial coefficients and to trade precision with range. Our function evaluator is based on first-order polynomials, and is suitable for applications requiring high performance with small area, at the expense of accuracy. The proposed method is illustrated using two functions, $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$, which have been used in Gaussian noise generation.

1 Introduction

The evaluation of functions is often the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\ln(x)$ or \sqrt{x} , and compound functions such as $\sqrt{-\ln(x)}$ or $\tan^2(x) + 1$. Computing these functions quickly and accurately is a major goal in computer arithmetic; software implementations are often too slow for numerically intensive or real-time applications. The performance of such applications depends on the design of a hardware function evaluator. Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. The principal contribution of this paper is a fast and efficient hardware function evaluator using polynomial approximations. The key novelties of our work include:

- a method for polynomial approximations with non-uniform segments;
- hardware architecture and implementation of the proposed method;
- evaluation of this method with a logarithmic function and a cosine function.

The rest of this paper is organized as follows. Section 2 covers background material and previous work. Section 3 explains our segmentation technique. Section 4 describes the hardware architecture. Section 5 presents a method for determining the placement of segment boundaries. Section 6 discusses evaluation and results, and Section 7 offers conclusion and future work.

2 Background

Polynomial approximation [13], [14] involves approximating a continuous function f with one or more polynomials p of degree n on a closed interval $[a, b]$. The aim is to minimize a distance $\|p - f\|$. There are two kinds of approximations: least squares approximations that minimize the average error, and least maximum approximations that minimize the worst-case error [15]. In both cases, the aim is to minimize a distance $\|p - f\|$. For least squares approximations, that distance is:

$$\|p - f\|_2 = \sqrt{\int_a^b w(x)(f(x) - p(x))^2 dx}, \quad (1)$$

where w is a continuous weight function for selecting parts of $[a, b]$ where we want the approximation to be more accurate. For least maximum (minimax) approximations, the distance is:

$$\|p - f\|_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|. \quad (2)$$

Our work is based on minimax polynomial approximations, which involve minimizing the worst-case error. Since we are interested in fixed-point number representation in our work, we will be concerned with the worst-case absolute errors. A recent study of minimax polynomial approximation on FPGAs can be found [21].

Much of the work on function evaluation is generally concerned with producing highly accurate approximation with complex designs. Instead, we will focus on applications that require very high speed and small area but not high accuracy. Examples of such applications include Gaussian noise generation [3] and belief propagation in LDPC decoding [20]. We will focus in this paper on first-order polynomials of the form $p(x) = c_1 \times x + c_0$, where c_1 is the gradient and c_0 is the y-intercept, which can be computed by two table lookups, a multiplication and an addition.

Previous work on polynomial approximations involves equally sized segments [4], [5], [6], [7], [8], [9], [10], [11], [12]. Approximations using such uniform segments are suitable for functions with linear regions, but they can be inefficient for non-linear functions. It is desirable to choose the boundaries of the segments to cater for the nonlinearities of the function. Highly non-linear regions may need smaller segments than linear regions. This approach minimizes the amount of storage required to approximate the function, leading to more compact and efficient designs.

3 Function Evaluation Based on Non-uniform Segmentation

The interval of approximation $[a, b]$ is divided into a set of sub-intervals, called segments. The best-fit straight line, in a minimax sense, to each segment is found. A lookup table is used to store the coefficients for each line segment, and the functions can then be evaluated using a multiplier and an adder to calculate the linear approximation [1].

Using well-known methods that compute elementary functions such as CORDIC [2], the evaluation of compound functions is a multi-stage process. Consider the evaluation

of the function $\sqrt{-\ln(x)}$ over the interval $(0, 1]$. Using CORDIC, the computation of this function is a two-stage process: the logarithm of x followed by the square root. With our approach, we look at the entire function over the given domain, and therefore we do not need to have two stages.

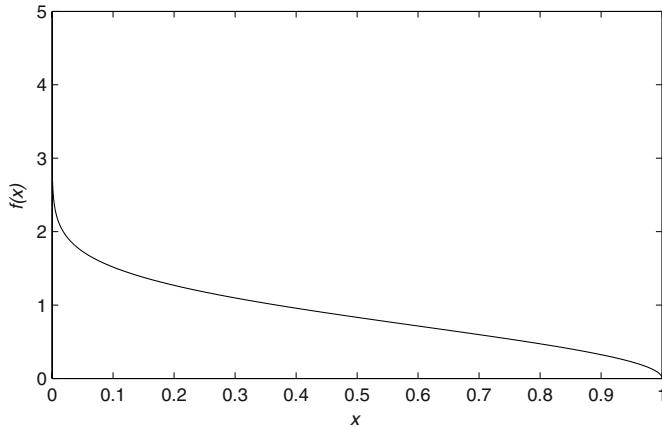


Fig. 1. $\sqrt{-\ln(x)}$ over $(0, 1]$.

As shown in Figure 1, the greatest non-linearities of the function $\sqrt{-\ln(x)}$ occur in the regions close to zero and one. If uniform segments are used, a large number of small segments would be required to get accurate approximations in the non-linear regions. However, in the middle part of the curve where it is relatively linear, accurate approximation can be obtained using relatively few segments. It would be efficient to use small segments for the non-linear regions, and large segments for linear regions. Arbitrary-sized segments would enable us to have the least error for a given number of segments; however, the hardware to calculate the segment address for a given input can be complex. Our objective is to provide near arbitrary-sized segments with a simple circuit to find the segment address for a given input.

We have developed a novel method which can construct piecewise linear approximation such that: (a) the segment lengths used in a given region depends on the local linearity, with more segments deployed for regions of higher non-linearity; and (b) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed. The proposed method consists of five steps.

1. Determine optimal placement of segment boundaries (see Section 5) – this would include dividing into regions such that in each region the function either monotonically increases or decreases.
2. For a non-linear region, if the non-linearity is monotonically increasing, then increase segment size by a factor of two or more at each step; if the non-linearity is

monotonically decreasing, then reduce segment size by a factor of two or more at each step.

3. The segment addresses can be obtained by computing the prefixes [16] with a simple combinational or pipelined circuit.
4. If necessary, divide the function into several intervals, then apply step 1–3 (see the function $\cos(2\pi, x)$ in Section 6).
5. If necessary, repeat the above steps with higher-order terms.

As an example to illustrate our approach, consider approximating $\sqrt{-\ln(x)}$ with an 8-bit input (Figure 1). Using the traditional approach, the most-significant bits of x are used to index the uniform segments. For instance if the most-significant four bits are used, 16 uniform segments are used to approximate the function. Using our approach, it is possible to use small segments for non-linear regions (regions near 0 and 1), and large segments for linear regions (regions around 0.5). The idea is to use segments that grow by a factor of two from 0 to 0.5, and segments that shrink by a factor of two from 0.5 to 1 in the x -axis of Figure 1. We use segment boundaries at locations 2^{n-8} and $1 - 2^{-n}$ where $0 \leq n < 8$. Up to 14 segments can be formed this way. A circuit based on prefix computation can be used for calculating segment addresses (Figure 2) for a given input x . It checks the number of leading zeros and ones to work out the segment address. A cascade of OR gates is used for segments that grow by factors of two, and a cascade of AND gates is used for segments that shrink by factors of two; these circuits can be pipelined and a circuit with shorter critical path but requiring more area can be used [16]. Note that the choice of segments does not have to be factors of two, it could be more. The appropriate taps are taken from the cascades depending on the choice of the segments and are added to work out the segment address. In Figure 2, the maximum available taps are taken, giving 14 segment addresses. Some taps would not be taken if the segments grow or shrink by more than a factor of two. It can be seen that the critical path of this circuit is the path from x_6 or x_7 to the output of the adder. By introducing pipeline registers between the gates, higher throughput can be easily achieved.

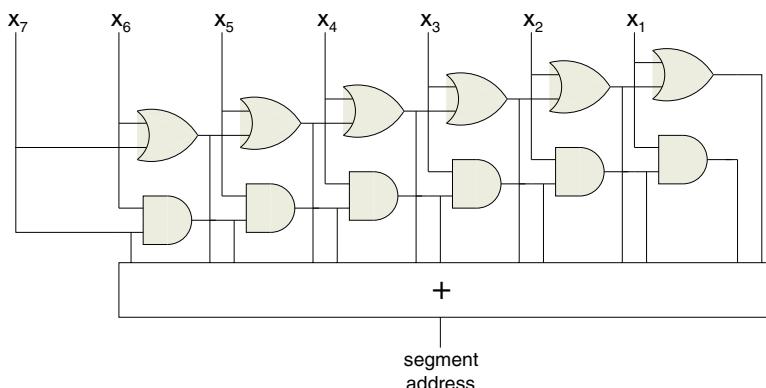


Fig. 2. Circuit to calculate the segment address for a given input x . The adder counts the number of ones in the output of the two prefix circuits.

When approximating $\sqrt{-\ln(x)}$ with 32-bit inputs based on polynomials of the form $p(x) = c_1 \times x + c_0$, the gradient of the steepest part of the curve is in the order of 10^8 , thus large multipliers would be required. To overcome this problem, we use scaling factors of multiples of two to reduce the magnitude of the gradient, essentially trading precision for range. This is appropriate since the larger the gradient, the less important precision becomes. The use of scaling factors provides the user the ability to control the precision for both c_1 and c_0 , resulting in variation of the size of the multiplier and adder. Hence for each segment four coefficients are stored: c_1 and its scaling factor, c_0 and its scaling factor.

It is also possible to divide the input interval into uniform or non-uniform intervals, and have uniform or non-uniform segments inside each interval. In this case, the most-significant bits are used to address the intervals, and the least-significant bits are used to address the segments inside each interval. It can be seen that one can have any number of nested combinations of uniform and non-uniform segments. This hybrid combination of nested uniform and non-uniform segments provides a flexible way to choose the segment boundaries. Currently, this segmentation step is done by hand, which is slow and far from optimal. A possible approach to automate this step is discussed in Section 5.

4 Hardware Architecture

The architecture of our function evaluator shown in Figure 3 is based on polynomials of the form $p(x) = c_1 \times x + c_0$. The most-significant bits are used to select the interval, and the least-significant bits are passed through the segment address calculator which calculates the segment address within the interval. The design shown is developed for the common cases, and has been used in the examples of this paper. For other cases, one could divide the input bits into more than two parts and apply the segment address calculation depending on whether the parts use uniform or non-uniform segments.

The ROM outputs the four coefficients for the chosen interval and segment. c_1 is multiplied by the input x and c_s_1 is used to scale the output. The scaling circuit involves shifters, which increase or decrease the value by powers of two. This scaled multiplication value is added to the scaled c_0 coefficient to produce the final result.

For high throughput applications, the segment address calculator, the multiplier and the adder can be pipelined. For typical applications targeting FPGAs, the ROM would be small and could be implemented on-chip using distributed RAM or block RAM. Often the multiplier would be the part taking up a significant portion of the area. Therefore it is important to minimize the multiplier size by finding out the minimum bit width for the coefficient c_1 . Also recent FPGAs, such as Xilinx Virtex-II devices, provide dedicated hardware resources for multiplication which can benefit the proposed architecture.

5 Placement of Segment Boundaries

Let f be a continuous function on $[a, b]$, and let an integer $m \geq 2$ specify the number of contiguous intervals into which $[a, b]$ has been partitioned: $a = u_0 \leq u_1 \leq \dots \leq u_m = b$. Let n_i and $d_i (i = 1, \dots, m)$ be non-negative integers and let P_i denote the set

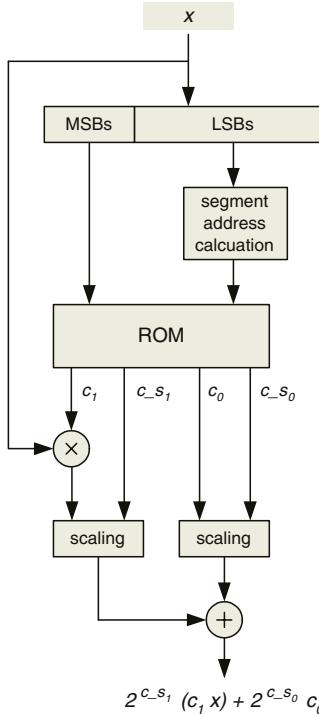


Fig. 3. Our function evaluator architecture.

of rational functions p_i whose numerators and denominators are polynomials of degrees less or equal to n_i and d_i , respectively. For $i = 1, \dots, m$, define

$$h_i(u_{i-1}, u_i) = \min_{p_i \in P_i} \max_{u_{i-1} \leq x \leq u_i} |f(x) - p_i(x)|. \quad (3)$$

Let $\mu = \mu(u) = \max_{1 \leq i \leq m} h_i(u_{i-1}, u_i)$. Lawson states in his paper [18] that the segmented rational minimax approximation problem is that of minimizing μ over all partitions u of $[a, b]$. It can be shown that if the error norm is a non-decreasing function of the length of the interval of approximation, that the function to be approximated is continuous and that the goal is to minimize the maximum error norm on each interval, then a balanced error solution is optimal; the term “balanced error” means that the error norms on each interval are equal.

Pavlidis and Maika present an iterative scheme for segmentation in their paper [19] which results in a suboptimal balanced error solution. The scheme is based on an iteration of the form

$$u_m^{k+1} = u_m^k + c(e_{m+1}^k - e_m^k), \quad m = 1, \dots, n-1. \quad (4)$$

Here u_m^k is the value of the m -th point and the k -th iteration, e_m^k is the error on $(u_{m-1}^k, u_m^k]$ and c is an appropriate small positive number. It can be shown that for sufficiently small

c the scheme converges to a solution [19]. In this algorithm, the number of segments is fixed and this determines the maximum error. However in many cases, it may be more useful to fix the accuracy desired and let the number of segments vary. Starting from *a* or *b* one could apply polynomial approximation in small increments, until the desired accuracy is reached. Then start a new segment from that point.

Once the segment boundaries have been found by using one of the two approaches above, the next step is to match the boundaries based on our addressing scheme as close to the suboptimum ones as possible. As discussed in Section 3, our addressing scheme is based on nested uniform and non-uniform segments. By carefully using these combinations of segments, it is possible to get a close approximation to the suboptimum segment boundaries. Our aim is to enable the user to input constraints such as maximum error norm and to apply the segmentation automatically to produce lookup tables and the corresponding circuits such as the one shown in Figure 3. A possible approach of such an automated method is shown in Figure 4.

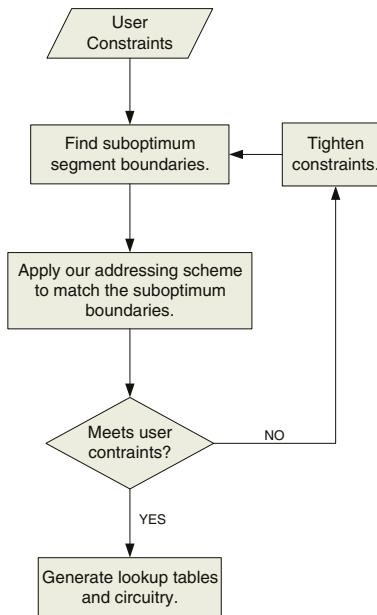


Fig. 4. Steps for automating segmentation.

6 Evaluation and Results

Our function evaluator has been successfully implemented for the Gaussian noise generator presented in [3]. Three functions are approximated: $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and

$\sin(2\pi x)$ over $[0, 1]$. 32-bit inputs are used for $\sqrt{-\ln(x)}$ and 16-bit inputs are used for $\cos(2\pi x)$ and $\sin(2\pi x)$.

We first consider the function $\sqrt{-\ln(x)}$. As stated earlier, the greatest non-linearities of this function occur in the regions close to zero and one. To be consistent with the change in linearity, we use line segment locations to boundaries at locations 2^{n-32} for $0 < x \leq 0.5$, and $1 - 2^{-n}$ for $0.5 < x \leq 1$, where $0 \leq n < 32$. A total of 59 segments are used to approximate this function as shown in Figure 5. Since $\sqrt{-\ln(x)}$ approaches infinity for x values close to zero, the smallest x value is $1/2^{32}$, resulting in a maximum output value of around 4.7.

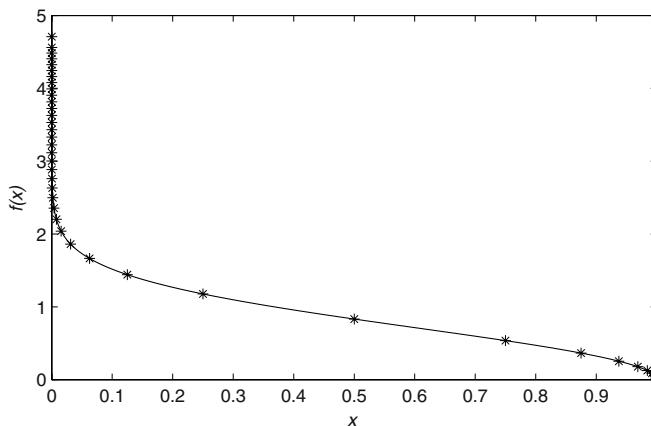


Fig. 5. The segments used to approximate $\sqrt{-\ln(x)}$ with 32-bit inputs. The asterisks indicate the segment boundaries of the linear approximations.

The maximum absolute error of this approximation is 0.020. However this is the case only if we have infinite precision for the coefficients, which is not realistic. Multipliers take significant amount of resources on FPGAs, therefore the coefficients for the gradient should be as small as possible. Tests are carried out to find the optimum number of bits for the gradient coefficients that provides the least absolute error. Figure 6 shows how the maximum absolute error varies with the number of bits used for the gradient of $\sqrt{-\ln(x)}$. The figure indicates that six bits are sufficient to give a maximum absolute error of 0.031. The approximation should differ from the true value by less than one unit in the last place (ulp) [17]; the least significant bit of the fraction of a number in its standard representation is defined to be the last place. With this error, it is sufficient to give an output accuracy of eight bits (three bits for integer and five for fraction). If uniform segments are used, small segment size would be needed in order to cope with the highly non-linear parts of the curve. In fact, one would require around 617 million segments to get the same maximum absolute error with uniform segments. This is a good example to demonstrate the effectiveness of our non-uniform approach. It is clear that our approach works well especially for functions with exponential behavior.

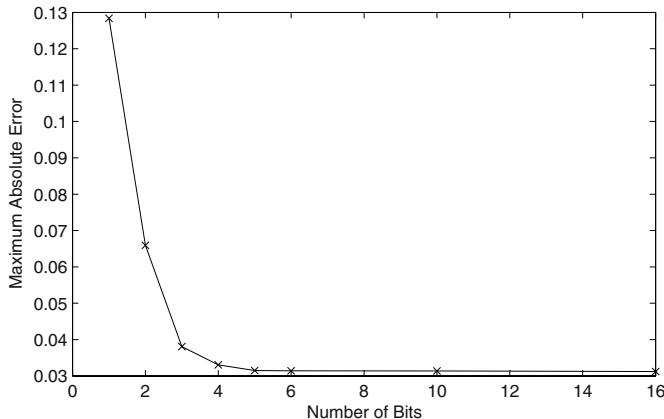


Fig. 6. Variation of function approximation error with number of bits for the gradient of $\sqrt{-\ln(x)}$.

To evaluate the functions $\cos(2\pi x)$ and $\sin(2\pi x)$, due to the symmetry of the sine and cosine functions, only the input range $[0, 1/4]$ for $\cos(2\pi x)$ needs to be approximated [15]. The specific axis-partitioning technique for $\sqrt{-\ln(x)}$ is unsuitable for $\cos(2\pi x)$, since the non-linearities of the two functions are different. If the same technique is used, there would be many unnecessary segments near the beginning and end of the curve, and not enough segments in the middle regions. As before we consider both the local linearity of the curve, and the computational concerns with respect to choosing specific segment boundary locations, leading to the approximations shown in Figure 7. The curve is divided into four uniform intervals and within each interval, non-uniform segmentation is applied. Note that for each interval, not all taps are taken from the segment address calculator. We use a total of 21 segments to approximate this function.

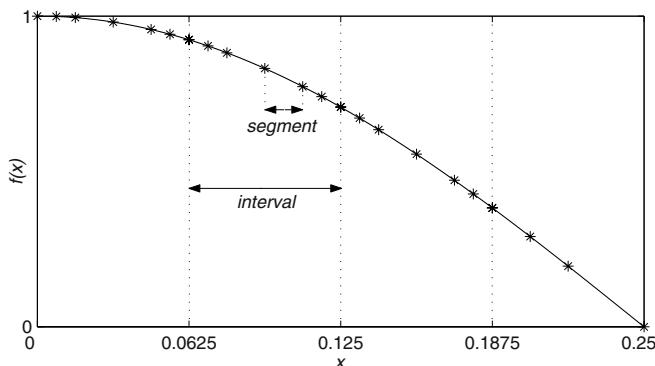


Fig. 7. Approximation for $\cos(2\pi x)$ over $[0, 1/4]$. The asterisks indicate the segment boundaries of the linear approximations.

With finite precision on the coefficients, the maximum absolute error of this approximation is 0.0035, which is sufficient to give an output accuracy of eight bits (all eight bits for fraction). Using uniform segments, the same error can be obtained with a slightly larger number of segments; this is because the curve does not have high non-linearities.

Table 1 shows a comparison of the number of segments for the two functions for non-uniform and uniform segmentation in order to achieve the same worst-case error. Note that for uniform segmentation, the number of segments needs to be a power of two. This is because the most-significant n bits are used for addressing. For instance, the actual number of uniform segments needed for the $\sqrt{-\ln(x)}$ function is 617 million, but 1 billion segments are used which is the next power of two (2^{30}). We do not have this kind of a restriction with our non-uniform addressing scheme. The table also shows the number of bits used for each coefficient in the look-up tables. The lookup tables for the three functions $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$ have a total size of just 3504 bits. With such small lookup table size, all the coefficients can be stored on-chip for fast access.

Table 1. Second column shows the comparison of the number of segments for non-uniform and uniform segmentation. Third column shows number of bits used for the coefficients to approximate the $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$ functions.

function	non-uniform	uniform	c_1	c_{-s_1}	c_0	c_{-s_0}
$\sqrt{-\ln(x)}$	59	1 billion	6	5	32	5
$\cos(2\pi x)$	21	32	8	4	16	4

The function evaluators for the three functions are written using the Handel-C hardware compiler from Celoxica [25], and are mapped and tested on a Xilinx Virtex-II XC2V4000-6 device [24]. The design occupies 1864 slices, four block multipliers and two block RAMs, and takes up around 7% of the device. A fully pipelined version of our design operates at 133 MHz with a latency of 14 clock cycles, and the function evaluators are capable of 133 million operations per second; the completion time for each input is given by $14 / 133$ million = 105 ns. The design has also been implemented on a low cost Xilinx Spartan-IIIE XC2S300E-7, which occupies 70% of the chip and is capable of 62 million operations per second. Our hardware implementations have been compared with software implementations (Table 2). The Virtex-based FPGA implementation is 158 times faster than the Athlon-based PC in terms of throughput, and 11 times faster in terms of completion time.

Well-known function evaluation methods, such as SBTM [5], [6], deal with the approximation of elementary functions over a fixed input range where the function is linear. Range reduction techniques such as those presented in [22] and [23] are used to bring the input within the linear range. However, range reduction is not possible for most compound functions. Our approach caters for both non-linear and linear regions, which makes it suitable for both elementary and compound functions. Currently, our approach tends to produce small lookup table sizes with low accuracy; we hope to improve accuracy by further work on automatic segmentation.

Table 2. Performance comparison: computation of $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$. All PCs are equipped with 512MB DDR RAM. The XC2V4000-6 FPGA belongs to the Xilinx Virtex-II family, while the XC2S300E-7 belongs to the Xilinx Spartan-II-E family. The software implementations are written in C generating single precision floating point numbers, and are compiled with the GCC 3.3 compiler [26].

platform	clock speed (MHz)	latency (clock cycles)	area (slices)	throughput (operations / second)	completion time (ns)
XC2V4000-6 FPGA	133	14	1864	133 million	105
XC2S300E-7 FPGA	62	14	2129	62 million	226
AMD Athlon PC	1400	-	-	0.84 million	1187
Intel Pentium 4 PC	2400	-	-	0.79 million	1261

7 Conclusion

This paper presents a novel method for evaluating functions using polynomial approximations by employing non-uniform segments. The non-uniform segments deal with the non-linearities of functions which occur frequently. A simple cascade of AND and OR gates can be used to rapidly calculate the segment address for a given input. Scaling factors are used to deal with large polynomial coefficients, trading precision with range. Two functions developed for the generation of Gaussian noise are used as examples to illustrate and to evaluate our approach. Results show the advantages of using non-uniform segments over uniform ones. Current and future work includes automating the selection of boundaries, and exploring the use of higher order polynomials for more accurate approximations. This would enable us to apply our approach to a wide range of functions and to obtain detailed comparison with other methods. We will also look at how our function evaluator can be used to speed up addition and subtraction functions in logarithmic number systems [12], which are highly non-linear functions.

Acknowledgment

The authors thank Jun Jiang and Shay Ping Seng for their assistance. The support of Celoxica Limited, Xilinx Inc., the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 55931 and GR/R 31409), and the U.S. Office of Naval Research is gratefully acknowledged.

References

1. O. Mencer, N. Boullis, W. Luk and H. Styles, "Parameterized function evaluation for FPGAs", *Field-Programmable Logic and Applications*, LNCS 2147, pp. 544–554, 2001.
2. J.E. Volder, "The CORDIC trigonometric computing technique", *IEEE Trans. on Elec. Comput.*, vol. EC-8, no. 3, pp. 330–334, 1959.
3. D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "A hardware Gaussian noise generator for channel code evaluation", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, 2003.

4. D. Das Sarma and D.W. Matula, “Faithful bipartite rom reciprocal tables”, *Proc. 12th IEEE Symp. on Comput. Arith.*, pp. 17–28, 1995.
5. M.J. Schulte and J.E. Stine, “Symmetric bipartite tables for accurate function approximation”, *Proc. 13th IEEE Symp. on Comput. Arith.*, vol. 48, no. 9, pp. 175–183, 1997.
6. M.J. Schulte and J.E. Stine, “Approximating elementary functions with symmetric bipartite tables”, *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 842–847, 1999.
7. J.A Pineiro, J.D. Bruguera and J.M. Muller, “Faithful powering computation using table look-up and a fused accumulation tree”, *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
8. J. Cao, B.W.Y. We and J. Cheng, “ High-performance architectures for elementary function generation”, *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
9. V.K. Jain, S.A. Wadecar and L. Lin, “A universal nonlinear component and its application to WSI”, *IEEE Trans. Components, Hybrids and Manufacturing Tech.*, vol. 16, no. 7, pp. 656–664, 1993.
10. H. Hassler and N. Takagi, “Function evaluation by table look-up and addition”, *Proc. of the IEEE 12th Symp. on Comp. Arith.*, pp. 10–16, 1995.
11. J. Detrey and F. de Dinechin, “Multipartite tables in JBits for the evaluation of functions on FPGAs”, *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, 2002.
12. D.M. Lewis, “Interleaved memory function interpolators with application to an accurate LNS arithmetic unit”, *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974–982, 1994.
13. J.F. Hart et al., *Computer Approximations*, Wiley, 1968.
14. J.R. Rice, *The Approximation of Functions*, vol. 1,2, Addison-Wesley, 1964, 1969.
15. J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.
16. R.E. Ladner and M.J. Fischer, “Parallel prefix computation”, *JACM*, vol. 27, no. 4, pp. 831–838, 1980.
17. I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.
18. C.L. Lawson, “Characteristic properties of the segmented rational minimax approximation problem”, *Numer. Math.*, vol. 6, pp. 293–301, 1964.
19. T. Pavlidis and A.P Maika, “Uniform piecewise polynomial approximation with variable joints”, *Journal of Approximation Theory*, vol. 12, pp. 61–69, 1974.
20. C. Jones, E. Vallés, C. Wang, M. Smith, R. Wesel and J. Villasenor, “High throughput Monte Carlo simulation for error floor testing in capacity achieving channel codes”, *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, 2003.
21. N. Sidahao, G.A. Constantinides and P.Y.K. Cheung, “Architectures for function evaluation on FPGAs”, *Proc. IEEE Int. Symp. on Circ. and Syst.*, 2003.
22. J.S. Walther, “A unified algorithm for elementary functions”, *Proc. Spring Joint Comput. Conf.*, 1971.
23. N.W. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, 1980.
24. Xilinx Inc., *Virtex-II User Guide v1.5*, 2002.
25. Celoxica Limited, *Handel-C Language Reference Manual*, ver. 3.1, document no. RM-1003-3.0, 2002.
26. GNU Project, *GCC 3.3 Manual*, <http://gcc.gnu.org>, 2003.

A Dual-Path Logarithmic Number System Addition/Subtraction Scheme for FPGA

Barry Lee and Neil Burgess

Cardiff school of Engineering, Cardiff University, Queen's Buildings,
The Parade, Cardiff. CF24 3TF U.K.
{Leebr2,BurgessN}@cf.ac.uk

Abstract. A new architecture for calculating the addition/subtraction function required in a logarithmic number system (LNS) is presented. A substantial logic saving over previous works is illustrated along with similarities with the dual-path floating-point addition method. The new architecture constrains the lookups to be of fractional width and uses shifting to achieve this. Instead of calculating the function $\log_2(1 \pm 2^{M-K})$ in two lookups the function arithmetic is performed (i.e. the two functions 2^{M-K} and $\log_2(\cdot)$, plus a correction function) as this allows logic sharing that maps well to FPGA. Better-than-floating-point (BTFP) accuracy is used to enable a future comparison with floating-point.

1 Introduction

The logarithmic number system (LNS) is a number system based on fixed-point arithmetic and has a high dynamic range comparable to floating-point. The LNS has the benefits of simplified multiplication, division and powering (including: square root, cube, square and reciprocal square root to name a few special cases) but at the cost of a complex addition/subtraction function. This paper describes the design of an FPGA core to evaluate the addition/subtraction function as a basis to efficiently port the LNS benefits to FPGA. The function is calculated with an error that is equivalent to floating-point to enable a fair comparison of the two systems (not covered in this work).

The paper is organised as follows. Section 2 introduces the LNS and gives some background details. Section 3 outlines the function partition method adopted. Section 4 describes the reasoning behind the BTFP accuracy. Section 5 gives details of the FPGA implementation including the function approximation methods and macro structures. Section 6 presents the results and finally section 7 concludes.

2 LNS Background

A base-2 LNS number is represented by the couple $\langle s_a, e_a \rangle$ where s_a is a 1-bit sign and e_a is an n -bit fixed-point number. Typically e_a is of two's complement form with integer (I) and fractional (F) sections as shown in figure 1. The real value R_v of an LNS number is given as,

$$R_v = (-1)^{s_a} \times 2^{e_a}. \quad (1)$$

This can be considered as a floating-point value that has an exponent with fractional precision and a significand of 1. A bias can be added to e_a to simplify comparisons similarly to the bias added to a floating-point exponent.

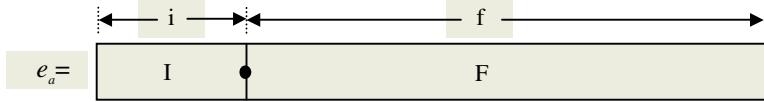


Fig. 1. The integer and fractional sections of the magnitude e_a of an LNS number

Consider two logarithmic values K and M, equations (2) and (3) respectively. The equivalent LNS operators to the real operators of multiplication, division, powering and addition are shown in (4), (5), (6) and (7) respectively.

$$K = \log_2(X). \quad (2)$$

$$M = \log_2(Y). \quad (3)$$

$$\log_2(X \times Y) = K + M. \quad (4)$$

$$\log_2(X / Y) = K - M. \quad (5)$$

$$\log_2(X^P) = X \times P. \quad (6)$$

$$\log_2(X \pm Y) = K + \log_2(1 \pm 2^{M-K}), \text{ where, } K \geq M. \quad (7)$$

Multiplication and division are fixed-point addition and subtraction operations with only simple overflow detection overheads and are exact operations. Powering is fixed-point multiplication, which can be considered as shifting for square root and squaring. The addition/subtraction function, a graph of which is shown in figure 2, is somewhat more complicated to perform. From figure 2 it can be seen that the addition function is well behaved and has a small range of (0,1]. Approximating the addition function is straight forward and a piecewise polynomial approach has been used in previous works [1] and [2]. The subtraction function is not so simple to approximate, which is primarily due to the singularity where the function value tends to $-\infty$ as the argument tends to 0. The region where the function tends to minus infinity is the lookup region required to calculate the difference of two numbers that are very similar and where catastrophic cancellation occurs (for floating-point). The difference of two very similar numbers produces a very small result and in the LNS this is represented by a very large (in magnitude) negative value.

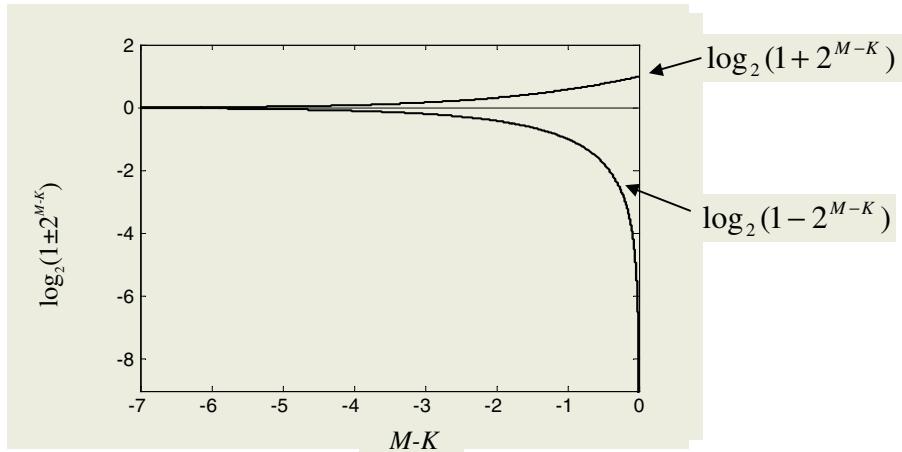


Fig. 2. A graph of the functions to be computed to enable LNS addition and subtraction

3 Function Partitioning

To arrive at the method implemented in this work the addition/subtraction function was first split into three sections. The three sections A, B and C are shown in figure 3.

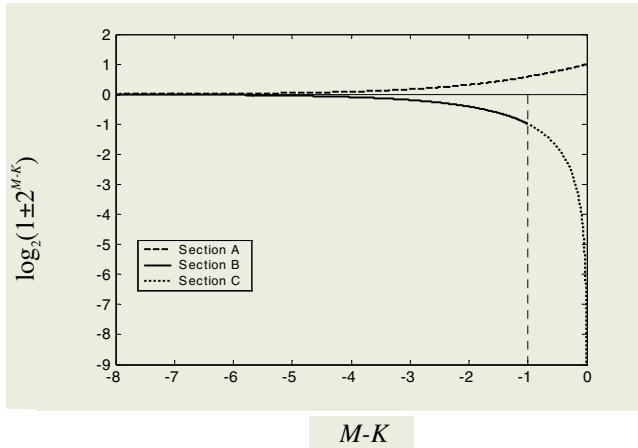


Fig. 3. The three section split of the addition/subtraction functions

To reduce the lookup sizes needed for the function approximations we calculate section A of the addition function and section B of the subtraction function by calculating the power function 2^{M-K} and the logarithm function $\log_2(\cdot)$. This idea was first suggested in [3] although it is applied in a different manner. For section A the addition function is calculated on the range $[T, 0]$, where T is the smallest value of $M-K$ that does not cause the addition function to evaluate to a number smaller than the

smallest number in the number system. This is analogous to the addition of a very small number and a very large number where the small number is shifted out of range of the large number and effectively an addition of zero produces the same result after rounding. The diagram of the required hardware to calculate the function in section A is shown in figure 4(a).

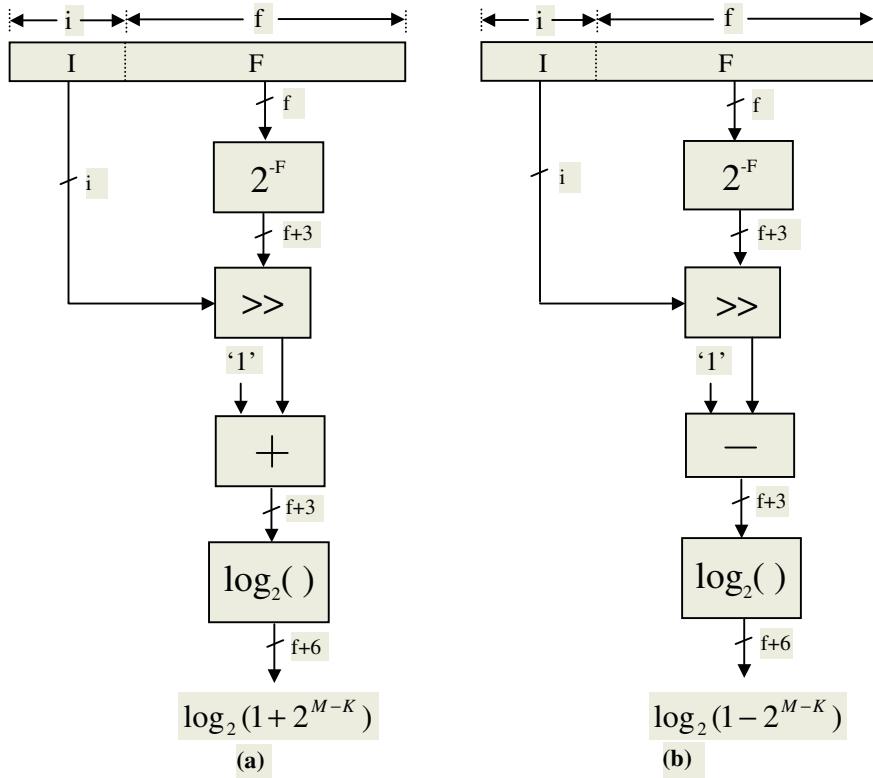


Fig. 4. (a) The required hardware to calculate the addition function on the range $[T,0]$. (b) The required hardware to calculate the subtraction function on the range $[T,1]$

For section B the subtraction function is calculated on the range $[T,1]$ where T has the same definition as for the addition function above. The hardware required for section B is illustrated in figure 4(b). For section C, the subtraction function on the range $(1,0)$, the hardware model of figure 4(b) cannot be used. This is due to the loss of accuracy, which can only be corrected by using very high precision approximations. To solve the accuracy dilemma we use an identity (9) introduced in [4].

$$\text{Let, } R = M - K. \quad (8)$$

$$\log_2(1 - 2^R) = \log_2\left(\frac{1 - 2^R}{-R}\right) + \log_2(-R). \quad (9)$$

The identity consists of two parts. The first part, which we will call the ‘correction function’, is a correction to the $\log_2(-R)$ function. A plot of the subtraction function, the $\log_2(-R)$ function and the correction function is shown in figure 5.

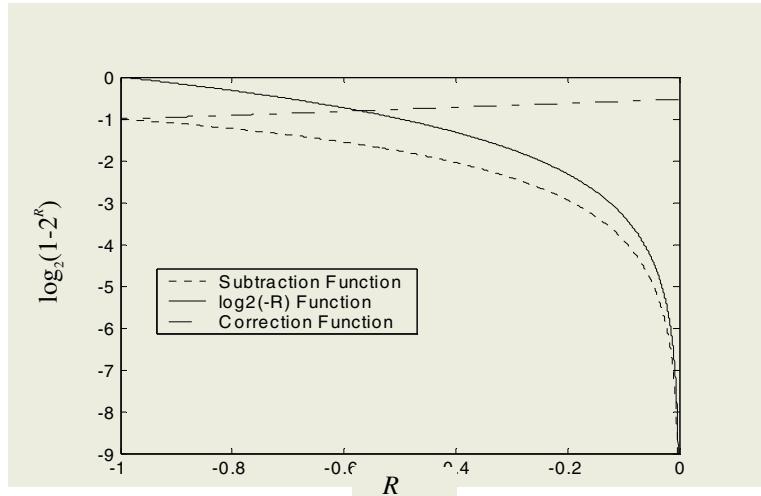


Fig. 5. A plot of the components that make up the subtraction function identity

From figure 5 we can see that the correction function plots a smooth regular curve of restricted range, which lends itself well to polynomial approximation. The $\log_2(-R)$ function can be easily evaluated using a shifting range reduction technique to reduce the approximation range to a single ‘binade’ and then adding the binary encoded shifting quantity to the approximation. The diagram of the required hardware to calculate the subtraction function for the range (1,0) is shown in figure 6.

To forge a complete hardware solution, similarities in the hardware diagrams are combined and hardware sharing is used where possible. The same hardware can be used to produce sections A and B with a minor modification to allow the addition component to perform a subtraction as necessary. The range of the function approximations is the same for each section, i.e. 2^f is evaluated on the range [0,1) and $\log_2(\cdot)$ is evaluated on the range [1,2). Sections A and B need to calculate $\log_2(\cdot)$ over the range [1,2) as does section C so the $\log_2(\cdot)$ approximation can be shared between all sections. This will not have a great impact on the delay as the function need only be calculated once for any approximation input. The correction function and the 2^f function can share the same arithmetic hardware (multipliers and adders) to calculate their approximations because again, only one of the functions is required for any approximation input. The diagram of the hardware model to calculate the whole LNS addition/subtraction function is shown in figure 7.

4 Better-than-FP Accuracy

This section on accuracy is adapted from the work by Lewis [1] and gives a bound on the permissible approximation error to produce BTFP results, a term coined by

Arnold [2]. Floating-point has a relative error $2^{-f-2} < \epsilon_{FP} \leq 2^{-f-1}$, so the worst-case error is 2^{-f-1} . The LNS has an absolute accuracy 2^{-f-1} corresponding to a relative accuracy of $\epsilon_{LNS} = 2^{2^{-f-1}} - 1 \approx 2^{-f-1.528}$ and giving a $\frac{1}{2}$ ulp accuracy improvement over floating-point. The worst-case relative error is achievable for a floating-point number system and should be for IEEE compliant systems [5]. The worst-case relative error is only achievable for LNS if exact arithmetic is possible.

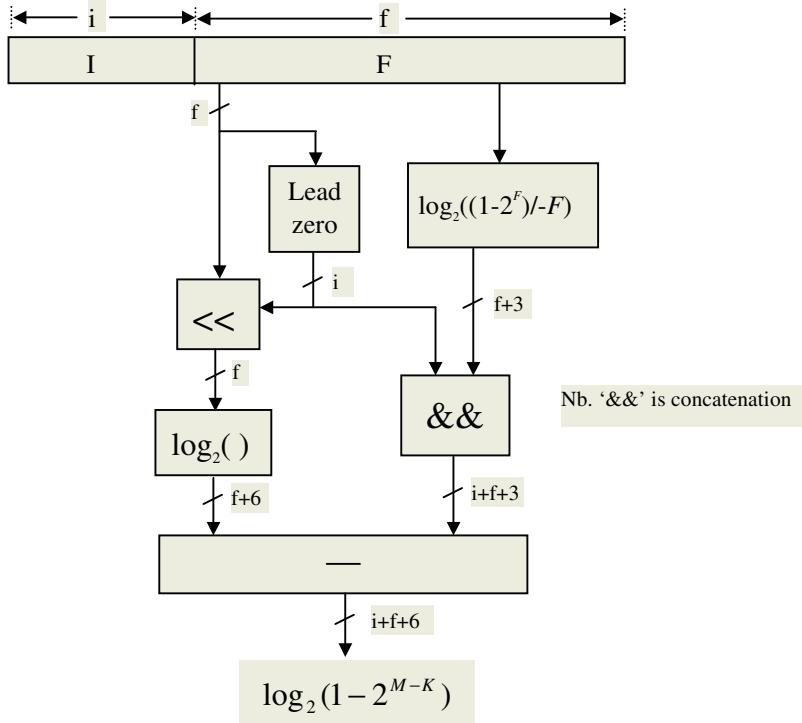


Fig. 6. A diagram of the hardware to calculate the subtraction function on the range (1,0)

To implement the logarithmic addition/subtraction function, transcendental functions need to be calculated either by using full table lookup or by function approximation. To guarantee correctly rounded results for function approximation methods the results need to be calculated with many extra bits of precision to overcome the ‘table-makers-dilemma’ problem [9]. In short the worst-case relative error is not achievable in any practical implementation so a compromise is taken.

The LNS addition/subtraction function will incur an error as a result of rounding, this is 2^{-f-1} . The addition/subtraction function approximation will also have an error ϵ_{dp} . The total error is $\epsilon_{dp} + 2^{-f-1}$. The relative error for a realistic LNS implementation (i.e. one with approximation errors) is,

$$\epsilon_{LNS} = 2^{(\epsilon_{dp} + 2^{-f-1})} - 1 = \log_e(2) \times (\epsilon_{dp} + 2^{-f-1}) \quad (10)$$

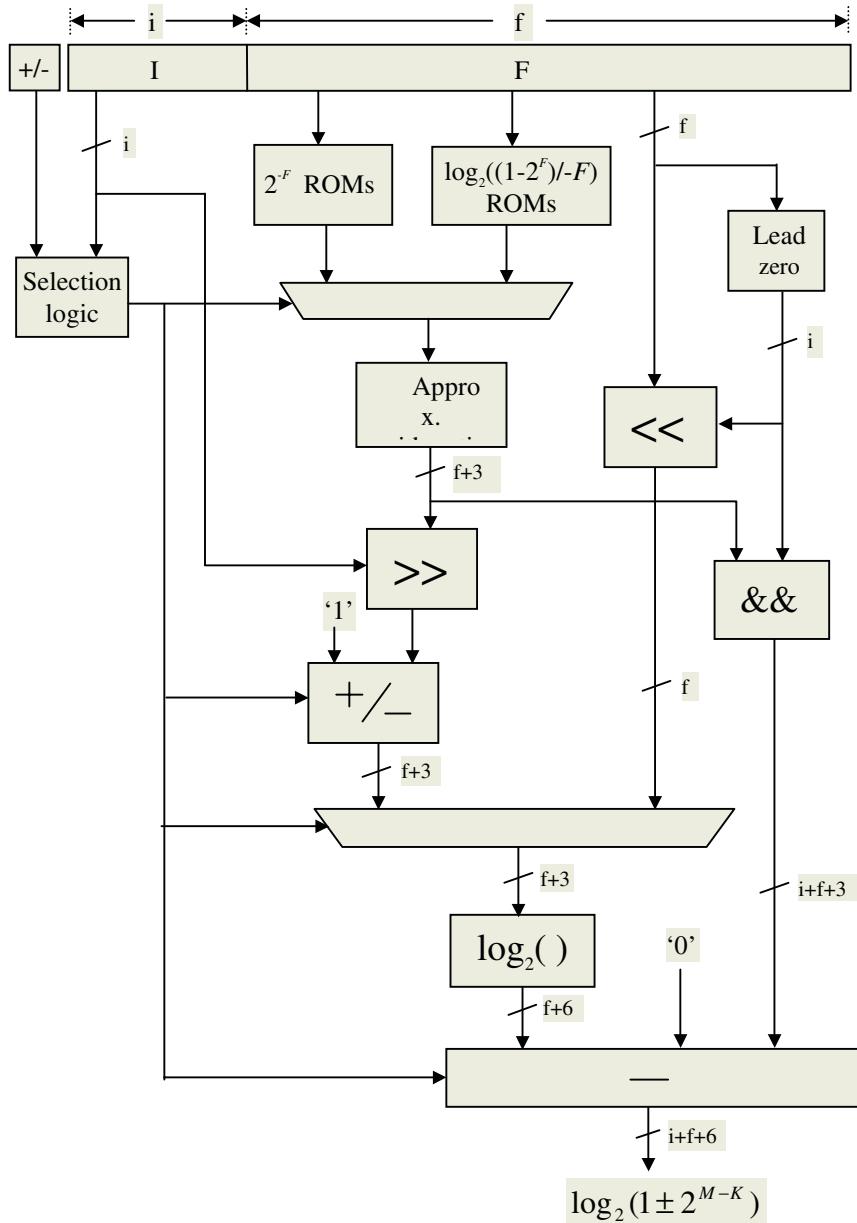


Fig. 7. The high-level hardware model to calculate the LNS addition/subtraction function

We would like the LNS to have an error that is equivalent or less than that of floating-point. The equation to calculate the permissible data path error that results in an equivalent accuracy of floating-point is,

$$\log_e(2) \times (\varepsilon_{dp} + 2^{-f-1}) \leq 2^{-f-1} \quad (11)$$

Solving for ε_{dp} gives,

$$\varepsilon_{dp} \leq \left(\frac{1}{\log_e(2)} - 1 \right) \times 2^{-f-1} \approx 1.771 \times 2^{-f-3} \approx 2^{-f-2.18} \quad (12)$$

Equation (12) requires the addition/subtraction function to be calculated correctly to three extra bits of precision to guarantee an error that is less than an equivalent floating-point system.

5 FPGA Implementation

A software model of the system has been developed using the MATLAB mathematical software package. The model allows the accuracy to be tested by enumeration to guarantee the BTFP accuracy criteria and provides a bit-true model to compare with the hardware model. The hardware model is written in VHDL with MATLAB being used to fill the coefficient ROMs needed for the function approximations.

5.1 Function Approximation

To calculate 2^F and the correction function a simple 2nd order Lagrange polynomial approximation evaluated by Horner's method is used. This is not the optimum approximation technique but it is simple to create and suffices for this study. The basic approximation is rearranged to allow the coefficient ROMs to be addressed by the MSBs of the argument and the LSBs to be the 'x-terms' of the polynomial. The $\log_2(\)$ function is evaluated by means of a third order Taylor approximation. The main choice for this is that such an approximation scheme has already been developed.

5.2 Shifter, Memory and Multiplier Structure

The shifters are constructed using layers of 4:1 and 2:1 multiplexers as done in [6] as these are the most efficient Virtex LUT-based multiplexer implementations. The shifters are binary encoded so each bit of the 'shift_amount' directly controls a multiplexer input. The multipliers are built based on the scheme given in [6], which uses one embedded multiplier and some extra logic to produce wider than 17X17 multipliers. The memory for the approximations is constructed using LUT-based memory. One LUT can be configured as a 4-bit in 1-bit out memory and many are multiplexed using dedicated embedded multiplexers to produce wider input memories.

6 Implementation Results

The addition/subtraction core has been implemented using Xilinx ISE 5.1.03i for an XC2V1000-4 part with results being generated for the two metrics of delay (ns) and area (slices). The delay is calculated by placing registers on the inputs and outputs and measuring the register-to-register delay as reported by the P&R tools. Table 1 shows the implementation results of an addition/subtraction core for an integer width of 8 and a fraction width of 23, which is the LNS format that is comparable to IEEE single precision floating-point. The results are given for implementations that use only the ‘logic-elements’; use a combination of the ‘logic-elements’ and 4 embedded multipliers; and a theoretical implementation that uses ‘logic-elements’, 4 embedded multipliers and 2 blockRAMs.

Table 1. Implementation results of the LNS addition/subtraction function for an integer width of 8 and a fractional width of 23

	‘Logic-elements’ only	‘Logic-elements’ and 4 embedded multipliers	‘Logic-elements’, 4 embedded multipliers and 2 blockRAMs
Area (slices)	1832	1289	1001
Delay (ns)	99	101	X

For the theoretical implementation the blockRAMs replace two 128X36-bit LUT-based ROMs. A single 128X36-bit ROM uses $8*36=288$ LUTs therefore two ROMs and two LUTs per slice means a saving of 288-slices.

Arnold [2] gives results for two FPGA implementations of the LNS addition function only. The unrestricted faithful rounding LNS adder uses 768-slices but has an accuracy that is worse than floating-point. Results for an unrestricted faithfully rounded subtraction function are not given. The estimated area for the BTFP LNS adder proposed by Lewis [1] is given in Arnold [2] as 2300-slices. Again, results for BTFP LNS subtraction are not given but would require a 10-fold increase in lookup ROM due to the singularity region [1]. Matousek et al. [8] implement a BTFP addition/subtraction function plus the logic to calculate the LNS sum using 1300-slices and 96 blockRAM cells. By using 96 blockRAMs the number of single precision LNS addition/subtraction components that can be fitted onto a single Virtex-II FPGA is restricted to one for all but the largest chip that could accommodate two.

7 Conclusion

A scheme has been presented that substantially reduces the lookup requirement and thus the area requirement of the implementation of an LNS addition/subtraction function on FPGA. Due to the area reductions it is now possible to accommodate 4 single-precision BTFP LNS addition/subtraction units on an XC2V1000 FPGA and potentially 45 on the largest 10M-gate chip [10]. A dual-path approach similar to that

of floating-point addition has been adopted and this reduces the lookups to be of fractional length. BTFP accuracy has been used as this allows a fair comparison with floating-point [6], which is a future work. Pipelining, delay reduction and efficient fixed-point to LNS conversions and vice-versa are also future works.

Acknowledgement

A special thanks to Richard Walke and the RTSL team at QinetiQ, Malvern.

References

1. Lewis, D.M.: Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit. IEEE Trans. on Comp. Aug. 1994. pp 974-982.
2. Arnold, M.G., Walter, C.: Unrestricted Faithful Rounding is Good Enough for Some LNS Applications. 15th symp. on Comp. Arith. 2001. pp 237-246.
3. Chen C., Chen R., Yang C.: Pipelined Computation of Very Large Word-length LNS Addition/Subtraction with Polynomial Hardware Cost. IEEE Trans. on Comp. Jul. 2000. pp 716-726.
4. Polioris, V., Stouraitis, T.: A Novel Algorithm for Accurate Logarithmic Number System Subtraction. IEEE Symp. on Circuits and Sys. May. 1996. pp 268-271.
5. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE std. 754. 1985.
6. Lee, B.R., Burgess, N.: Parameterisable Floating-Point operations on FPGA. 36th Asilomar Conf. on Signals, Systems and Comp. Nov. 2002.
7. Taylor, F.J., Gill, R., Joseph, J., Radke, J.: A 20-bit Logarithmic Number System Processor. IEEE Trans. on Comps. Feb. 1988. pp 190-199.
8. Matousek, R., Tichy, M., Pohl, M., Kadlec, J., Softley, C., Coleman, N.: Logarithmic Number System and Floating-Point Arithmetics on FPGA. Field-Programmable Logic and Applications, LNCS 2438, Sept. 2002. pp 626-636.
9. Schulte, M.J., Swartzlander, E.E, Jr.: Hardware Designs for Exactly Rounded Elementary Functions. IEEE Trans. on Comp. Aug. 1994. pp 964-973.
10. Xilinx. Virtex-II Platform FPGA Handbook. Dec. 2000.

A Modular Reconfigurable Architecture for Efficient Fault Simulation in Digital Circuits

J. Soares Augusto, C. Beltrán Almeida, and H. C. Campos Neto

INESC/IST**, Aptd. 13069, 1000-029 Lisboa, Portugal,
jose.augusto@inesc-id.pt, Tel: +351 213100294, Fax: +351 213145843

Abstract. In this paper, a modular reconfigurable architecture for efficient stuck-at fault simulation in digital circuits is described. The architecture is based on a *Universal Faulty Gate Block*, which models each 2-input gate by a 4-input Look-Up Table (LUT) and a Shift-Register (SR) with 3 stages, and relies on collapsing the stuck-at fault list of the gates using equivalence and dominance relations between faults. An example is presented, the expected performance is estimated and the applicability and limitations of the architecture are discussed.

1 Introduction

Fault simulation is an important issue in the design and in the test of electronic circuits. It measures the Fault Coverage (FC) obtained with a set of Test Vectors (TVs) applied to the Circuit Under Test (CUT). It also discovers redundant TVs, that is, TVs which aren't unique in detecting some fault [1, 2]. It is a fundamental tool in the evaluation of the testability of a given design and, in fault-tolerant circuits, fault simulation measures the degree of tolerance.

However, fault simulation is a very time consuming task and it is important to do it fast. Software techniques to accomplish this goal have been developed: the more important are the parallel, deductive and concurrent techniques [1, 2].

Hardware fault simulation is also well known. It relies on using specialized computer architectures that exploit the parallelism and/or concurrency in many CAD algorithms. Most hardware simulators are directed towards logic simulation, but many can also be used for fault simulation. Two examples are the logic simulation machine in [3], which was organized as a distributed processing architecture composed of separate processing units dedicated to specific tasks of the simulation algorithm, and the 'Yorktown Simulation Engine' [4] which exploited the parallelism of 256 processors. See [2] for a review of hardware fault simulation.

Reconfigurable Hardware (RHw) can speed-up many algorithms. In the last years, it was applied successfully to important ones. Many test and simulation algorithms are NP-complete [2], which means RHw is an important tool in these areas.

** This work was developed under the EC MEDEA+ A503 ASSOCIATE project, with funding from the Portuguese Government Agency "Agência de Inovação".

There is already a substantial amount of literature on the implementation of test-related algorithms in RHw. In [5] a critical path tracing algorithm was implemented in RHw. The satisfiability (SAT) problem, which is NP-complete and is very important to CAD and Test, was also implemented in RHw. In [6] a method for the emulation of the PODEM algorithm formulated as a SAT problem was proposed and an implementation of a small SAT problem appears in [9]. In [7] each instance of a SAT problem is solved through the creation of a specialized circuit: simulations predicted it could solve a random 3-SAT problem with 400 variables in 20 minutes, using a clock of 1 MHz. This approach of synthesizing each instance of SAT is discussed in [8] and in other articles from the same group.

In [10] a novel approach for generating test vectors that detect faults in combinational CUTs was introduced. It automatically built a circuit which implemented the D-algorithm (an Automatic Test Pattern Generation algorithm) specialized for the combinational CUT. It was estimated that the approach would be two times faster than a software implementation.

In [11] instance-specific accelerators for minimum-cost covering problems based on a branch-&-bound algorithm were presented. An instance-specific hardware architecture implementing branch-&-bound in 3-valued logic and using reduction techniques borrowed from software solvers, resulted in significant speed-ups in small-sized covering problems.

In [12] a method that does serial fault emulation using FPGAs, which must be reconfigured for each fault, was proposed. To improve the efficiency due to the overhead of the reconfiguration time spent in the mapping of numerous faulty circuits, independent faults are injected simultaneously, as well as some sets of dependent faults. This capability needs extra supporting circuitry.

A more efficient approach is followed in [13], where only *partial* FPGA reconfiguration is used to inject the faults.

Reconfiguration models can be classified into Compile-Time Reconfiguration (CTR) and Run-Time Reconfiguration (RTR), which is much more slower in practice. Our approach falls into the 'faster' CTR model. Other important issue in RHw is data transfer between the host computer and the hardware: the data must be transferred in as few 'batches' as possible to avoid the large time overhead in data transfer operations. Our approach also minimizes this issue.

In summary, in this paper a reconfigurable hardware architecture that emulates faulty digital circuits and pursues efficient fault simulation is proposed. In the following sections we discuss briefly the stuck-at fault model, fault list reduction and fault simulation, we present the reconfigurable architecture and its basic building block, and apply it to an example. Performance in moderately large (1000 gates) circuits is predicted from 3 typical implementations. We conclude the paper with an appreciation of the merits and of the limitations of the approach.

2 Stuck-at Faults and Fault Simulation

2.1 The Fault Simulation Problem

The fault simulation problem is formulated as follows: given a circuit and a fault list, apply a set of test vectors to the primary inputs of both the nominal/good circuit and the faulty circuits (circuits whose functionality suffers from the effect of a single fault) and collect the test responses in the primary outputs. The *single stuck-at zero/one fault* model is usually used in logic fault simulation.

The test responses which are collected are used for fault detection, diagnosis and for the selection of TVs for production test. This selection delivers a (minimal) test set that detects (almost) all the faults.

The simulation of the faulty circuits with a large set of TVs can be lengthy. This paper proposes RHw as a means of emulating the faulty circuits and simulating efficiently the faults.

2.2 Stuck-at Faults, Fault Equivalence and Fault Dominance

Although logic single stuck-at faults (SAFs) don't model all the possible defects, they are the main fault model used in practice. The reasons of this ubiquity are: SAFs are a simple model; the number of SAFs is finite (twice the number of nodes); and TVs that catch a large percentage of the SAFs (i. e., which have a good fault coverage) usually detect most of the defects, single or multiple, in the circuit (the correlation between SAF fault coverage and defects fault coverage is good).

There are two SAFs: the stuck-at-0 (SA0) and the stuck-at-1 (SA1) faults. If a node holds a SA0 fault, it will always have a logic '0'. In case it holds a SA1 fault, it will always be at '1'. The SAFs are associated with a logical block. A block with q terminals (primary inputs and outputs) has $2q$ SAFs: one of each per terminal. Thus, a 2-input gate has 6 faults (4 in the inputs and 2 in the output).

As fault simulation is time consuming, in practice only *single-faults* are simulated. There is also the empirical observation that tests with good single faults coverage have good multiple faults coverage.

Faults in a gate can be collapsed with *fault equivalence and dominance* [1].

Fault equivalence: faults in a logic block are equivalent iff the response in the output is the same for them all when all the input logic combinations (2^{n_i} , for n_i inputs) are applied.

Fault dominance: Let T_g be the set of all TVs that detect a fault g . Another fault f dominates the fault g iff f and g are functionally equivalent under T_g . (The concept of fault dominance implies that all the TVs that detect the fault g also detect the dominant fault f , but not vice-versa.)

Let's reduce the fault list of a NOR gate $z = \overline{x + y}$. Table 1 shows the nominal output (z) and the output under the effect of the faults. It is clear that x-SA1, y-SA1 and z-SA0 are equivalent: the corresponding columns are equal for all inputs. From this equivalent set, only z-SA0 will be kept in the fault list.

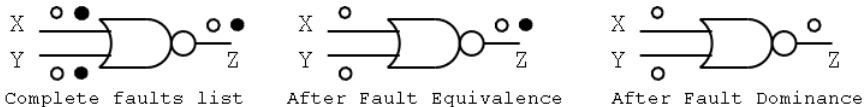
Table 1. Faults in a 2-input NOR gate. The boxed values illustrate fault dominance.

(x,y)	$z = \overline{x+y}$	x-SA0	x-SA1	y-SA0	y-SA1	z-SA0	z-SA1
00	1	1	0	1	0	0	1
01	0	0	0	1	0	0	1
10	0	1	0	0	0	0	1
11	0	0	0	0	0	0	1

In table 1 the columns corresponding to x-SA0, y-SA0 and z-SA1 have boxes in the entries that differ from the nominal z . The only TV of the fault x-SA0 is $(xy) = 10$ and the only TV of y-SA0 is $(xy) = 01$. Both these TVs detect z-SA1, which means that *the fault z-SA1 dominates both x-SA0 and y-SA0*.

In case x-SA0 or y-SA0 are detected, it is guaranteed that the dominant fault z-SA1 also is detected. If none of the dominated faults is detected, there is no information about the detection of z-SA1. Thus, the removal of the dominant fault from the fault list leaves uncertainty about fault coverage. However, in practice FC is usually above 99% which means that almost all the dominated faults are detected. This fact justifies collapsing the dominant faults (z-SA1 in the NOR).

Fault equivalence and fault dominance reduce the fault list of the NOR gate to z-SA0, x-SA0 and y-SA0. This is shown in figure 1 where a black dot and a circle represent SA1 and SA0 faults, respectively. The same procedure can be applied to 2-input AND, NAND, OR and 'x-input-negated' gates (which are useful in expanding the 2-input XOR). The reduced fault lists are in table 2.

**Fig. 1.** Fault list reduction in the NOR with equivalence and dominance.**Table 2.** Reduced fault lists in 2-input gates (inputs x, y , output z).

Gate	Reduced Fault List	Gate	Reduced Fault List
OR	z-SA1, x-SA0, y-SA0	OR/xNEG	z-SA1, x-SA1, y-SA0
NOR	z-SA0, x-SA0, y-SA0	NOR /xNEG	z-SA0, x-SA1, y-SA0
AND	z-SA0, x-SA1, y-SA1	AND/xNEG	z-SA0, x-SA0, y-SA1
NAND	z-SA1, x-SA1, y-SA1	NAND/xNEG	z-SA1, x-SA0, y-SA1

3 The Reconfigurable Architecture

The reconfigurable architecture we propose falls into the compile-time reconfiguration (CTR) model and is directed towards FPGA implementation. After the initial compilation into the FPGA, the nominal and all the faulty circuits are simulated in sequence with all the TVs the design/test-engineer wants to apply. The FPGA is re-programmed only when a different circuit is simulated.

3.1 The Universal Faulty Gate Block (UFGB)

The basic primitive block of the architecture is the *Universal Faulty Gate Block (UFGB)*. The UFGB models a 2-input AND, NAND, OR or NOR gate and its single SAFs. The gates with one input negated are also supported. However, *2-input NOR and 2-input NAND gates are sufficient* which means that any logic function can be implemented with only one of them. Thus, more complex gates are expanded, at compile time, to be represented by a combination of the aforementioned 2-input gates. In conclusion, the technique is applicable to any circuit and is limited only by the FPGA size and/or wiring and I/O limitations.

Fault equivalence and dominance collapse the 6 possible SAFs into 3 SAFs. Accordingly, the 'reconfigurable block' we propose implements 4 functions: the nominal function of the gate and the 3 stuck-at faults. This structure, that we call UFGB, replaces each gate in the CUT and, through 'soft' reconfiguration, permits to simulate the nominal gate and each single fault in the gate's fault list.

The UFGB consists of one Look-Up Table (LUT) with 4 inputs and 1 output, and of a 3 stage shift register (SR) wired as shown in figure 2. The UFGB structure is the same for all the supported 2-input gates: the differences between each gate are encapsulated in the LUT configuration.

As many FPGAs are based on LUTs with 4 (or more) inputs and 1 output, the LUT is a component already built into RHw. The SR is the serial connection of 3 flip-flops, or memory cells, which are also available in most reconfigurable devices.

Let's explain how the UFGB works using the NOR gate ($z = \overline{x+y}$) as an example. Figure 2 shows the original gate with the 3 faults to be simulated, and the UFGB that emulates the gate in the FPGA. 2 of the LUT's inputs are assigned to the original NOR inputs, x and y . The other 2 inputs are configuration signals named $c1$ and $c2$. The LUT is programmed as described in table 3 and works as a *multiplexer of logic functions*.

$c1$ and $c2$ configure the 'LUT gate' in nominal mode or in one of the 3 fault modes, which are simulated in sequence by applying a bit-stream '...0000110000...' into the $SRin$ input of the shift register (see the legend in figure 3). As the bit-stream is right-shifted, $c1$ and $c2$ span all the configuration options and the 3 faults are simulated in sequence. As soon as the 1's leave the nodes $c1$ and $c2$, the UFGB goes into nominal mode. The need of a 3-stage SR, instead of a 2-stage SR, is explained below.

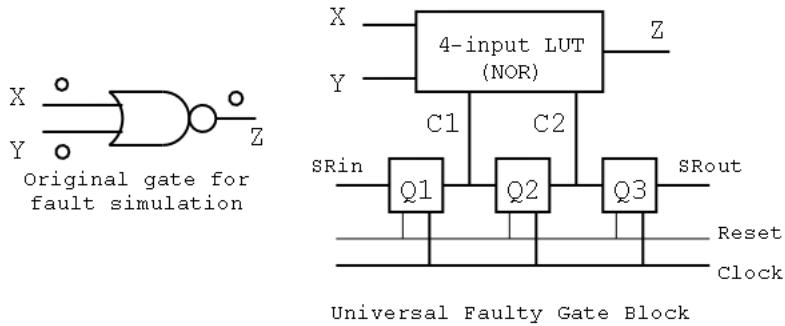


Fig. 2. Universal Faulty Gate Block (UFGB).

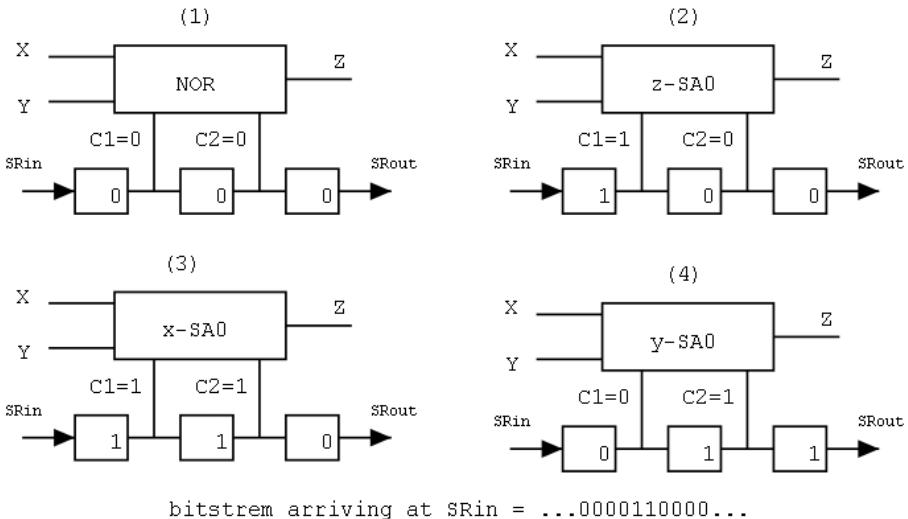


Fig. 3. UFGB configuration with a bit-stream '...000011000...' arriving at SRin: (1) with '000' in the SR, the LUT emulates a NOR; (2) with '100' in the SR the LUT emulates a z-SA0 fault; in (3) and (4) bit-stream shifting emulates the faults x-SA0 and y-SA0, respectively. With another shift, the SR is reset and the gate returns to nominal mode.

3.2 Fault Simulation in a Complete Circuit

Figure 4 shows a complete circuit built with one AND, one OR and one NOR, as well as the architecture which emulates it. The programming of the LUT for the NOR gate is shown in table 3. The other LUTs are programmed according to the gate nominal function and the fault list in table 2. In figures 3 and 4 the *Clock* and *Reset* lines shown in figure 2 are omitted in order to simplify the picture.

The flip-flops $m0a$ and $m0b$ are different from those in the UFGBs ($m1$ to $m9$): they have a *Set* input while the others have a *Reset* input as shown in

Table 3. LUT programming to replace the NOR gate $z = \overline{x + y}$.

(c1,c2)	Mode	Function	z	nominal	$z\text{-SA}0$	$x\text{-SA}0$	$y\text{-SA}0$
(x,y)				(c1,c2)			
00	nominal	$z = \overline{x + y}$		00	1	0	1
10	$z\text{-SA}0$	$z = 0$		10	0	0	0
11	$x\text{-SA}0$	$z = \overline{y}$		11	0	0	0
01	$y\text{-SA}0$	$z = \overline{x}$		01	0	0	1

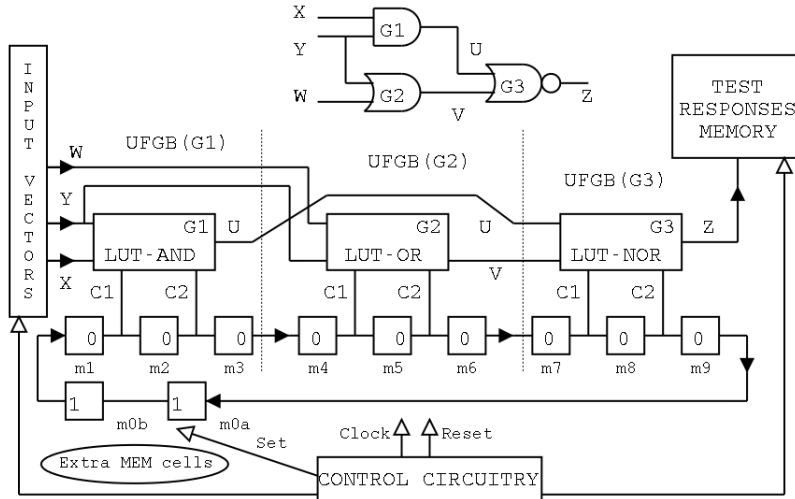


Fig. 4. Simulation of a circuit with 3 gates (top). The complete simulation architecture consists of 3 UFGBs (separated with dashed lines) with LUTs corresponding to each gate type, two extra memory cells $m0a$ and $m0b$, control circuitry, a block to apply the TVs and a block to save the test outputs ('Test Responses Memory'). In the beginning of the simulation (the logic values shown) the 'good' circuit is configured.

figure 2. In the beginning of the simulation $m0a$ and $m0b$ are set, while $m1$ to $m9$ are reset: a closed bit-stream with only two '1's is loaded in the flip-flops. In this stage, all the UFGBs are configured to the nominal function ($c1 = c2 = 0$ in all of them) and the nominal circuit is being simulated. After 1 clock cycle, the bit-stream rotates to the right and $c1 = 1, c2 = 0$ in the LUT-AND of G1: thus, we simulate the first fault in the fault list of G1 (according to table 2 it is the output-SA0 which in figure 4 is U-SA0). After a second clock cycle, $c1 = 1, c2 = 1$ in the LUT-AND of G1 and the second fault in this gate is simulated. And so on and so forth.

The third stage in each SR of the UFGBs can be explained with the help of figure 4. Consider, for instance, that $m3$ in the UFGB of G1 doesn't exist. When shifting the bit-stream there would be a state where $m1 = c1(G1) = 0$, $m2 = c2(G1) = 1$, $m4 = c1(G2) = 1$ and $m5 = c2(G2) = 0$ (remember $m3$ is

supposedly removed). With this configuration *a double fault is being simulated*, as the logic conditions $(c1, c2)(G1) = 01$ and $(c1, c2)(G2) = 10$ configure simultaneously the two LUTs associated to G1 and G2 in a faulty mode. $m3$ is thus inserted to act as a 'waiting buffer' to the front-end bit '1' while the rear bit '1' is still configuring the LUT of G1 to simulate one fault. This reasoning applies also to $m6$ and $m9$, the last flip-flops in the SRs belonging to the UFGBs of the gates G2 and G3.

The good circuit is simulated twice until the bit-stream returns to the initial position: one time in the beginning, as shown in figure 4, and another time when $m9 = m0a = 1$. One of the flip-flops $m0a$ or $m0b$ can be discarded if the last flip-flop in the chain, $m9$, is initially 'set' instead of 'reset'. This, however, destroys the modularity.

The TVs can be applied through a memory or using a generator such as a Linear Feedback Shift Register (LFSR). The outputs can be saved in the 'Test Responses Memory' or applied to an LFSR configured as a signature analyzer.

The control circuitry in figure 4 controls the clocks and the simulation modes, the set and reset of the SR flip-flops and of $m0a$ and $m0b$, the addressing of the memory with the TVs, the saving of the results into the 'test responses memory', it signals the end of the simulation and does an eventual intermediate data transfer with the host computer. The number of *Clock* pulses applied to the SR chain can be programmed in the control circuitry or, otherwise, the end of a 'bit-stream round trip' can be detected by AND'ing $m0a$ and $m0b$. Note that the only control signals applied to the emulation hardware are *Clock*, *Reset* and *Set* which usually use dedicated lines in the FPGA: thus, there is an economy of routing resources, which is a key advantage of this simulation architecture.

4 Performance Estimation

To estimate fault simulation times in real circuits, 1000 UFGBs were compiled into a Xilinx Spartan-II-200 FPGA. This reconfigurable device has 2352 slices (i.e., reconfigurable units).

Three circuits, all with 1000 UFGBs, were synthesized: 1000 UFGBs in series (the z output of each UFGB is linked to the next x and y inputs); a rectangular array of 10×100 UFGBs; and a rectangular array of 20×50 UFGBs. In these arrays, the number of inputs is the same as the number of outputs and is equal to 10 and 20, respectively. The inputs of the intermediate stages in the arrays were cross-wired to the outputs of the previous stage. The number of slices used in the FPGA before and after placement and routing (P&R), and the input/output delay after P&R are shown in table 4.

In all the 3 cases almost all of the slices in the FPGA were used, which gives a consumption figure of 2.35 slices per UFGB. However, before P&R only an average of 1.6 slices per UFGB was used. As the FPGA is almost completely populated, it is acceptable this overhead due to the P&R constraints.

Each fault simulation will run in a time approximately equal to the input/output delay shown in table 4, which is approximately proportional to the depth (number of levels) of the circuit. Between 1ns and 2 ns per level are

Table 4. Results from implementing 3 circuits with 1000 UFGBs.

Circuit	Slices before P&R	Slices after P&R	in/out delay (ns)
1000 UFGBs in series	1607	2350	1290
10 × 100 UFGBs (10 in/out)	1606	2350	190
20 × 50 UFGBs (20 in/out)	1617	2350	108

measured in the example. The compilation time of the FPGA will have to be accounted for in more exact calculations, but it can be neglected when simulating many faults [12].

The delay allows the estimation of simulation times. For instance, in the 20 × 50 UFGBs array the simulation of all the faults (3000) with all the possible TVs ($2^{20} \approx 10^6$) would last $3 \times 10^3 \times 2^{20} \times 108 \times 10^{-9} \approx 340$ seconds (i.e., 5.7 minutes). In the 10 × 100 array it would be $3 \times 10^3 \times 2^{10} \times 190 \times 10^{-9} \approx 0.58$ seconds. In conclusion, when the number of inputs is less than about 20, all the TVs can be simulated in a reasonable amount of time!

The *fault simulation time per gate* (t_{fsgate}) allows us to compare our results with software fault simulation. The third circuit in the table could be a real example (i.e., 1000 gates, 50 levels). Thus $t_{fsgate} = 108$ ps. According to [13], where are reported *software fault simulation results* in the ISCAS'89 benchmark circuits s5378 and s13207, with a widely used commercial fault simulation tool in a Sun Ultra 10/440 workstation with 1 GB Ram, t_{fsgate} is $3.7 \mu\text{s}/\text{gate}$ and $13.2 \mu\text{s}/\text{gate}$, respectively. This shows a potential of *more than a four order magnitude improvement in simulation time* of the approach here proposed.

5 Applicability and Limitations of the Architecture

All the faults (n gates will have $3n$ faults) can be simulated without reconfiguring the RHw. Fault simulation speed is limited by maximum clock speed in FPGA's flip-flops, paths' delays, and by the fill up of the test acquisition memory (a data transfer to the host computer must be done). Placement and routing can be a problem in case the maximum FPGA capacity is approached.

The architecture is directed towards combinational circuits. Unsupported gates can be synthesized with NANDs or NORs. Fault simulation of *sequential synchronous circuits* can be done by expanding them according to the combinational 'iterative array' model where each stage is a copy of the combinational block of the original sequential circuit. This approach is followed in the discovery of test vectors for faults in sequential circuits [1, 2]. Asynchronous circuits can eventually be tackled (their gates are replaced by UFGBs) but one must keep in mind that FPGAs' delays are different from ASICs' delays.

The validity of the emulation of the 'true' circuit by the proposed methodology is at logical level. Delays in FPGAs are different from ASICs' delays. Thus, delay-dependent issues such as timing verification, delay faults and stuck-open faults cannot be checked with our approach.

6 Conclusions and Further Work

We proposed a modular architecture suited to efficient stuck-at fault simulation in digital circuits. It is expected that in moderately sized circuits (e.g. with 1000 gates) one fault is simulated in less than 200 ns (5.000.000 faults/second simulation rate). The modularity of the architecture eases its generalization to simulate larger circuits using hardware 'virtualization'. The efficiency comes from the fact that *all the faults are simulated without performing total or partial reconfiguration*.

An automatic system is under development. It consists of tools that: convert unsupported gates (XORs and gates with more than 2 inputs) to 2-input gates; expand synchronous sequential circuits into the corresponding iterative array; extract the fault list from the circuit description; add control circuitry; define RAM partitions (where test vectors are kept, where the responses are saved); convert the circuit file into an HDL, compile it and download it into the RHw (FPGA); run the simulation, read the responses from the output memory, perform fault diagnosis and select test vectors.

References

1. M. Abramovici, M. Breuer and A. Friedman, "Digital Systems Testing and Testable Design" IEEE Press, 1990.
2. H. Fujiwara, "Logic Testing and Design for Testability", MIT Press, 1985.
3. M. Abramovici, Y. Levendel, P. Menon, "A logic simulation machine", Proc. 19th Design Automation Conference, pp. 65-73, 1982.
4. G. Pfister "The Yorktown Simulation Engine", Proc. 19th Design Automation Conference, pp. 51-54, 1982.
5. M. Abramovici, P. Menon, "Fault simulation on reconfigurable hardware", 5th IEEE Symposium on FPGA-Based Custom Computing Machines (FCCM'97), 1997.
6. M. Abramovici and D. G. Saab, "Satisfiability on Reconfigurable Hardware", 7th Int. Workshop on Field Programmable Logic and Applications, 1997.
7. T. Suyama, M. Yokoo, H. Sawada, "Solving Satisfiability Problems Using Logic Synthesis and Reconfigurable Hardware", 31st Hawaii Intl. Conf. on Sys. Sciences, 1998.
8. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware", Proc. IEEE Symp. on Field-Programmable Custom Computing Machines, April, 1998.
9. M. Abramovici, J. de Sousa, D. Saab, "A Massively-Parallel Easily-Scalable Satisfiability Solver Using Reconfigurable Hardware", Design Automation Conference (DAC'99), New Orleans, USA, 1999.
10. F. Kocan, D. G. Saab, "Concurrent D-Algorithm on Reconfigurable Hardware", Int. Conference on Computer Aided Design (ICCAD'99), 1999.
11. C. Plessl, M. Platzner, "Instance-Specific Accelerators for Minimum Covering", 1st Intl. Conf. on Eng. of Reconf. Systems and Algorithms, Las Vegas, USA, 2001.
12. K.-T. Cheng, S.-H Huang, W.-J. Dai, "Fault Emulation: a New Methodology for Fault Grading", IEEE Trans. CAD, vol. 18, no. 10, pp. 1487-95, 1999.
13. A. Parreira, J. Teixeira, M. Santos, "A Novel Approach to FPGA-Based Hardware Fault Modeling and Simulation", IEEE Int. Workshop on Design and Diag. of Elect. Circ. and Systems, Poland, April, 2003.

Evaluation of Testability of Path Delay Faults for User-Configured Programmable Devices*

Andrzej Krasniewski

Warsaw University of Technology, Institute of Telecommunications
Nowowiejska 15/19, 00-665 Warsaw, Poland
andrzej@tele.pw.edu.pl

Abstract. A model of the combinational section of a programmable device suitable for an analysis of testability of delay faults is proposed. All relevant factors that affect the evaluation of testability of path delay faults are identified and their impact on the outcome of the evaluation is discussed. A detailed analysis, supported by quantitative results, focuses on the selection of the set of target faults in terms of a class of logical paths and on the concept of defining testability measures for physical paths rather than for logical paths. Practical guidelines are formulated for the development of a procedure for the evaluation of testability of path delay faults.

1 Introduction

With the growing miniaturization, complexity and speed of today's ICs, increasingly more problems are associated with timing. These problems are caused not only by global and local disturbances of the fabrication process, but also by the impact of noise/interference effects, such as signal coupling (crosstalk), power supply noise and substrate noise, some of which are specific for the system environment [1] [2]. This has the following consequences:

- Timing-related problems cannot be adequately identified by testing an isolated chip by the manufacturer. This is especially true for programmable devices for which the operational (user-defined) configuration and clock frequency is not known at the time the device is fabricated. Therefore, although an application-independent test procedure aimed at delay faults, e.g. the one proposed in [3], might be useful, for highly dependable systems, it must be augmented with in-system test of a user-defined configuration (application-dependent test, application-oriented test [4]).
- It is extremely difficult, if not impossible, to generate deterministic patterns that would account for the above described timing-related effects [1]. Therefore, random testing might be a preferred solution.

Application-dependent random testing of delay faults in programmable devices can be implemented using the BIST techniques [5]. It should be emphasized that, for

* This work was supported by the State Committee for Scientific Research of Poland under grant no. 4 T11D 014 24

advanced FPGAs or CPLDs, by exploiting their reconfigurability or partial reconfigurability, BIST can be implemented at no circuitry or performance penalty [3] [5]-[7]. What is unclear about the BIST-based random testing is its quality.

The evaluation of efficiency of testing delay faults in FPGAs or CPLDs is difficult. As was shown in [8], the conventional methods for analysis of delay fault testability [9]-[11], developed for networks of simple gates, i.e. NOT, AND, NAND, OR, and NOR gates, are not directly applicable. This is because basic logic components of programmable devices (LUT-based FPGAs or similar) can implement arbitrary Boolean functions (and not only positive or negative unate functions) and, therefore, even such simple concepts like “a controlling value” do not apply.

Some specific aspects of the evaluation of quality of testing delay faults in a network of arbitrary logic components are presented in [8] [12]. In this paper, we discuss this problem in more detail.

2 Determining the Speed of a User-Programmed Device

The speed of a user-programmed FPGA/CPLD is determined by the maximum time it takes to propagate a transition along a path between two memory elements (we assume that all memory elements are controlled by the same clock). A path may include various types of logic components. We distinguish *active logic components* (ALCs), i.e. components that have at least two inputs which have not been fixed to constant values by programming the device. In our analysis, other (non-active) logic components are seen as part of the interconnection structure. This refers, in particular, to any component which - for a specific user-defined configuration - implements a single-argument function (an example being a multiplexer whose address is fixed or a 2-input gate whose one input is set to a non-controlling value). For a non-active component, a transition of a particular polarity (rising or falling) at its input always results in a transition of a specific polarity at its output. This may not be the case for an active component where for a transition of a particular polarity at its input, the occurrence of the output transition and its polarity depends on the state of the other inputs. In the case when a non-active logic component implements an inversion function, we “remove” this inversion and appropriately modify the function of ALCs driven by the considered component, so that not to change the network function.

The combinational part of a used-configured programmable device is represented as a network of single-output ALCs (a multiple-output component is represented as a set of single-output components) that implement arbitrary Boolean functions. Inputs and outputs of the network are special ALCs. Then, any path can be seen as an alternating sequence of ALCs and connections. This is illustrated in Fig. 1. It may be observed that the function implemented by ALC3 (which represents LUT2) has been modified to account for the “removed” inversion between the AND gate and LUT2.

The following assumptions are taken regarding propagation delays of components in the considered network of ALCs:

- delays are assigned to both ALCs and connections;
- ALC and connection delays may depend on the polarity of signal transitions;

- different ALC delays may be assigned to different ALC inputs;
- delays of a programmable ALC may depend on its specific user-defined function (e.g. different delays may be assigned to LUTs that implement different functions);
- for a multiple-destination connection (originating at some ALC and leading to several other ALCs), different delays may be assigned to its different branches.

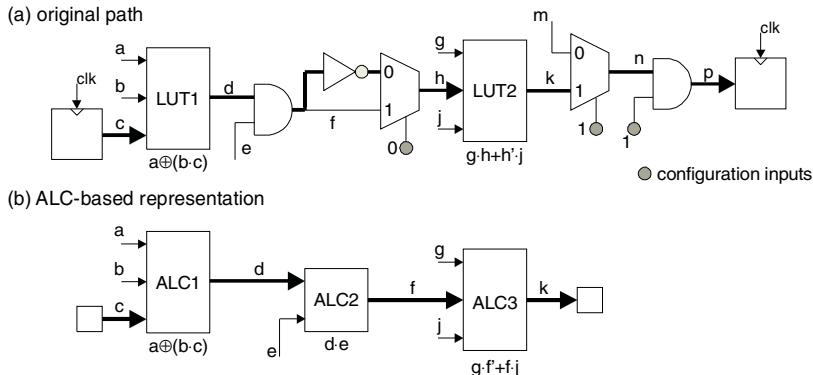


Fig. 1. Representation of a path in a network of ALCs

The propagation delay of a path depends on delays of its components - ALCs and connections. The delays of relevant portions of memory elements (flip-flops) at which the path originates and terminates are included in the delays of adjacent connections.

In the path-delay fault model, it is assumed that faults (excessive delays) are associated with logical paths [9] [10]. For a network of simple gates, a logical path is defined by a path (physical path, structural path) and the polarity of a transition that occurs at its input (or output); this is sufficient to determine the polarity of a transition at the input or output of each gate on the path. This may, however, not be the case for a network of ALCs. For example, for the path in Fig. 1(b), assuming that a rising transition at input c propagates through ALC1 (which requires $b = 1$), either a rising transition (for $a = 0$) or a falling transition (for $a = 1$) occurs at connection d.

For a network of ALCs, a *logical path* $\pi(\text{PTP})$ is defined by a path π and a *path transition pattern*, PTP, which specifies for each connection along the path, whether a rising (\uparrow) or a falling (\downarrow) transition occurs at the output of the ALC which feeds this connection. The path transition pattern must comply with the ALC functions, i.e. for each ALC along the path, transitions at the input and output of this ALC must have the same (different) polarity if the ALC function is positive (negative) unate in its on-path input [13]. Thus, the number of logical paths associated with a given (physical) path is 2^{K+1} , where K is the number of ALCs that implement functions which are binate in their on-path inputs. Path transitions patterns that do not comply with the ALC functions correspond to *pseudological paths*. For example, for path c-d-f-k in Fig. 1(b), the functions implemented by ALC1 and ALC3 are binate in their on-path inputs, whereas the function implemented by ALC2 is positive unate in its on-path input. Thus, the following 8 logical paths are associated with path c-d-f-k: $c \uparrow d \uparrow f \uparrow k \uparrow$, $c \uparrow d \uparrow f \uparrow k \downarrow$, $c \uparrow d \downarrow f \downarrow k \uparrow$, $c \uparrow d \downarrow f \downarrow k \downarrow$, $c \downarrow d \uparrow f \uparrow k \uparrow$, $c \downarrow d \uparrow f \uparrow k \downarrow$, $c \downarrow d \downarrow f \downarrow k \uparrow$, and $c \downarrow d \downarrow f \downarrow k \downarrow$.

Not all path transitions patterns that comply with the ALC functions can actually be produced in normal operation of the network, especially if the set of *functional input pairs*, i.e. vectors pairs that can occur at the input of the network in normal operation, does not contain all possible input pairs. A logical path $\pi(\text{PTP})$ is *feasible* if there exists a functional input pair which produces PTP.

For a feasible logical path $\pi(\text{PTP})$, there may or may not exist a functional input pair that - for a given *delay assignment* that specifies the propagation delay of each logical path in the network - sensitizes $\pi(\text{PTP})$, i.e. propagates a transition along π , producing PTP (the sensitization of a logical path is formally defined in [12]). If such a functional input pair exists, $\pi(\text{PTP})$ is called *true*, otherwise $\pi(\text{PTP})$ is called *false*.

For a given delay assignment, a logical path $\pi(\text{PTP})$ *determines the speed of the network* if for all possible functional input pairs, the latest transition at the network output is an effect of the sensitization of $\pi(\text{PTP})$ (by some of these pairs). Thus, to determine the speed of the network, propagation delays of true paths should only be examined. However, for a “real” device, the delay assignment is unknown - due to imperfections of the manufacturing process, path delays can vary significantly. Therefore, to determine the speed of the network, *irredundant* logical paths, i.e. logical paths that might be true for *some* delay assignment must be considered.

It might appear that any irredundant logical path determines the speed of the network under some delay assignment. However, in [12] it was shown that such a statement is incorrect and that to determine the speed of the network it is sufficient to examine a subset of irredundant paths, called *delay-essential* paths.

The following remarks should be made regarding the above defined concepts:

- Although “irredundant logical path” and “delay-essential logical path” are clearly timing-related concepts, no knowledge of network timing is necessary to decide whether or not a particular logical path is irredundant or delay-essential [12].
- The set of delay faults associated with delay-essential logical paths is an “ideal” set of target faults for any timing-oriented test procedure; this set contains all those and only those logical paths that determine the speed (maximum propagation delay) of the network under the unknown delay assignment.

3 Evaluation of Testability of Path Delay Faults

When evaluating the testability of path delay faults for a user-configured device, several factors should be considered, as shown in Fig. 2. We focus on the factors that are unique or particularly relevant for programmable devices, i.e. which differentiate networks of ALCs from networks of simple gates. These differences stem from that in a network of ALCs, a possibly large number of logical paths of possibly different testability characteristics are associated with each physical path, whereas in a network of simple gates, each physical path has exactly two corresponding logical paths.

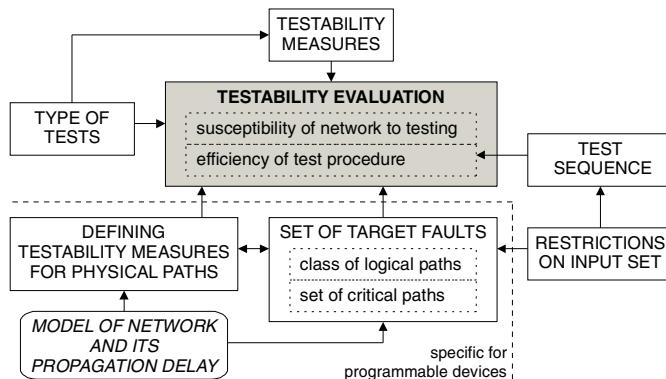


Fig. 2. Evaluation of testability of path delay faults for a user-configured programmable device

3.1 Testability Measures

There are two facets of the evaluation of testability: the evaluation of susceptibility of a network to testing and the evaluation of efficiency of a specific test procedure. The susceptibility of a network to testing is usually measured by *fault testability*, i.e. the percentage of faults that are detectable. The efficiency of a specific test procedure is usually measured by *fault coverage*, i.e. the percentage of faults that are detected by a test sequence produced by the considered procedure.

3.2 Type of Tests

For path delay faults, various types of tests are defined, including weak non-robust tests (WNR-tests), often referred to as non-robust tests, strong non-robust tests (SNR-tests), also referred to as restricted non-robust tests, and robust tests (R-tests). For networks of arbitrary logic components, the requirements for these types of tests can be found in [14]. The values of testability measures are calculated assuming a specific type of tests. Thus, we can report, for example, fault testability for robust tests (R-testability) or fault coverage by WNR tests (WNR-coverage).

3.3 Set of Target Faults

The value of a particular testability measure can only be calculated for a given *set of target faults* (fault model). As was stated earlier, in the considered network of ALCs, there is a one-to-one correspondence between logical paths and path delay faults. Therefore, the set of target faults is defined by a class of logical paths (a set of logical paths having a certain attribute) or by its appropriately specified subset.

a) classes of logical paths

A straightforward approach is to define the set of target faults (target logical paths) based on some specific type of testability. This way, the set of target faults can be defined as the set of WNR-testable logical paths, the set of SNR-testable logical paths, or the set of R-testable logical paths. Such an approach, taken in many publications on testing delay faults in gate networks, neglects, however, the key question of “which faults actually affect the speed of the network”.

As was stated in Section 2, the “ideal” set of target path delay faults in a network of ALCs is defined by the class of delay-essential logical paths. Although this class can be identified based exclusively on the network structure and functions implemented by ALCs, an examination of logical *multipaths* is required [12]. This might be too complex even for medium-size networks. Therefore, a practical solution would be to rely on a “reasonable approximation” of the set of delay-essential logical paths. The proposed solution is based on the relationship between the various classes of logical paths. As shown in Fig. 3, the set of delay-essential paths is a superset of the set of SNR-testable paths and is a subset of the set of irredundant paths. Thus, these two sets, whose identification does not involve an examination of multipaths, can be used to obtain lower- and upper-bound estimates on the “exact” testability measures. Other, less exact, estimates would rely on the set of feasible paths and all logical paths (lower-bound estimates) and on the set of R-testable paths (upper-bound estimates).

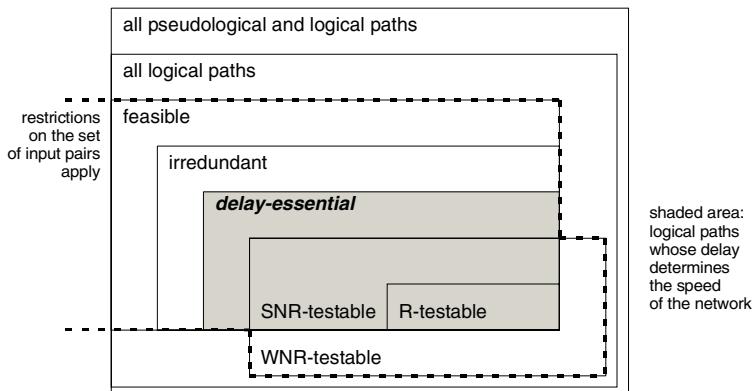
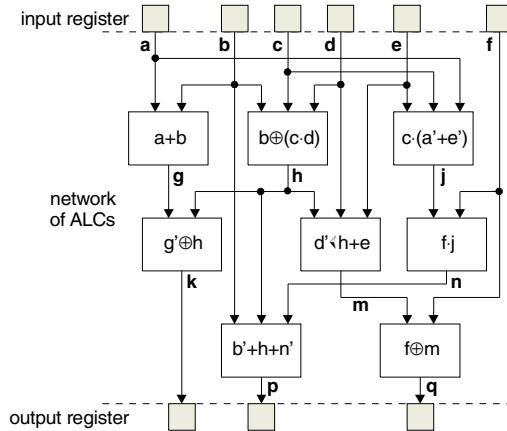


Fig. 3. Relationship between various classes of logical paths in a network of ALCs

The impact of various possible selections of the set of target faults on the values of testability measures is illustrated for the example network of ALCs shown in Fig. 4. For this network, the values of fault testability and fault coverage for the various types of testability are given in Table 1. The values of fault coverage are calculated for the specific test sequence composed of only 8 input vectors (7 input pairs) that has been developed so that to maximize the efficiency of testing; this sequence was selected out of $(2^6)^8 = 2.81 \cdot 10^{14}$ possible sequences using a combination of automatic search and manual optimization.

**Fig. 4.** Example network of ALCs

The data in Table 1 illustrate a strong impact of the class of logical paths that defines the set of target faults on the values of testability measures. For example, depending on the class of logical paths, the value of R-testability may change from 55.6% (or even from 26.0% if pseudological paths are also considered) to 100.0%, with the “exact” value (corresponding to the set of delay-essential paths) of 86.2%. Similarly, the WNR-coverage may change from 25.6% (or even from 12.0%) to 42.0%, with the “exact” value of 37.9%.

Table 1. Path delay fault testability and fault coverage for the network of Fig. 4

target logical paths	no. of paths	fault testability [%]			fault coverage [%]		
		WNR	SNR	R	WNR	SNR	R
all pseudological and logical	192	35.9	29.2	26.0	12.0	8.3	6.3
all logical	90	76.6	62.2	55.6	25.6	17.8	13.3
feasible	74	75.7	75.7	67.6	29.7	21.6	16.2
irredundant	70	80.0	80.0	71.4	31.4	22.9	17.1
delay-essential	58	96.6	96.6	86.2	37.9	27.6	20.7
WNR-testable	69	100.0	81.2	72.5	33.3	23.2	17.4
SNR-testable	56	100.0	100.0	89.3	39.3	28.6	21.4
R-testable	50	100.0	100.0	100.0	42.0	30.0	24.0

b) critical paths

An alternative to defining the set of target faults by the set of all logical paths of a certain class is to consider only critical paths in such a class, i.e. the “longest” paths, determined based on structural properties (number of components), timing simulation or some other method [15].

Most frequently, the attribute “critical” is associated with a physical path. Then, on average more logical paths are associated with a critical physical path than with a non-critical one (long paths include more ALCs than short ones). This, however, does not imply that the same relationship holds for delay-essential logical paths or other classes of logical paths that can be used to define the set of target faults. For example,

for the network of Fig. 4, under assumption that the set of critical paths includes all paths that contain 3 ALCs (4 connections), an average number of logical paths associated with a physical path is 5.0 for critical paths and 4.62 for non-critical paths, whereas an average number of delay-essential logical paths associated with a physical path is 2.33 for critical paths and 3.38 for non-critical paths. In general, if the set of target faults is restricted to critical paths, more-difficult-to-satisfy sensitization requirements lead to lower values of the testability measures.

3.4 Defining Testability Measures for Physical Paths

So far, we have assumed that all logical paths of a certain class (and the associated delay faults) equally contribute to the values of testability measures. Under such an assumption, a physical path having a large number of logical paths of a certain class has a significantly higher impact on the values of testability measures than a physical path having a small number of logical paths. If this is undesirable, an idea of defining testability measures for physical paths rather than logical paths can be considered.

The simplest way of implementing this idea is to assume that a physical path π has a certain testability-oriented feature, e.g. is delay-essential or is R-covered by some test sequence, if at least one logical path associated with π has this feature. The impact of such an assumption is illustrated in Table 2, which gives the values of fault testability and fault coverage for the network of Fig. 4 for the selected classes of physical paths. The fault coverage in Table 2 is calculated for the same test sequence as the fault coverage in Table 1. It is clearly seen that the values of testability measures calculated for physical paths (Table 2) are significantly higher than the corresponding values calculated for logical paths (Table 1).

Table 2. Testability measures for physical paths in the network of Fig. 4: simple approach

target physical paths	no. of paths	fault testability [%]			fault coverage [%]		
		WNR	SNR	R	WNR	SNR	R
all physical	19	94.7	89.5	84.2	73.7	63.2	42.1
irredundant	18	94.4	94.4	88.9	77.8	66.7	44.4
delay-essential	17	100.0	100.0	94.1	82.4	70.6	47.1
WNR-testable	18	100.0	94.4	88.9	77.8	66.7	44.4
SNR-testable	17	100.0	100.0	94.1	82.4	70.6	47.1
R-testable	16	100.0	100.0	100.0	81.3	68.8	50.0

A more sophisticated way of implementing the idea of defining testability measures for physical paths is to assume that a physical path π has a certain testability-oriented feature if all logical paths in a specific subset of $LP(\pi)$ ($LP(\pi)$ is the set of all logical paths of a certain class associated with π) have this feature. Such a subset of $LP(\pi)$, $SLP(\pi)$, must satisfy the following property: if $LP(\pi)$ includes a logical path that has a transition of a certain polarity associated with some connection c, then $SLP(\pi)$ also includes such a path. To illustrate this idea, consider path a-g-k in the network of Fig. 4. The set of delay-essential logical paths for this path is $LP(a-g-k) = \{a\uparrow g\uparrow k\uparrow, a\uparrow g\uparrow k\downarrow, a\downarrow g\downarrow k\uparrow, a\downarrow g\downarrow k\downarrow\}$. Thus, to decide whether or not a-g-k has a certain testability-oriented feature, it is sufficient to examine two logical paths: either those in

$SLP'(a-g-k) = \{a\uparrow g\uparrow k\uparrow, a\downarrow g\downarrow k\downarrow\}$ or those in $SLP''(a-g-k) = \{a\uparrow g\uparrow k\downarrow, a\downarrow g\downarrow k\uparrow\}$. The values of the testability measures for the network of Fig. 4 calculated based on the above described concept are given in Table 3 (the fault coverage is calculated for the same test sequence as the fault coverage in Table 1). It can be seen that, as expected, the values in Table 3 are significantly lower than those in Table 2.

Table 3. Testability measures for physical paths in the network of Fig. 4: more sophisticated approach

target physical paths	no. of paths	fault testability [%]			fault coverage [%]		
		WNR	SNR	R	WNR	SNR	R
all physical	19	94.7	89.5	84.2	36.8	21.1	15.8
irredundant	18	94.4	94.4	88.9	38.9	22.2	16.7
delay-essential	17	100.0	100.0	94.1	41.2	23.5	17.6
WNR-testable	18	100.0	94.4	88.9	38.9	22.2	16.7
SNR-testable	17	100.0	100.0	94.1	41.2	23.5	17.6
R-testable	16	100.0	100.0	100.0	43.8	25.0	18.8

3.5 Accounting for Restrictions on the Set of Input Pairs

The set of vector pairs that occur at the input of a given network of ALCs in normal operation is usually restricted. Such restrictions should be considered when defining the set of target faults and calculating the corresponding testability measures. This problem is discussed in more detail in [16]. Generally, it can be stated that the restrictions imposed on the set of input pairs affect the number of logical paths in most classes shown in Fig. 3. This obviously has an impact on the values of testability measures. This impact is particularly significant when the fault coverage is calculated for a test sequence that has been developed taking into account the restrictions on the input set.

4 Observations and Practical Guidelines

The results in Tables 1-3 show that for the network of Fig. 4, the values of path delay fault testability measures strongly depend on assumptions taken when calculating these values. For example, the coverage of the path delay faults, calculated for the considered test sequence under different assumptions, varies from 6.3% to 82.4%.

Our analysis of the impact of various factors on the outcome of the evaluation of path delay fault testability leads to the following observations and guidelines:

- The selection of the class of logical paths that defines the set of target faults poses an accuracy-complexity trade-off. The exact values of testability measures, corresponding to the set of delay-essential logical paths, are very difficult, if not impossible, to calculate. On the other hand, their easy-to-calculate approximations may be unacceptably inaccurate. As a compromise, the sets of target faults corresponding to irredundant and SNR-testable logical paths are recommended to provide reasonably accurate estimates of the exact values of testability measures.

- To reduce the computational load associated with the testability evaluation, one can decide to consider only critical paths. Such a decision may also be justified by a higher likelihood of affecting the speed of a device by faults associated with the longest paths. This approach would, however, lead to substantial savings only if the set of critical paths is a relatively small subset of the set of all paths, which is usually not the case for most speed-optimized circuits, for which the detection of timing-related problems is particularly important.
- As the number of physical paths is significantly lower than the number of logical paths, the idea of defining testability measures for physical paths looks attractive, even in the case when the calculation of such measures involves an examination of logical paths. Especially, the straightforward approach of assigning a testability-oriented feature to a physical path if some associated logical path has this feature would be recommended in the case when the rising and falling delays of individual components can be assumed equal or, at least, not significantly different.
- If the set of vector pairs that occur at the input of the network in normal operation is restricted, such restrictions should be considered when defining the set of target faults and evaluating the testability. This is of critical importance when the fault coverage is calculated for a test sequence that has been developed taking into account the restrictions on the input set.

5 Conclusion

We have shown that the evaluation of testability of delay faults for a user-configured programmable device is difficult both conceptually and computationally. To deal with this problem, we have developed a model of the combinational section of a programmable device (referred to as a network of ALCs) which contains a minimal number of components necessary for an analysis of path delay fault testability. We have identified all relevant factors that affect the evaluation of testability of path delay faults in a network of ALCs. An examination of the impact of these factors on the values of testability measures has resulted in several practical guidelines for the development of a procedure for the evaluation of testability of path delay faults.

The key conclusion of our study can be formulated as follows: Whenever the value of any measure of path delay fault testability for a programmable device is reported, a detailed explanation on what is reported should be given. This appears obvious, but is not. In many publications, the presented values are claimed to represent “the coverage of path delay faults by robust tests” or simply “the coverage of path delay faults”. This leads to uncertainty about the meaning of the reported results. For example, if some test procedure is claimed to cover 50% of path delay faults, this would probably mean an excellent test quality in the case when this value represents the robust coverage of delay-essential logical paths, but a rather poor test effort if it represents the coverage of robustly testable logical paths by non-robust tests.

References

1. Krstic, A., Liou, J.-J.: Delay Testing Considering Crosstalk-Induced Effects. In: Proc. IEEE Int. Test Conf. (2001) 558-567
2. Metra, C., Pagano, A., Ricco, B.: On-Line Testing of Transient and Crosstalk Faults Affecting Interconnections of FPGA-Implemented Systems. In: Proc. IEEE Int. Test Conf. (2001) 939-947
3. Abramovici, M., Stroud, C.: BIST-Based Delay-Fault Testing in FPGAs. In: Proc. IEEE Int. On-Line Testing Workshop (2002)
4. Renovell, M., Faure, P., Portal, J.M., Figueras, J., Zorian, Y.: IS-FPGA: A New Symmetric FPGA Architecture with Implicit SCAN. In: Proc. IEEE Int. Test Conf. (2001) 924-931
5. Krasniewski, A.: Application-Dependent Testing of FPGA Delay Faults. In: Proc. 25th EUROMICRO Conf., vol. 1 (1999) 260-267
6. Stroud, C., Konala, S., Chen, P., Abramovici, M.: Built-In Self-Test of Logic Blocks in FPGAs (Finally, a Free Lunch: BIST Without Overhead!). In: Proc. 14th VLSI Test Symp. (1996), 387-392
7. Harris, I.G., Menon, P.R., Tessier, R.: BIST-Based Delay Path Testing in FPGA Architectures. In: Proc. IEEE Int. Test Conf. (2001) 932-938
8. Krasniewski, A.: Testing FPGA Delay Faults: It Is Much More Complicated Than It Appears. In: Ciazynski W. et al. (eds.), Programmable Devices and Systems, Pergamon - Elsevier Science (2002) 281-286
9. Sparmann, U. et al.: Fast Identification of Robust Dependent Path Delay Faults. In: Proc. 32nd ACM/IEEE Design Automation Conf. (1995) 119-125
10. Cheng, K.-T., Chen, H.-C.: Classification and Identification of Nonrobust Untestable Path Delay Faults. IEEE Trans. on CAD, **8** (1996) 845-853
11. Krstic, A., Cheng, K.-T., Chakradhar, S.T.: Primitive Delay Faults: Identification, Testing, and Design for Testability. IEEE Trans. on CAD, **11** (1999) 669-684
12. Krasniewski, A.: Sensitization of Logical Paths in a Network of Arbitrary Logic Components: Theory and Application to Delay Fault Testing. In: Proc. IEEE Design and Diagnostics of Electronics Circuits and Systems Workshop (2003) 143-150
13. Krasniewski, A.: On the Set of Target Path Delay Faults in Sequential Subcircuits of LUT-Based FPGAs. In: Glesner, M., Zipf, P., Renovell, M. (eds.), Proc. FPL 2002, LNCS, vol. 2438, Springer Verlag (2002) 616-626
14. Underwood, B., Law, W.-O., Kang, S., Konuk, H.: Fastpath: A Path-Delay Test Generator for Standard Scan Designs. In: Proc. IEEE Int'l Test Conf. (1994) 154-163
15. Majhi, A.K., Agrawal, V.D., Jacob, J., Patnaik, L.M.: Line Coverage of Path Delay Faults. IEEE Trans. on VLSI Systems, **8**, (2000) 610-613
16. Krasniewski, A.: Evaluation of the Quality of Testing Path Delay Faults Under Restricted Input Assumption. In: Proc. IEEE Int. On-Line Testing Symp. (2003) (in printing)

Fault Simulation Using Partially Reconfigurable Hardware

A. Parreira, J.P. Teixeira, A. Pantelimon, M.B. Santos, and J.T. de Sousa

IST / INESC-ID, R. Alves Redol, 9, 1000-029 Lisboa, Portugal¹
marcelino.santos@inesc.pt

Abstract. This paper presents a fault simulation algorithm and that uses efficient partial reconfiguration of FPGAs. The methodology is particularly useful for evaluation of BIST effectiveness, and for applications in which multiple fault injection is mandatory, such as safety-critical applications. A novel fault collapsing methodology is proposed, which efficiently leads to the minimal stuck-at fault list at the look-up-tables' terminals. Fault injection is performed using local partial reconfiguration with small binary files. Our results on the ISCAS'89 sequential circuit benchmarks show that our methodology can be orders of magnitude faster than software or fully reconfigurable hardware fault simulation..

1 Introduction

Design for Testability (DfT) techniques are mandatory for cost-effective product development. Full scan based DfT techniques are very popular for sequential circuits, but these techniques impact device performance, area overhead, test application time and prevent at-speed testing, which is crucial to detecting dynamic faults [1]. To solve these problems partial scan or self-test strategies have been used, which requires testability analysis of the sequential blocks.

During the design phase, test quality may be assessed by Fault Simulation (FS). For complex devices FS may be a very costly process, especially for sequential circuits. In fact, circuit complexity, test pattern length and fault list size may lead to a large computational effort. Although many efficient algorithms have been proposed for software fault simulation (see for example [2] and [3]), for complex circuits it is still a very time-consuming task and can significantly lengthen the time-to-market.

FS can be implemented in software or hardware [1]. The ease of developing software tools for FS (taking advantage of the flexibility of software programming) made software simulation the most current approach. However, the recent advent of very complex programmable logic devices, namely Field Programmable Gate Arrays (FPGAs), created an opportunity for Hardware Fault Simulation (HFS), which may be an attractive solution for at least a subset of practical situations. In fact, evaluating Built-In Self-Test (BIST) effectiveness may require a heavy computational effort in fault simulation. Long test sessions are needed to evaluate systems composed of several modules that have to be tested simultaneously in order to evaluate aliasing.

¹ This work has been partially funded by FCT (Portugal), POCTI/ESE41788/2001.

One example is functional BIST in a System on a Chip (SoC) with data compression cores being used as part of the signature compression logic. Another fault simulation task not efficiently performed by software tools is *multiple* fault simulation, mainly because of the enormous number of possible fault combinations. Unfortunately, multiple fault simulation may become mandatory, namely in the certification process of safety-critical applications [4].

Different HFS approaches using FPGAs have been proposed in the literature to overcome the difficulties with Software Fault Simulation (SFS) for sequential circuits. *Dynamic* fault injection using dedicated extra hardware, and allowing the injection of different faults without reconfiguring the FPGA, was proposed in [5-8]. The additional hardware proposed for implementing dynamic fault injection uses a Shift Register (SR) whose length corresponds to the size of the fault list. Each fault is injected when the corresponding position in the SR has logic “1”, while all other positions have logic “0”. Upon initialization, only the first position of the SR is set to “1”. Then the “1” is shifted along the SR, activating one fault at a time. This technique was further optimized in [9]. However, a major limitation of this technique is the fact that the added hardware increases with the number of faults to inject, which limits the size of the circuits that can be simulated. In [6], it is also reported that some parallelism is possible by injecting independent faults at the same time. Unfortunately, the speedup reported was of only 1.36 times faster than serial fault simulation. In [10], a new approach that included a backward propagation network to allow critical path tracing [11] is proposed. This information allows multiple faults to be simulated for each test vector, but also requires heavy extra hardware and only combinational circuits have been analyzed.

A serial fault simulation technique that required only partial reconfiguration during logic emulation was proposed in [12], showing that no extra logic need be added for fault injection purposes, and that HFS can be about two orders of magnitude faster than SFS for designs over 100,000 gates. More recently, other hardware fault injection approaches were proposed in [13-15] using the JBITS [16] interface for partial FPGA reconfiguration. The JBITS software can achieve effective injection of faults on Look-Up-Tables (LUTs) [13] [14] or erroneous register values [15]. These works do not report any results on partial FPGA reconfiguration times for fault injection - only some predictions are given. Moreover, the approach that uses JBITS requires the Java SDK 1.2.2 [17] platform and the XHWIF [18] hardware interface.

The purpose of this paper is to present a Reconfigurable Hardware Fault Simulation (RHFS) methodology that uses partial reconfiguration to implement a novel, highly efficient hardware fault simulation tool, which we called **f's**. We show that partial FPGA reconfiguration can be accomplished very efficiently by deriving very small bit files for fault injection. These bit files are obtained from the full configuration bit file by means of its direct manipulation, without requiring any additional software tools. Moreover, the new RHFS tool does not require the use any specific hardware interface with the FPGA.

This paper is organized as follows. In section 2 the LUT extraction from the bit file is explained and the fault collapsing technique is proposed. In section 3 the new fault simulation tool, **f's**, is presented. Section 4 presents experimental results that compare SFS and HFS to the new RHFS approach using two ISCAS'89 benchmark circuits [19]. Finally, section 5 concludes the paper.

2 LUT Extraction, Fault Injection and Collapsing

Virtex FPGAs [20] are essentially composed of an array of Configurable Logic Blocks (CLBs) surrounded by a ring of I/O blocks. The CLBs are the building blocks for implementing custom logic. Each CLB contains two slices. Each slice contains two 4 input Look-Up-Tables (LUTs), 2 flip-flops and some carry logic. For the fault model used in the next section, only LUT extraction is required. Each LUT can be used to implement a function of 0, 1, 2, 3 or 4 inputs. It is important to identify the number of inputs relevant for each used LUT, in order to include the corresponding faults in the fault list. Since each LUT has 4 inputs, the LUT contents consist of a 16-bit vector, one vector for each combination of the inputs. Let y_{abcd} be the output value for the combination of input values $abcd$. The LUT configuration 16-bit vector can be denoted $y_{0000}, y_{0001}, y_{0010}, \dots, y_{1111}$, where each bit position corresponds to the LUT output value for the respective combination of the inputs i_3, i_2, i_1 and i_0 . If one input has a fixed value, then an 8 bit vector is obtained. For instance, if we have always $i_3=0$, then the relevant 8 bit vector is $y_{0000}, y_{0001}, y_{0010}, y_{0011}, y_{0100}, y_{0101}, y_{0110}, y_{0111}$.

The only circuit information required for our RHFS method is the configuration bit file of the FPGA. After retrieving the information of each LUT from the bit file [21], the relevance of each input i_x ($x=0, 1, 2$ and 3) is evaluated comparing the 8-bit vectors corresponding to $i_x=0$ and $i_x=1$. If these two vectors are identical then the input i_x is not active. This is how we extract the LUT types, LUT2, LUT3, LUT4, etc., according to their number of relevant inputs.

2.1 Line Stuck-AT Fault Injection and Collapsing

The most popular fault model for digital circuit fault simulation is the single Line-Stuck-At (LSA) fault model. For the Virtex FPGAs, RHFS injects LSA faults in the outputs and inputs of LUTs. After extracting the number of active inputs in each LUT, the LSA fault list is created. Each LUT has two LSA faults (LSA-0 and LSA-1) per input or output. Thus, the total number of faults is 2, 4, 6, 8 and 10 for LUTs with 0, 1, 2, 3 or 4 active inputs, respectively. The required data for each fault injection consists of the 16-bit LUT configuration vector, modified in order to inject the fault behavior.

In order to inject a LSA- v fault ($v = 0$ or $v = 1$) at the output of a LUT, all 16 bit positions of the LUT reconfiguration vector must be set to the logic value v . The reconfiguration vector that corresponds to the LUT input a stuck at value v is obtained by copying the values y_{vbcd} to y_{wbcd} for each bcd combination. For instance, the vector for the fault input a LSA-0 is obtained by copying y_{0000} to y_{1000}, y_{0001} to y_{1001}, y_{0110} to $y_{1110}, \dots, y_{0111}$ to y_{1111} .

After computing the 16-bit reconfiguration vectors for each fault, fault collapsing can be performed efficiently by grouping the faults with identical reconfiguration vectors. To illustrate this, consider the LUT functionality $F=a.b+c.d$. Table 1 shows the configuration vectors for the fault free LUT (F) and the configuration vectors that model all LSA faults in the LUT (i/v means line i stuck-at value v). The LUT input combination that corresponds to each vector position (row) is given in the four left most columns of Table 1. Analyzing this table, we notice that two fault pairs can be collapsed using the methodology presented: $(a/0, b/0)$ and $(c/0, d/0)$. The shaded cells

in the table indicate the LUT input combinations (or LUT test vectors) that activate the fault given in the respective column.

Table 1. Fault reconfiguration vectors for LSA faults of a LUT implementing $F=a.b+c.d$.

a	b	C	d	F	a/0	b/0	c/0	d/0	a/1	b/1	c/1	d/1	F/1	F/0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	1	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	1	1	0
0	0	1	1	1	1	1	0	0	1	1	1	1	1	0
0	1	0	0	0	0	0	0	0	1	0	0	0	1	0
0	1	0	1	0	0	0	0	0	1	0	1	0	1	0
0	1	1	0	0	0	0	0	0	1	0	0	1	1	0
0	1	1	1	1	1	1	0	0	1	1	1	1	1	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1	0
1	0	0	1	0	0	0	0	0	0	1	1	0	1	0
1	0	1	0	0	0	0	0	0	0	1	0	1	1	0
1	0	1	1	1	1	1	0	0	1	1	1	1	1	0
1	1	0	0	1	0	0	1	1	1	1	1	1	1	0
1	1	0	1	1	0	0	1	1	1	1	1	1	1	0
1	1	1	0	1	0	0	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	0

3 Reconfigurable Hardware Fault Simulation

A new tool for FPGA based fault simulation (**f's**) has been developed. One great advantage of **f's** is the fact that it takes as input solely the binary file with the complete FPGA configuration. From this file, **f's** identifies the target device for which the mapping was performed. This identification is mandatory in order to know the number of rows and columns for LUT bit positioning. LUT extraction is carried out as described in section 2.1 and the fault list is collapsed as described in section 2.2. After fault collapsing, **f's** creates a report with all the LUTs in use, the original fault list, the collapsed fault list and the estimated reconfiguration times for different hardware interfaces. Fig. 1 shows this report for the s5378 ISCAS'89 benchmark circuit. Next, all partial reconfiguration bit files are generated in sequence for each fault. In this way, each bit file reprograms only the minimum number of frames (smallest amount of data that can be read or written with a single command) required to eject the previous fault and inject the next one. Both single and multiple fault injection are supported.

Debugging and validation of the tool was carried out using a JTAG 200 Kb parallel III port. The experimental values obtained for the reconfiguration times matched the estimated values. Since the current version of the tool does not support any type of external test vector application, it is more suitable for BIST quality evaluation purposes. However, the methodology can be easily extended to apply external vectors supplied from an internal (FPGAs have internal RAM blocks) or external memory.

```
BIT_FILE_NAME: = s5378.ncd
BIT_PART_STRING: = v2000ebg560
BIT_DATE_STRING: = 2003/01/16
BIT_TIME_STRING: = 18:01:59
BIT_IMAGE: size 1269956
```

FAULTS STACK AT ZERO/ONE at LUT's INPUTS / OUTPUTS / SELECT RAM BITS

LUT -- LOCATION , TYPE , CONTENTS AND FAULT TYPE TO INJECT (equivalent faults , between – and separated by /)

```
row=49 col=19 slice=1 lut=G type=2 vector=0 0 0 0 0 1 1 0 0 0 0 0 0 1 1
faults= -- O_S_1/I1_S_0/I2_S_0 – O_S_0 – I2_S_1 – I1_S_1
```

...

```
...
row=55 col=73 slice=0 lut=G type=4 vector=0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
faults= -- O_S_1/I0_S_0/I1_S_0/I2_S_0/I3_S_1 – O_S_0 – I1_S_1 – I2_S_1
– I0_S_1 – I3_S_0
```

lut capacity	38400	
lut not used	37996	98.94791666666667
fauls to detect	3610	
fauls to inject	2707	0.749861495844875
lut type 1	0	0
lut type 2	66	0.171875
lut type 3	83	0.2161458333333333
lut type 4	255	0.6640625

TIME (hour:min:sec) to program and inject 0:6:51 (based on 200 Kb parallel III port speed – jtag)

TIME (hour:min:sec) to program and inject 0:0:16 (based on 5 Mb parallel IV port speed – jtag)

TIME (hour:min:sec) to program and inject 0:0:1 (based on 66 Mb usb port speed – serial)

Fig. 1. Fault list report from f's

4 Experimental Results

Software, Hardware and Reconfigurable Hardware Fault Simulation are compared in this section. The focus is to determine the trade-offs associated with the fundamentals of the three approaches: SFS costs increase steadily with circuit complexity, test length and fault list size, especially for sequential circuits. HFS has a significant initial cost, associated with FPGA synthesis and implementation, and a low cost in the hardware emulation process itself. Finally, RHFS represents a trade-off between the previous two approaches. Because reconfiguration is used in fault injection, hardware mechanisms for fault injection need not be added to the original circuit. This gives rise to a much simpler circuit that can be compiled faster. On the other the fault replacements by means of partial reconfiguration will take more time than simply shifting the fault register as in HFS. Thus the RHFS fault simulation times increase faster with the number of vectors when compared to HFS, but slower when compared to SFS.

In order to compare the three approaches, two ISCAS'89 circuits, s5378 and s13207, have been used as test vehicles. The complexity of these circuits and their fault list sizes before and after collapsing are shown in Table 2. From the analysis of this table, it is clear that the original fault list of LSAs at LUT terminals is efficiently collapsed by the tool. Table 3 shows the LSA fault collapsing results for the s5378 circuit obtained with the f's tool in terms of the distribution of the faults among the various types of extracted LUTs. Pseudo-random test vectors have been used in the experiments. The vectors are generated internally in the FPGA using an LFSR, as it is common for BIST. Since we are mainly interested in measuring the fault simulation time, we have not obtained fault coverage values. However fault coverage can be obtained by emulating the fault free circuit, saving the response sequences and later comparing them to the output response sequences of the faulty circuits. Work in this direction is in progress.

Table 2. LUT usage for a Virtex XCV2000E FPGA and corresponding fault list sizes for the two ISCAS'89 sequential circuit benchmarks.

Benchmark Circuit	#LUTs used	#LUTs unused	#LSA faults before collapsing	#LSA faults after collapsing
s5378	404	37996	3610	2707
s13207	795	37605	7126	5434

Table 3. Collapsed fault distribution over LUT types for the s5378 benchmark circuit.

LUT Type	# faults						
	4	5	6	7	8	9	10
LUT2	63		3				
LUT3		40	24	7	14		
LUT4			54	47	118	11	24

Software fault simulation has been carried out using a widely used commercial fault simulation tool running on a Sun ultra10/440 workstation of 1 GB of RAM capacity. The SFS time, t_{SFS} , is computed by the commercial EDA tool. The fault simulation time, t_{HFS} , for the HFS or RHFS approaches is computed by the following expression:

$$t_{HFS} = t_{comp} + t_{reconf} + \frac{\# faults \times \# vectors}{f_{HFS}}$$

where t_{comp} is the sum of the FPGA synthesis and configuration times, t_{reconf} is the time required for reconfiguration (which is zero in the case of HFS) and f_{HFS} is the frequency of vector hardware application. The value of t_{comp} is obtained after synthesis and mapping of the circuit using the Xilinx Synthesis Tool (XST). The **f's** tool analyses the bit stream and computes all reconfiguration frames for each fault. According to the number of generated reconfiguration frames, it reports the value of t_{reconf} for different hardware interfaces. Figs. 3 and 4 show the fault simulation times for the two ISCAS'89 benchmark circuits, s5378 and s13207, respectively, for $f_{HFS} = 50\text{MHz}$, using a Celoxica RC1000-PP board containing a Xilinx Virtex 2000E FPGA. Note that, unlike in SFS, in HFS or RHFS no fault dropping is available since all faults are simulated in sequence. The curve marked "RHFS (200 Kb)" refers to partial reconfiguration using a JTAG 200 Kb parallel III port. The curve marked "RHFS (5Mb)" refers to partial reconfiguration using a JTAG 5 Mb parallel IV port.

From these results it is clear that HFS (hardware fault simulation with a unique configuration file) is advantageous over SFS only when the number of test vectors exceeds a certain value that mainly depends on the FPGA compilation time t_{comp} [9]. Once the FPGA is compiled, the cost of applying test vectors is very small compared to SFS, despite the fact that the slope of t_{SFS} depends on the computational resources of the machine that is running it. On the other hand, RHFS (hardware fault simulation using partial reconfiguration files for fault replacement) is faster than SFS even for a small number of test vectors. This results from the fact that $t_{comp} + t_{recomp}$ for RHFS is much lower than t_{comp} for the HFS approach. In the final version of this paper, results comparing RHFS, HFS and SFS times for a larger set of benchmark circuits will be presented, which should also include larger circuits like the ones reported in [22].

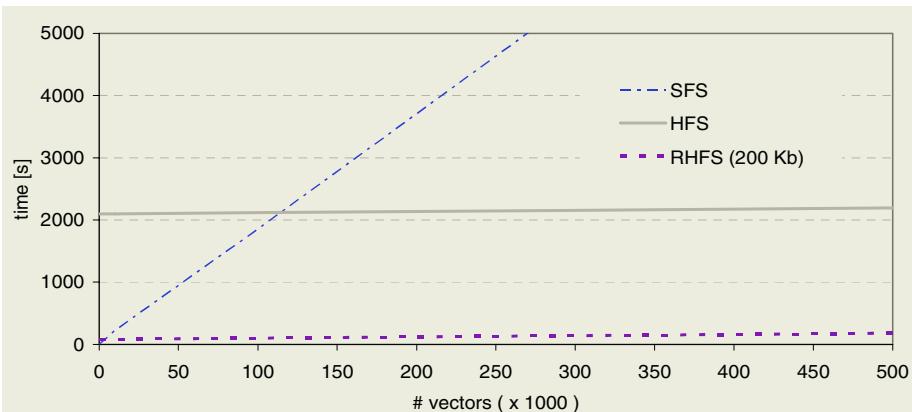


Fig. 2. Reconfigurable HW, fixed HW and SW fault simulation times for the s5378 benchmark circuit.

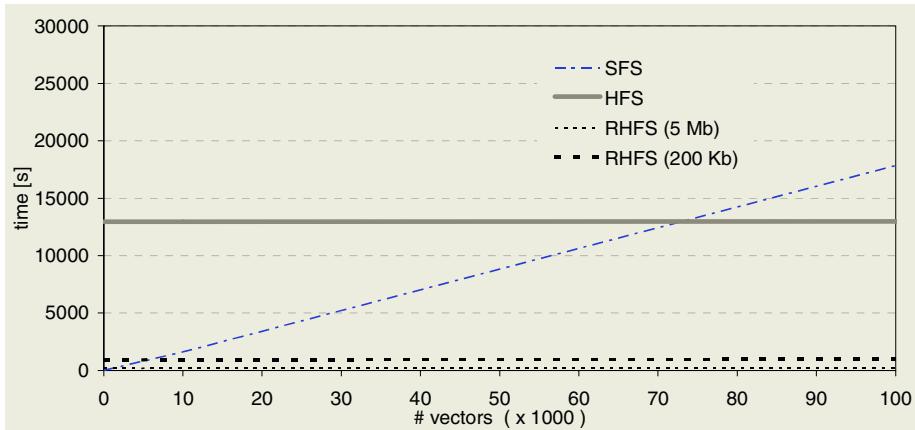


Fig. 3. Reconfigurable HW, fixed HW and SW fault simulation times for the s13207 benchmark circuit

Conclusions

In this work we have presented a novel approach to fault simulation using partial reconfiguration of FPGAs for fault replacement (RHFS). The approach has been compared to traditional software fault simulation (SFS), and to previous fixed hardware fault simulation (HFS). The superiority of the new approach has been demonstrated on two ISCAS'89 benchmark circuits.

Compared to software fault simulation, we found that RHFS is faster than SFS, even for a limited number of test vectors. After performing FPGA compilation during some tens of minutes, the simulation time for RHFS increases linearly with the number of test vectors at a rate that, for a 50MHz clock frequency, is two orders of magnitude lower than the increase rate observed for SFS.

Our fault simulation methodology uses the LSA fault model at LUT terminals. We presented a fault collapsing technique based on comparing the LUT reconfiguration vectors for each fault that leads to an efficiently optimized fault list.

Our RHFS methodology was embodied in a software tool, **f's**, which takes as unique input the binary file for FPGA configuration. This is a significant advantage over other partial reconfiguration techniques that depend on a variety of file formats and tools. When processing the bit file, **f's**, identifies the target FPGA device, extracts the LUTs in use and their types, and generates minimal reconfiguration bit files for fault injection. Current work is oriented towards supporting external vectors, automation of vector formatting and fault coverage evaluation. The goal is to make **f's** a competitive fault simulation tool.

References

- [1] M.L. Bushnel, V.D. Agrawal, "Essentials of Electronic Testing for Digital Memory and Mixed-Signal VLSI Circuits", Kluwer Academic Pubs., 2000.
- [2] T.M. Niermann, W. T. Cheng, J. H. Patel, "PROFS: A fast, memory-efficient sequential circuit fault simulator", IEEE Trans. Computer-Aided Design, pp.198-207, 1992.
- [3] E.M. Rudnick, J.H. Patel, "Overcoming the Serial Logic Simulation Bottleneck in Parallel Fault Simulation", Proc. of the IEEE International Test Conference (ITC), pp. 495-501, 1997.
- [4] F.M. Gonçalves, M.B. Santos, I.C. Teixeira and J.P. Teixeira, "Design and Test of Certifiable ASICs for Safety-critical Gas Burners Control", Proc. of the 7th. IEEE Int. On-Line Testing Workshop (IOLTW), pp. 197-201, July, 2001.
- [5] R.W.Wieler, Z. Zhang, R. D. McLeod, "Simulating static and dynamic faults in BIST structures with a FPGA based emulator", Proc. of IEEE Int. Workshop of Field-Programmable Logic and Application, pp. 240-250, 1994.
- [6] K. Cheng, S. Huang, W. Dai, "Fault Emulation: A New Methodology for Fault Grading", IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 10, pp1487-1495, October 1999.
- [7] Shih-Arn Hwang, Jin-Hua Hong and Cheng-Wen Wu, "Sequential Circuit Fault Simulation Using Logic Emulation", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 8, pp. 724-736, August 1998.
- [8] P.Civera, L.Macchiarulo, M.Rebaudengo, M.Reorda, M.Violante, "An FPGA-based approach for speeding-up Fault Injection campaigns on safety-critical circuits", IEEE Journal of Electronic Testing Theory and Applications, vol. 18, no.3, pp. 261-271, June 2002.
- [9] M.B. Santos, J. Braga, I. M. Teixeira, J. P. Teixeira, "Dynamic Fault Injection Optimization for FPGA-Based Harware Fault Simulation", Proc. of the Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS), pp. 370-373, April, 2002.
- [10] Miron Abramovici, Prem Menon, "Fault Simulation on Reconfigurable Hardware" , *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 182-190, 1997.
- [11] M. Abramovici, P. R. Menon, D. T. Miller, "Critical Path Tracing: An Alternative to Fault Simulation", IEEE Design Automation Conference, pp. 468 - 474 , 1984.
- [12] L. Burgun, F. Reblewski, G. Fenelon, J. Barbier, O. Lepape, "Serial fault simulation", Proc. Design Auomation Conference, pp. 801-806, 1996.
- [13] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp.405-413, October 2000.
- [14] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection Applications", IEEE Instrumentation and Measurement Technology Conference, vol. 3, pp.1773-1777, May 2001.
- [15] L.Antoni, R. Leveugle, B. Fehér, "Using Run-Time Reconfiguration for Fault Injection in Hardware Prototypes", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 245-253, 2002.
- [16] S. Guccione, D. Levi, P.Sundararajan, "Jbits: A Java-based Interface for Reconfigurable Computing", Proc. of the 2nd Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD), pp. 27, 1999.
- [17] E. Lechner, S. Guccione, "The Java Environment for Reconfigurable Computing", Proc. of the 7th International Workshop on Field-Programmable Logic and Applications (FPL), Lecture Notes in Computer Science 1304, pp.284-293, September 1997.

- [18] P.Sundararajan, S.Guccione, D.Levi, "XHWIF: A portable hardware interface for reconfigurable computing", Proc. of Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, SPIE 4525, pp.97-102, August 2001.
- [19] F. Brglez, D. Bryan, K. Kominski, "Combinational Profiles of Sequential Benchmark Circuits", Proc. Int. Symp. on Circuits and Systems (ISCAS), pp. 1229-34, 1989.
- [20] Xilinx Inc., "Virtex-E 1.8V Field Programmable Gate Arrays", Xilinx DS022, 2001.
- [21] Xilinx Inc., "Virtex Series Configuration Architecture User Guide", Application Note: Virtex Series, XAPP151 (v1.5), September 27, 2000.
- [22] H. Cha, E. Rudnick, J.Patel, R. Iyer, G.Shoi, "A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults", IEEE Transactions on Computers, vol. 45, no.11, pp1248-1256, November 1996.

Switch Level Fault Emulation

Seyed Ghassem Miremadi and Alireza Ejlali

Sharif University of Technology, Department of Computer Engineering

Azadi Ave., Tehran, Iran
miremadi@sharif.edu
ejlali@ce.sharif.edu

Abstract. The switch level is an abstraction level between the gate level and the electrical level, offers many advantages. Switch level simulators can reliably model many important phenomena in CMOS circuits, such as bi-directional signal propagation, charge sharing and variations in driving strength. However, the fault simulation of switch level models is more time-consuming than gate level models. This paper presents a method for fast fault emulation of switch level circuits using FPGA chips. In this method, gates model switch level circuits and we can emulate mixed gate-switch level models. By the use of this method, FPGA chips can be used to accelerate the fault injection campaigns into switch level models.

1 Introduction

Logic emulation systems [1], [2] are now commercially available for fast prototyping, real-time operation, and logic verification. In many cases emulation is about 10^3 to 10^6 faster than simulation [3]. A logic emulator can implement a synthesizable model on a board composed of dozens of field programmable gate array (FPGA) chips. It cannot be used to emulate switch level models. However, in this paper, we present a method for modeling switch level circuits by gate level models. The resulted gate level model may be used as a starting point for synthesis into an interconnection of logic cells or FPGA layouts. In contrast to most cases for which transistors are grouped together to form gates, in this method, gates are grouped together to form switch models of transistors.

The advantages of this method are:

1. Use of FPGAs for fast emulation of CMOS switch level or mixed switch-gate level circuits.
2. Fault emulation [4] of switch level fault models such as transistor-stuck-on and transistor-stuck-off faults.
3. Emulation of three-state CMOS gates by the use of FPGAs, which do not support three-state logic directly.

Section 2 presents a brief review of switch level models. Section 3 presents the method for modeling switch level circuits by logic gates. Section 4 describes how the presented model can be used in switch level fault injection. Section 5 describes the mixed-mode approach, which is used for emulating CMOS circuits. In section 6, the

experimental evaluation of the presented method is discussed. Finally, the conclusion is in Section 7.

2 A Review of Switch Level Models

The switch level is an abstraction level between the gate level and the electrical level, offers many advantages. By operating directly on the transistor network, switch level simulators can reliably model many important phenomena in CMOS circuits, such as bi-directional signal propagation, charge sharing and variations in driving strength [5][6].

On the switch level, transistors are viewed as switches and the circuit state is described by discrete values, consisting of a logic state (e.g. 0, 1, U - unknown, Z - high impedance) and a strength (e.g. weak, strong, medium, ...) [7][8].

Different number of strength levels may be used in switch level models. In some methods, only two levels of strengths are used (e.g. weak and strong) [7], while others use more levels. For example, Verilog is a hardware description language, which supports switch level modeling with eight different levels of strength [8].

Each switch is a bi-directional element, which is controlled by the gate terminal G. A point at which two or more transistors have a common connection is called a node. A node may be driven by signals, which have different strengths. If a node is driven with the driver signals S_1, S_2, \dots, S_n , the value (i.e. the logic state and the strength) of the node is equal to the value of the strongest driver signal. If two driver signals, with same strengths and different logic states, have the highest strength, the logic state of the node will be U (unknown).

In some methods, resistive switches are used. These switches reduce the strength of signals that propagate through them. For instance, in Verilog HDL both resistive and non-resistive switches can be used [8].

3 Switch Level Modeling Using Gate Level Circuits

We present in this section a method for modeling switch level circuits by logic gates. In this method, each transistor switch and each node is replaced by a corresponding gate level circuit. These circuits model the behavior of transistor switches and nodes. Figure 1b shows the gate level circuit, which models an NMOS transistor, and Figure 1d shows another gate level circuit, which models a node. We will refer to these circuits as *pseudo transistor* and *pseudo node* respectively.

Drain and source terminals of a MOS transistor switch are bi-directional connections, but all wires in a gate level circuit are uni-directional, so two connections with different directions are used for each of the drain and source terminals as shown in Figures 1b and 1d.

As an illustration, we will show how pseudo transistors and pseudo nodes can be used for modeling a CMOS NAND gate. Figure 2 shows a CMOS NAND gate and its counterpart gate level circuit, which is constructed with pseudo transistors and pseudo nodes.

Each terminal of a pseudo transistor or a pseudo node consists of K wires, which transfer K-bit codes between pseudo transistors and pseudo nodes. For example, S1 terminal of the pseudo transistor shown in Figure 1b consists of K wires.

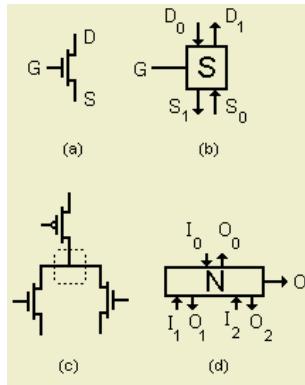


Fig. 1. (a) An NMOS switch, (b) The pseudo transistor,(c) A node, (d) The psudo node.

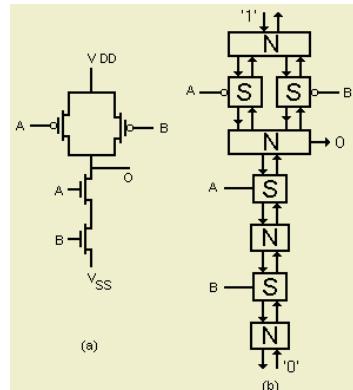


Fig. 2. A CMOS NAND gate (a) The CMOS Circuit, (b) The gate level model

The K-bit code is used for coding different logic states and strengths. This K-bit code is divided into two parts: state code and strength code. State code is a 2-bit code, which codes the logic state of the terminal, while strength code is an n-bit code, which codes the strength level of it. The width of the strength code depends on the number of strength levels.

In this paper, strengths are denoted by L_i, where i identifies the strength level. For example, if four different strength levels are used, the possible strengths will be L₀, L₁, L₂, and L₃, where L₀ is the weakest one and L₃ is the strongest one.

The logic states, which are commonly used in switch level simulations, are '1', '0', 'U' and 'Z' [7]. Here, only '1', '0' and 'U' states are coded by state codes and 'Z' state is indicated by the strength L₀.

3.1 The Operation of the Pseudo Transistors

Let L(X) be the logic state of the terminal or wire X and let S(X) be its strength. The pseudo transistor, shown in Figure 1b, is a combinational circuit, which operates according to the following rules:

```

1. If L(G)=0 and S(G)>L0 then
begin
  S(D1)=L0;
  S(S1)=L0;
  L(D1)= don't care;
  L(S1)= don't care;
end;
```

Rule 1 states that when a pseudo transistor is off, it passes the binary code of 'Z' value to pseudo node circuits which are connected to it indicating that the transistor does not take part in resolving the value of the nodes. Note that 'Z' value is represented by strength L0

```
2. If L(G)=1 and S(G)>L0 then
begin
    L(D1)=L(S0) ;
    L(S1)=L(D0) ;
    S(D1)=S(S0) ;
    S(S1)=S(D0) ;
end;
```

Rule 2 states that when a pseudo transistor is on, it passes the values through itself.

```
3. If L(G)=U or S(G)=L0 then
beign
    L(D1)=U;
    L(S1)=U;
    S(D1)=S(S0) ;
    S(S1)=S(D0) ;
end;
```

Rule 3 states that when the gate value is unknown or high impedance, there is no way to determine whether the transistor switch is on or off. Therefore, drain and source values will be unknown. Since the transistor switch may be on or off, it may pass strengths through itself or not. In this case, by considering the worst case it is supposed that strengths pass through the transistor switch.

These three rules model the behavior of NMOS switches. The corresponding rules for PMOS switches can be easily obtained from the above rules by interchanging 'L(G)=1' and 'L(G)=0'.

3.2 The Operation of the Pseudo Nodes

When all of the MOS transistors, which are connected to a node are off, the node retains its previous value. This is caused by the charge stored on the stray capacitance C associated with the node. For this reason, a sequential circuit is used as a pseudo node. Memory elements (flip-flops) are used for modeling gate capacitance of MOS transistors or capacitance of wires, which can store signal values.

It should be noted that the stored values are weak values because they result from small charges, which have not the current drive capability [8]. Therefore, in this paper, the strength L1 is assigned to each stored value. The strength L0 can not be used for this purpose, because it indicates the 'Z' state.

Figure 3 shows the internal structure of a pseudo node. In this figure, the output O is called the main output of the node and the outputs O_i are called returned outputs.

The boxes labeled "Resolution circuit" and "Attenuator" in Figure 3 are combinational circuits, while box labeled "Pseudo capacitor" is a sequential circuit.

As mentioned previously, if a node is driven with the driver signals I_1, I_2, \dots, I_n , the value of the node is equal to the value of the strongest driver signal, and if two driver

signals, with same strengths and different logic states, have the highest strength, the logic state of the node will be U (unknown). This is the way a resolution circuit resolves a single value from multiple driving values.

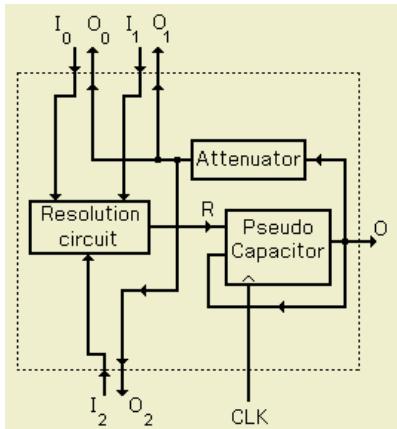


Fig. 3. The internal structure of a pseudo node

Table 1. The state table of the pseudo capacitor circuit

Present State	Next State						
	$R=U_n$	$R=1_n$	$R=0_n$	$R=U_1$	$R=1_1$	$R=0_1$	$R=Z$
U_m	U_n	1_n	0_n	U_1	U_1	U_1	U_1
1_m	U_n	1_n	0_n	U_1	1_1	U_1	1_1
0_m	U_n	1_n	0_n	U_1	U_1	0_1	0_1
U_1	U_n	1_n	0_n	U_1	U_1	U_1	U_1
1_1	U_n	1_n	0_n	U_1	1_1	U_1	1_1
0_1	U_n	1_n	0_n	U_1	U_1	0_1	0_1
Z	U_n	1_n	0_n	U_1	1_1	0_1	Z

$n > 1, m > 1$

A pseudo capacitor models the behavior of the parasitic capacitance associated with the node. If we consider the state of this sequential circuit to be its output, its operation can be summarized in Table 1 as a state table.

By considering the operation of a node capacitance in a switch level circuit, the derivation of this state table is carried out most easily. For example, according to the first three columns of the table (columns labeled $R=U_n$, $R=1_n$ and $R=0_n$), when the value of the input R have the strength greater than $L1$, the output O will be equal to input R . This is because the values stored in a node capacitance can be overridden by stronger values supplied by turned-on transistor switches. As another example, consider the row 'Present State= 1_m ' and the column ' $R=0_1$ ', in this case the next state will be ' U_1 '. This is because a ' 1_1 ' value is stored in the stray capacitance because of the previous ' 1_m ' value of the node. When the transistor switches connected to the node supply the weak value ' $R=0_1$ ' to the node, the result is ' U_1 '. Note that the input R of the pseudo capacitor circuit is the output of the resolution circuit, therefore its value is determined by all the pseudo transistors, which are connected to the node.

The combinational circuit labeled "Attenuator", reduces the strength of the output O , and transfers the result to all the transistor switches, which are connected to the node. It should be noted that this circuit does not change the logic state of the output O . Also it does not change the strength of the output O if its strength is $L0$ (i.e. ' Z ' state). The benefits of the attenuator circuit are as follows:

- This circuit ensures that a value with the strength greater than $L1$ can not be stored in pseudo nodes. Note that when two pseudo nodes are connected with a turned on pseudo transistor switch, they send their values for each other at each clock pulse. In this case the attenuator circuits of these nodes ensure that strong values can not loop between these two pseudo nodes. They attenuate such looping values, therefore only weak values with strength $L0$ can be stored in such loops.

- By the use of this circuit, resistive transistor switches can be modeled, because the strength of a signal is reduced by 1, whenever it propagates from a node to another node through a transistor switch.

Since pseudo nodes are sequential circuits, when a combinational circuit, such as the NAND gate shown in Figure 2a, is modeled using the presented model (See Figure 2b), a sequential circuit is resulted.

If one wants to model a CMOS sequential circuit by this method, two different clock signals will be needed. One clock signal is used for the modeled sequential circuit itself and the other is used for pseudo nodes. The former is called main clock, while the latter is called emulation clock.

In this paper, only combinational circuits have been considered for fault emulation, therefore only one clock signal (emulation clock) is used.

4 Fault Injection at the Switch Level

Some of the advantages of the fault injection at the switch level over the fault injection at the gate level are as follows:

- Many regions are not reachable for gate level fault injection, as they are internal to the gates. But real defects may occur inside a gate.
- There are some defects, which can not be modeled by gate level stuck-at faults.

This section describes how the presented model can be used for injecting transistor-stuck fault models into switch level circuits.

A transistor is stuck-on if it remains on regardless of its gate value. In a similar manner, a transistor is stuck-off if it remains off regardless of its gate value.

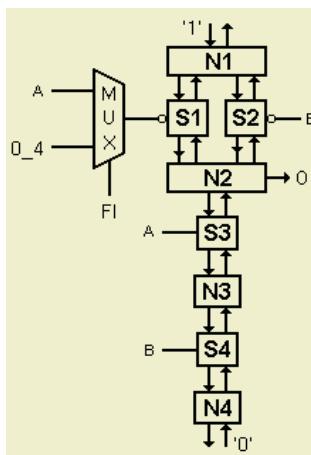


Fig. 4. Injection of transistor-stuck-on fault

Figure 4 shows how a transistor-stuck fault can be injected into the switch level model of a CMOS NAND gate (Shown in figure 2) using the presented gate level model. The signal "FI" in the Figure 4 is a fault injector signal, which is used for

activating a fault. This signal is the selection line of a 2-line to 1-line multiplexer. When the input A is selected, the circuit is fault free, and when the input 0_4 (with the logic state 0 and the strength L4) is selected a transistor-stuck-on fault is injected because, in this case, the pseudo transistor "S1" is on regardless of the value of A.

It is not necessary to choose the value 0_4 in order to turn on the pseudo transistor "S1". In fact, any value with the logic state 0 and the strength greater than L0 can be used to turn on the pseudo transistor. Note that each input and output of the multiplexer consists of K wires, which transfer K-bit codes.

5 Mixed-Mode Emulation

If the presented model is used for emulating a switch level circuit using FPGAs, a large number of FPGA resources may be consumed. In order to attain a better utilization of FPGA resources a mixed-mode emulation approach has been used. In this approach parts of the circuit are emulated at the gate level while the rest of the circuit (such as faulty portions of the circuit) is emulated at the switch level. This method is similar to the method, which is used in mixed-mode fault simulators [9]. The difference is that the presented mixed-mode method has been used to address the FPGA resources utilization problem while the mixed-mode simulators have been used to address the time explosion problem.

[4] discusses a dual-railed logic which is used for emulating gate level circuits. The idea is to use two wires to represent a three-valued logic signal. In a similar manner, in this paper a dual-railed logic with three-valued logic signals (0, 1 and U) is used for gate level parts of the mixed-model. This is because switch level parts have logic values such as "U" and "Z" which cannot propagate through two-valued logic gates.

Whenever a signal propagates from the switch level parts to the gate level parts of the circuit only its state code is transferred and its strength code is omitted. As mentioned in Section 3, state code is a 2-bit code, which codes the logic state of a signal. Therefore, it can be applied to dual-railed logic gates.

Figure 5 shows how a signal propagates from the switch level portion of a mixed-model to its gate level portion.

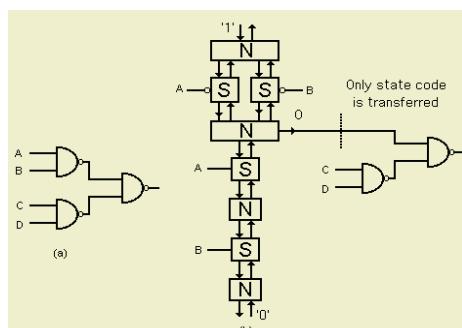


Fig. 5. mixed-mode switch-gate level emulation (a) a gate level circuit (b) its mixed-mode model for emulating

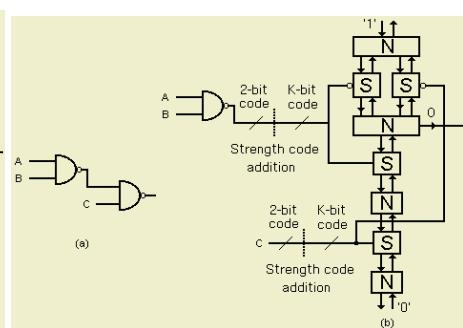


Fig. 6. mixed-mode switch-gate level emulation (a) a gate level circuit (b) its mixed-mode model for emulating

When a signal propagates from the gate level portion of a circuit to its switch level portion, only a strength code with the strength greater than L0 is added to the propagated signal.

Figure 6 shows how a signal propagates from the gate level portion of a mixed-model to its switch level portion. In most cases, signals, which propagate from gate level parts to the switch level parts, are connected to the gates of the transistor switches. In such cases, the value of the added strength code is insignificant (See rules 1,2 and 3 in Section 3). It only has to be greater than L0.

6 Experimental Results

This section presents the results, which are obtained from the experimental evaluation of the method by the use of two Altera FPGAs (FLEX10K200SFC484 with 9984 LCs and 10160 DFFs, and FLEX10K50RC240 with 2880 LCs and 3184 DFFs). The experimented model has 8 different levels of strength.

Table 2 shows the resources, which are used by each pseudo transistor. As shown in Table 2, pseudo transistors do not consume DFFs (these circuits are combinational). Table 3 shows the number of resources, which are used by pseudo nodes. This table is made up of three parts based on the number of the pseudo transistors, which are connected to the pseudo node.

As shown in Table 3, the number of DFFs, which the pseudo node uses is 5 regardless of the number of the pseudo transistors, which are connected to it. This is because only the pseudo capacitor of a node needs DFFs.

Table 2. Resources used by pseudo transistors (with 8 levels of strength)

P type pseudo transistor		
FPGA	LCs	DFFs
EPF10K200SFC484	11	0
EPF10K50RC240	11	0
N type pseudo transistor		
FPGA	LCs	DFFs
EPF10K200SFC484	11	0
EPF10K50RC240	11	0

Table 3. Resources used by pseudo nodes (with 8 levels of strength)

Pseudo node with 2 connections		
FPGA	LCs	DFFs
EPF10K200SFC484	30	5
EPF10K50RC240	30	5
Pseudo node with 3 connections		
FPGA	LCs	DFFs
EPF10K200SFC484	57	5
EPF10K50RC240	56	5
Pseudo node with 4 connections		
FPGA	LCs	DFFs
EPF10K200SFC484	63	5
EPF10K50RC240	63	5

The width of the strength code is 3 and the state code is a 2-bit code. Therefore, 5 DFFs are used.

Table 4 shows the resources, which are used by dual-railed NAND gates. These results show the importance of the mixed-mode switch-gate level emulation presented in the previous section.

Our experiments show that the speedup grows with the circuit size and the emulation of a switch level circuit can be about 27 to 532 times faster than its simulation using Verilog switch level models. In these experiments a ModelSim simulator (Version 5.4a) is used which is run on a Pentium II system (333MHz, RAM=192 MB). For example for an 8-bit Magnitude comparator circuit, which its gate level model is shown in Figure 7, the resulted speedup is 417 when 10000 different inputs are applied to the circuit. In this experiment mixed-mode emulation is not used and circuit is emulated using a pure switch level model. In fact, each gate of the circuit shown in Figure 7 is replaced with its corresponding switch level model.

Table 4. Resources used by dual-railed NAND gates

NAND gate with 2 inputs		
FPGA	LCs	DFFs
EPF10K200SFC484	2	0
EPF10K50RC240	2	0
NAND gate with 3 inputs		
FPGA	LCs	DFFs
EPF10K200SFC484	3	0
EPF10K50RC240	3	0
NAND gate with 4 inputs		
FPGA	LCs	DFFs
EPF10K200SFC484	4	0
EPF10K50RC240	4	0



Fig. 7. An 8-bit Magnitude Comparator used in the experiments

7 Conclusions

In this paper a novel method is presented in order to emulate switch level circuits and switch level faults. The advantages of this method are:

1. Use of FPGAs for fast emulation of CMOS switch level or mixed switch-gate level circuits.
2. Fault emulation of switch level fault models such as transistor-stuck-on and transistor-stuck-off faults.
3. Emulation of three-state CMOS gates by the use of FPGAs, which do not support three-state logic directly.

The experiments show that the emulation of a switch level circuit can be very faster than its simulation

It is shown that a mixed-mode emulation approach can be used in order to optimize the utilization of FPGA resources when switch level emulation is used.

References

1. J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," IEEE Trans. VLSI Syst., vol. 1, pp. 171-174, June 1993.
2. S. Walters, "Computer-aided prototyping for ASIC-based system," IEEE Design Test Comput., pp. 4-10, June 1991.
3. U. R. Khan, H. L. Owen, J. L. Hughes, "FPGA Architecture for ASIC Hardware Emulator", Proc. 6, IEEE ASIC Conference, p.336, 1993.
4. K.T. Cheng, S.Y. Huang, and W.J. Dai, "Fault Emulation: A New Methodology for Fault Grading," IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 18, no. 10, pp. 1487-1495, October 1999.
5. P. Dahlgren, P. Liden, "Efficient Modeling of Switch-Level Networks Containing Undetermined Logic Node States", Proc. IEEE/ACM Int. Conf. on CAD, pp. 746-752, 1993.
6. R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems", IEEE Trans. Computers, Vol. C-33, No. 2, pp. 160-177, Feb 1984.
7. M. Abramovici, M. A. Breuer, and A. D. Friedman, "Digital Systems Testing and Testable Design", Revised edition, IEEE Press, 1995.
8. Verilog Hardware Descriptor Language Reference Manual (LRM) DRAFT, IEEE 1364, April, 1995.
9. G. S. Choi, and R. K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis", IEEE Trans. Computers, vol. 41, no. 12, pp.1515-1526, Dec. 1992.

An Extensible, System-On-Programmable-Chip, Content-Aware Internet Firewall^{*}

John W. Lockwood, Christopher Neely, Christopher Zuver,
James Moscola, Sarang Dharmapurikar, and David Lim

Applied Research Laboratory
Washington University in Saint Louis
1 Brookings Drive, Campus Box 1045
Saint Louis, MO 63130 USA

{lockwood,cen1,cz2,jmm5,sarang,dlim}@arl.wustl.edu
<http://www.arl.wustl.edu/arl/projects/fpx/>

Abstract. An extensible firewall has been implemented that performs packet filtering, content scanning, and per-flow queuing of Internet packets at Gigabit/second rates. The firewall uses layered protocol wrappers to parse the content of Internet data. Packet payloads are scanned for keywords using parallel regular expression matching circuits. Packet headers are compared to rules specified in Ternary Content Addressable Memories (TCAMs). Per-flow queuing is performed to mitigate the effect of Denial of Service attacks. All packet processing operations were implemented with reconfigurable hardware and fit within a single Xilinx Virtex XCV2000E Field Programmable Gate Array (FPGA). The single-chip firewall has been used to filter Internet SPAM and to guard against several types of network intrusion. Additional features were implemented in extensible hardware modules deployed using run-time reconfiguration.

1 Introduction

Recently, demand for Internet security has significantly increased. Internet connected hosts are now frequently attacked by malicious machines located around the world. Hosts can be protected from remote machines by filtering traffic through a firewall. By actively dropping harmful packets and rate-limiting unwanted traffic flows, the harm caused by attacks can be reduced.

While some types of attacks can be thwarted solely by examination of packet headers, other types of attacks – such as network intrusion, Internet worm propagation, and SPAM proliferation – require that firewalls process entire packet payloads [1]. Few existing firewalls have the capability to scan entire packet payloads. Of those that do, most are software-based and cannot process packets at the high-speed rates used by modern networks. Hardware-accelerated firewalls are needed to process entire packet payloads at high speeds.

* This research was supported in part by a grant from Global Velocity, the National Science Foundation (ANI-0096052), and a gift from Xilinx

Application Specific Integrated Circuits (ASICs) have been used in firewalls to implement some packet filtering functions. ASICs allow firewalls to achieve high throughput by processing packets in deep pipelines and parallel circuits. But ASICs can only protect networks from threats known to the designer when the chip was fabricated. Once an ASIC is fabricated, its function is static and cannot be altered. The ability to protect networks against both the present and future threats is the real challenge in building modern firewalls [2].

2 System-On-Programmable-Chip (SOPC) Firewall

A System-On-Programmable-Chip (SOPC) Internet firewall has been implemented that protects high-speed networks from present and future threats. In order to protect networks against current threats, the SOPC firewall parses Internet protocol headers, scans packet payloads, filters traffic, and performs per-flow queuing. In order to protect against future threats, the SOPC is highly extensible. This paper details the implementation of that firewall in a single Xilinx Virtex XCV2000E FPGA.

The top-level architecture of the SOPC firewall is shown in Figure 1. When a packet first enters the SOPC firewall, it is processed by a set of *layered protocol wrappers*. These wrappers segment and reassemble frames; verify and compute checksums; and read and write the headers of the Internet packet. Once the packet has been parsed by the wrappers, a *payload scanner* searches the entire content of the packet for keywords and regular expressions. Next, a *Ternary Content Addressable Memory (TCAM) filter* classifies packets based on a set of reconfigurable rules. The packet then passes through one or more *extensible modules*, where additional packet processing or packet filtering is implemented. Next, the data and flow ID are passed to a *queue manager*, which schedules the packets for transmission from a *flow buffer*. Finally, the packet is transmitted to a switch or to a Gigabit Ethernet or SONET line card. Details of these components are given in the sections that follow.

2.1 Protocol Processing

To process packets, a set of layered protocol wrappers was implemented to parse protocols at multiple layers [3]. At the lowest layer, data is segmented and reassembled from short cells into complete frames. At the network layer of the protocol stack, the fields of the Internet Protocol (IP) packets are verified and computed. At the transport layer, the port numbers and packet lengths are used to extract application-level data.

2.2 Payload Processing with Regular Expressions

Many types of Internet traffic cannot be classified by header inspection [4]. For example, junk email (SPAM) typically contains perfectly valid network and transport-layer data. In order to determine that a message is SPAM, a filtering

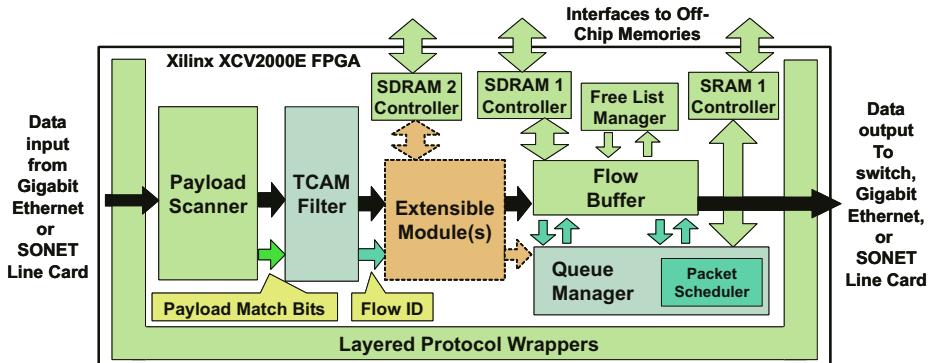


Fig. 1. Block Diagram of System-On-Chip Firewall

device must be able to classify packets based on the content rather than just the values that appear in the packet headers.

Dynamically reconfigurable hardware performs content classification functions effectively. Unlike an ASIC, new finite automata can be programmed into hardware to scan for specific content. In order to scan packet payloads, a regular expression matching circuit was implemented. Regular expressions provide a shorthand means to specify the value of a string, a substring (specified by ‘()’), alterative values (separated with ‘|’), any one character (specified by ‘.’), zero or one characters (specified by ‘?’), or zero or more characters (specified by ‘*’). For example, to match all eight case variations of the phrase “make money fast” and to allow any number of characters to reside between keywords, the expression: “(M|m)ake.*(M|m)oney.*(F|f)ast” would be specified.

The architecture of a payload scanner with four parallel content scanners searching for eight Regular Expressions, $RE[1]-RE[8]$, is illustrated in Figure 2. A match is detected when a sequence of incoming data causes a state machine to reach an accepting state. In order to differentiate between multiple groups of regular expressions, the scanner instantiates a sequence of parallel machines that each search for different expressions. In order to maximize throughput, multiple sets of parallel content matching circuits are instantiated. When data arrives, a *flow dispatcher* sends it to an available buffer which then streams data through the sequence of regular expression search engines. Finally, a *flow collector* combines the traffic into a single outgoing stream of data. The final result of the circuit is a set of *payload match bits* that identify which regular expressions were present in each data flow. To implement other high-speed payload scanning circuits, an automated design process was created to generate circuits from a specification of regular expressions [5].

2.3 Rule Processing

A diagram of the rule matching circuit used on the SOPC firewall is shown in Figure 3. An on-chip TCAM is used to classify packets as belonging to a specific

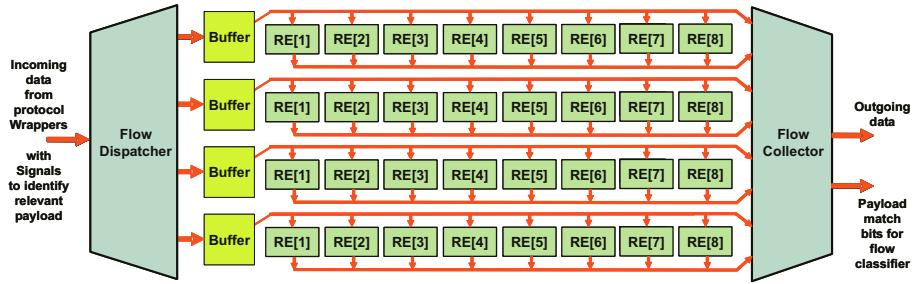


Fig. 2. Regular Expression Payload Scanner

flow. As an Internet Protocol (IP) packet arrives, *bits in the IP header*, which include the *source address*, *destination address*, *source port*, *destination port*, and *protocol* as well as the *payload match bits* are simultaneously compared to the *CAM value* fields in all rows of the TCAM table. Once all of the values have been compared, a *CAM mask* is applied to select which bits of each row must match and which bits can be ignored. If all of the values match in all of the bit locations that are unmasked, then that row of the TCAM is considered to be a match. The *flow identifier* associated with the rule in the highest-priority matching TCAM is then assigned to the flow. Rules can be added or modified by sending control packets to the firewall to change values in the *CAM mask*, *CAM value*, and *flow ID* fields. Large CAMs can be implemented on the FPGA by using block RAMs [6].

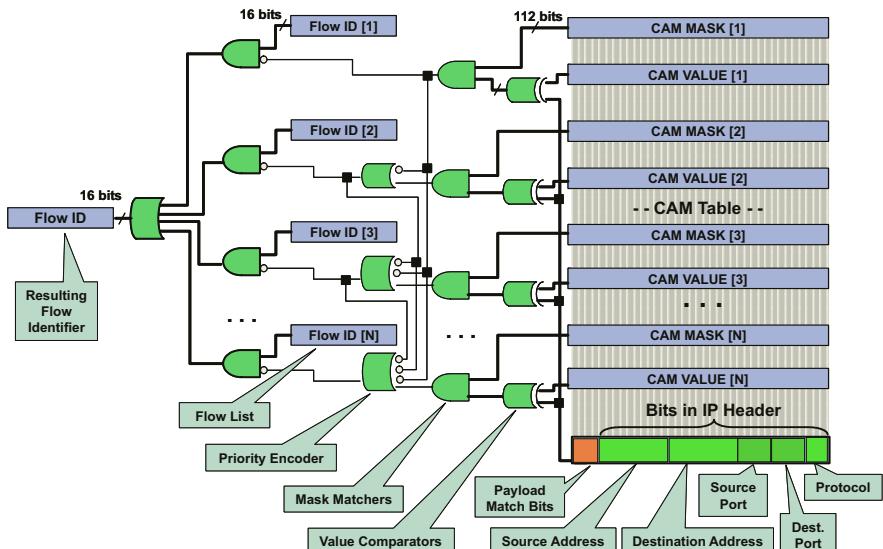


Fig. 3. Header Processing Module

2.4 Flow Buffering

To provide different levels of Quality of Service (QoS) for different traffic passing through the network, the SOPC firewall implements both class-based and per-flow queuing. Class-based queuing allows certain types of traffic to receive better service than other traffic. Per-flow queuing ensures that no single traffic flow consumes all of the network's available bandwidth [7].

To support multiple classes of service, traffic flows are organized by the firewall into four priority classes. Within each class, multiple linked lists of packets are maintained. Management of queues and tracking of free memory space is performed by the FPGA hardware in constant-time using linked-list data structures.

A diagram of the flow buffer and queue manager is shown in Figure 4. The queue manager includes circuits to enqueue traffic flows, dequeue traffic flows, and to schedule flows for transmission. Within the scheduler, four separate queues of flows are maintained (one for each class).

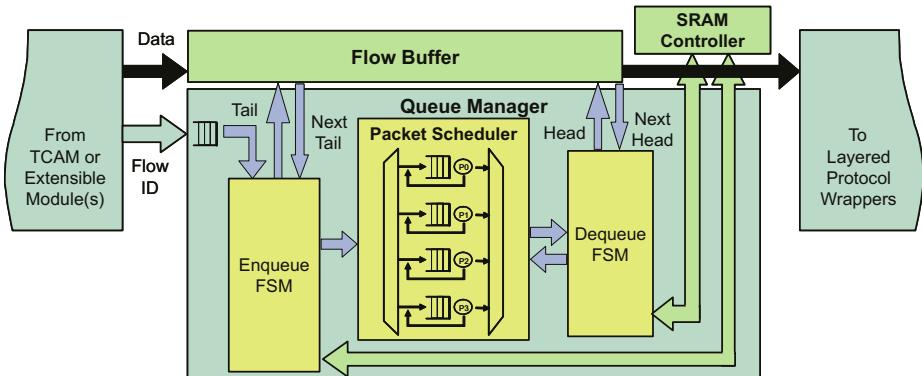


Fig. 4. Queue manager for organizing traffic flows

When a packet arrives, the packet's data is delivered to the *flow buffer* and the packet's flow identifier is passed to the *enqueue FSM* in the *Queue Manager*. Using the *flow ID*, the *enqueue FSM* reads SRAM to retrieve the flow's state. Each entry of the flow's state table contains a pointer to the *head* and *tail* of a linked list of stored packets in SDRAM as well as counters to track the number of packets read and written to that flow.

Meanwhile, the flow buffer stores the packet into memory at the location specified by the enqueue FSM's tail pointer. The flow buffer includes a controller to store the packet in Synchronous Dynamic Random Access Memory (SDRAM) [8]. After writing a packet to the next available memory location, the value of the tail pointer is passed to the queue manager to identify the next free memory location.

Within each class of traffic, the queue manager performs round-robin queuing of individual traffic flows. When the first packet of a flow arrives that has no packets already buffered, the flow identifier is inserted into a scheduling queue for that packet's class of service and the flow state table is updated. When another packet of a flow that is already scheduled arrives, the packet is simply appended to the linked list and the packet write count is incremented.

To transmit packets, the *dequeue FSM* reads a flow ID from the scheduler. The scheduler dequeues the flow from the next available flow in the highest priority class of traffic. The dequeue FSM then reads the flow state to obtain a pointer to the data in the flow buffer. The flow identifier is then removed from the head of the scheduler's queue and re-enters at the tail of the same queue if that flow has additional packets to transmit.

2.5 Extensible Features and Design Tools

Custom functionality is programmed into the SOPC firewall to implement specific functions demanded for a particular network application or service. Extensible modules that perform data encryption, enhanced packet filtering functions, and other types of packet scheduling have been implemented. One such module blocks patterns of TCP Synchronize/Acknowledge (SYN/ACK) packets characteristic of a DoS attacks. Other extensible functions that have been implemented as modules that fit into the SOPC firewall are listed in Table 1 [9] [10]. The set of features in the firewall is only limited by the size of the FPGA.

Table 1. SOPC Firewall Modules

Virus blocking	Content filtering
Denial of Service Protection	AES and 3DES Decryption
Bitmap image filtering	Network Address Translation (NAT)
Internet route lookup	IP Version 6 (IPV6) tunneling
Resource Reservation (RSVP)	Domain Name Service (DNS) caching

To facilitate development of extensible modules, an integration server was created that allows modules to be automatically configured using an application developer's information submitted via the web [11]. The tool transforms time-consuming and mistake-prone task of individually wiring ports on newly created modules into the relatively simple task of selecting where a module fits within the system interface. Developers upload their extensible modules to the integration server, which registers them in a database. The server parses the uploaded VHDL files and prompts the user to supply the port mapping to one or more of the standard, modular interfaces on the SOPC firewall. Then a bitfile is generated that contains the new module integrated into the baseline firewall, which is then returned to the user for testing. The server can also be used to create a bitfile for a SOPC firewall with any subset of previously uploaded extensible modules.

The integration server tracks and displays the estimated device utilization for each module to aid the selection process.

To further ease the development of extensible modules, a networked test server was developed to allow new extensible modules to be tested over the Internet [12]. After developing an extensible module on the integration server, a developer submits the resulting placed and routed design via a web interface to the test server. The test server: (1) remotely programs the FPGA platform, (2) generates test traffic in the form of network packets, and (3) displays the resulting packets. The traffic generated by the test server is processed by the firewall and returned to the test server. The developer then compares the contents of the resulting packets to expected values to verify correctness. Automation simplifies the process of migrating designs from concept to implementation.

3 Results

The results after place and route for the synthesized SOPC Firewall on the Xilinx Virtex XCV2000E are listed in Table 2. The core logic occupied 43% of the logic and 39% of the block RAMs. Placement of the SOPC circuitry was constrained using Synplicity's Amplify tool to lock the location of modules into specific regions of the FPGA. A view of the placed and routed Xilinx Virtex XCV2000E is shown in Figure 5. Note that the center region of the chip was left available for insertion of extensible modules.

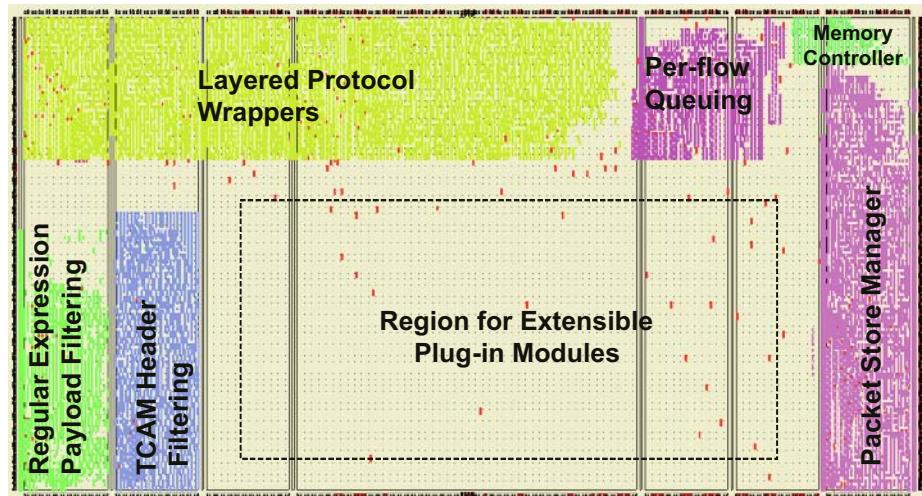


Fig. 5. FPGA Layout of the SOPC Firewall

Table 2. SOPC Firewall Implementation Statistics

Resource	Virtex XCV2000E Utilization	
	Device Utilization	Percentage
Logic Slices	8342 out of 19200	43%
BlockRAMs	63 out of 160	39%
External IOBs	286 out of 512	55%

3.1 Throughput

The components of the SOPC firewall that implement the protocol wrappers, CAM filter, flow buffer, and queue manager were synthesized and operated at 62.5 MHz. Each of these components process 32 bits of data in every cycle, thus giving the SOPC firewall a throughput of $32 \times 62.5\text{MHz} = 2$ Gigabits/second. The regular expression scanning circuit for the SPAM pipeline synthesized at 37 MHz. Given that each pipeline processes 8 bits of data per cycle, the throughput of the SPAM filter is $8 \times 37\text{MHz} = 296$ Megabits/second per pipeline. By running 8 SPAM pipelines in parallel, the payload matching circuit achieves a throughput of $8 \times 8 \times 37\text{MHz} = 2.368$ Gigabits/second.

3.2 Testing the SOPC Firewall on the FPX Platform

The Field Programmable Port Extender (FPX) platform was used to evaluate the performance of the SOPC firewall with real Internet traffic. The FPX is an open hardware platform that supports partial run-time reconfiguration [13] [14]. As with other Internet-based reconfigurable systems, the FPX allows some or all of the hardware to be dynamically reprogrammed over the network [15] [16]. On the FPX, however, all reconfiguration operations are performed in hardware and the network interfaces run at multi-gigabit/second rates [17].

The core components of the SOPC firewall include the layered protocol wrappers, TCAM packet filters, the payload processing circuit, and the per-flow packet buffer with the SRAM and SDRAM controllers. All of these components were synthesized into the Virtex XCV2000E device that implements the Reprogrammable Application Device (RAD) on the FPX. The resulting bitfile was uploaded into the RAD and in-system testing was performed.

Using the features described above, a SPAM filter was implemented on the SOPC firewall to detect and filter unwanted junk email, commonly referred to as SPAM. By scanning the payloads of packets passing through our network, potentially unwanted messages were filtered before they reached their targeted endpoint. To implement our SPAM filter, we observed that within our network, emails could be classified into one of eight general categories. Within these categories, we identified several regular expressions that were commonly present or absent in SPAM-infested email. Our listed included the terms: “MAKE MONEY FAST”, “Limited Time Offer”, as well as 32 other terms. These terms were programmed into hardware as regular expressions in a way that allowed case-insensitive matching where needed. Meanwhile, the TCAM was programmed

with rules to assign SPAM-infested packets to put SPAM-infested email in lower-priority traffic flows [10].

Actual network traffic was sent to the hardware over the network from remote hosts. The SOPC firewall filtered traffic passing between a local area network and a wide area network, as shown in Figure 6. Malicious packets were dropped, the SPAM was rate-limited, and all other flows received a fair share of bandwidth.

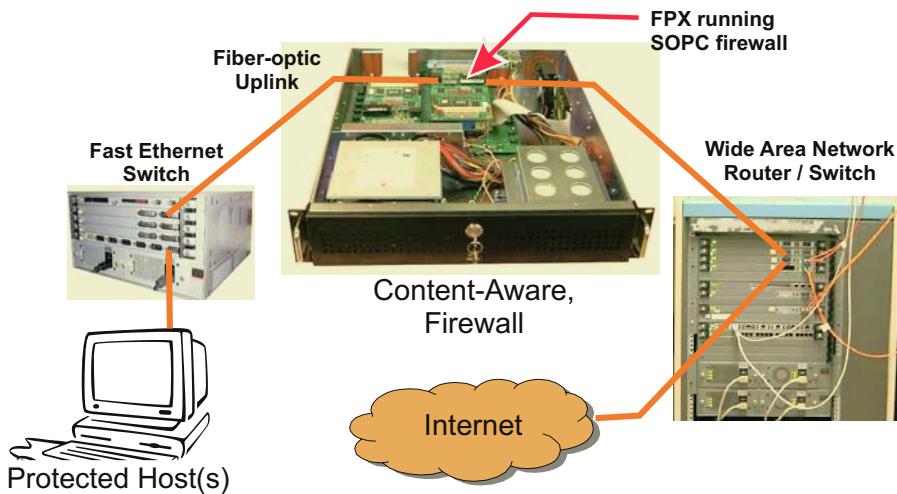


Fig. 6. Firewall Configuration

4 Conclusions

An extensible firewall has been implemented as a reconfigurable System-On-Programmable-Chip (SOPC). In addition to the standard features implemented by other Internet firewalls, the SOPC firewall performs payload scanning and per-flow queuing. The circuit was implemented on a Xilinx XCV-2000E FPGA. The resulting bitfile was tested on the Field Programmable Port Extender (FPX) network platform.

By using parallel hardware and deeply pipelined circuits, the SOPC firewall can process protocol headers with TCAMS and search the entire payload using regular expression matching at rates over 2 Gigabits/second. Attacks are mitigated by sorting each traffic flow into its own queue and scheduling traffic from all flows. These features allow the firewall to filter SPAM and protect networks from intrusion. A region of gates in the FPGA was left available to be used for extensible plugins. By using reprogrammable this hardware, new modules can be added and the firewall can protect a network against future threats.

References

1. R. Franklin, D. Carver, and B. L. Hutchings, "Assisting network intrusion detection with reconfigurable hardware," in *FCCM*, (Napa, CA), Apr. 2002.
2. J. W. Lockwood, "Evolvable internet hardware platforms," in *The Third NASA/DoD Workshop on Evolvable Hardware (EH'2001)*, pp. 271–279, July 2001.
3. F. Braun, J. Lockwood, and M. Waldvogel, "Reconfigurable router modules using network protocol wrappers," in *Field-Programmable Logic and Applications (FPL)*, (Belfast, Northern Ireland), pp. 254–263, Aug. 2001.
4. Y. Cho, S. Nahab, and W. H. Mangione-Smith, "Specialized hardware for deep network packet filtering," in *Field Programmable Logic and Applications (FPL)*, (Montpellier, France), Sept. 2002.
5. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an Internet firewall," in *FCCM*, (Napa, CA), Apr. 2003.
6. J.-L. Brelet, "Using block RAM for high performance read/write CAMs." Xilinx XAPP204, May 2002.
7. H. Duan, J. W. Lockwood, S. M. Kang, and J. Will, "High-performance OC-12/OC-48 queue design prototype for input-buffered ATM switches," in *INFO-COM'97*, (Kobe, Japan), pp. 20–28, Apr. 1997.
8. S. Dharmapurikar and J. Lockwood, "Synthesizable design of a multi-module memory controller." Washington University, Department of Computer Science, Technical Report WUCS-01-26, Oct. 2001.
9. "Acceleration of Algorithms in Hardware." <http://www.arl.wustl.edu/~lockwood/class/cs535/>, Sept. 2001.
10. "Reconfigurable System-On-Chip Design." <http://www.arl.wustl.edu/~lockwood/class/cs536/>, Dec. 2002.
11. D. Lim, C. E. Neely, C. K. Zuver, and J. W. Lockwood, "Internet-based tool for system-on-chip integration," in *International Conference on Microelectronic Systems Education (MSE)*, (Anaheim, CA), June 2003.
12. C. E. Neely, C. K. Zuver, and J. W. Lockwood, "Internet-based tool for system-on-chip project testing and grading," in *International Conference on Microelectronic Systems Education (MSE)*, (Anaheim, CA), June 2003.
13. E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *Design Automation Conference (DAC)*, (New Orleans, LA), June 2002.
14. T. Sproull, J. W. Lockwood, and D. E. Taylor, "Control and configuration software for a reconfigurable networking hardware platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM)*, (Napa, CA), Apr. 2002.
15. S. McMillan and S. Guccione, "Partial run-time reconfiguration using JRTR," in *Field-Programmable Logic and Applications (FPL)*, (Villach, Austria), pp. 352–360, Aug. 2000.
16. H. Fallside and M. J. S. Smith, "Internet connected FPL," in *Field-Programmable Logic and Applications (FPL)*, (Villach, Austria), pp. 48–57, Aug. 2000.
17. J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

IPsec-Protected Transport of HDTV over IP*

Peter Bellows¹, Jaroslav Flidr¹, Ladan Gharai¹, Colin Perkins¹, Paweł Chodowiec²,
and Kris Gaj²

¹ USC Information Sciences Institute, 3811 N. Fairfax Dr. #200, Arlington VA 22203, USA;
{pbellows|jflidr|ladan|csp}@isi.edu

² Dept. of Electrical and Computer Engineering, George Mason University, 4400 University
Drive, Fairfax VA 22030, USA; {pchodow1|kgaj}@gmu.edu

Abstract. Bandwidth-intensive applications compete directly with the operating system’s network stack for CPU cycles. This is particularly true when the stack performs security protocols such as IPsec; the additional load of complex cryptographic transforms overwhelms modern CPUs when data rates exceed 100 Mbps. This paper describes a network-processing accelerator which overcomes these bottlenecks by offloading packet processing and cryptographic transforms to an intelligent interface card. The system achieves sustained 1 Gbps host-to-host bandwidth of encrypted IPsec traffic on commodity CPUs and networks. It appears to the application developer as a normal network interface, because the hardware acceleration is transparent to the user. The system is highly programmable and can support a variety of offload functions. A sample application is described, wherein production-quality HDTV is transported over IP at nearly 900 Mbps, fully secured using IPsec with AES encryption.

1 Introduction

As available network bandwidth scales faster than CPU power[1], the overhead of network protocol processing is becoming increasingly dominant. This means that high-bandwidth applications receive diminishing marginal returns from increases in network performance. The problem is greatly compounded when security is added to the protocol stack. For example, the IP security protocol (IPsec) [2] requires complex cryptographic transforms which overwhelm modern CPUs. IPsec benchmarks on current CPUs show maximum throughput of 40-90 Mbps, depending on the encryption used [3]. With 1 Gbps networks now standard and 10 Gbps networks well on their way, the sequential CPU clearly cannot keep up with the load of protocol and security processing. By contrast, application-specific parallel computers such as FPGAs are much better suited to cryptography and other streaming operations. This naturally leads us to consider using dedicated hardware to offload network processing (especially cryptography), so more CPU cycles can be dedicated to the applications which use the data.

This paper describes a prototype of such an offload system, known as “GRIP” (Gigabit-Rate IPsec). The system is a network-processing accelerator card based on

* This work is supported by the DARPA Information Technology Office (ITO) as part of the Next Generation Internet program under Grants F30602-00-1-0541 and MDA972-99-C-0022, and by the National Science Foundation under grant 0230738.

Xilinx Virtex FPGAs. GRIP integrates seamlessly into a standard Linux implementation of the TCP/IP/IPsec protocols. It provides full-duplex gigabit-rate acceleration of a variety of operations such as AES, 3DES, SHA-1, SHA-512, and application-specific kernels. To the application developer, all acceleration is completely transparent, and GRIP appears as just another network interface. The hardware is very open and programmable, and can offload processing from various levels of the network stack, while still requiring only a single transfer across the PCI bus. This paper focuses primarily on our efforts to offload the complex cryptographic transforms of IPsec, which, when utilized, are the dominant performance bottleneck of the stack.

As a demonstration of the power of hardware offloading, we have successfully transmitted an encrypted stream of live, production-quality HDTV across a commodity IP network. Video is captured in an HDTV frame-grabber at 850 Mbps, packetized and sent AES-encrypted across the network via a GRIP card. A GRIP card on a receiving machine decrypts the incoming stream, and the video frames are displayed on an HDTV monitor. All video processing is done on the GRIP-enabled machines. In other words, the offloading of the cryptographic transforms frees enough CPU time for substantial video processing with no packet loss on ordinary CPUs (1.3 GHz Pentium III).

This paper describes the hardware, device driver and operating system issues for building the GRIP system and HDTV testbed. We analyze the processing bottlenecks in the accelerated system, and propose enhancements to both the hardware and protocol layers to take the system to the next levels of performance (10 Gbps and beyond).

2 GRIP System Architecture

The overall GRIP system is diagrammed in figure 1. It is a combination of an accelerated network interface card, a high-performance device driver, and special interactions with the operating system. The interface card is the SLAAC-1V FPGA coprocessor board [4] combined with a custom Gigabit Ethernet mezzanine card. The card has a total of four FPGAs which are programmed with network processing functions as follows. One device (X0) acts as a dedicated packet mover / PCI interface, while another (GRIP) provides the interface to the Gigabit Ethernet chipset and common offload functions such as IP checksumming. The remaining two devices (X1 and X2) act as independent transmit and receive processing pipelines, and are fully programmable with any acceleration function. For the HDTV demonstration, X1 and X2 are programmed with AES-128 encryption cores.

The GRIP card interfaces with a normal network stack. The device driver indicates its offload capabilities to the stack, based on the modules that are loaded into X1 and X2. For example in the HDTV application, the driver tells the IPsec layer that accelerated AES encryption is available. This causes IPsec to defer the complex cryptographic transforms to the hardware, passing raw IP/IPsec packets down to the driver with all the appropriate header information but no encryption. The GRIP driver looks up security parameters (key, IV, algorithm, etc.) for the corresponding IPsec session, and prefixes these parameters to each packet before handing it off to the hardware. The X0 device fetches the packet across the PCI bus and passes it to the transmit pipeline (X1). X1 analyzes the packet headers and security prefix, encrypting or providing other security

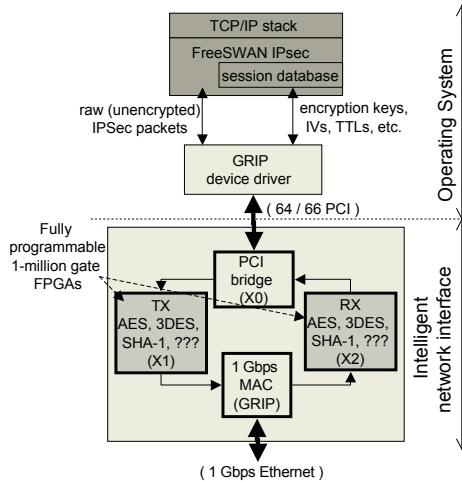


Fig. 1. GRIP system architecture

services as specified by the driver. The packet, now completed, is sent to the Ethernet interface on the daughter card. The receive pipeline is just the inverse, passing through the X2 FPGA for decryption. Bottlenecks in other layers of the stack can also be offloaded with this “deferred processing” approach.

3 GRIP Hardware

3.1 Basic Platform

The GRIP hardware platform provides an open, extensible development environment for experimenting with 1 Gbps hardware offload functions. It is based on the SLAAC-1V FPGA board, which was designed for use in a variety of military signal processing applications. SLAAC-1V has three user-programmable Xilinx Virtex 1000 FPGAs (named X0, X1 and X2) connected by separate 72-bit systolic and shared busses. Each FPGA has an estimated 1 million equivalent programmable gates with 32 embedded SRAM banks, and is capable of clock speeds of up to 150 MHz. The FPGAs are connected to 10 independent banks of 1 MB ZBT SRAM, which are independently accessible by the host through passive bus switches. SLAAC-1V also has an on-board flash/SRAM cache for storing FPGA bitstreams, allowing for rapid run-time reconfiguration of the devices. For the GRIP project, we have added a custom 1 Gigabit Ethernet mezzanine card to SLAAC-1V. It has a Vitesse 8840 Media Access Controller (MAC), and a Xilinx Virtex 300 FPGA which interfaces to the X0 chip through a 72-bit connector. The Virtex 300 uses 1 MB of external ZBT-SRAM for packet buffering, and performs common offload functions such as filtering and checksumming.

The GRIP platform defines a standard partitioning for packet processing, as described in section 2. As described, the X0 and GRIP FPGAs provide a static framework that manages basic packet movement, including the MAC and PCI interfaces. The X0 FPGA

contains a packet switch for shuttling packets back and forth between the other FP-GAs on the card, and uses a 2-bit framing protocol (“start-of-frame” / “end-of-frame”) to ensure robust synchronization of the data streams. By default, SLAAC-1V has a high-performance DMA engine for mastering the PCI bus. However, PCI transfers for a network interface are small compared to those required for the signal processing applications targeted by SLAAC-1V. Therefore for the GRIP system, the DMA engine was tuned with key features needed for high-rate network-oriented traffic, such as dynamic load balancing, 255-deep scatter-gather tables, programmable interrupt mitigation, and support for misaligned transfers.

With this static framework in place, the X1 and X2 FPGAs are free to be programmed with any packet-processing function desired. To interoperate with the static framework, a packet-processing function simply needs to incorporate a common I/O module and adhere to the 2-bit framing protocol. SLAAC-1V’s ZBT SRAMs are not required by the GRIP infrastructure, leaving them free to be used by packet-processing modules. Note that this partitioning scheme is not ideal in terms of conserving resources - less than half of the circuit resources in X0 and GRIP are currently used. This scheme was chosen because it provides a clean and easily programmable platform for network research. The basic GRIP hardware platform is further documented in [5].

3.2 X1/X2 IPsec Accelerator Cores

A number of packet-processing cores have been developed on the SLAAC-1V / GRIP platform, including AES (Rijndael), 3DES, SHA-1, SHA-512, SNORT-based traffic analysis, rules-based packet filtering (firewall), and intrusion detection [6–8]. For the secure HDTV application, X1 and X2 were loaded with 1 Gb/s AES encryption cores. We chose a space-efficient AES design, which uses a single-stage iterative datapath with inner-round pipelining. The cores support all defined key sizes (128, 192 and 256-bit) and operate in either CBC or counter mode. Because of the non-cyclic nature of counter mode, the counter-mode circuit can maintain maximum throughput for a single stream of data, whereas the CBC-mode circuit requires two interleaved streams for full throughput. For this reason, counter mode was used in the demonstration system.

The AES cores are encapsulated by state machines that read each packet header and any tags prefixed by the device driver, and separate the headers from the payload to be encrypted / decrypted. The details of our AES designs are given in [6]. We present FPGA implementation results for the GRIP system in section 7.

4 Integrating GRIP with the Operating System

The integration of the hardware presented in the section 3 is a fairly complex task because, unlike ordinary network cards or crypto-accelerators, GRIP offers services to three layers of the OSI architecture: the physical, link and network layers. To make a complex matter worse, the IPsec stack - the main focus of current GRIP research - is located in neither the network nor link layers. Rather, it could be described as a link-layer component “wrapped” in the IP stack (figure 2). Thus care must be taken to provide a continuation of services even though parts of higher layers have been offloaded.

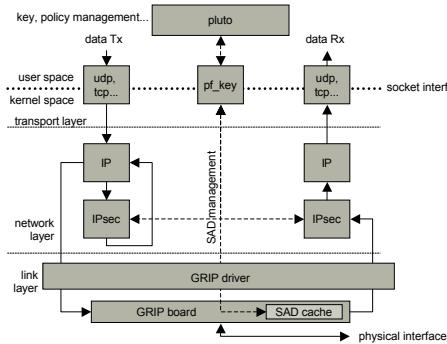


Fig. 2. The IPsec (FreeSWAN) stack in the kernel architecture.

For this study we used FreeSWAN, a standard implementation of IPsec for Linux [9]. FreeSWAN consists of two main parts: KLIPS and Pluto. KLIPS (KerneL IP Security) contains the Linux kernel patches that implement the IPsec protocols for encryption and authentication. Pluto negotiates the Security Association (SA) parameters for IPsec-protected sockets. Figure 2 illustrates the integration of FreeSWAN into the system architecture. Pluto negotiates new security associations (SA's) using the ISAKMP protocol. When a new SA is negotiated, it is sent to the IPsec stack via the pf_key socket, where it is stored in the Security Association Database (SAD). At this point, the secure channel is open and ready to go. Any time a packet is sent to an IPsec-protected socket, the IPsec transmit function finds the appropriate SA in the database and performs the required cryptographic transforms. After this processing, the packet is handed back to IP which passes it to the physical interface. The receive mode is the inverse but somewhat less complex. When there are recursive IPsec tunnels or multiple IPsec interfaces, the above process can repeat many times.

In order to accommodate GRIP acceleration we made three modifications. First, we modified Pluto so that AES Counter mode is the preferred encryption algorithm for negotiating new SA's. Second, we altered the actual IPsec stack so that new SA's are communicated to the GRIP device driver using the driver's private space. The driver then caches the security parameters (encryption keys, etc.) on the GRIP card for use by the accelerator circuits. Finally, the IPsec transmit and receive functions were slightly modified to produce proper initialization vectors for AES counter mode. Any packet associated with an AES SA gets processed as usual - IPsec headers inserted, initialization vectors generated, etc. The only difference is that the packet is passed back to the stack without encryption. The GRIP driver recognizes these partially-processed packets and tags them with a special prefix that instructs the card to perform the encryption.

5 Example Application: Encrypted Transport of HDTV over IP

5.1 Background

To demonstrate the performance of the GRIP system, we chose a demanding real-time multimedia application: transport of High Definition Television (HDTV) over IP. Studios

and production houses need to transport uncompressed video through various cycles of production, avoiding the artifacts that are an inevitable result of multiple compression cycles. Local transport of uncompressed HDTV between equipment is typically done with the SMPTE-292M standard format for universal exchange [10]. When production facilities are distributed, the SMPTE-292M signal is typically transported across dedicated fiber connections between sites, but a more economical alternative is desirable. We consider the use of IP networks for this purpose.

5.2 Design and Implementation

In previous work [11] we have implemented a system that delivers HDTV over IP networks. The Real-time Transport Protocol (RTP) [12] was chosen as the delivery service. RTP provides media framing, timing recovery and loss detection, to compensate for the inherent unreliability of UDP transport. HDTV capture and playout was via DVS HDstation cards [13], which are connected via SMPTE-292M links to an HDTV camera on the transmitter and an HDTV plasma display on the receiver. These cards were inserted into dual-processor Dell PowerEdge 2500 servers with standard Gigabit Ethernet cards and dual PCI busses (to reduce contention with the capture/display cards). Since the GRIP card appears to the system as a standard Ethernet card, it was possible to substitute a GRIP card in place of the normal Ethernet, and run the HDTV application unmodified.

The transmitter captures the video data, fragments it to match the network MTU, and adds RTP protocol headers. The native data rate of the video capture is slightly above that of gigabit Ethernet, so the video capture hardware is programmed to perform color sub-sampling from 10 to 8 bits per component, for a video rate of 850 Mbps. The receiver code takes packets from the network, reassembles video frames, corrects for the effects of network timing jitter, conceals lost packets, and renders the video.

5.3 Performance Requirements

As noted previously, the video data rate (after colour subsampling) is 850 Mbps. Each video frame is 1.8 million octets in size. To fit within the 9000 octet gigabit Ethernet MTU, frames are fragmented into approximately 200 RTP packets for transmission. The high packet rates are such that a naive implementation can saturate the memory bandwidth; accordingly, a key design goal is to avoid data copies. We implement scatter send and receive (implemented using the `recvfrom()` system call with `MSG_PEEK` to read the RTP header, followed by a second call to `recvfrom()` to read the data) to eliminate data marshalling overheads. Throughput of the system is limited by the interrupt processing and DMA overheads. We observe a linear increase in throughput as the MTU is increased, and require larger than normal MTU to successfully support the full data rate. It is clear that the system is operating close to the limit, and that adding IPsec encryption will not be feasible without hardware offload.

6 Related Work

Two common commercial implementations of cryptographic acceleration are *VPN gateways* and *crypto-accelerators*. The former approach is limited in that it only provides security between LANs with matching hardware (datalink layer security), not end-to-end (network layer) security. The host-based crypto-accelerator reduces the CPU overhead by offloading cryptography, but overwhelms the PCI bus at high data rates. GRIP differs from these approaches in that it is a reprogrammable, full system solution, integrating accelerator hardware into the core operation of the TCP/IP network stack.

A number of other efforts have demonstrated the usefulness of dedicated network processing for accelerating protocol processing or distributed algorithms. Examples of these efforts include HARP[14], Typhoon[15], RWCP's GigaE PM project[16], and EMP [17]. These efforts rely on embedded processor(s) which do not have sufficient processing power for full-rate offload of complex operations such as AES, and are primarily focused on unidirectional traffic. Other research efforts have integrated FPGAs onto NICs for specific applications such as routing [18], ATM firewall [19], and distributed FFT [20]. These systems accelerate end applications instead of the network stack, and often lacked the processing power of the GRIP card.

7 Results

7.1 System Performance

The HDTV demonstration system was built with symmetric multiprocessor (SMP) Dell PowerEdge 2500 servers (2x1.3 GHz) and Linux 2.4.18 kernels, as described in section 5.2, substituting a GRIP card in place of the standard Ethernet. The full, 850 Mbps HDTV stream was sent with GRIP-accelerated AES encryption and no compression. In addition, we tested for maximum encrypted bandwidth using iperf [21]. Application and operating system bottlenecks were analyzed by running precision profiling tools for 120 second intervals on both the transmitter and receiver. Transmitter and receiver profiling results are comparable, therefore only the transmitter results are presented for brevity. The profiling results are given in figure 7.1.

<i>Library/Function</i>	bandwidth	idle	kernel	IPsec	grip driver	appplcation	libc
<i>HDTV-SMP</i>	893 Mbps	62%	28%	4%	3%	<1%	< 1%
<i>iperf-SMP</i>	989 Mbps	47%	35%	4%	4%	2%	8%
<i>iperf-UP</i>	989 Mbps	0%	70%	9%	4%	3%	12%

Fig. 3. Transmitter profiling results running the HTDV and iperf applications, showing percentage of CPU time spent in various functions.

The HDTV application achieved full-rate transmission with no packets dropped. Even though the CPU was clearly not overloaded (idle time > 60%!), stress tests such as running other applications showed that the system was at the limits of its capabilities. Comparing the SMP and UP cases under iperf, we can see that the only change (after

taking into account the 2X factor of available CPU time under SMP) is the amount of idle time. Yet in essence, the performance of the system was unchanged.

To explain these observations, we consider system memory bandwidth. We measured the peak main memory bandwidth of the test system to be 8 Gbps with standard benchmarking tools. This means that in order to sustain gigabit network traffic, each packet can be transferred at most 8 times to/from main memory. We estimate that standard packet-processing will require three memory copies per packet: from the video driver's buffer to the hdtv application buffer, from the application buffer to the network stack, and a copy within the stack to allow IPsec headers to be inserted. The large size of the video buffer inhibits effective caching of the first copy and the read-access of the second copy; this means these copies consume 3 Gbps of main memory bandwidth for 1 Gbps network streams. Three more main memory transfers occur in writing the video frame from the capture card to the system buffer, flushing ready-to-transmit packets from the cache, and reading packets from memory to the GRIP card. In all, we estimate that a 1 Gbps network stream consumes 6 Gbps of main memory bandwidth on this system. Considering that other system processes are also executing and consuming bandwidth, and that the random nature of network streams likely reduces memory efficiency from the ideal peak performance, we conclude that main memory is indeed the system bottleneck.

7.2 Evaluating Hardware Implementations

Results from FPGA circuit implementations are shown in figure 7.2. As shown in the figure, the static packet-processing infrastructure easily achieves 1 Gbps throughput. Only the AES and SHA cores have low timing margins. Note that there are more than enough resources on SLAAC-1V to combine both AES encryption and a secure hash function at gigabit speeds. Also note that the target technology, the Virtex FPGA family, is five years old; much higher performance could be realized with today's technology.

<i>Design</i>	<i>CLB Util.</i>	<i>BRAM Util.</i>	<i>Pred. Perf.</i> (MHz / Gbps)	<i>Measured Perf.</i> (MHz / Gbps)
X0	47%	30%	PCI: 35 / 2.24 I/O: 54 / 1.73	33 / 2.11 33 / 1.06
X1 / X2 (AES)	17%	65%	CORE: 90 / 1.06 I/O: 47 / 1.50	90 / 1.06 33 / 1.06
GRIP	35%	43%	41 / 1.33	33 / 1.06
<i>Other modules:</i>				
3DES	31%	0%	77 / 1.57	83 / 1.69
SHA-1	16%	0%	64 / 1.00	75 / 1.14
SHA-512	23%	6%	50 / 0.62	56 / 0.67

Fig. 4. Summary of FPGA performance and utilization on Virtex 1000 FPGAs

8 Conclusions and Future Work

Network performance is currently doubling every eight months [1]. Modern CPUs, advancing at the relatively sluggish pace of Moore's Law, are fully consumed by full-rate data at modern line speeds, and completely overwhelmed by full-rate cryptography. This disparity between network bandwidth and CPU power will only worsen as these trends continue. In this paper we have proposed an accelerator architecture that attempts to resolve these bottlenecks now and can scale to higher performance in the future. The unique contributions of this work are not the individual processing modules themselves; for example, 1 Gbps AES encryption has been demonstrated by many others. Rather, we believe the key result is the full system approach to integrating accelerator hardware directly to the network stack itself. The GRIP card is capable of completing packet processing for multiple layers of the stack. This gives a highly efficient coupling to the operating system, with only one pass across the system bus per packet. We have demonstrated this system running at full 1 Gbps line speed with end-to-end encryption on commodity PCs. This provides significant performance improvements over existing implementations of end-to-end IPsec security.

As demonstrated by the HDTV system, this technology is very applicable to signal processing and rich multimedia applications. It could be applied to several new domains of secure applications, such as immersive media (e.g. the collaborative virtual operating room), commercial media distribution, distributed military signal processing, or basic VPNs for high-bandwidth networks.

We would like to investigate other general-purpose offload capabilities on the current platform. A 1 Gbps secure hash core could easily be added to the processing pipelines to give accelerated encryption and authentication simultaneously. More functions could be combined by using the rapid reconfiguration capabilities of SLAAC-1V to switch between a large number of accelerator functions on-demand. Packet sizes obviously make a big difference - larger packets mean less-frequent interrupts. The GRIP system could leverage this by incorporating TCP/IP fragmentation and reassembly, such that PCI bus transfers are larger than what is supported by the physical medium. Finally, several application-specific kernels could be made specifically for accelerating the HDTV system, such as RTP processing and video codecs.

Our results suggest that as we look towards the future and consider ways to scale this technology to multi-gigabit speeds, we must address the limitations of system memory bandwidth. At these speeds, CPU-level caches are of limited use because of the large and random nature of the data streams. While chipset technology improvements help by increasing available bandwidth, performance can also greatly improve by reducing the number of memory copies in the network stack. For a system such as GRIP, three significant improvements are readily available. The first and most beneficial is a direct DMA transfer between the grabber/display card and the GRIP board. The second is the elimination of the extra copy induced by IPsec, by modifying the kernel's network buffer allocation function so that the IPsec headers are accommodated. The third approach is to implement the zero-copy socket interface.

FPGA technology is already capable of multi-gigabit network acceleration. 10-Gbps AES counter mode implementations are straightforward using loop-unrolling [22]. Cyclic transforms such as AES CBC mode and SHA will require more aggressive tech-

niques such as more inner-round pipelining, interleaving of data streams, or even multiple units in parallel. We believe that 10 Gbps end-to-end security is possible with emerging commodity system bus (e.g. PCI Express), CPU, and network technologies, using the offload techniques discussed.

References

1. Calvin, J.: Digital convergence. In: Proceedings of the Workshop on New Visions of Large-Scale Networks: Research and Applications, Vienna, Virginia (2001)
2. IP Security Protocol (IPsec) Charter: Latest RFCs and Internet Drafts for IPsec, <http://ietf.org/html.charters/ipsec-charter.html>. (2003)
3. FreeS/WAN: IPsec Performance Benchmarking, http://www.freeswan.org/freeswan_trees-freeswan-1.99/doc/performance.html. (2002)
4. Schott, B., Bellows, P., French, M., Parker, R.: Applications of adaptive computing systems for signal processing challenges. In: Proceedings of the Asia South Pacific Design Automation Conference, Kitakyushu, Japan (2003)
5. Bellows, P., Flidr, J., Lehman, T., Schott, B., Underwood, K.D.: GRIP: A reconfigurable architecture for host-based gigabit-rate packet processing. In: Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA (2002)
6. Chodowiec, P., Gaj, K., Bellows, P., Schott, B.: Experimental testing of the gigabit IPsec-compliant implementations of Rijndael and Triple-DES using SLAAC-1V FPGA accelerator board. In: Proc. of the 4th Int'l Information Security Conf., Malaga, Spain (2001)
7. Gembowski, T., Lien, R., Gaj, K., Nguyen, N., Bellows, P., Flidr, J., Lehman, T., Schott, B.: Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512. In: Proc. of the 5th Int'l Information Security Conf., Sao Paulo, Brazil (2002)
8. Hutchings, B.L., Franklin, R., Carver, D.: Assisting network intrusion detection with reconfigurable hardware. In: Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA (2002)
9. FreeS/Wan: <http://www.freeswan.org/>. (2003)
10. Society of Motion Picture and Television Engineers: Bit-serial digital interface for high-definition television systems (1998) SMPTE-292M.
11. Perkins, C.S., Gharai, L., Lehman, T., Mankin, A.: Experiments with delivery of HDTV over IP networks. Proc. of the 12th International Packet Video Workshop (2002)
12. Schulzrinne, H., Casner, S., Frederick, R., Jacobson, V.: RTP: A transport protocol for real-time applications (1996) RFC 1889.
13. DVS Digital Video Systems: <http://www.dvs.de/>. (2003)
14. Mumment, T., Kosak, C., Steenkiste, P., Fisher, A.: Fine grain parallel communication on general purpose LANs. In: In Proceedings of 1996 International Conference on Supercomputing (ICS96), Philadelphia, PA, USA (1996) 341–349
15. Reinhardt, S.K., Larus, J.R., Wood, D.A.: Tempest and typhoon: User-level shared memory. In: International Conference on Computer Architecture, Chicago, Illinois, USA (1994)
16. Sumimoto, S., Tezuka, H., Hori, A., Harada, H., Takahashi, T., Ishikawa, Y.: The design and evaluation of high performance communication using a Gigabit Ethernet. In: International Conference on Supercomputing, Rhodes, Greece (1999)
17. Shivam, P., Wyckoff, P., Panda, D.: EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In: Proc. of the 2001 Conference on Supercomputing. (2001)
18. Lockwood, J.W., Turner, J.S., Taylor, D.E.: Field programmable port extender (FPX) for distributed routing and queueing. In: Proc. of the ACM International Symposium on Field Programmable Gate Arrays, Napa Valley, CA (1997) 30–39

19. McHenry, J.T., Dowd, P.W., Pellegrino, F.A., Carrozzi, T.M., Cocks, W.B.: An FPGA-based coprocessor for ATM firewalls. In: Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA (1997) 30–39
20. Underwood, K.D., Sass, R.R., Ligon, W.B.: Analysis of a prototype intelligent network interface. Concurrency and Computing: Practice and Experience (2002)
21. National Laboratory for Applied Network Research: Network performance measuring tool, <http://dast.nlanr.net/Projects/Iperf/>. (2003)
22. Jarvinen, K., Tommiska, M., Skytta, J.: Fully pipelined memoryless 17.8 Gbps AES-128 encryptor. In: Eleventh ACM International Symposium on Field- Programmable Gate Arrays (FPGA 2003), Monterey, California (2003)

Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System

Ioannis Soudris and Dionisios Pnevmatikatos

Microprocessor and Hardware Laboratory,
Electronic and Computer Engineering Department,
Technical University of Crete, Chania, GR 73 100, Greece
`{soudris,pnevmati}@mhl.tuc.gr`

Institute of Computer Science (ICS),
Foundation for Research and Technology-Hellas (FORTH),
Vasilika Vouton, Heraklion, GR 71110, Greece
`pnevmati@ics.forth.gr`

Abstract. Intrusion Detection Systems such as Snort scan incoming packets for evidence of security threats. The most computation-intensive part of these systems is a text search against hundreds of patterns, and must be performed at wire-speed. FPGAs are particularly well suited for this task and several such systems have been proposed. In this paper we expand on previous work, in order to achieve and exceed a processing bandwidth of 11Gbps. We employ a scalable, low-latency architecture, and use extensive fine-grain pipelining to tackle the fan-out, match, and encode bottlenecks and achieve operating frequencies in excess of 340MHz for fast Virtex devices. To increase throughput, we use multiple comparators and allow for parallel matching of multiple search strings. We evaluate the area and latency cost of our approach and find that the match cost per search pattern character is between 4 and 5 logic cells.

1 Introduction

The proliferation of Internet and networking applications, coupled with the widespread availability of system hacks and viruses have increased the need for network security. Firewalls have been used extensively to prevent access to systems from all but a few, well defined access points (ports), but firewalls cannot eliminate all security threats, nor can they detect attacks when they happen.

Network Intrusion Detection Systems (NIDS) attempt to detect such attempts by monitoring incoming traffic for suspicious contents. They use simple rules (or search patterns) to identify possible security threats, much like virus detection software, and report offending packets to the administrators for further actions. Snort is an open source NIDS that has been extensively used and studied in the literature [1–4]. Based on a rule database, Snort monitors network traffic and detect intrusion events. An example of a Snort rule is:

```
alert tcp any any ->192.168.1.0/24 111(content: "idc|3a3a|"; msg:  
"mountd access";)
```

A rule contain fields that can specify a suspicious packet's protocol, IP address, Port, content and others. The "content" (idc|3a3a|) field contains the pattern that is to be matched, written in ascii, hex or mixed format, where hex parts are between vertical bar symbols "|". Patterns of Snort V1.9.x distribution contain between one and 107 characters.

NIDS rules may refer to the header as well as to the payload of a packet. Header rules check for equality (or range) in numerical fields and are straightforward to implement. More computation-intensive is the text search of the packet payload against hundreds of patterns that must be performed at wire-speed [3, 4]. FPGAs are very well suited for this task and many such systems have been proposed [5–7]. In this paper we expand on previous work, in order to achieve and exceed a processing bandwidth of 10Gbps, focusing on the string-matching module.

Most proposed FPGA-based NIDS systems use finite automata (either deterministic or non-deterministic) [6, 8, 9] to perform the text search. These approaches are employed mainly for their low cost, which is reported to be is between 1 and 1.5 logic elements per search pattern character. However, this cost increases when other system components are included. Also, the operation of finite automata is limited to one character per cycle operation. To achieve higher bandwidth researchers have proposed the use of packet-level parallelism [6], whereby multiple copies of the automata work on different packets at lower rate. This approach however may not work very well for IP networks due to variability of the packet size and the additional storage to hold these (possibly large) packets. Instead, in this work we employ full-width comparators for the search. Since all comparators work on the same input (one packet), it is straightforward to increase the processing bandwidth of the system by adding more resources (comparators) that operate in parallel. We evaluate the implementation cost of this approach and suggest ways to remedy the higher cost as compared to (N)DFA. We use extensive pipelining to achieve higher operating frequencies, and we address directly the fan-out of the packet to the multiple search engines, one of limiting factors reported in related work [2]. We employ a pipelined fan-out tree and achieve operating frequencies exceeding 245 MHz for VirtexE and 340 MHz for Virtex2 devices (post place & route results).

In the following section we present the architecture of our FPGA implementation, and in section 3 we evaluate the performance and cost of our proposed architecture. In section 4 we give an overview of FPGA-based string matching and compare our architecture against other proposed designs, and in section 5 we summarize our findings and present our conclusions.

2 Architecture of Pattern Matching Subsystem

The architecture of an FPGA-based NIDS system includes blocks that match header fields rules, and blocks that perform text match against the entire packet

payload. Of the two, the computationally expensive module is the text match. In this work we assume that it is relatively straightforward to implement the first module(s) at high speed since they involve a comparison of a few numerical fields only, and focus in making the pattern match module as fast as possible.

If the text match operates at one (input) character per cycle, the total throughput is limited by the operating frequency. To alleviate this bottleneck suggested using packet parallelism where multiple copies of the match module scan concurrently different packet data. However, due to the variable size of the IP packets, this approach may not offer the guaranteed processing bandwidth. Instead, we use discrete comparators to implement a CAM-like functionality. Since each of these comparators is independent, we can use multiple instances to search for a pattern in a wider datapath. A similar approach has been used in [7].

The results of the system are (i) an indication that there was indeed a match, and (ii) the number of the rule that did match. Our architecture uses fine grain pipeline for all sub-modules: fan-out of packet data to comparators, the comparators themselves, and for the encoder of the matching rule. Furthermore to achieve higher processing throughput, we utilize N parallel comparators per search rule, so as to process N packet bytes at the same time. In the rest of this section we expand on our design in each of these sub-modules. The overall architecture we assume is depicted in Figure 1. In the rest of the paper we concentrate on the text match portion of the architecture, and omit the shaded part that performs the header numerical field matching. We believe that previous work in the literature have fully covered the efficient implementation of such functions [6, 7]. Next we describe the details of the three main sub-systems: the comparators, the encoder and the fan-out tree.

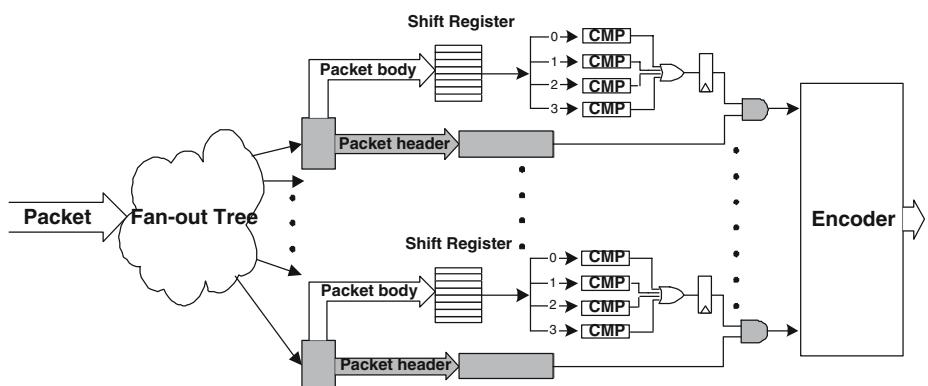


Fig. 1. Envisioned FPGA NIDS system: Packets arrive and are fan-out to the matching engines. N parallel comparators process N characters per cycle (four in this case), and the matching results are encoded to determine the action for this packet. Shaded is the header matching logic that involves numerical field matching.

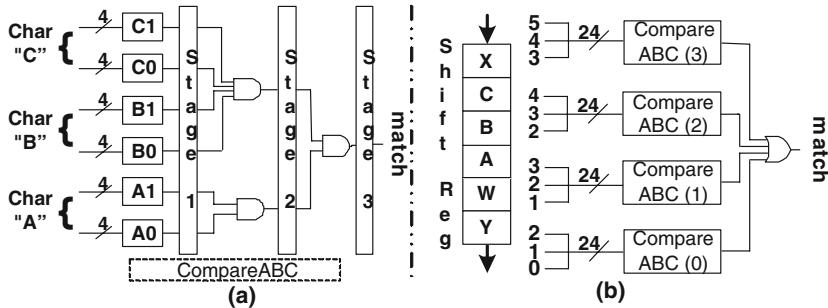


Fig. 2. (a) Pipelined comparator, which matches pattern "ABC". (b) Pipelined comparator, which matches pattern "ABC" starting at four different offsets.

2.1 Pipelined Comparator

Our pipelined comparator is based on the observation that the minimum amount of logic in each pipeline stage can fit in a 4-input LUT and its corresponding register. This decision was made based on the structure of Xilinx CLBs, but the structure of recent Altera devices is very similar so our design should be applicable to Altera devices as well. In the resulting pipeline, the clock period is the sum of wire delay (routing) plus the delay of a single logic cell (one 4-input LUT + 1 flip-flop). The area overhead cost of this pipeline is zero since each logic cell used for combinational logic also includes a flip-flop. The only drawback of this deep pipeline is a longer total delay (in clock cycles) of the result. However, since the correct operation of NIDS systems does not depend heavily on the actual latency of the results, this is not a crucial restriction for our system architecture. In section 3 we evaluate the latency of our pipelines to show that indeed they are within reasonable limits.

Figure 2(a) shows a pipelined comparator that matches the pattern "ABC". In the first stage comparator matches the 6 half bytes of the incoming packet data, using six 4-input LUTs. In the following two stages the partial matches are AND-ed to produce the overall match signal. Figure 2(b) depicts the connection of four comparators that match the same pattern shifted by zero, one, two and three characters (indicated by the numerical suffix in the comparator label). Comparator comparator_ABC(0) checks bytes 0 to 2, comparator_ABC(1) checks bytes 1 to 3 and so on. Notice that the choice of four comparators is only indicative; in general we can use N comparators, allowing the processing of N bytes per cycle.

2.2 Pipelined Encoder

After the individual matches have been determined, the matching rule has to be encoded and reported to the rest of the system (most likely software). We use a hierarchical pipelined encoder. In every stage, the combinational logic is

described by at most 4-input, 1-output logic functions, which is permitted in our architecture.

The described encoder assumes that at most one match will occur in order to operate correctly (i.e. it is not a priority encoder). While in general multiple matches can occur in a single cycle, in practice we can determine by examining the search strings whether this situation can occur in practice. If all the search patterns have distinct suffixes, then we are ensured that we will not have multiple matches in a single cycle. However, this guarantee becomes more difficult as we increase the number of concurrent comparators. To this end we are currently working on a pipelined version of a priority encoder, which will be able to correctly handle any search string combination.

2.3 Packet Data Fan-Out

The fan-out delay is major slow-down factor that designers must take into account. While it involves no logic, signals must traverse long distances and potentially suffer significant latencies. To address this bottleneck we created a register tree to "feed" the comparators with the incoming data. The leaves of this tree are the shift registers that feed the comparators, while the intermediate nodes of the tree serve as buffers and pipeline registers at the same time. To determine the best fan-out factor for the tree, we experimented with the Xilinx tools, and we determined that for best results, the optimal fan-out factor changes from level to level. In our design we used small fan-out for the first tree levels and increase the fan-out in the later levels of the tree up to 15 in the last tree level. Intuitively, that is because the first levels of the tree feed large blocks and the distance between the fed nodes is much larger than in last levels. We also experimented and found that the optimal fan-out from the shift-registers is 16 (15 wires to feed comparators and 1 to the next register of shift register).

2.4 VHDL Generator

Deriving a VHDL representation starting from a Snort rule is very tedious; to handle tens of hundred of rules is not only tedious but extremely error prone. Since the architecture of our system is very regular, we developed a C program that automatically generates the desired VHDL representation directly from Snort pattern matching expressions, and we used a simple PERL script to extract all the patterns from a Snort rule file.

3 Performance Evaluation

The quality of an FPGA-based NIDS can be measured mainly using performance and area metrics. We measure performance in terms of operating frequency (to indicate the efficiency of our fine grain pipelining) and total throughput that can be serviced, and we measure total area, as well as area cost per search pattern character.

We evaluate our proposed architecture we used three sets of rules. The first is an artificial set that cannot be optimized (i.e. at every position all search characters are distinct), that contains 10 rules matching 10 characters each. We also used the "web-attacks.rules" from the Snort distribution, a set of 47 rules to show performance and cost for a medium size rule set, and we used the entire set of web rules (a total of 210 rules) to test the scalability of our approach for large rule sets. The average search pattern length for these sets was 10.4 and 11.7 characters for the Web-attack and all the Web rules respectively.

We synthesized each of these rule sets using the Xilinx tools (ISE 4.2i) for several devices (the $-N$ suffix indicates speed grade): Virtex 1000-6, VirtexE 1000-8, Virtex2 1000-5, VirtexE 2600-8 and Virtex2 6000-5. The structure of these devices is similar and the area cost of our design is expected (and turns out) to be almost identical for all devices, with the main difference in the performance.

The top portion of Table 1 summarizes our performance results. It lists the number of bits processed per cycle, the device, the achieved frequency and the corresponding throughput (in Gbps). It also lists the total area and the area required per search pattern character (in logic cells) of rules, the corresponding device utilization, as well as the dimensions of the rule set (number of rules and average size of the search patterns). For brevity we only list results for four parallel comparators, i.e. for processing 32 bits of data per cycle. The reported operating frequency gives a lower bound on the performance using a single (or fewer) comparators.

In the top portion of the table we can see that for our synthetic rule set (labeled 10x10) we are able to achieve throughput in excess of 6 Gbps for the simplest devices and over 12 Gbps for the advanced devices. For the actual Web attack rule set (labeled 47x10.4), we are able to sustain over 5 Gbps for the simplest Virtex 1000 device (at 171 MHz), and about 11 Gbps for a Virtex2 device (at 345 MHz). The performance with a VirtexE device is almost 8 Gbps at 245 MHz. Since the architecture allows a single logic cell at each pipeling stage, and the percentage of the wire delay in the critical path is around 50%, it is unlikely that these results can be improved significantly.

However the results for larger rule sets are more conservative. The complete set of web rules (labeled 210x11.7) operates at 204MHz and achieve a throughput of 6.5 Gbps on a VirtexE, and at 252MHz having 8 Gbps throughput on a Virtex2 device. Since the entire design is larger, the wiring latency contribution to the critical path has increased to 70% of the cycle time. The total throughput is still substantial, and can be improved by using more parallel comparators, or possibly by splitting the design in sub-modules that can be placed and routed in smaller area, minimizing the wire distances and hence latency.

In terms of implementation cost of our proposed architecture, we see that each of the search pattern characters costs between 15 and 20 logic cells depending on the rule set. However, this cost includes the four parallel comparators, so the actual cost of each search pattern character is roughly 4-5 logic cells multiplied by N for N times larger throughput.

Table 1. Detailed comparison of string matching FPGA designs

Description	Input Bits/ c.c.	Device	Freq. MHz	Throu- ghput (Gbps)	Logic Cells ¹	Logic Cells/ char	Utili- zation	#Patterns × #Characters
Sourdis-Pnevmatikatos Discrete Comparators	32	Virtex 1000	193 171	6.176 5.472	1,728 8,132	17.28 16.64	7% 33%	10 × 10 47 × 10.4
		VirtexE 1000	272 245	8.707 7.840	1,728 7,982	17.28 16.33	7% 33%	10 × 10 47 × 10.4
		Virtex2 1000	396 344	12.672 11.008	1,728 8,132	16.86 16.64	16% 80%	10 × 10 47 × 10.4
		VirtexE 2600	204	6.524	47,686	19.40	94%	210 × 11.7
		Virtex2 6000	252	8.064	47,686	19.40	71%	210 × 11.7
Sidhu et al.[9] NFAs/Reg. Expression	8	Virtex 100	93.5 57.5	0.748 0.460	280 1,920	~31 ~66	11% 80%	(1×) 9 ⁴ (1×) 29 ⁴
Franklin et al.[8] Regular Expressions	8	Virtex 1000	31 99 63.5	0.248 0.792 0.508	20,618 314 1726	2.57 3.17 3.41	83% 1% 7%	8,003 ⁵ 99 506
		VirtexE 2000	50 127 86	0.400 1.008 0.686	20,618 314 1726	2.57 3.17 3.41	53% 1% 4%	8,003 99 506
Lockwood[6] DFAs 4 Parallel FSMs on different Packets	32	VirtexE 2000	37	1.184	4,067 ²	16.27	22% ²	34 × 8 ³
Lockwood[10] FSM+counter	32	VirtexE 1000	119	3.808	98	8.9	0.4%	1 x 11
Gokhale et al.[5] Discrete Comparators	32	VirtexE 1000	68	2.176	9,722	15.2	39%	32 × 20
Young Cho et al.[7] Discrete Comparators	32	Altera EP20K	90	2.880	N/A	10	N/A	N/A

¹ Two *Logic Cells* form one *Slice*, and two Slices form one *CLB* (4 Logic Cells).² These results does not includes the cost/area of infrastructure and protocol wrappers³ 34 regular expressions,with 8 characters on average, (about 250 character)⁴ One regular Expression of the form $(a \mid b)^*a(a \mid b)^k$ for $k = 8$ and 28. Because of the $*$ operator the regular expression can match more than 9 or 29 characters.⁵ Sizes refer to Non-meta characters and are roughly equivalent to 800, 10, and 50 patterns of 10 characters each.

We compute the latency of our design taking into account the three components of our pipeline: fan-out, match, encode. Since the branching factor is not fixed in the fan-out tree, we cannot offer a closed form for the number of stages. The pipeline depths for the designs we have implemented are: $3 + 5 + 4 = 12$ for the Synth10 rule set, $3 + 6 + 5 = 14$ for the Web Attacks rule set, and $5 + 7 + 7 = 19$ for the Web-all rule set. For 1,000 patterns and pattern lengths of 128 characters, we estimate the total delay of the system to be between 20 and 25 clock cycles.

We also evaluated resource sharing to reduce the implementation cost. We sorted the 47 web attack rules, and we allowed two adjacent patterns to share comparator i if their i^{th} characters were the same, and found that the number of logic cells required to implement the system was reduced by about 30%. Due to space limitations, we do not expand on this option in detail. However, it is a very promising approach to reduce the implementation cost, and allow even more rules to be packed in a given device.

4 Comparison with Previous Work

In this section we attempt a fair comparison with previous reported research. While we have done our best to report these results with the most objective way, we caution the reader that this task is difficult since each system has its own assumptions and parameters, occasionally in ways that are hard to quantify.

One of the first attempts in string matching using FPGAs, presented in 1993 by Pryor, Thistle and Shirazi [11]. Their algorithm, implemented on Splash 2, succeeded to perform a dictionary search, without case sensitivity patterns, that consisted of English alphabet characters (26 characters). Pryor et al managed to achieve great performance and perform a low overhead AND-reduction of the match indicators using hashing.

Sidhu and Prassanna [9] used Regular Expressions and Nondeterministic Finite Automata (NFAs) for finding matches to a given regular expression. They focused in minimizing the space $-O(n^2)$ - required to perform the matching, and their automata matched 1 text character per clock cycle. For a single regular expression, the constructed NFAs and FPGA circuit was able to process each text character in 17.42-10.70ns (57.5-93.5 MHz) using a Virtex XCV100 FPGA.

Franklin, Carver and Hutchings [8] also used regular expressions to describe patterns. The operating frequency of the synthesized modules was about 30-100 MHz on a Virtex XCV1000 and 50-127 MHz on a Virtex XCV2000E, and in the order of 63.5 MHz and 86 MHz respectively on XCV1000 and XCV2000E for a few tens of rules.

Lockwood used the Field Programmable Port Extender (FPX) platform, to perform string matching. They used regular expressions (DFAs) and were able to achieve operation at 37 MHz on a Virtex XCV2000E [6]. Lockwood also implemented a sample application on FPX using a single regular expression and were able to achieve operation at 119 MHz on a Virtex V1000E-7 device [10].

Gokhale, et al [5] using CAM to implement Snort rules NIDS on a Virtex XCV1000E. Their hardware runs at 68MHz with 32-bit data every clock cycle, giving a throughput of 2.2 Gbps, and reported a 25-fold improvement on the speed of Snort v1.8 on a 733MHz PIII and an almost 9-fold improvement on a 1 GHz PowerPC G4.

Closer to our work is the recent work by Cho, Navab and Mangione-Smith [7]. They designed a deep packet filtering firewall on a FPGA and automatically translated each pattern-matching component into structural VHDL. The content pattern match unit micro-architecture used 4 parallel comparators for every

pattern so that the system advances 4 bytes of input packet every clock cycle. The design implemented in an Altera EP20K device runs at 90MHz, achieving 2.88 Gbps throughput. They require about 10 logic cells per search pattern character. However, they do not include the fan-out logic that we have, and do not encode the matching rule. Instead they just OR all the match signals to indicate that some rule matched.

The results of these works are summarized in the bottom portion of Table 1, and we can see that most previous works implement a few tens of rules at most, and achieve throughput less than 4 Gbps. Our architecture on the same or equivalent devices achieves roughly twice the operating frequency and throughput. In terms of best performance, we achieve 3.3 times better processing throughput compared with the fastest published design which implements a single search pattern. Our 210-rule implementation achieves at least a 70% improvement in throughput compared to the fastest existing implementation.

5 Conclusions and Future Work

We have presented an architecture for Snort rule match in FPGAs. We propose the use of extensive fine grain pipelining in order to achieve high operating frequencies, and parallel comparators to increase the processing throughput. This combination proves very successful, and the throughput of our design exceeded 11 Gbps for about 50 Snort rules. These results offer a distinct step forward compared to previously published research. If latency is not critical to the application, fine grain pipelining is very attractive in FPGA-based designs: every logic cell contains one LUT and one Flip-Flop, hence the pipeline area overhead is zero. The current collection of Snort rules contains less than 1500 patterns, with an average size of 12.6 characters. Using the area cost as computed earlier, we need about 3 devices of 120,000 logic cells to include the entire Snort pattern matching, and about 4 devices to include the entire snort rule set including header matching. These calculations do not include area optimizations, which can lead to further significant improvements.

Throughout this paper we used four parallel comparators. However, a different level of parallelism can also be used depending on the bandwidth demands. Reducing the processing width leads to a smaller, possibly higher frequency design, while increasing the processing width leads to a bigger and probably lower frequency design. Throughput depends on both frequency and processing width, so we need to seek for the cost effective tradeoff of these two factors.

Despite the significant body of research in this area, there are still improvements that we can use to seek better solutions. In our immediate goals is to use the hierarchical decomposition of large rule set designs, and attempt to use the multiple clock domains. The idea is to use a slow clock to drive long wide busses to distribute data and a fast clock for local processing that only uses local wiring. The target would be to retain the frequency advantage of our medium-sized design (47 rules) for a much larger rule set. All the devices we used already support multiple clock domains, and with proper placement and routing tool

support this approach will also be quicker to implement: each of the modules can be placed and routed locally one after the other, reducing the memory and processing requirements for placement and routing.

Furthermore, future devices offer significant speed improvements, and it would be interesting to see whether the fine grain pipelining will be as effective for these devices (such as the Virtex 2 Pro) as it was for the devices we used in this work.

Acknowledgments

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union.

References

1. SNORT official web site: (<http://www.snort.org>)
2. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of LISA'99: 13th Administration Conference. (1999) Seattle Washington, USA.
3. Desai, N.: Increasing performance in high speed NIDS. In: www.linuxsecurity.com. (2002)
4. Coit, C.J., Staniford, S., McAlerney, J.: Towards faster string matching for intrusion detection or exceeding the speed of snort. In: DISCEXII, DAPRA Information Survivability conference and Exposition. (2001) Anaheim, California, USA.
5. Gokhale, M., Dubois, D., Dubois, A., Boorman, M., Poole, S., Hogsett, V.: Granidt: Towards gigabit rate network intrusion detection technology. In: Proceedings of 12th International Conference on Field Programmable Logic and Applications. (2002) France.
6. Moscola, J., Lockwood, J., Loui, R.P., Pachos, M.: Implementation of a content-scanning module for an internet firewall. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. (2003) Napa, CA, USA.
7. Young H. Cho, S.N., Mangione-Smith, W.: Specialized hardware for deep network packet filtering. In: Proceedings of 12th International Conference on Field Programmable Logic and Applications. (2002) France.
8. Franklin, R., Carver, D., Hutchings, B.: Assisting network intrusion detection with reconfigurable hardware. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2002)
9. Sidhu, R., Prasanna, V.K.: Fast regular expression matching using fpgas. In: IEEE Symposium on Field-Programmable Custom Computing Machines. (2001) Rohnert Park, CA, USA.
10. Lockwood, J.W.: An open platform for development of network processing modules in reconfigurable hardware. In: IEC DesignCon '01. (2001) Santa Clara, CA, USA.
11. Pryor, D.V., Thistle, M.R., Shirazi, N.: Text searching on splash 2. In: Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. (1993) 172–177

Irregular Reconfigurable CAM Structures for Firewall Applications

T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu, and N. Dulay

Department of Computing,

Imperial College, 180 Queen's Gate, London SW7 2BZ, UK

{tk197, sy99, w.luk, m.sloman, e.c.lupu, n.dulay}@doc.ic.ac.uk

Abstract. Hardware packet-filters for firewalls, based on content-addressable memory (CAM), allow packet matching processes to keep in pace with network throughputs. However, the size of an FPGA chip may limit the size of a firewall rule set that can be implemented in hardware. We develop two irregular CAM structures for packet-filtering that employ resource sharing methods, with various trade-offs between size and speed. Experiments show that the use of these two structures are capable of reduction, up to 90%, of hardware resources without losing performance.

1 Introduction

FPGA-based firewall processors have been developed for high-throughput networks [3, 6, 7]. Such firewall processors must be able to carry out packet matching effectively based on filter rules. Each filter rule consists of a set of logical operations on different fields of an input packet header. A ‘don’t care’ condition indicates that a field can match any value.

Content-Addressable Memory (CAM) is a searching device that consists of an array of storage locations. A search result can be obtained in constant time through parallel matching of the input with the data in the memory array. CAM based hardware packet-filters [4] are fast and support various data widths [3]. However, the size of an FPGA may limit the number of filter rules that can be implemented in hardware [7]. We describe two hardware structures that employ resource sharing methods to reduce hardware resource usage for packet-filtering firewalls. Resource usage reduces approximately linearly with the degree of grouping of the filter rules in a rule set. These two structures, when applied to CAM based packet-filters, offer various trade-offs between speed and size under different situations involving parallel and pipelined implementations. The contributions described in this paper include:

1. two hardware irregular CAM structures for implementing filter rules;
2. a strategy to generate hardware firewall processors;
3. an evaluation of the effectiveness of the irregular CAM structures, comparing them against regular CAMs.

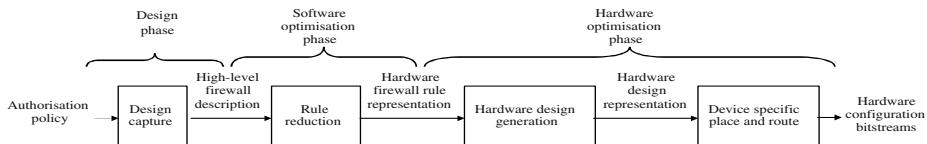


Fig. 1. An overview of our development framework for reconfigurable firewall processors. There are three main phases: the *design phase*, the *software optimisation phase*, and the *hardware optimisation phase*.

The rest of the paper is organised as follows. Section 2 gives an overview of our design framework. Section 3 describes our hardware structures for filter rules. Section 4 outlines the design generation. Section 5 evaluates the performance of our approach in terms of speed and size, and Section 6 provides a summary of current and future work.

2 Framework Overview

As shown in Figure 1, our framework for developing reconfigurable-hardware packet filtering firewalls consists of three main phases: the design phase, the software optimisation phase, and the hardware optimisation phase.

During the *design phase*, the requirements of a firewall are captured as a high-level description. We use a subset of Ponder [2], a policy specification language, to create our firewall description language [5]. This firewall description uses Ponder's parameterised types and the concept of domains. Our high-level firewall description supports abstraction from details of the hardware implementation. It uses constraints to specify low-level hardware requirements such as placement and partitioning, run-time reconfiguration, timing and size requirements, and hardware software co-operation.

During the *software optimisation phase*, high-level firewall rules are reduced and converted to a hardware firewall rule representation, using parameterised library specifications. We have developed a series of rule reduction steps to reduce hardware usage by employing rule elimination and rule sharing methods [5]. A rule set is divided into a number of groups of hardware filter rules. Sequencing is performed to preserve the ordering and the semantics of a rule set. Reordering and partitioning is conducted to facilitate the grouping process. Each group consists of either a list of rules related by common attributes, or a singleton rule if no sharing with other rules can be found within the same partition.

During the *hardware optimisation phase*, hardware firewall rule representations are converted to a hardware design which is then used to produce the hardware configuration bitstreams for downloading onto an FPGA. The next section describes the irregular CAM structures that we develop, and their use in implementing a rule set in hardware and in facilitating resource sharing.

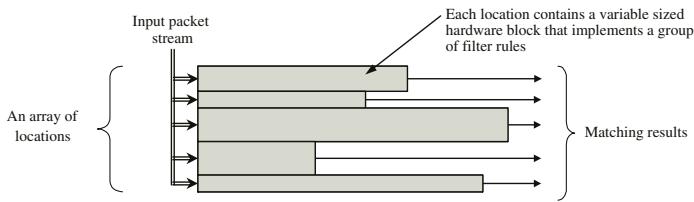


Fig. 2. A rule set in hardware implemented as an irregular CAM structure. Each location contains a *variable sized* hardware block that implements a *group* of filter rules. An array of hardware blocks together form a CAM structure that supports the whole rule set.

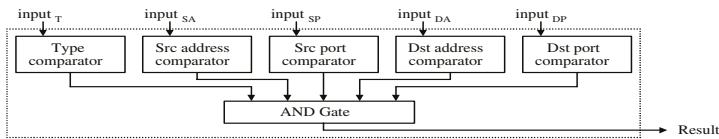


Fig. 3. A hardware block for a group of singleton filter rule. Instead of having a single matching processor as in a regular CAM, our hardware blocks for filter rules contain several field-level hardware comparators that correspond to different fields of an input packet. When a hardware block is instantiated, filter rules that contain ‘don’t care’ fields will have the corresponding hardware comparators eliminated. This example shows the situation when no ‘don’t care’ fields are involved in a filter rule.

3 Architecture of Irregular CAM for Firewall Rules

Our approach for implementing a firewall rule set in hardware is to construct a specialised CAM structure, as shown in Figure 2, to perform packet-filtering. Conventional regular CAM structures store a *single* matching criterion in each memory location. However, instead of having a one-to-one mapping of a filter rule to a CAM location, we construct each CAM location as a hardware block that implements a *group* of filter rules.

A rule set is divided into several groups of filter rules as described in Section 2. Each of these groups is implemented as a hardware block that corresponds to a CAM location. These *variable sized* hardware blocks together then produce an irregular CAM structure that represents the whole filter rule set.

Hardware blocks are instantiated according to the types of grouping and the combinations of field attributes. Figure 3 shows a hardware block for a group of singleton filter rule. It contains several field-level hardware comparators that correspond to different fields of the input. Matching results are obtained as the unified result from all the individual comparators. This design is functionally the same as a regular CAM, except that a regular CAM design will normally use only one comparator for the input data and does not need the AND gate. Separating the matching processor into field-level comparators, however, allows

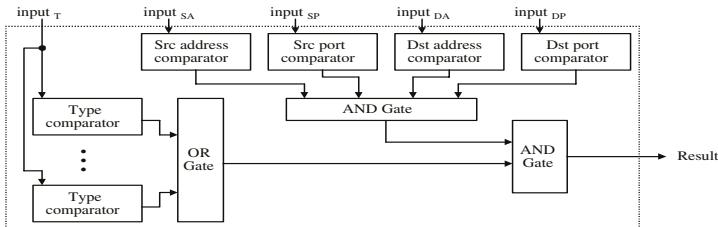


Fig. 4. A hardware block of a group of shared filter rules using the *Siamese Twins* structure. Individual fields of the filter rules having identical data values are shared by using the same hardware comparators. Fields that cannot be shared have their corresponding parts OR-ed together. This example shows that a block is instantiated with all but the *Type*-field comparator being shared.

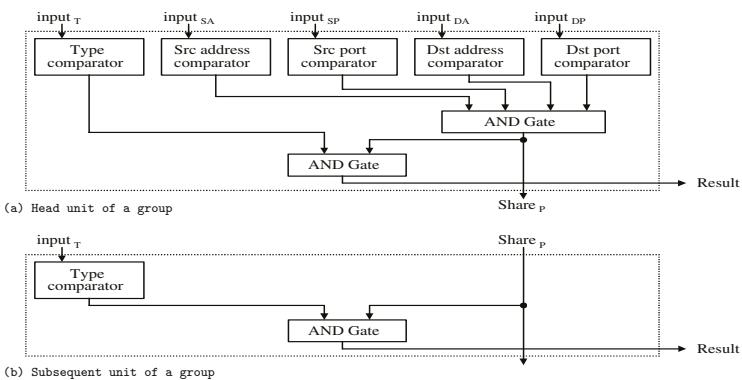


Fig. 5. Hardware blocks of a group of shared filter rules using the *Propaganda* structure. A group of filter rules are sub-divided into a head unit and a number of subsequent units chained to the head. Individual fields of the filter rules having identical data values are shared by using the same hardware comparators in the head unit. The comparison result of the shared fields is propagated from the head unit to each of the subsequent units in the group. Fields that cannot be shared are AND-ed with the propagating result individually. This example shows that the blocks are instantiated with all but the *Type*-field being shared.

us to achieve reduction in resource usage at the expense of introducing a multi-input AND-gate. When a hardware block is instantiated, filter rules that contain ‘don’t care’ fields will have the corresponding hardware comparators eliminated.

Within a group of rules, hardware resources are shared by attributes that are common. There are two levels of sharing: field-level sharing and bit-level

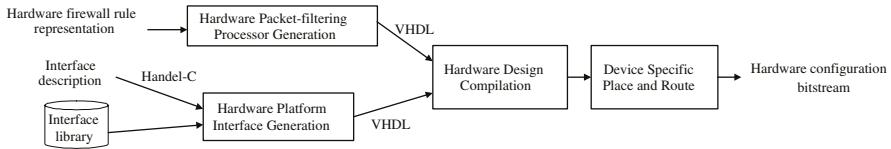


Fig. 6. Hardware firewall processor generation. To improve flexibility, the packet-filtering processing unit and the platform-specific interfaces are separately generated.

sharing [5]. We develop two irregular CAM structures: the *Siamese Twins* and the *Propaganda*. They both provide field-level sharing but have different trade-offs between size and speed. A group of shared filter rules are implemented as hardware blocks using either the Siamese Twins structure, which is optimised for area, or the Propaganda structure, which is optimised for speed. Figure 4 and Figure 5 show some examples of hardware blocks for a group of filter rules using the two structures.

In the Siamese Twins structure, the fields of a group of filter rules with identical data values are shared by using the same field-level hardware comparators. Fields that cannot be shared have their corresponding parts OR-ed together. This organisation has the advantage of a simple design, and results in reduction of resource usage by eliminating redundant hardware comparators.

In the Propaganda structure, a group of filter rules are sub-divided into a head unit and a number of subsequent units chained to the head. Individual fields of the filter rules with identical data values are shared by using the same field-level hardware comparators in the head unit. The comparison result of the shared fields is propagated from the head unit to each of the subsequent units in the group. Fields that cannot be shared are AND-ed with the propagating result individually. Filter rules implemented using the Propaganda structure result in a list of hardware blocks joined together. Each filter rule within a group corresponds to a hardware block. The length of the list varies with the number of rules in a group. This is in contrast to the hardware blocks of Siamese Twins or singleton filter rules, where there is only one unit.

4 Hardware Firewall Processor Generation

To generate our hardware firewall processors (Figure 6), we separate the generation of the platform-specific interfaces from the generation of the filtering processor unit. The interfaces are written in the Handel-C language [1], which facilitates porting the design to various hardware platforms.

We design the hardware code generator that takes the hardware firewall rule representation as input, and generate the hardware packet-filtering processor (Figure 6) in VHDL. During the implementation of a CAM location, a hardware block is instantiated according to the attributes of the field in a group of filter rules. These include the combinations of fields that are shared and not shared, and the number of rules in a group. Furthermore, there are structures

for replicating and connecting the non-shared fields for a group of rules. All our hardware blocks can be used in both parallel and pipelined mode.

Our implementations target the Xilinx Virtex series FPGAs. We follow the vendor's recommendation [8] of reprogramming lookup tables as Shift Registers (SRL 16). The JBits tool is then used to reconfigure the SRL 16 blocks to desired matching values, for various locations in our irregular CAMs.

5 Performance Evaluation

To analyse the performance of our irregular CAM, we compare implementations that employ the Siamese Twins and the Propaganda structures against those based on regular CAMs. We evaluate the implementations in terms of clock speed and hardware resource consumption.

In addition to using rule sets from network sites, we also generate artificial rule sets. Our rule set generator is based on real filter rule sets and covers a wider spectrum of possible real situations as well as some worst-case scenarios. The test data include the effects of ‘don’t care’ fields, the degree to which rules are grouped, and the size of rule sets. For the purpose of the experiments, ‘degree of grouping’ means the percentage of rules within a rule set that are in a shared group. The resource usage figures include resource to support the I/O to RAM, which is a fixed overhead and is insignificant when compared with the overall resources required by a rule set. All the experiments are performed on a Celoxica RC1000-PP reconfigurable hardware platform that contains a Xilinx Virtex XCV1000 FPGA device.

5.1 Resource Usage

Figure 7 shows the resource usage for rule sets with different degrees of grouping, and the effects of ‘don’t care’ fields. The resource usage of regular CAM remains unchanged as the degree of grouping varies.

When the degree of grouping is at 0% as shown on the left-hand-side of Figure 7 (a) and (b), there is no reduction in resource usage in the case of no ‘don’t care’ fields, but there is around 45% reduction in the case with ‘don’t care’ fields for both the Siamese Twins and the Propaganda structures. A rule set that does not contain any ‘don’t care’ fields in all of its rules is unrealistic. In reality, most rule sets contain a certain amount of ‘don’t care’ fields. This suggests that both Siamese Twins and Propaganda will achieve reduction in resource usage over a regular CAM, whenever a ‘don’t care’ field exists in a rule set.

For the parallel versions, the resource usage of Siamese Twins and Propaganda are about the same as shown in the lower parts of Figure 7 (a) and (b), where their corresponding graphs almost overlap. For the pipelined version, Propaganda uses noticeably more resources than Siamese Twins. This is due to the additional pipeline registers. This suggests that both Siamese Twins and Propaganda are suitable for implementations involving parallel structures. However, if an implementation must involve pipelining and when resource usage is the main concern, Siamese Twins is the preferred choice.

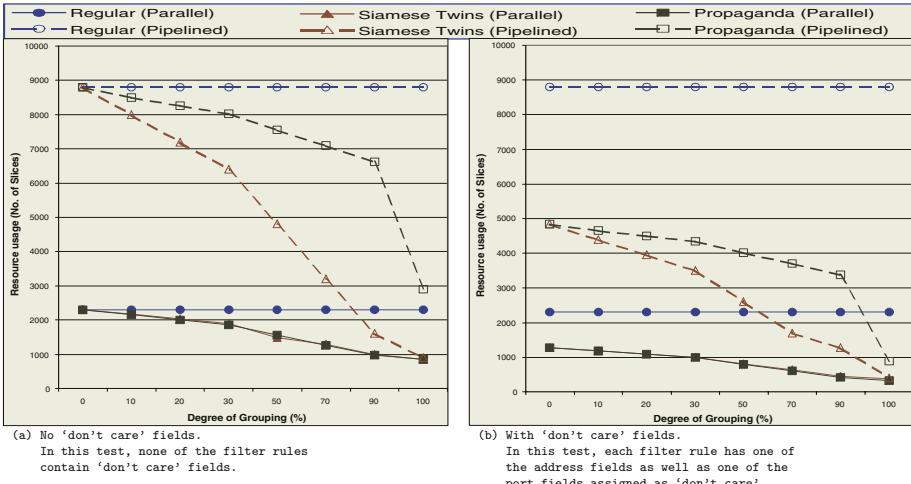


Fig. 7. Resource usage versus degree of grouping. Resource usage of both Siamese Twins and Propaganda reduce approximately linearly with the degree of grouping. For the parallel versions, both structures perform almost identically. For the pipelined versions, Siamese Twins is considerably better. Note that for this test, one of the address field is not shared in a group. Since the address field is the largest field in a filter rule, it gives the worst case resource usage for a single non-shared field.

When the degree of grouping is low (less than 10%), the pipelined versions consume 3.8 times more resources than their parallel counterparts. This shows the well-known trade-offs between speed and size. However, when the degree of grouping is high (larger than 70% in the case of no 'don't care' fields, and larger than 50% in the case with 'don't care' fields), the pipelined versions of Siamese Twins consume comparable or fewer resources than the parallel versions of the regular CAM. These two figures correspond to 138% (in the case of no 'don't care' fields) and 188% (in the case with 'don't care' fields) of the speed of the regular CAM. This suggests that, in situations when both size and speed should be optimised, a pipelined version of Siamese Twins can be better than a parallel version of the regular CAM.

5.2 Speed

Figure 8 shows the speed performance for rule sets with different degrees of grouping, and the effects of 'don't care' fields. The maximum operating frequency of regular CAM remains unchanged as the degree of grouping varies.

Results for the parallel versions are shown in the lower parts of Figure 8 (a) and (b). Both Siamese Twins and Propaganda performs approximately the same as the regular CAM. Results for the pipelined versions are shown in the upper parts of Figure 8 (a) and (b). While Propaganda performs comparable

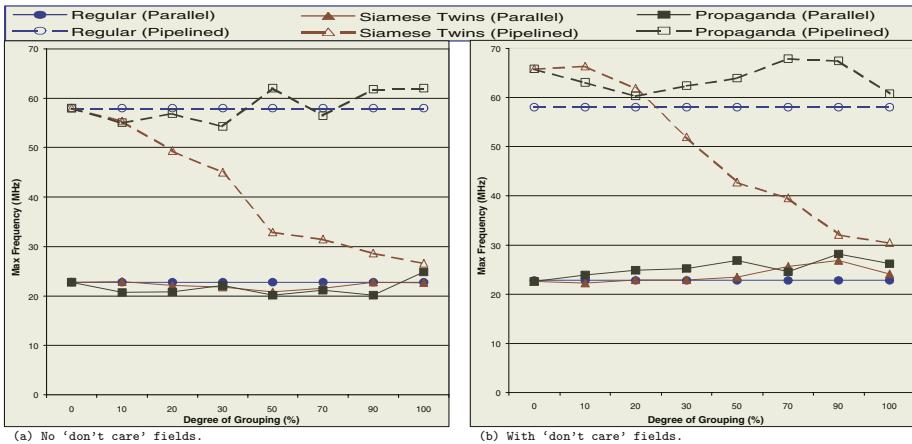


Fig. 8. Speed performance versus degree of grouping. Propaganda consistently achieves comparable performance to the regular CAM. Siamese Twins, while having approximately the same performance as regular CAM in the parallel version, suffers from performance degradation when the degree of grouping increases in the pipelined versions. Note that for this test, all the shared rules using the Siamese Twins structure are grouped into a single CAM location. This produces the highest propagation delay and so the lowest performance.

to or sometimes slightly better than the regular CAM, the Siamese Twins suffers from performance degradation when the degree of grouping increases. This reduction in performance is due to the increased routing and propagation delay of the enlarged OR-structure inside the Siamese Twins. When the degree of grouping is low (less than 10%), both structures are 2.5 times faster than their parallel counterparts. When the degree of grouping is at 100%, the performance of Siamese Twins is reduced by nearly 50% to have similar performance to its parallel counterpart. This suggests that both Siamese Twins and Propaganda are suitable for implementations involving parallel structures. However, if implementations involve pipelining and when speed is also a major concern, Propaganda can be a better choice.

Figure 9 shows that maximum operating frequency is determined not only by the degree of grouping, but also by the maximum group size. For the parallel versions, both Siamese Twins and Propaganda do not vary much with the degree of grouping. For the pipelined versions as shown in the top-left parts of Figure 9 (a) and (b), when the group size is small, the performance of Siamese Twins is comparable to the regular CAM even at 100% degree of grouping. When the group size is large (100 rules/location), its performance decreases by nearly 50%.

The effects of maximum group size suggest that there can be a trade-off between resource utilisation and the maximum operation frequency. In order to avoid performance degradation at a high degree of grouping, one can choose

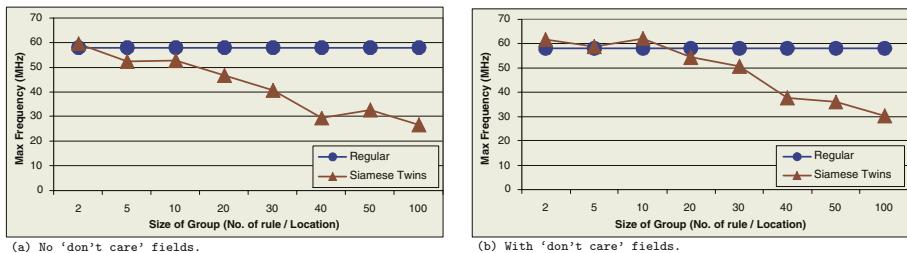


Fig. 9. Speed performance versus maximum group size for pipelined implementations. When the maximum group size is small (less than 20 rules/location in the case of no ‘don’t care’ fields, and less than 30 rules/location in the case with ‘don’t care’ fields) the performance of Siamese Twins and the regular CAM are comparable. In this test, the degree of grouping is always 100%, but the group of shared filter rules are broken down into a number of smaller groups.

to impose either a ceiling group size or a maximum degree of grouping for the pipelined versions of Siamese Twins. For example, groups with number of rules exceeding the ceiling group size can be broken down into several smaller groups. This method can maintain performance, but at the expense of using more hardware resources to implement the additional groups.

5.3 Results Summary

The analysis results are discussed in Section 5.1 and Section 5.2. The maximum reduction in hardware usage and maximum group size before performance degradation are shown respectively in Table 1 and Table 2. For the purpose of the experiments, performance degradation is defined as no more than 10% reduction in speed, when compared to corresponding regular CAMs.

Table 1. Maximum reduction in hardware usage before performance degradation.

		Reduction in hardware usage	Degree of grouping
Siamese Twins	Parallel	84%	100%
	Pipelined	60% with ‘don’t care’	30%
		18% without ‘don’t care’	20%
Propaganda	Parallel	84%	100%
	Pipelined	90%	100%

Table 2. Maximum group size before performance degradation (Siamese Twins pipelined).

	Rule / Location	Reduction in hardware usage
With ‘don’t care’	30	90%
Without ‘don’t care’	10	85%

6 Conclusion

We have presented the Siamese Twins and the Propaganda irregular CAM structures. These two structures employ resource sharing to reduce hardware usage for packet-filtering firewalls. Experiments show that resource usage reduces approximately linearly to the degree of grouping of the filter rules in a rule set. These two irregular CAM structures offer various trade-offs between speed and size, under different situations involving parallel and pipelined implementations. Both structures are capable of reduction, up to 90%, of hardware resources of regular CAMs without losing performance.

Current and future work includes the use of bit-level sharing to achieve further reduction in hardware usage, and global and local optimisations of irregular CAM using the Siamese Twins and the Propaganda structures.

Acknowledgements

The support of UK Engineering and Physical Sciences Research Council (Grant number GR/R 31409, GR/R 55931 and GR/N 66599), Celoxica Limited and Xilinx, Inc. is gratefully acknowledged.

References

1. Celoxica Limited, *Handel-C v3.1 Language Reference Manual*, <http://www.celoxica.com/>.
2. N. Damianou, N. Dulay, E. Lupu and M Sloman, “The Ponder Policy Specification Language”, in *Proc. Workshop on Policies for Distributed Systems and Networks*, LNCS 1995, Springer, 2001, pp. 18-39.
3. J. Ditmar, K. Torkelsson and A. Jantsch, “A Dynamically Reconfigurable FPGA-based Content Addressable Memory for Internet Protocol Characterization”, *Field Programmable Logic and Applications*, LNCS 1896, Springer, 2000.
4. P.B. James-Roxby and D.J. Downs, “An Efficient Content-addressable Memory Implementation Using Dynamic Routing”, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2001.
5. T.K. Lee, S. Yusuf, W. Luk, M. Sloman, E. Lupu and N. Dulay, “Compiling Policy Descriptions into Reconfigurable Firewall Processors”, in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 2003.
6. J.T. McHenry and P.W. Dowd, “An FPGA-Based Coprocessor for ATM Firewalls” in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, IEEE Computer Society Press, 1997.
7. R. Sinnappan and S. Hazelhurst, “A Reconfigurable Approach to Packet Filtering”, *Field Programmable Logic and Applications*, LNCS 2147, Springer, 2001.
8. Xilinx Inc., *Designing Flexible, Fast CAMs with Virtex Family FPGAs*, 1999, <http://www.xilinx.com/>.

Compiling for the Molen Programming Paradigm

Elena Moscu Panainte¹, Koen Bertels¹, and Stamatis Vassiliadis¹

Computer Engineering Lab

Electrical Engineering Department, TU Delft, The Netherlands

{E.Panainte,K.Bertels,S.Vassiliadis}@et.tudelft.nl

Abstract. In this paper we present compiler extensions for the Molen programming paradigm, which is a sequential consistency paradigm for programming custom computing machines (CCM). The compiler supports instruction set extensions and register file extensions. Based on pragma annotations in the application code, it identifies the code fragments implemented on the reconfigurable hardware and automatically maps the application on the target reconfigurable architecture. We also define and implement a mechanism that allows multiple operations to be executed in parallel on the reconfigurable hardware. In a case study, the Molen processor has been evaluated. We considered two popular multimedia benchmarks: mpeg2enc and jpeg and some well-known time-consuming operations implemented in the reconfigurable hardware. The total number of executed instructions has been reduced with 72% for mpeg2enc and 35% for jpeg encoder, compared to their pure software implementations on a general purpose processor (GPP).

1 Introduction and Related Work

In the last decade, several approaches have been proposed for coupling an FPGA to a GPP. For a classification of these approaches the interested reader is referred to [1]. There are four shortcomings of current approaches, namely:

1. **Opcode space explosion:** a common approach (e.g. [2], [3], [4]) is to introduce a new instruction for each portion of application mapped into the FPGA. The consequence is the limitation of the number of operations implemented into the FPGA, due to the limitation of the opcode space. More specifically stated, for a specific application domain intended to be implemented in the FPGA, the designer and compiler are restricted by the unused opcode space.
2. **Limitation of the number of parameters:** In a number of approaches, the operations mapped on an FPGA can only have a small number of input and output parameters ([5], [6]). For example, in the architecture presented in [5], due to the encoding limits, the fragments mapped into the FPGA have at most 4 inputs and 2 outputs; also, in Chimaera [6], the maximum number of input registers is 9 and it has one output register.

3. No support for **parallel execution** on the FPGA of sequential operations: an important and powerful feature of FPGA's can be the parallel execution of sequential operations when they have no data dependency. Many architectures [1] do not take into account this issue and their mechanism for FPGA integration cannot be extended to support parallelism.
4. No **modularity**: each approach has a specific definition and implementation bounded for a specific reconfigurable technology and design. Consequently, the applications cannot be (easily) ported to a new reconfigurable platform. Further there are no mechanisms allowing reconfigurable implementation to be developed separately and ported transparently. That is a reconfigurable implementation developed by a designer A can not be included without substantial effort by the compiler developed for an FPGA implementation provided by a designer B.

A general approach is required that eliminates these shortcomings. In this paper, a programming paradigm for reconfigurable architectures [7], called the Molen Programming Paradigm and a compiler are described that offer alternatives and a solution to the above presented limitations.

The paper is organized as follows: in the next section, we discuss related research and present the Molen programming paradigm. We then describe a particular implementation, called the Molen processor that uses microcoded emulation for controlling the reconfigurable hardware. Consequently, we present the two main elements of the paper, namely the Exchange Register mechanism and the compiler extension for the Molen processor. We finally discuss an experiment comparing the Molen reconfigurable processor with the equivalent non-reconfigurable processor, using two well-known multimedia benchmarks: mpeg2 and jpeg.

2 The Programming Paradigm

The Molen programming paradigm[7] is a sequential consistency paradigm for programming CCMs possibly including a general purpose computational engine(s). The paradigm allows for parallel and concurrent hardware execution and it is intended (currently) for single program execution. It requires only a one time architectural extension of few instructions to provide a large user reconfigurable operation space. The added instructions include:

- Two instructions¹ for controlling the reconfigurable hardware, namely:
 - SET < *address* >: at a particular location the hardware configuration logic is defined
 - EXECUTE < *address* >: for controlling the executions of the operations on the reconfigurable hardware

¹ Actually, five if partial reconfiguration, pre-loading of reconfiguration and executing microcode are also explicitly assumed [7].

- Two move instructions for passing values of to and from the GPP register file and the reconfigurable hardware.

Code fragments constituted of contiguous statements (as they are represented in high-level programming languages) can be isolated as generally implementable functions (that is code with multiple identifiable input/output values). The parameters stored in registers are passed to special reconfigurable hardware registers denoted as Exchange Registers(XRs). The Exchange Register mechanism will be described later in the paper. In order to maintain the correct program semantics, the code is annotated and CCM description files provide the compiler with implementation specific information such as the addresses where the SET and EXECUTE code are to be stored, the number of exchange registers, etc. It should be noted that this programming paradigm allows modularity, meaning that if the interfaces to the compiler are respected and if the instruction set extension (as described above) is supported, then:

- custom computing hardware provided by multiple vendors can be incorporated by the compiler for the execution of the same application.
- the application can be ported to multiple platforms with mere recompilation.

Finally, it is noted that every user is provided with at least $2^{(n-op)}$ directly addressable functions, where n represents the instruction length and 'op' the opcode length. The number of functions can be easily augmented to an arbitrary number by reserving opcode for indirect opcode accessing. From the previous discussion, it is obvious that the programming paradigm and the architectural extensions resolve the aforementioned problems as follows:

- There is only a one time architectural extension of few new instructions to include an arbitrary number of configuration.
- The programming paradigm allows for an arbitrary (only hardware real estate design restricted) number of I/O parameter values to be passed to/from the reconfigurable hardware. It is only restricted by the implemented hardware as any given technology can (and will) allow only a limited hardware.
- Parallelism is allowed as long as the sequential memory consistency model can be guaranteed.
- Assuming that the interfaces are observed, modularity is guaranteed because the paradigm allows freedom of operation implementation.

Parallelism and Concurrency: As depicted in Figure 1, the split-join programming paradigm suggests that the SET instruction does not block the GPP because it can be executed independently from any other instruction. Moreover, a block of consecutive resource conflict free SET instructions (e.g. set op1, set op2 in our example) can be executed in parallel. However, the SET-instruction (set op3) following a GPP-instruction can only be executed after the GPP-instruction is finished. As far as the EXECUTE-instruction is concerned, we distinguish between two distinct cases, one that adds a new instruction and one that does not:

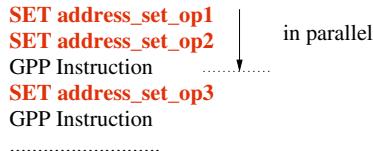


Fig. 1. SET instructions performed concurrently with GPP instructions

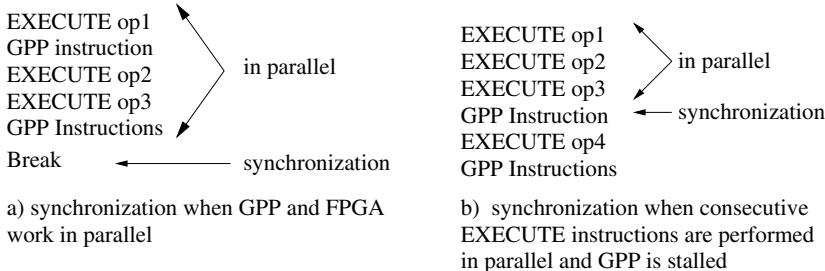


Fig. 2. Models of synchronization

1. If it is found that there is a substantial performance to be gained by parallel execution between GPP and FPGA, then the GPP and EXECUTE-instructions can be issued and executed in parallel. The sequence of instructions performed in parallel is initiated by an EXECUTE instruction. The end of the parallel execution requires an additional instruction (BREAK in the example) indicating where the parallel execution stops (see Figure 2 (a)). A similar approach can be followed for the SET instructions.
2. If such performance is not to be expected (which will most likely be the case for reconfigured “complex” code and GPP code with numerous data dependencies), then a block of EXECUTE-instructions can be executed in parallel on the FPGA while the GPP is stalled. An example is presented in Figure 2(b) where the block of EXECUTE instructions which can be processed in parallel contains the first three consecutive EXECUTE instructions and it is delimited by a GPP instruction.

We note that parallelism is guaranteed by the compiler, that checks whether there are data dependencies and whether the parallel execution is supported by the reconfigurable unit. Moreover, if the compiler detects that a block of SET/EXECUTE instructions cannot be performed in parallel, it separates them by introducing appropriate instructions. In the remaining of the paper, we assume that the separating instruction for SET/EXECUTE is a GPP instruction.

The Molen Reconfigurable Processor: The Molen $\rho\mu$ -coded processor has been designed having in mind the programming paradigm previously presented. The Molen machine organization is depicted in Figure 3.

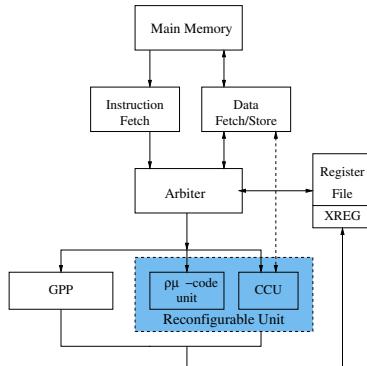


Fig. 3. The Molen machine organization

The arbiter performs a partial decoding of the instructions fetched from the main memory and issues them to the corresponding execution unit. The parameters for the FPGA reside in the Exchange Registers. In the Molen approach, an extended microcode - named reconfigurable microcode - is used for the emulation of both SET and EXECUTE instructions. The microcode is generated when the hardware implementation for a specific operation is defined and it cannot be further modified.

3 Compiler Extensions

In this section we present in detail the mechanism and compiler extensions required to implement the Molen programming paradigm.

The Exchange Registers: The Exchange Registers are used for passing operation parameters to the reconfigurable hardware and returning the computed values after the operation execution. In order to avoid dependencies between the RU and GPP, the XRs receive their data directly from the GPP registers. Therefore, move instructions have to be provided for this communication.

During the EXECUTION phase, the defined microcode is responsible for taking the parameters of its associated operation from XRs and returning the result(s). A single EXECUTE does not pose any specific challenge because the whole set of exchange registers is available. However, when executing multiple EXECUTE instructions in parallel, the following conventions are introduced:

- All parameters of an operation are allocated by the compiler in consecutive XRs and they form a block of XRs.
- The (micro)code of each EXECUTE instruction has a fixed XR, which is assigned when the microcode is developed. The compiler places in this XR a link to the block of XRs where all parameters are stored. This link is the number of the first XR in the block.

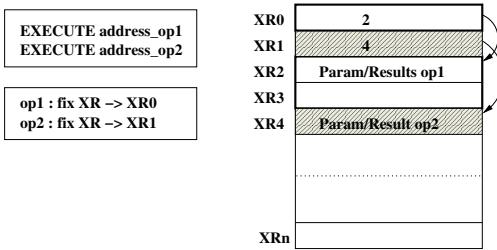


Fig. 4. Exchange Registers allocation by the compiler

Based on these conventions, the parameters for all operations can be efficiently allocated by the compiler and the (micro)code for each EXECUTE instruction is able to determine the associated block of parameters. An example is presented in Figure 4, where two operations, namely *op1* and *op2*, are executed in parallel. Their fix XRs (XR0 and XR1) are communicated to the compiler in a FPGA description file. As indicated by the number stored in XR0, the compiler allocates for operation *op1* two consecutive XRs for passing parameters and returning results, namely XR2 and XR3. The operation *op2* requires only one XR for parameters and results, which in the example is XR4, as indicated by the content of XR1.

Compiler Extensions: The compiler system relies on the Stanford SUIF2[8] (Stanford University Intermediate Format) Compiler Infrastructure for the front-end, while the back-end is built over the framework offered by the Harvard Machine SUIF[9]. The last component has been designed with retargetability in mind. It provides a set of back-ends for GPPs, powerful optimizations, transformations and analysis passes. These are essential features for a compiler targeting a CCM. We have currently implemented the following extensions for the x86 processor:

- Code identification: for the identification of the code mapped on the reconfigurable hardware, we added a special pass in the SUIF front-end. This identification is based on code annotation with special pragma directives (similar to [2]). In this pass, all the calls of the recognized functions are marked for further modification.
- Instruction Set extension: the Instruction Set has been extended with SET/EXECUTE instructions at both MIR (Medium Intermediate Representation) level and LIR (Low Intermediate Representation) level.
- Register file extension: the Register File Set has been extended with the XRs. The register allocation algorithm allocates the XRs in a distinct pass applied before the GPR allocation; it is introduced in Machine SUIF, at LIR level. The conventions introduced for the XRs are implemented in this pass.
- Code generation: code generation for the reconfigurable hardware (as previously presented) is performed when translating SUIF to Machine SUIF IR, and affects the function calls marked in the front-end.

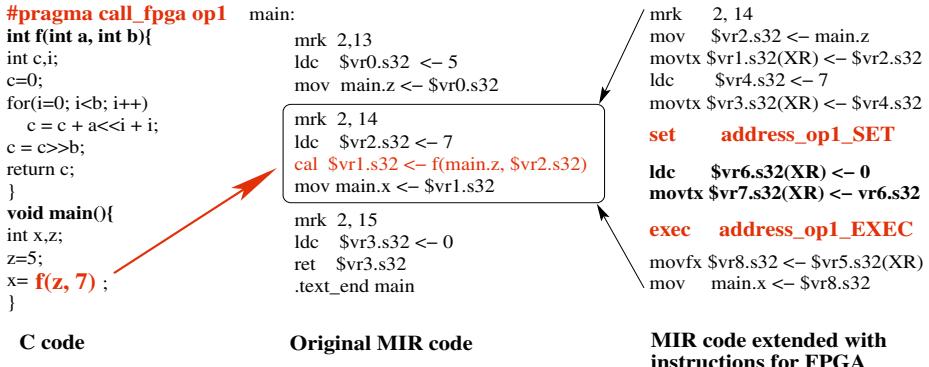


Fig. 5. Code Generation at MIR level

An example of the code generated by the extended compiler for the Molen programming paradigm is presented in Figure 5. In the first part, the C program is given. The function implemented in reconfigurable hardware is annotated with a pragma directive named *call_fpga*. It has incorporated the operation name, *op1* as specified in the description file. In the central part of the picture, the code generated by the original compiler for the C program is depicted. The pragma annotation is ignored and a normal function call is included. The last part of the picture presents the code generated by the compiler extended for the Molen programming paradigm; the function call is replaced with the appropriate instructions for sending parameters to the reconfigurable hardware in XRs, hardware reconfiguration, preparing the fix XR for the microcode of the EXECUTE instruction, execution of the operation and the transfer of the result back to the GPP. The presented code is at MIR level and the register allocation pass has not been applied.

The compiler extracts from a description file the information about the target architecture such as microcode address of SET and EXECUTE instructions for each operation implemented in the reconfigurable hardware, the number of XRs, the fix XR associated with each operation, etc.

4 A Case Study

In order to evaluate the performance improvements provided by the Molen processor, we used two well-known multimedia benchmarks, namely *mpeg2enc* and *ijpeg* for which we perform a pure software analysis. We made the following assumptions:

- the parts of the applications which can be implemented in the reconfigurable hardware are isolated in functions. This constitutes the base model for the comparison between the GPP and the Molen processor;
- the input data are:

- for mpeg2enc: the frames included in the benchmark
- for ijpeg: specmun, 1024 * 688

The parts of the applications that are candidates for the reconfigurable hardware implementation are the well-known time-consuming multimedia operations[7]: SAD (sum of absolute-difference), DCT (2 dimensional discrete cosine transform), IDCT (inverse DCT) and VLC (variable length coding). In order to study the performance improvements, we use the *Halt* library[10] available in Machine SUIF and which we modified to suit our purpose. This library is an instrumentation package that allows the compiler to change the code of the program being compiled in order to collect information about the program own behavior (at run-time).

For the above considered applications, the following is measured for their pure software implementation on the GPP (x86):

- The exact types and numbers of instructions - generated by the compiler- which are executed in the whole application and in each chosen function for hardware implementation plus their exact number of calls
- The number of cycles for the whole application and for each function chosen for hardware implementation

Based on these data, the following information can be computed for the Molen reconfigurable processor:

1. The code reduction as a result of implementation of parts of the application in reconfigurable hardware
2. An approximation of the maximum performance improvement of processor cycles for the whole application and for a particular implementation of one operation

However, because we lack a real implementation of the Molen processor, we cannot yet provide the second set of data for a particular implementation. We therefore restrict ourselves to indicating what functions are most likely to yield the highest performance improvement.

We introduced an additional pass in order to instrument the basic blocks of a program with the number and type of the included instructions. We also developed two sets of run-time analysis routines. The first set of routines is used to collect the type and number of instructions executed in the whole application and each specific function; it uses the instrumentation pass previously mentioned in this section. The second set of run-time analysis routines provides the number of cycles spent in the whole program or in a specific function. The measurements for the processor clock cycles have been performed on a Pentium II at 300MHz and we used the Pentium benchmarking instruction RDTSC - Read Time Stamp Counter - which returns the number of processor clock cycles since the CPU was reset. In this manner, the finest granularity is achieved (the code instrumentation does not affect the results).

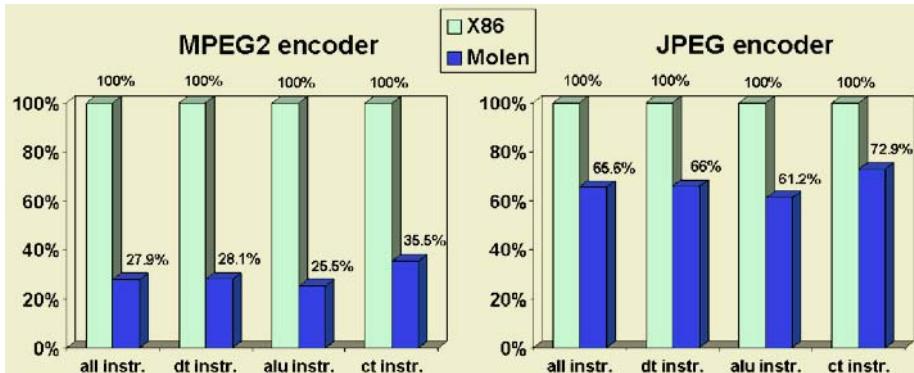


Fig. 6. mpeg2enc and ijpeg encoder instruction results

For example, we compute the total number of instructions executed in the Molen approach, when the above described functions have been implemented in hardware as follows:

$$n_{all(MOLEN)} = n_{all(GPP)} - \sum_{i=1}^N n_{f_i(GPP)} + \sum_{i=1}^N n_i(N_{call_i(MOLEN)} - N_{call_i(GPP)})$$

where n_{all} is the number of all instructions executed by the application, N is the number of functions implemented on the reconfigurable hardware, n_{f_i} represents the total number of instructions executed in the function f_i for all its calls, n_i is the number of calls of function f_i and N_{call} is the fixed number of instructions used for passing parameters, function call .

The measured data for the GPP alone and the computed data for the Molen processor are compared for mpeg2enc and ijpeg in Figure 6. The most important categories of instructions have been considered, namely data transfer (dt) instructions, arithmetic and logical (alu) instructions and control transfer(ct) instructions. From these pictures, a substantial reduction of the number of instructions is achieved by the Molen reconfigurable processor compared to the GPP: 72.1% for mpeg2enc and 34.4 % for ijpeg encoder. Also it is obvious that in both cases the alu instructions are the most reduced category of instructions, while the ct instructions are the least reduced instructions. This conclusion is confirmed by the inspection of the function code since it contains a large number of arithmetical computation and only a small number of branches. In table 1, the cycle measurements are reported. From these results, we can identify those functions that potentially give the highest performance improvement, given an efficient hardware implementation. The numbers suggest that the SAD function is the most promising candidate for hardware implementation, while the rest of the functions can provide at best a moderate performance improvement.

Table 1. mpeg2enc (left) and ijpeg encoder (right) cycle result

Fct	Cycles	% Total
SAD	149.947.461	55.2 %
DCT	42.529.647	15.7 %
VLC	3.946.954	1.4 %
IDCT	3.693.986	1.36 %
mpeg2enc Application	271.616.655	100 %

Fct	Cycles	% Total
DCT	40.206.773	12.5 %
VLC	36.571.622	10.5 %
ijpeg enc Application	341.316.466	100 %

5 Conclusions

In this paper, we presented the Molen Set-Execute paradigm that addresses a number of previously unresolved issues such as parameter passing and parallel execution of operations into the reconfigurable hardware. The paradigm involves the instruction set extension and requires on behalf of the FPGA developers only the address where the configuration(SET) and execution(EXECUTE) code is stored. A particular architectural implementation was presented, where the microcoded emulation of the SET and EXECUTE instructions are included.

The compiler extensions allow to generate code where the functions mapped on the reconfigurable hardware are automatically (rather than manually) substituted by the appropriate SET-EXECUTE instructions. It has been shown through experimentation that the compiler can be used as an important tool to support the design process focusing on the identification of good candidates for the reconfigurable hardware implementation. The presented results show a substantial reduction of the executed number of instructions and potential reduction of processor cycles for two multimedia benchmarks for their execution on the Molen reconfigurable processor compared to their pure software implementation on the GPP.

References

1. M. Sima, S. Vassiliadis, S. Cotofana, J. van Eijndhoven, and K. Vissers, “Field-Programmable Custom Computing Machines – A Taxonomy,” in *12th International Conference on Field Programmable Logic and Applications (FPL)*, Montpellier, France, Sep 2002, pp. 79–88.
2. M. Gokhale and J. Stone, “Napa C: Compiling for a Hybrid RISC/FPGA Architecture,” in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, California, April 1998, pp. 126–137.
3. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, “The Chimaera Reconfigurable Functional Unit,” in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, Napa, California, 1997, pp. 87–96.
4. A. L. Rosa, L. Lavagno, and C. Passerone, “Hardware/Software Design Space Exploration for a Reconfigurable Processor,” in *Proc. of the DATE 2003*, 2003, pp. 570–575.

5. F. Campi, R. Canegallo, and R. Guerrieri, "IP-Reusable 32-Bit VLIW Risc Core," in *Proc. of the 27th European Solid-State Circuits Conference*, Villah, Austria, Sep 2001, pp. 456–459.
6. Z. Ye, N. Shenoy, and P. Banerjee, "A C Compiler for a Processor with a Reconfigurable Functional Unit," in *ACM/SIGDA Symposium on FPGAs*, Monterey, California, USA, 2000, pp. 95–100.
7. S. Vassiliadis, S. Wong, and S. Cotofana, "The MOLEN $\rho\mu$ -Coded Processor," in *11th International Conference on Field Programmable Logic and Applications (FPL)*, Springer-Verlag LNCS, vol. 2147, Belfast, UK, Aug 2001, pp. 275–285.
8. <http://suif.stanford.edu/suif/suif2>.
9. <http://www.eecs.hardward.edu/hube/research/machsui.html>.
10. M. Mercaldi, M. D. Smith, and G. Holloway, "The Halt Library," in *The Machine-SUIF Documentation Set*, Hardvard University, 2002.

Laura: Leiden Architecture Research and Exploration Tool

Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis, and Ed Deprettere

Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
`{claus,stefanov,kienhuis,edd}@liacs.nl`

Abstract. At Leiden Embedded Research Center (LERC), we are building a tool chain called *Compaan/Laura* that allows us to map fast and efficiently applications written in Matlab onto reconfigurable platforms. In this chain, first the Matlab code is converted automatically to executable Kahn Process Network (KPN) specification. Then a tool called *Laura* accepts this specification and transforms the specification into design implementations described as synthesizable VHDL. In this paper, we present our methodology implemented in the *Laura* tool, to automatically convert KPNs to synthesizable VHDL code targeted for mapping onto FPGA-based platforms. With the help of *Laura*, a designer is able to either fast prototype signal processing and multimedia applications directly in hardware or to extract very fast valuable low-level quantitative implementation data such as performance in terms of clock cycles, time delays and silicon area.

1 Introduction

The potential of achieving high-performance implementations onto FPGA-based systems (platforms) has been demonstrated by the FPGA research community for applications in the domain of signal processing, multimedia, and imaging. These performance improvements depend very much on the expertise of the hardware designer, who has to possess an accurate knowledge of the underlying FPGA platform and the application. Moreover, the mapping of applications onto this type of platforms is in most cases done manually, which leads to a slow, difficult, and error prone design process. Therefore, we have developed a methodology that allows fast and efficient mapping of a class of multimedia and signal processing applications onto FPGA-based platforms. Part of this methodology is captured in the *Laura* tool that we present in this paper. Central to our methodology is the use of the Kahn Process Network (KPN) [3] model of computation to specify applications. The *Laura* tool accepts applications written in this KPN model and produces synthesizable VHDL code that implements the application for a specific FPGA platform.

Our methodology uses the KPN model of computation as it is a convenient model to specify imaging applications like Stereo Vision, multimedia applications like MJPEG, and classical signal processing applications like Digital Beam-forming. The model reveals the inherent parallelism of an application that is exploited when mapping the application onto FPGA platforms that are inherently fine-grained parallel platforms.

The KPN specification represents an application in terms of distributed control and distributed memory, which in our case is derived from a sequential code written in Matlab using a tool called *Compaan*. The distributed control and distributed memory are key to obtain efficient implementations on FPGAs for stream oriented applications. This is in great contrast to the original Matlab code that is using a single thread of control and shared memory. Other work describing the mapping of Matlab code (or C for that matter) onto FPGA uses other computational models like CDFG [2] or CSP [5]. These models are well suited for control dominated applications, but less for stream oriented applications.

We present our methodology to map an application written in Matlab onto an FPGA platform in Section 2. In Section 3, we look in more detail at the *Laura* tool that we have developed. In Section 4, we explain in more detail, using a running example, how *Laura* constructs an architecture in VHDL. In Section 5, we present experiments that have been obtained by using *Laura* for three applications. We conclude this paper in Section 6.

2 Integrating *Laura* in an FPGA-Based Design Flow

The *Laura* tool takes as an input a KPN specification of a given application and generates synthesizable VHDL code that targets a specific FPGA platform. In general, specifying an application as a KPN is a difficult task. Therefore, we use our compiler called *Compaan* [4] that fully automates the transformation of Matlab code into Kahn Process Networks (KPNs). The applications *Compaan* can handle, have to be specified as parameterized static nested loop programs, which is a subset of the Matlab language. We have designed the *Laura* tool to operate as a back-end of the *Compaan* compiler, realizing a fully automated design flow that maps sequential algorithms written in Matlab onto reconfigurable platforms. This design flow is shown in Figure 1.

In the first part of the design flow, an application specification is given in Matlab. This is because *Compaan* only accepts Matlab code. Nevertheless, the design flow is equally applicable to C code or Java code, as their model of computation is equal to the imperative model of computation of Matlab. The *Compaan* compiler itself is composed of a number of tools. One tool in *Compaan* performs an aggressive array-dataflow analysis by exploring all data-dependencies in the original program. The result of this tool is a data structure representing the dependence graph of the program. Another tool in *Compaan* converts this data structure into a KPN specification.

In the second part of the design flow, *Laura* transforms a KPN specification together with predefined IP cores into synthesizable VHDL code. The IP cores are needed as they implement the functionality of the functions used in the original Matlab program. They are provided to *Laura* by the *IP cores* box in Figure 1.

In the third part of the design flow, the generated VHDL code is processed by *Commercial Tools* to obtain quantitative results. These results can be interpreted by designers, leading to new design decisions. These decisions are reflected by writing a new Matlab program that exposes, for example, more or less parallelism. For that purpose, we have developed a tool called *MatTransform* that manipulates the Matlab input specification in a Source-to-Source fashion to generate more instances of the application, in which each instance exposes a different level of concurrency without

altering the algorithm's behavior [8]. The concurrency is altered by performing high-level transformations like loop unrolling (unfolding), retiming (skewing), and code merging. By rewriting Matlab code, we can explore different mappings of a Matlab algorithm in an efficient way. When an obtained algorithm instance meets the requirements of the designer, the corresponding VHDL output is synthesized by a commercial tool and mapped onto an FPGA platform.

3 The Laura Tool

The KPN model of computation [3] assumes concurrent autonomous processes that communicate in a point to point fashion over unbounded FIFO channels, using a *blocking-read* synchronization primitive. Each process in the network is specified as a sequential program that executes an internal function. At each execution (also referred to as an iteration) this function reads/writes data from/to different FIFO channels. Because of the unboundedness of the FIFO channels, the KPN cannot be translated directly into a VHDL representation and mapped onto a hardware platform. Instead, a *blocking-write* primitive is needed next to the *blocking-read*. Also, the FIFO channel sizes now need to be fixed such that no deadlock occurs. Using the method presented in [6], we find a bound on the size of the FIFOs such that the network will not deadlock.

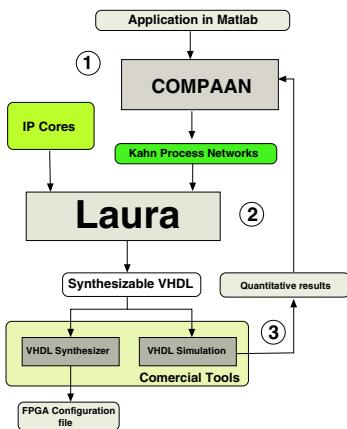
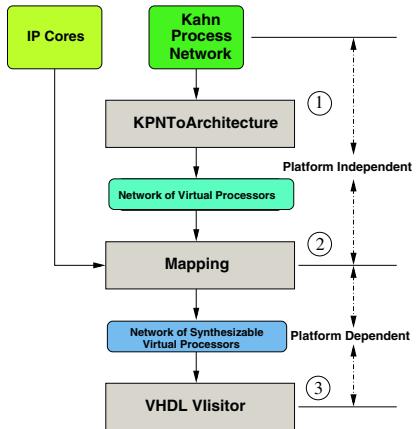
To convert a KPN specification into hardware, we have implemented in Laura a strategy that divides the conversion process into two parts: a platform independent part and a platform dependent part. In the platform independent part, we define an abstract model of the architecture on which we map a KPN application. The model of architecture defines the key components of the architecture and their attributes. It also defines the semantic model, i.e., how the various components interact with each other. Hence, the architecture also implements autonomous processes, that communicate over channels using blocking read and blocking write semantics.

The abstract architecture model is captured in Laura in terms of a class-hierarchy. This class hierarchy describes a network of virtual processors. Each of them is composed of four units: a *Read unit*, a *Write unit*, an *Execute unit* and a *Controller unit*. The first three units are synchronized by the Controller unit of the processor. Each FIFO channel in the KPN specification is represented by a *Hardware Channel* unit.

In the platform dependent part we start to add information to the abstract architecture model that is specific for the target platform. At this stage, we include IP cores in the Execute units that implement the functions of the original application. Also, we set attributes of the components like bit-width and size of the Hardware Channels.

When an architecture model is established for a given KPN specification, we convert the architecture model into VHDL code using a Visitor Design Structure. For each component in the abstract architecture, we have a small piece of VHDL code that expresses how to represent that component on the target architecture. The visitor structure gives Laura a lot of flexibility. If needed, the output can easily be converted to other formats like Verilog or SystemC.

The steps that make up the Laura tool are shown in Figure 2. In the first step, the *KPN-ToArchitecture* method converts the given KPN specification into an equivalent network of virtual processors (*Network of Virtual Processors*). This is a platform independent

**Fig. 1.** The Compaan/Laura tool chain**Fig. 2.** Steps in Laura

step as no information on the target platform is taken into account. In the second step, platform specific information is mapped onto the abstract architecture model leading to a network of Synthesizable Processors (*Network of Synthesizable Processors*). In the third step, the architecture model is visited by a VHDL visitor to generate the VHDL code.

4 Laura in Action

To make clear how a Matlab program is converted into a VHDL code, we explain the steps done in Laura using the very simple Matlab program given in Figure 4. This program consists of three loops. In the first loop, variable $a(j)$ is initialized using function `Init`, which represents a *Source*. In the second loop, the function `Compute` performs an operation on $a(j-1)$, introducing a self-loop. Finally, the last loop takes the result of $a(6)$ using function `Pass`, representing a *Sink*. The Matlab program is given to the Compaan compiler that converts it into a KPN representation consisting of three different processes. A graphical representation of this KPN is given in the top-part of Figure 3. One process (P1) is the Source, one process implements the `Compute` function (P2), and one process is the Sink (P3). The picture clearly shows the self-loop of function `Compute`. As said before, each process contains a sequential program. In Figure 5, the sequential program for process P2 is given in C++ using the YAPI [1] format.

The sequential program produced by Compaan always follows a particular sequence of events. These events are highlighted by the three different boxes in Figure 5. The first box, contains the code that reads data from input ports. The actual computation takes place in the second box (i.e., performing the function `Compute` from the Matlab program of Figure 4). In the third box, we show the code that writes out data produced by the computation. The three boxes are enclosed by a for-loop, indicating that the sequence of events needs to be repeated for a given number of times. As a consequence, this

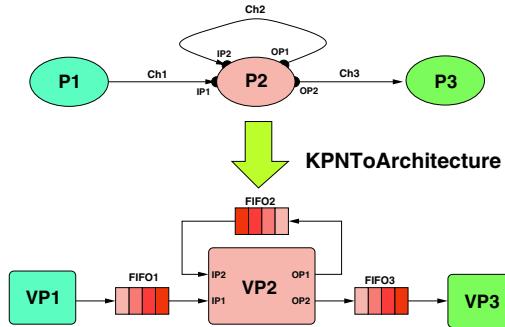


Fig. 3. An example of *KPNToArchitecture* step in Laura

process operates in a stream based fashion, an operation model which is very applicable to multi-media and digital signal processing applications.

4.1 KPNToArchitecture

The KPN shown in the upper part of Figure 3 is mapped by the KPNToArchitecture step in Laura onto an abstract architecture model. This model is composed of *Virtual Processors* and bounded hardware communication channels. The lower part of the Figure 3 represents the network of virtual processors that has the same topology as the input KPN. This is because Laura currently performs a *one-to-one* mapping. The three processes P_1 , P_2 , and P_3 are mapped onto the virtual processors VP_1 , VP_2 , and VP_3 , respectively. The KPN unbounded FIFO channels Ch_1 , Ch_2 , and Ch_3 are mapped onto the bounded hardware FIFOs $FIFO_1$, $FIFO_2$, and $FIFO_3$, respectively.

Every virtual processor is composed of four units: a *Read* unit, a *Write* unit, an *Execute* unit, and a *Controller unit*, as shown in Figure 6. The Execute unit is the computational part of a virtual processor. It has *Input Arguments* that provide to the unit the necessary data for execution and *Output Arguments* that are the result of the computation process. In our model, the Execute unit fires when all the input arguments have data and always produces data to all the output arguments. The Read unit is responsible for assigning all the input arguments of the Execute unit with valid data. Since there are more input ports than arguments, the Read unit has to select from which port to read data. This information is stored in the *Control Table* of the Read unit. The *Input Port* is the input interface that connects the virtual processor with a communication channel. The *Output Port* is the output interface that connects the virtual processor with a communication channel. The Write unit is responsible for distributing the results of the Execute unit to the relevant processors in the network. A write operation can be executed only when all the output arguments of the execute unit are available for the write unit. A *Control Table* is used to select the proper Output Port according to the current iteration of the virtual processor.

The virtual processor's *Controller* synchronizes all the processor's units and keeps track of how many times the processor has already fired. The Read unit and the Write unit can block the next firing when a blocking-read or a blocking-write situation occurs,

```

1 void P2 ::main() {
2   for (int i = 2 ; i <= 6 ; i += 1 ) {
3     if (i-2 == 0) {                               READ
4       /*reads a token from a channel
5       in_0 = read(IP1);
6     }
7     if (i-3 >= 0) {
8       /*reads a token from a channel
9       in_0 = read(IP2);
10    }
11   out_0 = Compute(in_0) ;      EXECUTE
12
13   if (-i+5 >= 0) {                  WRITE
14     /*writes a token to a channel
15     write(OP1, out_0);
16   }
17   if (i-6 == 0) {
18     /*writes a token to a channel
19     write(OP2, out_0);
20   } // for i
21 }
```

Fig. 4. A very simple Matlab Program

thereby stalling the complete processor. A blocking-read situation occurs when data is not available at a given input port. A blocking-write situation occurs when data cannot be written to a particular output port.

Let us consider the P2 process as it is specified by the sequential code given in Figure 5. This code is analyzed in the KPNToArchitecture step to instantiate the corresponding components of the virtual processor. The Read unit is generated based on the information contained between lines 3 and 10. Two input ports, **IP1** and **IP2**, are required to read the input argument **in_0** of the Execute unit. Because a 2-to-1 relationship exists between the input ports and the input argument, a Control Table is needed to select the proper input port for reading the input argument at a particular iteration of the processor. For the example, the Control Table $c = [1, 0, 0, 0, 0]$ is derived based on the number of firings (line 2) and the **if** statements from lines 3 and 7. The Write unit is instantiated according to the lines 12 to 19. It requires two output ports **OP1** and **OP2** to write the output argument **out_0** of the Execute unit. Again a Control Table is derived based on the number of firings (line 2) and the **if** statements from lines 12 and 16. The Control table is equal to $d = [0, 0, 0, 0, 1]$. The Control unit of the processor is instantiated as a counter that iterates i from 2 to 6. For the Execute unit an interface is defined, based on the information contained in line 11. This interface is used again in the Mapping step (Figure 2) when an IP core is connected to the Execute unit. The complete virtual processor that corresponds to process P2, is shown in Figure 7.

Fig. 5. Process P2

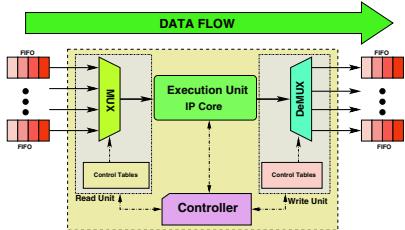


Fig. 6. The Virtual Processor model

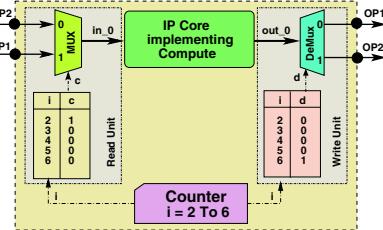


Fig. 7. VP2 in the example shown in Fig. 3

4.2 Mapping

The Mapping step is used to include additional information to the abstract architecture model. This is information about the IP cores used by the virtual processors and the bit-width of data. At this step, the width and size of a hardware channel is provided. Furthermore, the notion of a clock event is taken into consideration.

We use IP cores in designing new hardware applications to reduce the design time. This means that we add in the Mapping step the functionality of the Execute unit in terms of an IP core. In order to select the appropriate IP core, the Mapping step searches through a library of predefined cores until it matches the required functionality. The found IP core is subsequently associated to the Execute unit of the virtual processor. For the IP cores which are pipelined, additional information needs to be provided to the Control unit to accommodate the control for the pipelining.

The final result of the mapping step is an annotated architectural model called *Network of Synthesizable Virtual Processors* (NSVP) that is targeted to a particular FPGA platform.

4.3 Visitor

The last step of Laura generates the correct VHDL code for the NSVP structure. First, the communication network is generated, followed by the various processors, and finally a test bench. Within the Visitor there is a well-defined relationship between the components of the abstract architecture model and its representation in VHDL. This means that we have a VHDL template for each component. For example, there is a template for the various units in a processor as well as for the processor itself. The relationship is often one-to-one, but for example in the case of the hardware communications channels, a one-to-many relationship exists. A hardware communication channel operates as a data buffer that can be realized using flip-flops, a look-up table, or internal BRAM memory. This gives the Visitor a lot of flexibility to derive alternative VHDL code taking advantage of specific elements of the target platform.

5 The Experiments

Our experimental results are obtained by evaluating the synthesizable VHDL code generated by Compaan/Laura for three computational intensive algorithms. The first one

is the *Sequence Alignment* algorithm [7] from the field of bio-informatics. Using the unfolding transformation provided by the MatTransform [8] tool box of Compaan, we generate three different networks. The application specific processor uses an IP core called *Match* that is composed of two adders and a comparator. The second algorithm is the implementation of the 2D-DCT function that is used in data compression algorithms such as MJPEG. In this case, we used the freely available 2D-DCT IP core from the Xilinx web site. The third one implements the QR factorization algorithm used in signal processing applications. It has two IP cores, *Vectorize* and *Rotate*, provided by QinetiQ, Ltd [10]. Table 1 shows the complexity of the input KPNs given by the number of processors and communication channels that has to be handled by the Laura tool. The complexity of the IP cores used to implement the application specific processor is given by the number of hardware multipliers and the pipeline depth used to implement the core.

Table 1. The Process Network complexity

Experiment	No. of Processors	No. of Channels	Pipeline Stages	Multipliers
Sequence Alignment	7	13	0	0
Seq. Alignment Unfold 2x2	10	40	0	0
Seq. Alignment Unfold 3x3	15	83	0	0
2D-DCT	4	4	92	6
QR(Rotate, Vectorize)	5	18	55, 42	8, 8

For each benchmark algorithm, a description of the algorithm in Matlab was written and passed through Compaan and Laura. We verified the hardware in two ways. The first way is by simulating the generated hardware using a VHDL simulator and comparing the results to the output of the algorithm executed in the Matlab interpreter. The second way is by implementing the generated hardware onto our reconfigurable platform and comparing the results to the Matlab output. The VHDL simulator provided the total number of cycles needed to execute a given algorithm, as shown in the **Cycles** column of Table 2. We use the XST synthesizer and the Xilinx Foundation 5.1i tool to synthesize, place, and route the output of Laura. The clock delay and the total amount of slices needed to implement the networks onto a Virtex II-6000 are also provided in Table 2.

Table 2. Experimental Results

Experiment	Cycles	Clock delay (ns)	Used Slices	Used Area Virtex II-6000
Sequence Alignment	865	16.030	1321	3%
Seq. Alignment Unfold 2x2	466	15.751	3127	9%
Seq. Alignment Unfold 3x3	293	18.511	5874	17%
2D-DCT	364	19.733	1610	4 %
QR(N=7,T=21)	19181	24.390	11270	33 %

To study the overhead introduced by our methodology in terms of cycle delays and area (i.e., used slices), we conducted a second experiment. In this experiment, we compare a single IP core with the same core embedded in a network. For a single IP core we determine its clock speed and area and compare this to the speed and area taken by the same IP core used in an application network. This gives an indication about the overhead introduced by our methodology. Table 3 shows the delays and the area used by the IP cores, the influence of communication on clock delay (**Delay Overhead**), and the used area (**Area Overhead**). We notice that for fine-grained core implementations the area needed to communicate data in a distributed way is dominant. For example, in case of Sequence Alignment, 20 times more area is needed than a stand-alone version of the *Match* IP core. The communication takes more than 2 times longer in terms of clock-delay than the stand-alone version, due to the routing of the hardware channels on the FPGA. The network of embedded coarse-grained cores, i.e., 2D-DCT, Vectorize and Rotate, introduce considerable less clock-delay than the network of embedded fine-grained cores, i.e., Match. The area overhead depends mainly on the network complexity in terms of channels used. See the difference in number of channels between 2D-DCT and QR in Table 1.

Table 3. Trade off between Computation and Communication

Experiment	Working Processor	Clock Delay	Slices	Delay Overhead	Area Overhead
Sequence Alignment	Match	6.156	66	2×	20×
Seq. Alig. Unfold 2x2	4×Match	6.156	264	2×	11.8×
Seq. Alig. Unfold 3x3	9×Match	6.156	594	3×	10×
2D-DCT	2D-DCT	13.656	1365	1.4×	1.17×
QR	Vectorize, Rotate	15.862	3442	1.5×	3.27×

6 Conclusions and Limitations

In this paper, we have presented the Laura tool that implements our methodology to map KPNs generated by the Compaan tool onto a reconfigurable platform such as FPGAs. Although the tool generates only VHDL code, it can be reconfigured to generate other kinds of output, such as Verilog or SystemC. A number of experiments have been conducted for applications in the field of bio-informatics, image processing, and signal processing. The experiments show that we are able to derive fully automatically a hardware implementation from Matlab code. Because Laura implements Kahn Process Networks into hardware, it is well suited for stream oriented applications. Laura is not suited to map control dominated applications. To study the impact of the KPN model on the hardware realization, we investigated the trade off between a stand-alone IP core and an integrated IP core. We found that for more coarse-grained IP cores, the presented methodology gives the best results.

A number of limitation can still be found in Laura. The first issue is that Laura can handle only FIFO communication between processors. High-level code transformations,

such as unfolding and skewing, can introduce out-of-order communication between processors [9]. In such case a FIFO can no longer be used in the communication between processes. Future work includes extending the communication components to include this out-of-order communication. The second issue is that Laura generates hardware implementations for non-parameterized KPN models, while Compaan is capable of deriving parameterized descriptions. Future work will focus on generating parameterized hardware networks. The third issue is that communication channels are not always used at their full capacity. We would like to collapse some of these channels onto one channel to share its hardware to reduce communication requirements.

Acknowledgments

We would like to acknowledge Alexandru Turjan of the LERC group, Leiden University, for his very valuable input and sharing his insights on the Laura work and his substantial effort to integrate the Laura work with Compaan. Also, we would like to thank to Steven Derrien for his insides toward the processor synchronization issues.

References

1. E. de Kock, G. Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proc. 37th Design Automation Conference (DAC'2000)*, pages 402–405, Los Angeles, CA, June 5-9 2000.
2. M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. A system for synthesizing optimized fpga hardware from matlab. In *Proc. Int. Conf. on Computer Aided Design*, San Jose, CA, Nov. 2001.
3. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.
4. B. Kienhuis, E. Rypkema, and E. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign (CODES)*, San Diego, USA, May 2000.
5. I. Page. Constructing hardware-software systems from a single description. In *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.
6. T. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.
7. T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
8. T. Stefanov, B. Kienhuis, and E. Deprettere. Algorithmic transformation techniques for efficient exploration of alternative application instances. In *10th Int. Symposium on Hardware/Software Codesign (CODES'02)*, pp. 7-12, Estes Park, Colorado, USA, May 6-8, 2002.
9. A. Turjan, B. Kienhuis, and E. Deprettere. Realizations of the extended linearization model in the compaan tool chain. In *proceedings of the 2nd Samos workshop*, Samos, Greece, Aug. 2002.
10. R. Walke, R. Smith, and G. Lightbody. 20Gflops QR processor on a Xilinx Virtex-E FPGA. In *Proc. SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations X*, pages 300–310, 2000.

Communication Costs Driven Design Space Exploration for Reconfigurable Architectures

Lilian Bossuet, Guy Gogniat, and Jean-Luc Philippe

LESTER Lab
University of South Brittany,
Lorient, France

{lilian.bossuet, guy.gogniat, jean-luc.philippe}@univ-ubs.fr

Abstract. In this paper we propose a design space exploration method targeting reconfigurable architectures that takes place at the algorithmic level and aims to rapidly highlight architectures that present good performance vs. flexibility tradeoffs. The exploration flow is based on a functional model to describe the architectures that the designer wants to compare. The paper mainly focuses on the projection step of our flow and presents an allocation heuristic that is based on communication costs reduction.

1 Introduction

The new telecommunication and multimedia applications need to have a reduction of actual systems on chip (SoC) power consumption and an increase of SoC flexibility. Although the evolutions of integration and design are more and more important, they are not sufficient to face these challenges. It is necessary to provide novel approaches that work at the system level to design more efficiently, and to target new technologies.

Reconfigurable architectures are becoming more attractive in terms of capacity, performances, low-power consumption and flexibility (through the possibilities of run-time reconfiguration and multi-granularity resources) [1][2]. They correspond to an efficient solution to the SoC challenge and will be unavoidable in a near future. But the design space of reconfigurable architectures is very large, because these architectures can be extremely heterogeneous in term of processing, memory and routing resources. Hence, it is very complex to find the best reconfigurable architecture for a panel of applications where each application can be dynamically configured on the architecture.

In order to help the designer it is necessary to develop tools that compare several architectures for different applications. We propose in this paper an original method of design space exploration for reconfigurable architectures that works at the algorithmic level.

The paper is organized as follows. Section 2 describes related work dealing with design space exploration methodologies. Section 3 describes major issues in design space exploration at the algorithmic level and section 4 presents our approach. Section 5 details the projection step. Section 6 gives some results for different reconfigurable architectures. Finally, section 7 concludes the paper and exposes future direction.

2 Related Work

Many research teams are focusing on reconfigurable architecture [1]. Some are working on design space exploration methodology in order to find the best architecture for a panel of applications.

Two ways are possible to explore reconfigurable architectures. The first one is to synthesize all the applications for the different target architectures, and to compare the overall performance results. In that case the results are very accurate, but it is necessary to have a specific synthesis tool for each architecture (which is not always available in the case of architecture exploration) or to use generic synthesis tools [3][4]. However, synthesis steps use very complex algorithms, which conducts to a limited and slow exploration. Furthermore it is necessary to have a very good knowledge of the target architectures when using generic synthesis tools since it is necessary to provide them a model of the target architectures. Hence, this method is not really adapted for a large and rapid architecture exploration and is more dedicated to do some architecture refinement steps.

The second way is to perform estimations. In that case it is necessary to consider a generic architecture model to describe the different architectures to target. The objective is to make relative performance estimations (speed, power consumption and area) in order to compare very quickly different architectures. Although the estimations do not give necessarily real and accurate performance results, it is enough to compare architectures since the important point in that case is that estimations are faithful and an absolute error is not the major concern.

Both exploration methods require having an architecture model. It is possible to consider a physical model. Then it is necessary to know precisely the physical parameters of the architecture (technology, routing type and size, routing switch resources, clusters size, etc). Versatile Place and Route (VPR) tool, developed at the Toronto University, is a very interesting approach that works on a physical model [3]. VPR is a synthesis tool that works at the logic level and is oriented for island style fine-grained architectures (as FPGA). It is not suitable for coarse-grained architectures. It is also possible to model architecture with a functional model. Each element of the architecture is described by the functions it can execute. The functional model enables to describe a large panel of architectures and the description are technological independent. This model is used in the generic place and route tool for fine-grained reconfigurable architectures called Madeot-Bet [4] that works at the logic level.

VPR and Madeot-Bet are not the only tools that use an architecture model, but there are very representative. Both are FPGAs oriented, but other approaches target reconfigurable architectures. In [5] the design space exploration flow targets mesh architecture called KressArray - a fast reconfigurable ALU. The exploration tool, Xplorer works at the algorithmic level and aims to assist the designer in finding a suitable architecture for a given set of applications. This tool is architecture-dependent, but the use of fuzzy logic to analyze the results of the exploration is a very attractive approach.

[6] presents the design space exploration for the Raw Microprocessor as an example of a tiled architecture. The Raw Microprocessor is reminiscent of coarse-grained FPGA and comprises a replicated set of tiles coupled together by a set of

compiler orchestrated, pipelined, switches. Each tile contains a RISC-like processing core and SRAM memory for instructions and data.

3 Design Space Exploration Flow Principles

Design space exploration can be performed at different levels of abstraction in order to reduce progressively the number of solutions. More the abstraction level is refined more accurate results can be obtained since a lower number of solutions need to be considered.

At the algorithmic level the objective is to rapidly identify target architectures that present a high performance and versatility potential. To reach such a goal, design space exploration methods must promote the flexibility, the rapidity and the fidelity. Encouraging (i) the flexibility means that performance and versatility can be estimated for a wide variety of reconfigurable architectures, (ii) the rapidity enables to estimate performances without the time-consuming computation of programs such as Place & Route algorithms and (iii) the fidelity points out that the relative comparisons between two alternative architectures must be close to the relative errors that would be obtained after the synthesis steps even if there may be significant absolute errors in the performance estimation at the algorithmic level.

Another major concern is to promote the interactivity with the designer at all the abstraction levels in order to take benefit from his experience. Thus, the refinement process at a given abstraction level can be performed through several runs of the exploration method in order to converge progressively to an efficient mapping between the application and the architecture. Between several runs, the designer can improve the architecture model according to the previous results of the exploration. Once, the designer has selected some efficient architectures he can refine his results by decreasing the abstraction level and thus using more accurate architecture model and exploration tools.

In this paper we proposed a design space exploration method targeting reconfigurable architectures that addresses the previous highlighted principles. Our method is based on an estimation approach and takes place at the very first steps of the design flow since it works at the algorithmic level. A functional model is used to describe the target architectures since such model as proven its efficiency to characterize a wide variety of reconfigurable architectures [7].

4 Proposition of a Design Space Exploration Flow

The design space exploration flow that we propose is depicted on the left of the figure 1. The specification is provided in a high level language (C language) and is first translated into an intermediate representation - the HCDFG model. This model is a Hierarchical Control and Data Flow Graph allowing efficient algorithm characterization and exploration of complex applications including control flow and multi-dimensional data.

The first part of the flow (figure 1) is the **System Estimation** step [8][9], during which the application is characterized and scheduled. The results computed are

defined as **Costs Profiles** i.e., scheduling for all the resources used by the application and for different time constraints. The available processing and memory resources to perform the scheduling are defined in a file called **User Abstract Rules**.

Relative Estimation, the second step of the flow (figure 1), aims to estimate the application performances on several reconfigurable architectures. Relative Estimation gives designer information to improve progressively the architecture definition with several runs in the design exploration flow.

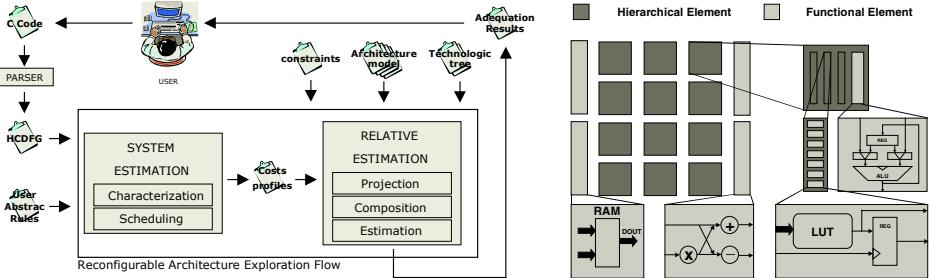


Fig. 1. The design exploration flow and an example of reconfigurable architecture that can be specified with the HF model.

4.1 Specification

The application is specified with the C language. Once it has been functionally validated, it is translated into an intermediate model, which is a Hierarchical Control and Data Flow Graph (HCDFG) [9][10]. For sake of simplicity we can say that an application is modeled with several DFG connected through control structures, hierarchy and dependence relations. An important characteristic of the HCDFG model is that processing and data are explicitly represented with nodes in the graph. This decomposition enables to emphasize the hierarchy and the potential parallelism of the application, which is an essential characteristic to perform an efficient algorithmic and architecture exploration.

4.2 System Estimation

System Estimation [8] consists in two steps; **Characterization** and **Scheduling**. During the Characterization step, the application orientation is analyzed through three axes - processing, control and memory. Specific metrics are used to find this orientation [9]. Once the application orientation has been exhibited the scheduling of the application is performed accordingly. For example, if the application is processing oriented, the processing resources are first scheduled and the memory resources are scheduled in a second step (the opposite if the application is memory oriented). The main objective of the System Estimation is to show up the intrinsic processing and memory parallelisms of the application and to give some guidance on how to build the architecture (pipeline, parallelism, memory hierarchy etc.). Further details on the system estimation step are beyond the scope of this paper. Interested reader can refer to [9].

4.3 Relative Estimation

Relative Estimation is linked with the System Estimation through the cost profiles. The cost profiles describe the scheduling results (for all the processing and memory operations) for different time constraints. These values characterize the application to be implemented.

This step is composed of three tasks: **projection**, **composition** and **estimation** that are performed sequentially. In order to evaluate an application on different architectures, it is necessary to specify the target reconfigurable architectures. For this, a functional model is used since such model is suitable for rapid relative comparison and is efficient to describe architectures at a high abstraction level. This model called **HF model** [7], enables to describe functionally the elements of the architecture and to represent different architectural styles and different architecture elements. Although, the routing resources are not explicitly described, they are taken into account in the estimation flow with connection costs in the HF model. There are two types of elements in the model: the hierarchical elements and the functional elements. The hierarchical elements are used to describe the architecture hierarchy. A hierarchical element can be composed of functional elements and other hierarchical elements. The functional elements are used to describe the architecture resources. They can be logical, input/output, memory and processing resources. An extension of the HF model exists to model tile-based architectures, where the communications between the tiles must be explicitly modeled [11]. Figure 1 (on the right) shows an example of architecture with several levels of granularity that can be modeled with the HF model.

Relative Estimation begins with the **Projection** step that makes the link between the functional needs (processing and memory) of the application and the available resources of the architecture. If a difference of resource granularity exists between the functional needs (from the application) and the available resources (in the architecture), the necessary resources can be split as fine-grained resources using a library called **Technological Trees** (the same method is used with the PipeRench reconfigurable architecture [12]).

The **Composition** step takes into account the application scheduling obtained during the System Estimation step in order to refine previous results. Since it is necessary to add resources dedicated to realize the scheduling like multiplexer, register or states machine. These additional resources are taken into account in the last step of the estimation process.

The **Estimation** step computes the global application performances on the selected architectures. The estimations take into account the static costs of the model (interconnect costs between two hierarchical elements and the costs of the functional elements), and the dynamic costs of the application (critical path, operator communications, memory reads/writes). The results of this step are gathered into a file where the application is characterized for the target architectures. Composition and estimation steps are not the topic of this article.

5 Projection Algorithms

In this section we focus on the projection step which is particularly important in the flow since it has a strong impact on the quality of the final estimated performances. The goal of this step is to allocate the resources of the architecture that will support the application operations (processing and memory). The allocation algorithms aim to assign in a same hierarchical level of the architecture the resources that communicate the most in order to reduce the cost overhead due to communications. In our model architecture hierarchy corresponds to routing topology. Communication costs are smaller inside a hierarchical element than between two hierarchical elements. More hierarchical levels are crossed by a communication higher is the communication cost.

Why focussing on communication costs for allocation algorithms? Studies on power repartition in fine-grain reconfigurable architecture like FPGA show that the major contribution to power consumption is due to routing resources (wires and switch). It is always better to reduce communication paths [13][14] since implementations are more energy efficient when wires are short and number of switches is low. The clock frequency can also be increased when communication paths are short since critical path is reduced.

In our approach we enhance the spatial locality between resources that most communicate. Different cost functions have been defined, to estimate the communication costs, which are computed in three algorithms that give respectively a lower and an upper bound and a mean value for the total communication costs.

Since our approach works at the algorithmic level it does not target a specific synthesis tool and does not consider an accurate physical architecture model. Hence instead of giving designers a single communication cost value that may present a significant absolute error due to backend synthesis algorithms and architecture refinement we propose to give them some bounds and an average value. Such approach as shown figure 2 enables designers to select architectures at the algorithmic level with the guaranty that the final performance will belong to the estimated performance interval.

Such approach also enables to give designer metrics on allocation algorithm impact. On the example (figure 2) the architecture **C** has a narrow performance interval so allocation algorithms will have a small effect on the final performance and low complexity algorithms can be considered. The architecture **B** has a large performance interval so allocation heuristics will have a strong impact on final performance and it might be important to consider better allocation algorithms.

Results of the relative estimation step provide designers resources utilization rate and estimations of communication costs. Based on these results designers can perform a first architectures performance comparison and remove architectures that present a poor synergy with the application (e.g. unadapted granularity) or too important communication costs.

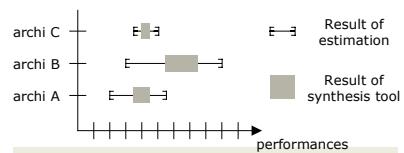


Fig. 2. Bound performance results.

5.1 Average Communications Graph (ACG)

In order to make this projection, the first step builds, from the HCDFG model, a new graph with only processing nodes, i.e. a graph without memory node and control structure. The edges between two nodes represent the communications. This graph is then reduced in order to take into account the scheduling result, the final graph is called Average Communications Graph (ACG). This graph exhibits how each type of processing resources communicates with the other types of processing resources. This graph is used during the projection step to enhance the spatial locality of communicating resources. In the ACG each edge represents the communications between two types of processing nodes.

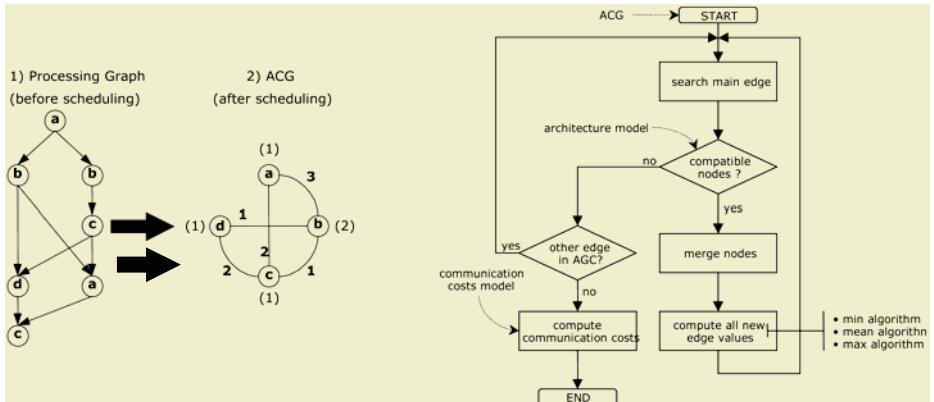


Fig. 3. Transformation of the processing graph (1) into ACG (2), and the generic projection algorithm

Figure 3 shows on a simple example how to transform a processing graph in an ACG. In the ACG, each node corresponds to a type of processing resources. In our example the type of the processing resources correspond to a letter (a, b, c or d). Several differences exist between the two graphs. The ACG is realized after the processing graph scheduling. There are fewer nodes in the ACG than in the processing graph, since the ACG has only one node for one type of processing resource. Its edges are not oriented, the communications are take into account in all directions. Several attributes are added in the ACG to better describe the communications inter-node.

The number in brackets beside a node is the number of operators that the scheduling has allocated (see section 4.3). The boldface number beside an edge is the total number of communications between two processing types. In order to know what pair of nodes communicates the most, it is necessary to compute the relative number of communications between two processing types. This value is obtained with the following expression:

$$\text{RelativeComm}_{Op1-Op2} = \frac{\text{TotalComm}_{Op1-Op2}}{\text{NumberOp1} + \text{NumberOp2}}$$

Where $\text{RelativeComm}_{Op1-Op2}$ is the relative number of communications, $\text{TotalComm}_{Op1-Op2}$ is the total number of communications, NumberOp1 and NumberOp2 are the

numbers of allocated operators of each type. It is very fast to build the ACG from the HCDFG of the application. The ACG is the input of the projection algorithms.

5.2 Generic Projection Algorithm

The projection step makes the link between the necessary (application) and the available (architecture) resources with the challenge that the most communicating resources must be assigned in a same hierarchical level of the architecture. Figure 3 shows the proposed algorithm. The algorithm begins with the ACG, and the first step searches in the graph the pair of nodes that communicates the most. This step searches the edge with the highest relative value.

When a pair of nodes is determined it is necessary to know if the two types of processing node can be implemented by functional elements in a same hierarchical element. If the two nodes are compatible, they are assigned to the hierarchical element, and they form a new node, a composite node. This composite node has as parameter the processing types of the two previous nodes. Since the ACG has a new node, it is not the same graph, so it is necessary to re-computed all this edge values and make all the necessary transformations due to the composite edge presence. To do this, we use three algorithms that give respectively a lower and an upper bound and a mean value for the total communication number. These algorithms are detailed in the paragraphs 5.3, 5.4 and 5.5. Each of them gives the total number of communications in the architecture to support the application.

If, after the search of the main edge, the pair of nodes is not compatible, the search re-starts with other nodes. If all the ACG edges have already been selected, and if any pair of nodes is compatible (it is not possible to implement it in functional elements in a same hierarchical element), then the projection algorithm stops.

To compute the communication costs a communication costs model that is based on the architecture hierarchy must be considered. A communication between two hierarchical elements does not represent the same cost than a communication in a single hierarchical element. More details are given in the section 5.6.

5.3 Min Algorithm (Lower Bound)

To illustrate the execution of the min algorithm, figure 4 shows with a very simple example the different steps of computation. The modifications of the architecture are presented on the right side of the figure. The architecture has two hierarchical levels (represented by two hierarchical elements H1 and H2). The hierarchical element H2 has three functional elements, one functional element can realize one multiplication and the two other functional elements can realize one addition or one subtraction (depends on the configuration). During the process the functional elements are progressively allocated (in grey). On the left side of the figure the modifications of the ACG during the process are detailed. At the beginning of the process, the ACG has no composite node. At each step one composite node is created by merging two nodes, and all the ACG edge values are re-compute to take into account the new composite node.

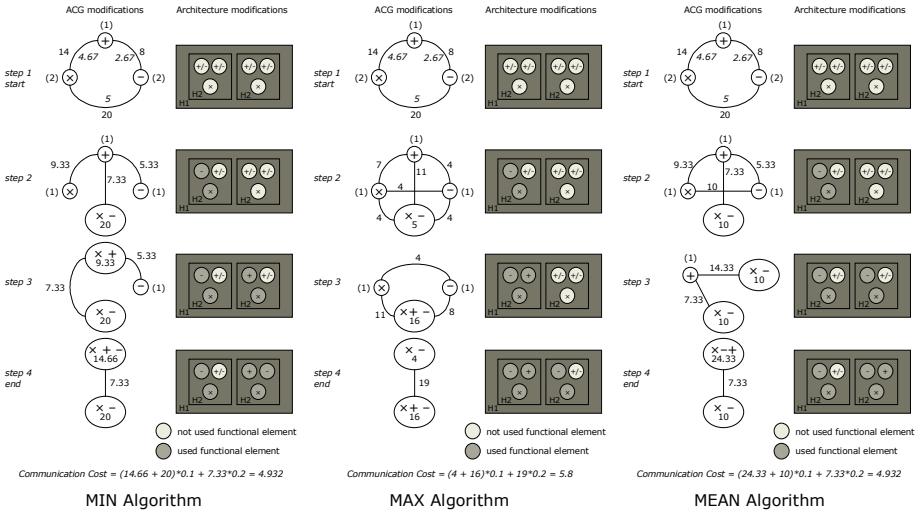


Fig. 4. The Min, Max and Mean algorithms.

The strategy of the min algorithm is to consider that if two operators of distinct processing types can be assigned in a same hierarchical element, they will perform all the communications between all the nodes of both processing types. It is the reason why in the start of the algorithm no edge is created between the new composite node (that describe a hierarchical realization) and one of the two nodes that communicate the most. For the example of the figure 4, the node multiplier and the node subtracter shape the pair of nodes that communicates the most (the relative value of the edge between this pair of node is the higher of the ACG). So in the second step of the algorithm, a composite node is created with two operations; one multiplication and one subtraction. The numbers (in brackets) of allocated operator multiplier and subtracter are decremented. The edge between the pair of nodes is deleted but this value (20) is transformed into internal communications of the new composite node.

However if another node has some communications (i.e. an edge) with one of the two previous nodes, then this node has an edge with the new composite node, and it preserves its other edges but with new values. It is the case of the node adder in the example of the figure 4.

The process stops when it is not possible to merge nodes. When the process ends, the communication cost is computed. In this example, the cost to communicate between two hierarchical elements H2 (in the H1 element) is 0.2, and the cost to communicate in a H2 element is 0.1. With this algorithm most communications are executed in composite nodes. As the communication costs in composite nodes are low (because there is not communication across several hierarchical levels) the total cost of communication will represent a lower bound.

5.4 Max Algorithm (Upper Bound)

As in the case of the min algorithm, figure 4 shows with the same example the different steps of computation. The strategy of the max algorithm is to consider that

all operators communicate uniformly. When a composite node is created with a pair of nodes (that communicates the most in the ACG) two edges are created between the new composite node and the pair of nodes. The communications are uniformly distributed between the two edges value and inside the new composite node. With this algorithm the communications are uniformly executed between the different hierarchical elements, which corresponds to an upper bound. In that case the allocation algorithm do not take benefit from the architecture hierarchy, hence, the communication costs correspond to maximum costs.

5.5 Mean Algorithm (Mean Value)

Figure 4 shows for the same example the different steps of computation. The idea of the mean algorithm is to consider that two operators in a same hierarchical element must communicate more than two operators in two different hierarchical elements. The number of communication in a composite node, is the maximum communication value between two nodes. That is the reason why the MIN function is used to determine the internal communications in the new composite node.

We can see on the figure 4 that contrary to the min and max algorithms, with the mean algorithms, the two first created composite nodes are with one multiplier and one subtracter, that is most appropriated to this application. But as this example is very simple, when the communication costs are compute, the result is the same with the mean that with the min algorithm. But with a bigger application it is not the case as we can see in the section 6.

5.6 Communication Costs

On figure 3, the final step of the projection algorithm uses a communication costs model to compute the total communication cost. This model describes the costs to perform a data transfer in a hierarchical element and between several hierarchical elements. These costs depend on the routing topologies (mesh, segmented base, hierarchical, etc) and on the routing resources (channel of wires, bus, crossbar, etc). A survey of interconnects architectures for reconfigurable architecture is done in [15]. In our approach these costs are relative since we want to compare two architecture styles instead of giving an absolute performance value. However, if the designer has a good knowledge of the target architecture, precisely costs can be considered.

Once the communication costs are determined, it is necessary to compute the number of communication between each hierarchical element in function of the graph result (result of one of the three previous algorithms). The total communication cost is given by the following expression:

$$\text{CommCost} = \sum_i^n \left(\sum_j^{m_i} C_j \right) \times k_i$$

Where there are n hierarchical elements in the target architecture. The total number of communication in the element i is the sum of communication C_j inside each of the m_i hierarchical element i . The cost for one communication in the hierarchical element i is k_i . In this expression the first sum with the i index allows to scan all the

architecture hierarchical elements. The second index j allows to sum all the communication inside each hierarchical element i .

6 Application and Results

The one dimension Lee-DCT, a typical application of video compression [16], has been chosen to exhibit the exploration process and its ability to compare several architectures. Figure 5 presents the processing graph and the ACG for this application. The processing scheduling during the system estimation step has given four multipliers, four subtracters and three adders to perform this application. Four architectures have been targeted during the projection step, with different hierarchical levels, and cluster sizes (number of functional elements for a hierarchical element). Figure 5 depicts it.

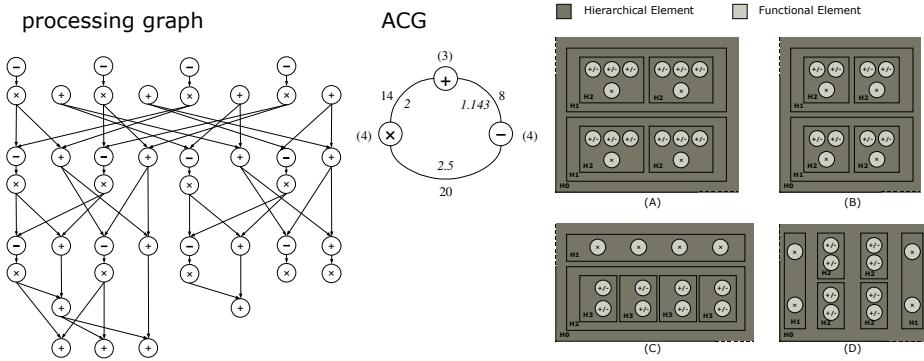


Fig. 5. Processing graph and ACG for the 1-D Lee DCT, and four examples of target architectures.

The results of the projection step is the total number of communications between resources weighted by the costs to communicate between hierarchical elements (communication costs model). For the considered architectures the communication costs model is the following (for the example of architecture A): Cost of communication internal H2 = 0.1, cost of communication H2 to H2 = 0.2, cost of communication H2 to H1 = 0.3

Results are given in figure 6. They highlight several interesting elements. Architectures **A** and **B** are better for this application than architectures **C** and **D**, since in the application the base structure is a MAC so it is necessary to have in a same hierarchical element the possibility to assign one multiplier, one adder and one subtracter.

We can notice that the results are close between the MIN and the MAX algorithm for the architecture **C** and **D**. Since these architectures have a limited exploration potential for this application, so the allocation will not have a strong impact on the final performances.

The last row of the table gives the utilization rate of functional elements in the architecture. Although the **A** architecture is better to reduce communication costs, it has not a high functional element utilization rate. The **A** architecture is larger than the

other architectures, so its static power consumption is higher than the other architectures. With the utilization rate information, we can see that **B** architecture is a good tradeoff between communication cost reduction vs. functional element utilization rate.

In order to show the ability of our approach to make architecture exploration, we have computed the mean communication cost for architectures **A** and **B** with several numbers of hierarchical elements H2 in the hierarchical H1. We can see, on figure 6, that the larger is the cluster H1, the better is the communication cost. Indeed if H1 has many H2, local communications are promoted. This experiment show how is easy to explore some architecture characteristics.

architecture	A	B	C	D
MIN	4.5	4.5	11	7.6
MEAN	6	6.5	11.3	7.9
MAX	9.2	9.45	11.6	8.2
USE	68.7%	91.6%	91.6%	91.6%

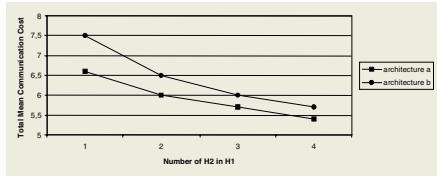


Fig. 6. Table of projection algorithms results, and curves of exploration of cluster size.

7 Conclusions and Future Work

In this paper we have presented a rapid allocation algorithm that takes into account the communications between application's operations. This algorithm uses the hierarchical view of the target architecture that is modeled with a functional model. Communication costs are taken into account with a communication cost model that depends on the routing structure in the target architecture. This algorithm is part of a design exploration flow that works at the algorithmic level and enables to compare quickly several reconfigurable architectures.

This approach which has been integrated in the codesign environment *Design Trotter* [8], enables to explore a large design space at an early stage of the design cycle and to characterize each solution in terms of area versus delay. Future work will complete the design exploration flow development and test it on a tile-based architecture [11].

References

- [1] R. Hartenstein. A Decade of Reconfigurable Computing : a Visionary Retrospective. In *DATE'01, Munich, Germany, 13-16 March, 2001*.
- [2] J. M. Rabaey. Reconfigurable Processing: The Solution to Low-Power Programmable DSP. In *IEEE, ICASSP'97, Munich, Germany, April 1997*.
- [3] V. Betz, J. Rose, A. Marquart. *Architecture and CAD for Deep Submicron FPGAs*. Kluwer Academic Publishers, 1999.
- [4] L. Lagadec, D. Lavenier, E. Fabiani, B. Pottier. Placing, Routing and Editing Virtual FPGAs. In *FPL'01, August 2001*.

- [5] U. Nageldinger. Coarse-Grained Reconfigurable Architecture Design Space exploration. *Ph.D. Thesis, University of Kaiserlautern, Germany, June 2001.*
- [6] C. A. Moritz, D. Yeung, A. Agarwal. Exploring Optimal Cost-Performance designs for Raw Microprocessors. In *FCCM'98, Napa, CA, USA, April 1998.*
- [7] L. Bossuet, G. Gogniat, J.P. Diguet, J.L. Philippe. A Modeling Method for Reconfigurable Architecture. In *IWSOC'02, Banff, Canada, July, 2002.*
- [8] Y. Le Moullec, J.P. Diguet, J.L. Philippe. Design Trotter: a Multimedia Embedded Systems Design Space Exploration Tool. In *IEEE MMSP'02, Virgin Island, USA, 9-11 December, 2002.*
- [9] Y. Le Moullec, N.Ben Amor, J.P. Diguet, M. Abid, J.L. Philippe. Multi-Granularity Metrics for the Era of Strongly Personalized SOCs. In *DATE 03, Munich, Germany, 3-7 March, 2003.*
- [10] J. P. Diguet, G. Gogniat, P. Danielo , M. Auguin, J.L Philippe. The SPF model. In *FDL'00, Tübingen, Germany, September, 2000.*
- [11] L. Bossuet, W. Burleson, G. Gogniat, V. Anand, A. Laffely, J.L. Philippe. Targeting Tiled Architectures in Design Exploration. *RAW'03, Nice, France, April, 2003.*
- [12] M. Budiu, S. C. Goldstein. Fast Compilation for PipeRench Reconfigurable fabrics. In *ACM/SIGDA FPGA'99, Monterey, CA, USA, February, 1999.*
- [13] L. Shang, A. Kaviani, K. Bahala. Dynamic Power in VirtexTM-II FPGA Family. In *ACM/SIGDA FPGA'02, Monterey, Californie, USA, February, 2002.*
- [14] A. Garcia, W. Burleson, J.L. Danger. Power Modeling in Field Programmable Gate Arrays (FPGA). In *FPL'99, Glasgow, Scotland, September 1999.*
- [15] H. Zhang, M. Wan, V. George, J. Rabaey. Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSPs. *IEEE Computer Society Workshop on VLSI, April 1999.*
- [16] V. Bhaskaran, K. Konstantinides. *Image and Video Compression Standards : Algorithms and Architectures*. Kluwer Academic Publishers, 1995.

From Algorithm Graph Specification to Automatic Synthesis of FPGA Circuit: a Seamless Flow of Graphs Transformations

Linda Kaouane¹, Mohamed Akil¹, Yves Sorel², and Thierry Grandpierre¹

¹ Groupe ESIEE-Laboratoire A2SI, BP 99 – 93162 Noisy-le-Grand, France
`{kaouanel,akilm,grandpit}@esiee.fr`

² INRIA Rocquencourt-OSTRE, BP 105 – 78153 Le Chesnay Cedex, France
`yves.sorel@inria.fr`

Abstract. The control, signal and image processing applications are complex in terms of algorithms, hardware architectures and real-time/embedded constraints. System level CAD softwares are then useful to help the designer for prototyping and optimizing such applications. These tools are oftently based on design flow methodologies. This paper presents a seamless design flow which transforms a data dependence graph specifying the application into an implementation graph containing both data and control paths. The proposed approach follows a set of rules based on the RTL model and on mechanisms of synchronized data transfers in order to transform automatically the initial algorithmic graph into the implementation graph. This transformation flow is part of the extension of our AAA (Algorithm-Architecture Adequation) rapid prototyping methodology to support the optimized implementation of real-time applications on reconfigurable circuits. It has been implemented in SynDEX¹, a system level CAD software tool that supports AAA.

1 Introduction

The increasing complexity of signal, image and control processing algorithms in embedded applications requires high computational power to meet real-time constraints. This power can be achieved by high performance mixed hardware architectures built from different types of programmed components (RISC or CISC processors, DSP,...) to perform high level tasks and/or specific components (dedicated boards, ASIC, FPGA,...) used to perform efficiently low level tasks such as signal and image processing and devices control. Implementing these complex algorithms on such distributed and heterogenous architectures while verifying the severe real-time constraints is generally a difficult and complex task. This explain the need for dedicated high level design environnement based on efficient system-level design methodology to help the real-time application designer to solve the specification, validation and synthesis problems [1].

¹ <http://www-rocq.inria.fr/syndex>

In order to cope with these increasing needs, we have developed the AAA rapid prototyping methodology [2] which helps the real-time application designer to obtain rapidly an efficient implementation of his application algorithm on his heterogenous multiprocessor architecture and to generate automatically the corresponding distributed executive. This methodology is based on an unified model of factorized graphs [3], as well to modelize the applicative algorithm and the multicomponent architecture, than to deduce the possible implementations in terms of graphs transformations.

Based on this model, we have extended the AAA methodology to support the implementation of real-time applications on reconfigurable circuits. This extension uses a single factorized graph model, from the algorithm specification down to the architecture implementation, through optimizations expressed in terms of defactorization transformations applied to the algorithmic graph [4]. This optimization aims to satisfy the real-time constraints while minimizing the required hardware resources. In prospect, this extension is expected to allow the AAA methodology to be used for optimized hardware/software codesign.

In this paper, we focus on the rules used to synthesize both the data and the control paths of the circuit corresponding to an algorithm specified as a factorized data dependence graph. It is known that control path synthesis is more difficult to carry out than data path synthesis. We show here that it is possible to synthesize the control path in a secure and systematic way by using a technique of data transfers synchronization based on the RTL model. This approach allows us to carry out an automatic generator of synthesizable VHDL in a simple way. The remainder of the paper is organized as follows: in the next section, we briefly present the transformation flow used by our extended methodology to automate the hardware implementation process of an application algorithm on reconfigurable circuits. In section 3, we present the factorized data dependence graph model used to specify the application algorithm. As critical portions of control, signal and image processing algorithms often consist of regular computations generally expressed as nested loop, we will use a motivating example of matrix-vector product to illustrate the proposed transformation design flow. Section 4 gives rules to automate the synthesis of data and control paths extracted from the algorithm specification while the principles of optimization by defactorization are shown in section 5. We also show in section 6, the results of the implementation of the matrix-vector product algorithm onto a *Xilinx* FPGA following these rules. Finally, we conclude and discuss future work in section 7.

2 AAA Methodology for Circuits

Given an algorithm graph $G_{al} = (O, D)$ specifying the application, we transform it to an implementation graph $G_{im} = (O'', D'')$ following a set of graphs transformations as described in Fig.1. This seamless transformation flow is composed of the generation of the data-path graph $G_{dp} = (O''_1, D''_1)$ and the control-path graph $G_{cp} = (O''_2, D''_2)$ ($G_{im} = G_{dp} \cup G_{cp}$). Data-path transformations are quite simple, but control-path transformations are not trivial and require

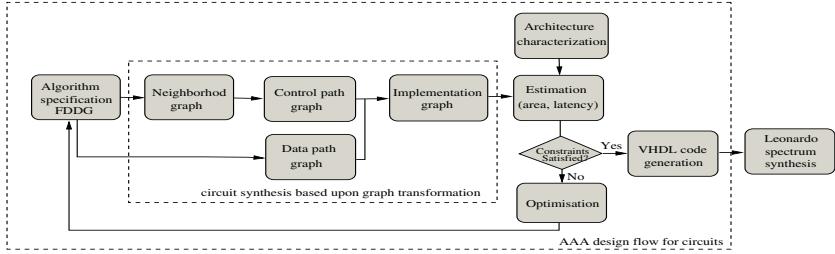


Fig. 1. The AAA methodology for circuits

to build first a neighborhod graph $G_{ng} = (O', D')$. Finally the implementation graph containing both data and control graphs is then charaterized in order to estimate time and area performance. If the deduced implementation does not meet the user specified constraints, we apply a defactorization process in order to reduce the latency by increasing the hardware resources. Since there is a large number of possible defactorized implementations with different characteristics (FPGA area required, latency,...) among which we need to select the most efficient one, we need to use heuristics guided by their cost function. Finally, the VHDL code corresponding to the optimized FPGA implementation is generated.

3 Algorithm Model

The algorithm specification is the starting point of the process of hardware implementation of an algorithm application onto an architecture. According to the AAA methodology, the algorithm model is an extention of the directed data dependence graph (direct acyclic hypergraph DAG), where each node models an operation (more or less complex, e.g. an addition or a filter), and each oriented hyperedge models a data, produced as output of a node, and used as input of an other node or several other nodes (data diffusion). The extended model provides specification of loops through factorization nodes (fork, join, iterate, diffuse), leading to an algorithm model called Factorized Data Dependence Graph (FDDG) [3]. This algorithm graph may be specified directly by the user or it may be generated from high level specification languages such as the synchronous languages (Esterel, Signal,...), which perform formal verifications in terms of events ordering in order to prevent dead-locks [5].

3.1 Factorized Data Dependence Graphs Model

As described in [3], an algorithm specification contains regular parts (repetitive subgraph) and non-regular parts. In fact, these spatial repetitions of operation patterns (identical operations that operate on different data) are usually reduced by a factorization process to reduce the size of the specification and to highlight its regular parts. Graph factorization consists in replacing a repeated pattern,

- slow-downstream: "slow" side of a consumer FF ;
- fast-upstream: "fast" side of a producer FF ;
- fast-downstream: "fast" side of a consumer FF ;
- slow-upstream : "slow" side of a producer FF .

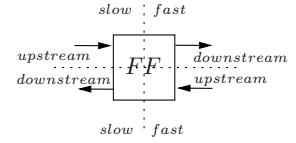


Fig. 2. Node of neighborhood graph for a frontier FF

i.e. a subgraph, by only one instance of the pattern, and in marking each edge crossing the pattern frontier with a special "factorization" node, and the factorization frontier itself by a dashed line crossing these nodes. The type of the factorization node depends on the way the data are managed when crossing a factorization frontier, it may be: a Fork node ' F ' (array partition in as many subarrays as repetitions of the pattern), a Join node ' J ' (array composition from results of each repetition of the pattern), a Diffusion node ' D ' (diffusion of a data to all repetitions of the pattern) or an Iterate node ' I ' (data dependence between iterations of the pattern).

Note that from the algorithm specification point of view, the factorization reduces only the size of the specification, without any modification of its semantics. However, from the implementation point of view, the factorization describes also in intention all the possible implementations, from the entirely parallel one to the entirely sequential one, with all the intermediate cases mixing both sequential and parallel. Obviously, each of these implementation will have different characteristics in terms of area and response time.

3.2 Neighborhood Graph

Every factorization frontier may be a consumer (located downstream) or/and a producer (located upstream) relatively to another frontier according to the data dependences relating them. Two frontiers are neighbors if there is at least one relation of direct dependence that does not cross a third frontier.

Based on these neighborhood relations between the factorization frontiers, we build a neighborhood graph denoted $G_{ng} = (O', D')$. The nodes $o'_{F_i} \in O'$ of such graph represent the factorization frontiers and the oriented edges $d'_i = (o'_{F_i}, o'_{F_j}) \in D'$ represent the data flow between factorization frontiers. The edge orientation describes the consumption/production relation: an edge starts at a producer (o'_{F_i}) and ends at a consumer (o'_{F_j}).

In the case of a sequential implementation, every factorization frontier, called FF , separates two regions, the first one called "fast", being repeated relatively to the second one, called "slow". These slow and fast sides of a frontier are due to the difference of data rate on each side of the factorization frontier. Every node of the neighborhood graph is then subdivided in four parts (see Fig.2).

This neighborhood graph, deduced automatically from the algorithm graph (FDDG), is used during the implementation in order to establish the control relationships between frontiers.

3.3 Example: Specification of MVP (Matrix-Vector Product)

We now use a Matrix-Vector Product example (MVP) to illustrate the algorithm model of specification and its use for the building of the neighborhood graph. So the MVP of one matrix $A \in R^m \times R^n$ by a vector $B \in R^n$ gives a vector $C \in R^m$, and can be written in a factorized form as follows:

$$C = \left[\sum_{j=1}^n a_{ij} b_j \right]_{i=1}^m$$

This equation allows us to obtain the graph corresponding to the algorithm specification of the factorized MVP (Fig. 3). The interface with the physical environment is delimited by input (F_A^∞ et F_B^∞) and by output (J_C^∞). It corresponds to the factorization frontier of the infinitely repeated pattern of the graph (FF_1) since we deal with reactive applications that interact infinitely with the physical environment. The square brackets $[]_{i=1}^m$ correspond to a second frontier (FF_2), delimited by factorization nodes of a finitely repeated pattern. This frontier selects the m lines of the matrix A (F_{21}), diffuses the vector B (D_{21}) and collects the result vector C (J_{21}). The functor $\sum_{j=1}^n$ corresponds to a third frontier (FF_3),

also delimited by factorization nodes of a second finitely repeated pattern. This frontier selects the a_{ij} elements of the i th line of the matrix A (F_{31}) and the elements b_j of the vector B (F_{32}) and it supplies the result of the sum of products between a_{ij} et b_j for every line of matrix A (I_{31}). The “slow” and “fast” sides of each frontier are labeled “s” and “f”, respectively.

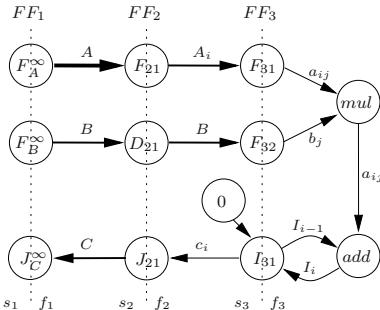
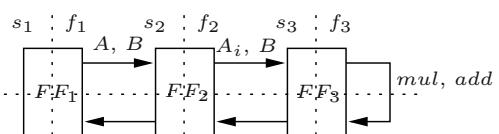


Fig. 3. Factorized data dependence graph of MVP

Fig. 4. Neighborhood graph of MVP: relations between frontiers

The neighborhood graph between factorization frontiers, obtained from the factorized data dependence graph specifying the MVP algorithm, is shown in Fig.4. Because the factorization frontier FF_1 is infinite, it does not have neighbor on its “slow” side which corresponds to the physical environment. FF_1 is, at the same time, a producer (edges A and B) and a consumer (edge C) compared to FF_2 . FF_2 is also a producer (edges A_i and B) and a consumer (edge C_i) compared to FF_3 . FF_3 is a producer and a consumer, compared to itself through the arithmetic operations mul and add .



4 Circuit Synthesis from a Neighborhood Graph

High-level circuit synthesis transforms a high-level behavioral specification into a register-transfert-level implementation (RTL) [6]. The resulting RTL design containing both the data path and control path can then be synthesized using logic synthesis tools that map components such as adders, multipliers,... to gates, perform optimization and finally generate the netlist of the final design. The automation of this synthesis process reduces significantly the development cycle of the circuit, and allows the exploration of different hardware implementations, seeking for an efficient compromise between area and response time of the circuit. Afterward we will present the rules used to automatically generate the data path (denoted by G_{dp}) and the control path (denoted by G_{cp}) of the circuit, from the factorized data dependence graph G_{al} and the neighborhood graph G_{ng} .

4.1 Data Path Synthesis

The hardware implementation of operations consists in providing for every node ($o_i \in O$) of the algorithm graph $G_{al} = (O, D)$ a matching operator ($o''_{op_i} \in O''_1$) (by instantiating the corresponding component of a VHDL library). The matching operator is a logic function in the case of an operation node, or it is composed of a multiplexer and/or registers in the case of a factorization node (F is implemented by a multiplexer, J by a demultiplexer with a memory array, I by a register with initialization value,...). The hardware implementation of the data dependences between operations consists in providing, for each edge ($d_i \in D$) of the G_{al} , a matching connection ($d''_i \in D''_1$) between the corresponding operators. The resulting graph G_{dp} of operators (O''_1) and their interconnections (D''_1) composes the data path of the circuit.

4.2 Control Path Synthesis

The control path corresponds to the logic functions that must be added to the data path, in order to control the multiplexers and the transitions of the registers composing the factorization operators. It is then obtained by data transfer synchronization between registers. However, two conditions must be satisfied to allow a register to change state: the new upstream data to the register must be stable, and all downstream consumers of the register must have finished the use of previous data. If moreover upstream data of a circuit comes from various producers with different propagation time, it is necessary to have a synchronized circuit. This synchronization is possible through the use of a request/acknowledge communication protocol. Consequently, the synchronization of the circuit implementing the algorithm is reduced to the synchronization of the request/acknowledge signals of the set of factorization operators.

These operators are gathered in factorization frontier and their data consumption and production are done in a synchronous way at the level of the frontier. We propose then a control system where each factorization frontier will have its own control unit. This delocalized control approach allows the CAD

tools used for the synthesis to place the control units closer to the operators to control rather than a centralized control approach.

Control Unit: As mentioned above, each factorization frontier has upstream and downstream relations on both sides, “slow” and “fast”. The relations between upstream/downstream and request/acknowledge signals on both sides of a frontier are implemented by the “control unit” of the factorization frontier (Fig.5). This control unit contains a counter C with d states (corresponding to the d factorized repetitions) and an additional logic function in order to generate, in the one hand the communication protocol between frontiers (the slow and fast, request and acknowledge signals at the upstream and downstream sides), and in the other hand the counter value cnt and the enable signal (en), that control the frontier operators. The counter value (cnt) controls the frontier operators: F , J and I . The enable signal (en) determines the clock cycles where the registers of the frontier operators (J , I , F^∞ “sensor”, J^∞ “actuator”) will change state. Note that, the signal ($init$) resets the counter while the signal (end) indicates that the counter is in its last state ($d - 1$).

All the other signals are the request (r) and acknowledge (a) signals generated by the frontier(s) located upstream or diffused to the frontier(s) located downstream. They are separated in two groups: those which relate to the frontier(s) located on the “slow” side and those which relate to the frontier(s) located on the “fast” side, corresponding to the four parts of the control unit: slow-upstream (su), slow-downstream (sd), fast-upstream (fu) and fast-downstream (fd).

Thus, the control path, modeled by the graph $G_{cp} = (O''_2, D''_2)$, is mainly composed of the set of control units associated to the corresponding factorization frontiers. These control units are then inter-connected in an automatic way based on relationships between the factorization frontiers deduced from the neighborhood graph. In this control path graph, the nodes $o''_{UC_i} \in O''_2$ correspond to the control units and the edges $d_{UC_i, UC_j}^{rd-rs} \in D''_2$ correspond to the request signals transmitted between the control units. The acknowledge signals are transmitted, in the opposite direction of the associated request signals, between the same control units. When several signals occur at the same input of a control unit, the conjunction is performed by a logical gate AND.

5 Implementation Optimization: Principles

As previously mentioned, the optimized implementation of a factorized algorithm graph onto an application specific integrated circuit or a FPGA, is formalized in terms of graph transformations, i.e defactorization. When we defactorize a graph we expect to reduce the latency by increasing the number of hardware resources. Thus, the implementation space, which must be explored in order to find the best solution, is composed of all the possible defactorizations of a factorized algorithm graph. For instance, for a given algorithm graph with n frontiers, we have at least 2^n defactorized implementations. Moreover, each frontier can

be partially defactorized: a factorization frontier of r repetitions can be decomposed in f factorization frontiers of r/f repetitions. Consequently, for a given algorithm graph, there is a large, but finite, number of possible implementations which are more or less defactorized, and among which we need to select the most efficient one, i.e which satisfies real-time constraints (upper bound on latency), and which uses as less as possible the hardware resources, logic gates for ASIC and number of Configurable Logic Blocks CLB for FPGA. This optimization problem is known to be NP-hard, and its size is usually huge for realistic applications. This is why we use heuristics guided by their cost function, in order to compare the performances of different defactorizations of the specification. These heuristics, using tricks related to practice, allows us to explore only a small subset of all the possible defactorizations into the implementation space. These cost functions take into account the characteristics of an implementation: hardware resources required (number of gates or CLBs) and latency [3].

6 Example: Synthesis of MVP Implementation on FPGA's Circuit

The Fig.6 represents the hardware implementation of the factorized MVP corresponding to the algorithm specification given in Fig.3. The data path is composed of the factorization frontier operators ($F_{i,j}$, $D_{i,j}$, $J_{i,j}$ and $I_{i,j}$) and of the combinatorial operators *mul* and *add* delimited by a dotted box. The control path is composed of the control units UC_1 , UC_2 and UC_3 , and of the control signals r (request), a (acknowledge), cnt and en . The interconnections between the request and acknowledge signals, is based on the relationships between the factorization frontiers, namely the neighborhood graph (Fig.4) built from the algorithm graph.

In Fig.7 we present the hardware implementation of a defactorized solution corresponding to the partial defactorization of the frontier FF_2 by a factor of 2. The FF_2 frontier has been replaced by two frontiers FF_{2a} , FF_{2b} being repeated $m/2$ times (m : factor of repetitions of FF_2). The factorization frontier FF_3 remain unchanged but it has been duplicated (FF_{3a} , FF_{3b}) due to the partial defactorization of FF_2 . The data path is then composed of the factorization frontier operators, the combinatorial operators delimited by dotted boxes and of the operators X_1 (array-decomposition operation), M_1 (array-composition operation). The control path is composed of the control units UC_1 , UC_{2a} , UC_{2b} , UC_{3a} and UC_{3b} . The synchronisation of frontiers FF_{2a} , FF_{2b} is assured by the AND gate at the upstream request and the downstream acknowledge of UC_1 .

Tab.8 shows the synthesis result of the generated VHDL code of hardware implementation of MVP (6×6 matrix and 6 elements vector, coded on 3 bits) onto a *Xilinx* FPGA XL4000XL-3 4005xIPC84, using the CAD tool *Leonardo Spectrum 2003*, developed by *Exemplar Logic Inc.*. The implementation results are presented in function of, the area (hardware ressources: number of CLBs), the number of clock cycles required by the algorithm execution, the maximum frequency of operators in *MHz*, and finally the latency in *ns*.

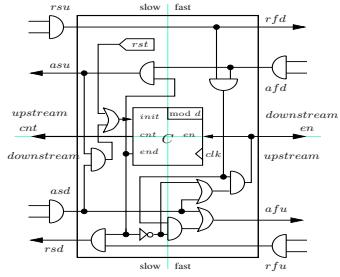


Fig. 5. Control unit

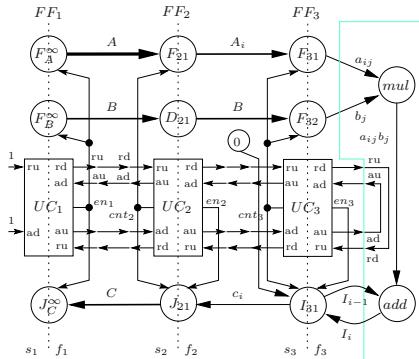


Fig. 6. Implementation graph of MVP

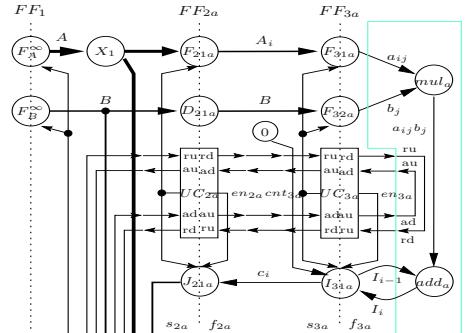


Fig. 7. A defactorized implementation graph of MVP

7 Conclusion and Future Work

We have shown that from an algorithm specification based on a factorized data dependence graph model it is possible to generate automatically a hardware implementation onto an FPGA circuit, employing a set of rules for the data path and the control path synthesis. The delocalized control approach presented in this paper allows the CAD tools used for the synthesis to place the control units closer to the operators to control rather than a centralized control approach. We validated this seamless graph based design flow on several examples of low-level image processing applications that includes interesting cases of data factorization like: mean filtering [4], edge detector operators (sobel, deriche [7], ...).

This work is part of the extension of the AAA methodology implemented in the software tool SynDEx to support implementation onto reconfigurable circuits. Basically, AAA/SynDEx for multiprocessors, allows to generate automatically the dead-lock free executive for the optimized implementation of the given algorithm onto the specified multiprocessor architecture [8].

The principles described in this paper allowed us to carry out an automatic generator of structural synthesizable VHDL for mono-FPGA (one FPGA) architectures, that has been added to SynDEx [9]. The generated VHDL code which

<i>Implementation</i>	<i>Area</i> (CLB)	<i>Nb.</i> <i>cycl.</i>	<i>Freq.</i> (MHz)	<i>Lat.</i> (ns)
Factorized Spec.	76	36	12,4	2916
Part.defac. by FF_2	99	18	13,5	1332
Fully. defac. by FF_2	168	6	14,3	420
Part. defac. by FF_3	92	30	10,8	2790
Fully. defac. by FF_3	79	6	9,0	660
Fully. defactorized	234	1	11,4	87

These results represent some possible implementations explored by the optimization heuristic by partial defactorization (as described in [4]) of the initial factorized implementation. Note that these defactorized solutions allow to reduce the latency of the implementation, but they increase the number of required hardware resources (CLB).

Fig. 8. Optimization results for the implementation of MVP onto FPGA

corresponds to the optimized FPGA implementation obtained by successive defactorizations of the factorized algorithm graph, is then used by a CAD tool in order to generate the netlist needed for the FPGA configuration.

Currently we are working on the control involved by the conditioning in the algorithm specification, in addition to the control involved by repetition of operation. After that we plan to extend the proposed methodology to the case of multi-FPGAs architectures. To support such architectures, the optimization heuristic will address both defactorization and partitioning issues.

References

1. S. Edwards, L. Lavagno, E.A. Lee, A. Sangiovanni-Vincentelli. *Design of embedded systems: formal models, validation, and synthesis*. Proc. of IEEE, v.85, n.3, March 1997.
2. T. Grandpierre, C. Lavarenne, Y. Sorel. *Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors*. CODES'99 7th Intl. Workshop on Hardware/Software Co-Design, Rome, May 1999.
3. L.Kaouane, M. Akil, Y. Sorel, T. Grandpierre. *A methodology to implement real-time applications on reconfigurable circuits*. ERSA'03, Intl. Conference on Engineering of Reconfigurable Systems and Algorithms, Las vegas, USA, June 2003.
4. A. F. Dias, C. Lavarenne, M. Akil, Y. Sorel. *Optimized implementation of real-time image processing algorithms on field programmable gate arrays*. Proc. of the 4th Intl. Conference on Signal Processing, Beijing, Oct. 1998.
5. N. Halbwachs. *Synchronous programing of reactive systems*. Kluwer Academic Publishers, Dordrecht Boston, 1993.
6. D. Gajski, F. Vahid. *Specification and design of embedded hardware-software systems*. IEEE Design & Test of Computers, Spring 1995, p. 53-67.
7. L.Kaouane, M. Akil, Y. Sorel, T. Grandpierre. *An automated design flow for optimized implementation of real-time image processing applications on FPGA*. Eurocon'03, Intl. Conference on computer as a tool, Ljubljana, Slovenia, sept. 2003.
8. T. Grandpierre, Y. Sorel. *From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations*. MEMOCODE03, Intl. Conference on Formal Methods and Models for Codesign, Mont Saint-Michel, France, June 2003.
9. R. Vodisek, M. Akil, S.Gailhard, A.Zemva. *Automatic Generation of VHDL code for SynDex v6 software*. Electro technical and Computer Science conference, Portoroz, Slovenia, september 2001.

Adaptive Real-Time Systems and the FPAAs

Stuart Colsell and Reuben Edwards

Department of Communication Systems, Lancaster University,

Lancaster, Lancashire, U.K., LA1 4YR

s.colsell@lancaster.ac.uk, r.edwards@lancaster.ac.uk

Abstract. Adaptive real-time systems are often typically considered by design engineers to be a solely digital preserve. However, in recent years analogue technologies such as the Field programmable Analogue Array (FPAAs) have been developed to support real-time reconfiguration. These technologies can offer an analogue platform for a vast number of applications requiring adaptive processing and form the subject of this paper. Although in terms of their technology and software/algorithmic support, FPAAs are still in the relative infancy compared with digital techniques, we aim to show their advantages and how present devices can be exploited in real-time applications.

1 Introduction

The application of adaptable devices has migrated from the prototyping stages of design, into the final commercial architectures of real-time systems. This is particularly so, within the field of communications where flexible systems such as Software Defined Radio (SDR), require the ability to support of multiple-standards and protocols. Modern design techniques and the demand for fast time-to-market products have driven the development of reconfigurable devices. These devices can, and have been used in very diverse products, both in the development stage and within the finished article. Until recently these flexible devices have materialized predominately within the digital domain. But now analogue circuitry has achieved the adaptability required, by using FPAAs. This paper discusses the challenges faced by both FPAAs device designers and end users.

2 Real-Time Requirements of FPAAs-Based Systems

There are a number of real-time applications, which would benefit greatly from the flexibility of FPAAs devices. The key application for this research is the use of these devices within real-time communication systems. Such systems require that; any reconfiguration of its current processing architecture should be carried out with minimum disruption to the data flow. This is very challenging as the buffering of data is not always an option. In order to meet the system performance criteria the reconfiguration of the devices needs to be as fast as possible with minimal degradation to the implementation. Often the algorithms used to ensure a high

performance implementation can hinder the speed in which the device can be configured. This presents an argument for more application specific architectures in order to minimize the analysis of place and route options, thus moving toward tuning a circuit to suit the new requirements rather than having an architecture that can be completely changed. Another alternative would be to use parallel circuits within a single device using redundant resources. This method would ensure that the replacement circuit could be placed and routed carefully before removing the previous circuit from the data path.

Current and Chu have presented a new maintenance approach for increasing the reliability of analogue ICs by the direct self-calibration and built-in self-test of off line replacement sub-circuits for an analogue function block, and the commutation, switching, of replacement sub-circuits into the signal path without disruption of the analogue signal path or any loss of analogue functionality. The commutation scheme connects the replacement sub-circuit in parallel with the active sub-circuit to be replaced before it is removed. The new commutation control signal switching edge is carefully designed to minimize “clock” feed-through noise. The strategy for on-chip analogue function maintenance is to create analogue functions with adjustable figures of merit, test and adjust (calibrate) redundant functional blocks while they are off-line, and replace on-line active functional blocks with calibrated redundant blocks when desired. Steps required in the use of the strategy are to select an analogue function that is to be maintained on-chip, define the figures of merit that indicate acceptable performance, coalesce figure-of-merit-adjustment circuitry with the analogue function, create the necessary on-chip excitation and measurement circuitry, create circuitry for delivery off-chip of the analogue test results of digital indications of the test results [1].

The use of parallel devices to support seamless reconfiguration may seem like an expensive approach in terms of power consumption, PCB real estate, and cost. But in fact, when using a filter implementation to compare analogue and digital reconfigurable solutions [2], using two analogue devices consumes less power, PCB real estate, and costs less than using one FPGA. Often with smaller FPGAs only one filter can be implemented on a device. So we have a trade-off between the digitals implementation accuracy and software support and the analogue’s power efficiency, smaller size, reduced cost and lack of latency. The question is, can the analogue devices provide acceptable accuracy to out weigh the digital advantage? This will be explored in the following section.

3 Circuit Performance Repeatability

It is important to consider the performance repeatability of circuits based within reconfigurable technologies. When a designer complies a circuit with a performance criterion, he or she requires that it be maintained each time the circuit is downloaded or transferred from one device to another. This is particularly important when considering multipath systems because they rely on the real and complex paths being very closely matched. This is particularly important when considering matched filters within I and Q paths of a communication receiver.

Matched filter imperfections can have a detrimental effect on the recovery of the wanted signal. Imperfections materialize in the form of Amplitude and phase error.

Amplitude error is the main cause of signal mirroring errors in the passband and it is caused solely by resistor mismatch. Phase errors however occur at the edges of the passband and are caused by both resistor and capacitor mismatch.

If FPAA technology is to be used in complex multipath systems, it is important to investigate how well it can support the filtering requirements. For the purpose of this experiment a 9 KHz channel filter design was chosen to demonstrate the ability of an FPAA to produce matched filters suitable for an Intermediate Frequency stage of a DRM or HF receiver. The filter design was implemented twice within a single switched capacitor based Anadigm AN10E40 FPAA.

4 Results

The matched filters within the FPAA were tested using both a frequency sweep from a spectrum analyser. In this section we will show the result of the measurements. A frequency sweep was made first to obtain the response of one of the filters and compare it to the noise floor. The plot from this test can be seen in figure 1. It is clear from this plot that there is some non-linearity in the response of the filter; this however, can be corrected with more accurate impedance matching. The plot also shows that the filter has a dynamic range of approximately 25dB in the passband. This is due to gain compression and the high noise floor. This high level of noise can be partially attributed to the chips clock signal and internal switching. This is partly because the clock input pin is positioned next to the analogue output pins of the device. The clock problem and the internal noise have been addressed in the new Anadigm Vortex family of devices.

The frequency response of both filters was measured and can be seen in figure 2. The plot shows a very slight amplitude mismatch when studied closely. This however can be removed easily using an Active Gain Control algorithm within the DSP section of a communications receiver. Using an oscilloscope, an attempt was made to observe any phase mismatch between the two filters but it was too slight to view and record.

5 Conclusions

Currently FPAA technologies can be used for a wide range of applications including instrumentation, control systems, analogue baseband and Low/Medium frequency transceiver front-end circuitry. The most recent field programmable analogue devices have shown great improvements in their ability to support more practical applications. This is most apparent when considering the 400% increase in bandwidth over the last six years. However it would be unrealistic to expect such a high rate of improvement to continue unless a routing scheme is developed, which introduces a minimal amount of performance degrading effects. We can see from the results of the circuit performance repeatability experiment that when using switched capacitor based FPAs the matching of circuit performance is quite good and can be deemed acceptable for communication systems such as DRM and HF.

FPAA technologies can provide adaptive processing without compromising battery life and portability. It is clear that further development is required especially to the

software support/design tools and speed of configuration. Comparisons based on circuit complexity, chip size, power consumption and latency show that reconfigurable analogue can be considered a very efficient alternative solution to digital devices such as the FPGA. This is particularly apparent when considering operations such as filtering [2] within power critical mobile applications.

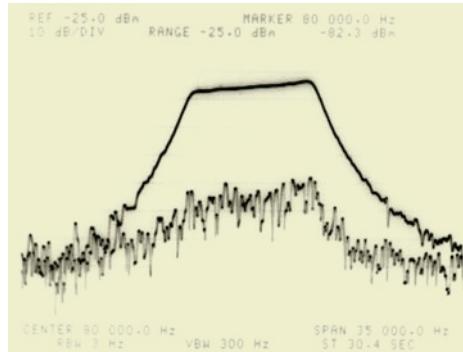


Fig. 1. Filter frequency response & noise floor.

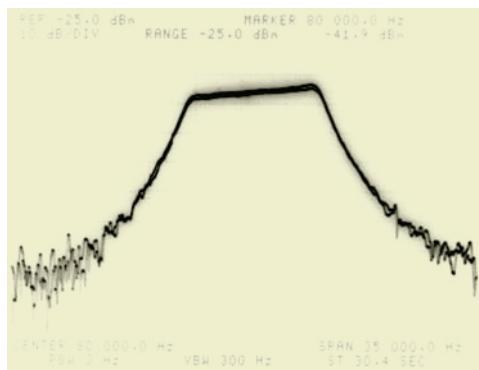


Fig. 2. Overlaid frequency responses of I & Q filters.

References

1. Current, K. W. & Chu, W. 2001, Demonstration of an Analogue IC function maintenance strategy, including direct calibration, built-in self-test, and commutation of redundant functional blocks, *Analogue integrated circuits and signal processing*, No.26, 129-140.
2. Colsell, S. A. & Edwards, R. 2002, FPAD Versus FPA For Future Mobile Communications, *Proc. 9th International workshop on systems, signals and image processing (IWSSIP)*, Manchester, UK, pp. 181-185.

Challenges and Successes in Space Based Reconfigurable Computing

Mark E. Dunham, Michael P. Caffrey, and Paul S. Graham

Los Alamos National Laboratory
Los Alamos, NM 87545

mdunham@lanl.gov, mpc@lanl.gov, graham@lanl.gov

Abstract. For 6 years Los Alamos has developed space-compatible versions of Reconfigurable Computing (RCC). Several such designs are now operational. We describe the key research steps required to make commercial silicon processes amenable to Radiation Tolerant operations, and the limits on algorithms imposed by reliability concerns.

1 RadHard-by-System-Design for an FPGA

Over the last 6 years a new means of using near-commercial silicon processes for Radiation Tolerant applications in space has emerged[1], which we call Radiation Hardness by System Design (RHSD). This method allows the identical mask sets used for commercial products to be used in creating Radiation Tolerant FPGA versions on epitaxial or SOI substrates, thereby removing the enormous technology lags now seen in space processing ASICs, while offering useful total dose levels (TLD) and latch-up immunity (SEL). Single event upsets due to sporadic radiation must still be tolerated however; which is the subject of this FPGA research work.

Unlike hardware approaches to mitigating upsets, our techniques do not attempt to make FPGA circuits 100 % tolerant of SEUs, but instead allow them to operate through SEU without requiring system resets. While a number of techniques are used to achieve high SEU reliability, our primary means of mitigation depends on being able to read back the state of the FPGA continuously, which is a Xilinx feature. Heavy-ion testing has shown that the Xilinx XQR300 average saturation cross-section per bit is $8\text{e-}8 \text{ cm}^2$ and that the cross-section measured for the Single Event Functional Interrupts (SEFIs) is $1\text{e-}5 \text{ cm}^2$ total per device[2]. The set of all upsets includes categories with different consequences and different cross-sections. Most categories are common with many digital devices. But the Virtex has some unique sensitivities such as the configuration bitstream, half-latches, and the configuration management controller. Results from several Los Alamos tests of the Virtex XQV300 FPGA are shown in the overleaf table. Lookup table (LUT) and configuration bits are both observable in the configuration readback bitstream, which the device can provide through the SelectMAP configuration interface while the design is in operation. This makes the vast majority of static upsets directly observable while the system is in service. Primary bit stream and LUT errors are corrected via hardware monitoring techniques described in Section 3. Recent Los Alamos work has shown that “half-

Table 1. Xilinx XQVR1000 SEU Test Results, partitioned by source in the FPGA

Resource	Contribution in Bits	Fraction of Total
Configuration	5,603,456	91%
LUT Bits	393,216	6.4%
Block Select RAM	131,072	2.1%
User Flip-Flops	26,112	0.4%
Single Event Functional	?	<.0021%
Transients	?	?
Half Latches	?	?

latches,” are also susceptible to SEUs, and are a significant contributor to residual SEU not addressed by hardware monitoring. When upset, the output values of these circuits remains inverted until the device is fully reprogrammed, and this inversion is not directly observable by reading back the configuration bitstream. To address this problem we created a tool, RadDRC,[3] which parses the XDL representation of a design to locate half-latch issues and generates an FPGA Editor script to automate their removal .

2 RHSD Architectures for Signal Processing

In all FPGA instantiations, many Virtex resources are left unused, so not every upset results in incorrect processing. The Virtex SelectMAP interface allows the device's configuration data to be read back while the device is in use, the primary feature we exploit to detect upsets. In addition, the Virtex can be partially configured, allowing rewrite of only the corrupted segment. These are key features of our Radiation Tolerant space processor designs, allowing mitigation of SEU without significant interruption in dataflow through the processor.

Our reconfigurable processor module shown in Figure 1 uses three Xilinx Virtex XQVR1000 FPGAs as the data processors. The FPGAs each have identical pin-outs so they can share configuration files. The module has two high bandwidth TTL busses for data input and output, and a ring bus for inter-processor data flow. The busses run at 50 MHz to deliver 200 Mbytes/sec per bus. Signaling is compatible with the VME FPDP specification, with 32 data lines, strobe and data valid. Each module also has a resident Actel RT54SX32S device that acts as a microprocessor interface and board controller. The Actel provides watchdog monitoring for the three Xilinx FPGAs as well as a configuration interface. It also manages the Xilinx bitstream SEU detection by continuously reading each Xilinx configuration, available for reconfiguration). Each FPGA configuration is read every 180 msec calculating a cyclic-redundancy check (CRC) for each frame (the finest granularity while the device is in operation, with no interruption in service, resulting in better than .999 unperturbed dataflow for typical low earth and geostationary orbits.

A disadvantage of SRAM-based FPGA technology for space is the significant power consumption and corresponding concerns for thermally induced mechanical failure. One key consideration is that the power consumption is a function of the processing

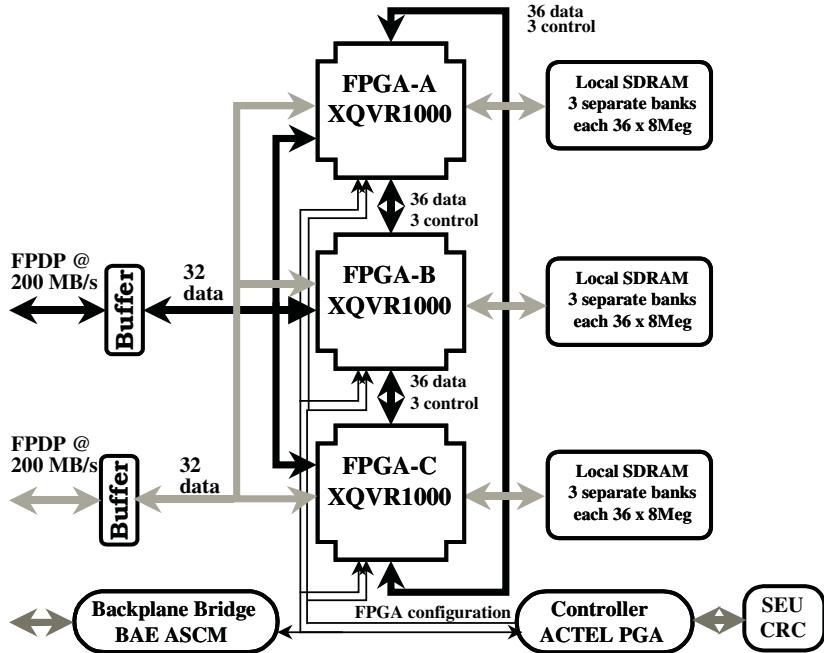


Fig 1. CIBOLA Space Based Reconfigurable Computer Module

being performed. Different configurations will have widely varying power requirements. We are limiting peak power consumption to $\sim 6\text{W}$ / FPGA, which is still significant in light of a 560 grid array package that is 42.5 mm^2 , cavity down. See references for detail on how we have chosen to manage assembly thermal issues[4].

3 Gabor Software Radio as an Application Example

The overlapped FFT process, more formally known as a Gabor Transform when near-perfect reconstruction is demanded of the processed data, is a highly useful practical example[5]. Here we have done extensive work, driven by our interest in software radio. Modern Gabor processing is derived from previous work in Trans-Multiplexer filter banks, but uses frame theory to derive more rigorous design methods for the reconstruction window operator. Two benefits accrue from such a rigorous formalism: 1) Ability to reconstruct analyzed data to arbitrarily fine precision using modern window designs, and 2) ability to relax the required block oversampling from 2:1 to 4:3 or even 5:4, with the same savings in processing rates. For input resolutions of up to 16 bits, we find that rate 4:3 gives errors of less than -100 dB RMS , which is more than adequate for most practical applications.

To the basic Forward Gabor Transform, we add two key steps in order to achieve a compressed output datastream, suitable for further processing. A magnitude and phase representation of the complex data is much more natural than rectangular format for most applications. CORDIC operators are efficient means of implementing

polar conversion in an FPGA. Then, log compressing the magnitude only arrives in a natural domain for multiplicatively modulated signals. This sequence of operations, and the effect on convolution of two signals is shown below diagrammatically.

$$\begin{aligned} S_1(t) \otimes S_2(t) &\xrightarrow{\text{FFT}} S_1(f) S_2(f) \xrightarrow{\text{Polar}} |S_1| |S_2| (\angle_1 + \angle_2) \\ & \& \\ |S_1| |S_2| (\angle_1 + \angle_2) &\xrightarrow{\text{Log}} (\text{Log}|S_1| + \text{Log}|S_2|)(\angle_1 + \angle_2) \end{aligned}$$

A case study for the current XQVR1000 architecture shows a 4096 point FFT with 1/3 overlap, windowing, and log magnitude/phase post-processing has been designed over 2 FPGA, operating at a 100 MHz data input rate with ~12 watts dissipation. For this design, it was necessary to use internal, dual-ported Block SelectRAM and internal LUT RAM to provide the required rates. However the cores run at 100 MHz, beyond the required 67 MHz rate, leaving idle time to run test vectors. This demonstrates a second means of bitstream SEU test, allowing the use of fast LUT RAM in a radiation environment. The work also shows how valuable new hybrid ASIC/FPGA designs will be in reducing die power dissipation.

4 Conclusions and Directions

We have described an exciting new capability for space processing, that provides many useful features needed for current and future commercial payloads. This capability is now moving into use in various government and commercially sponsored projects. A major attribute of RCC is that new generations of FPGA are constantly being released, but each new release stretches the boundaries of RHSD design. Therefore, space RCC is far from mature, and many topics for further research and development remain. We gratefully acknowledge the contributions of a large team in performing this work, and the sponsorship of the US Department of Energy.

References

1. Graham, P. et. al. Reconfigurable Computing in Space: From Current Technology to Reconfigurable Systems on Chip. *Proc. IEEE Aerospace Conference, Big Sky, MT.* (2003).
2. Fuller, E. et. al.: Radiation Testing Update, SEU Mitigation, & Availability Analysis of the Virtex FPGA for Space Reconfigurable Computing. *Proc. 3rd Conf. On Military & Aerospace Programmable Logic.* Columbia, MD (2000)
3. Graham, P.: RadDRC software LACC-02-39. Available via a license through Los Alamos National Laboratory, Industrial Business Development Office. Los Alamos, NM (2003)
4. Caffrey, M.: A Space Based Reconfigurable Radio. In *Proc. Engineering of Reconfigurable Systems and Algorithms.* Las Vegas, NV (2002)
5. Qian, S., Chen, D.: **Joint Time-Frequency Analysis.** Prentice-Hall PTR, Upper Saddle River, NJ (1996) 45-74

Adaptive Processor: A Dynamically Reconfiguration Technology for Stream Processing

Shigeyuki Takano

Graduate School of Computer Science and Engineering
University of Aizu
Aizu-Wakamatsu, Fukushima-ken 965-8580, Japan
`d8021103@u-aizu.ac.jp`

Abstract. In order to improve the performance of reconfigurable computing, the number of reconfigurable units is increased with advance of semiconductor technology. The array of reconfigurable units can be configured to application-specific pipelined processing datapath. Then configuration overhead will be critical overhead of total execution time for dynamic reconfiguration based system. In this paper, models of efficient configuration methodology and application-specific pipelined stream processing are proposed. Adaptive processor architecture is also proposed, and discussed in summary.

1 Introduction

Performance is improved by three factors of array processing, clock cycle time, and overhead. Increasing the number of reconfigurable units provides higher performance of the array processing. The array construct an application-specific pipelined datapath that makes clock cycle time be short and hides the global communication delay. In addition, set of data may be composed to a stream that flows the configured datapath. Generally, an overhead decreases the effects of array processing and application-specific pipelining. Regarding reconfigurable computing, the reconfiguration time is critical overhead of total execution time with enlarging the size of array. However, the overhead reduction is not yet discussed well.

Section 2 proposes computation model, how to construct a stream processing and its coarser data granularity datapath efficiently. Section 3 proposes a processor architecture that focuses onto compiler workload reductions, eliminating place and route, and eliminating communication and interaction between kernel on host and kernel on reconfigurable unit. Finally, the model and processor architecture are summarized.

2 Computation Model

Concept of Object

In order to sequence configuration for coarser data granularity datapath, concept of an *object* is proposed. The *logical object* consists of set of information, result data and status.

An object ID (a tag) is assigned to each logical object to identify. By the addressing of object ID, the logical object is loaded from main memory into *physical* object(s) which is prefabricated hardware. Then configuration data called as static configuration data addressed by object ID is also loaded from main memory into the physical object. A dynamic configuration data is also loaded into the physical object from instruction at the same time. Physical object consists of storages for the logical object, object ID, static and dynamic configuration data, router, and reconfigurable fabric. Let's call pair of logical object and its configuration data as object. Any hardware unit in the processor is an object. The set of objects constructs an object space in the processor.

Instruction Scheduling and Configuration Cache

Application program is partitioned into two parts, set of logical object and instruction stream. Object specifies an operation. The detail of operation is specified by the static and dynamic configuration data. Instruction consists of processing object's ID field, two referenced object's ID fields for binomial model, and dynamic configuration data. There is no opcode, and instruction decoder and its pipeline stage are not necessary. In order to schedule instruction, general resource systems proposed by [1] is applied to it. Resource is the objects. The systems perform request, acquirement, and release of objects. Output data dependency is equivalent to resource conflict in the systems. At the resource conflict, adaptive processor pipeline waits and stalls for release of acquired object. The adaptive processor pipeline is as follows.

- Pipeline Stage 1: Request

Instruction requests the objects. If all requested objects are in the object space and processing object is not acquired, then goto Pipeline Stage 2. If requested object is not in the object space, then the object is stored into physical object. If there is no physical object for the storing, then not acquired object is replaced for the store.

- Pipeline Stage 2: Acquirement

During the acquirement phase, object processes operation and sends result data and local control signal to chained following object(s). When release token is fired, then goto Pipeline Stage 3.

- Pipeline Stage 3: Release

The firing of release token sends result data, local control signal, and release token to chained following object(s). Object does not process configured operation.

In order to explain stream processing and its configuration methodology, execution of dot product is used as an example. Figure 1 (a) shows the dot product program. The necessary objects makes instruction stream as shown in Figure 1 (b). Name of object is shrinked as "Obj" in the figure. Object ID is named by a number. As shown in Figure 1 (b), the object ID is assigned to each necessary object. Then the ID addresses the logical object and its static configuration data. Instruction stream constructs coarser data grain datapath, a data flow graph based on node of object. True data dependency between instructions is used for chaining objects to construct the datapath. Based on the instruction scheduling, instruction stream is sequenced and it constructs the datapath in the object space as shown in Figure 1 (c), (d), and (e). Then streaming of vector elements

is overlapped with the following sequence including the configuration. Black circle on paths from object 5 to two sequencers shows release token. Scalar data always has the release token. Flow of the release token and its firing technique makes in-order release for objects. Although in this example object keeps result, there are cases of when the vector store is used for generating vector result.

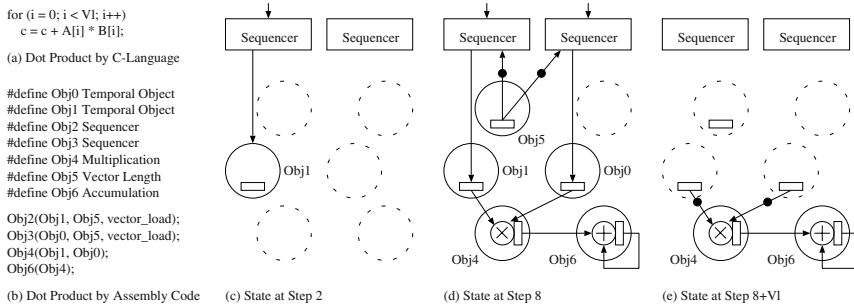


Fig. 1. Dot Product Execution

Logical object information and its configuration data are in object space after the processing. When the object is requested by other following instruction, the object is in the object space. This is a cache hit, called as configuration cache technology [2]. Placement location and replacement are decided by the cache technology. The replacement decides unused object for placement of requested and cache miss object. Not acquired objects are idle, however, they play as role of the configuration cache.

3 Adaptive Processor Architecture

Figure 2 (a) explains basic organization without instruction sequence datapath to simplify the figure. It consists of instruction register (IR), working set register file (WRF), cache miss handler (CH), and physical objects. The IR is similar to conventional processor. WRF holds acquired instructions. Acquiring an instruction is to store it into the WRF. Search for requests, cache hit signals, and acquirement signal from a selected register of WRF perform an instruction scheduling, the pipeline transition, and does chaining objects based on true data dependency between instructions. After a register of WRF receives release token from a fired object, the register is free, the instruction (objects) is released. CH performs replacement and placement of requested cache miss object. The adaptive processor architecture is tightest coupling model, and there is no host processor. There is also no register file, thus, no spill and fill instructions, and no register allocation. Figure 2 (b) explains a router simplified basic organization. A bus is only used to chain objects for one true dependency, no arbitration exist. Physical objects, set of buses, planes, and working set registers are replicated to scale the processor.

The configuration cache architecture extends the replacement algorithm studied by [3]. The contents of list are a set of information which physical object has.

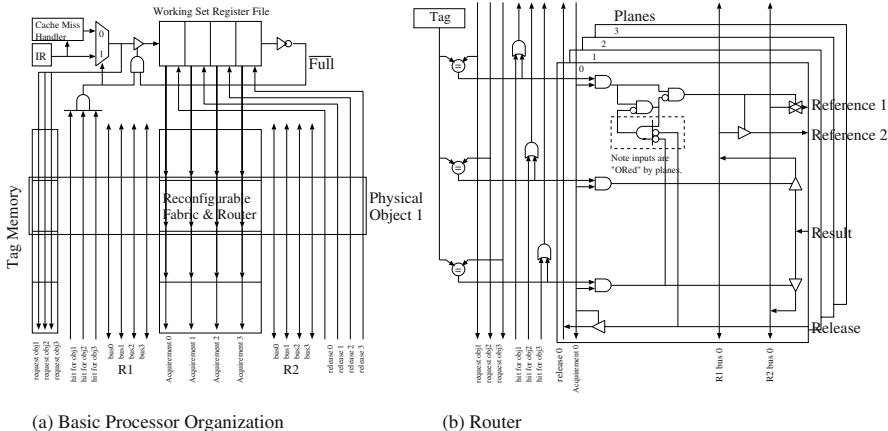


Fig. 2. Adaptive Processor Basic Organization

4 Summary

In this paper, models of efficient configuration methodology and application-specific pipelined stream processing were proposed. Concept of object, release token technique, and configuration cache, are introduced and used respectively for the sequence. Instruction is scheduled by general resource systems. Configuration cache efficiently uses silicon resource for physical objects. The adaptive processor architecture is also proposed, and it is tightest coupling model that does not need to partition application to kernel on host and kernel on reconfigurable fabric, and not need to analyze the interaction. It does not need to place and route for the objects. The instruction stream (excepts for a part of dynamic configuration data) and set of logical object are compatible information for any adaptive processor architectures. We can select an interconnection network for chaining and reconfigurable fabric architectures on demand. Compared to other models, speedup for first architecture comes from overhead reduction, overlapping configuration and processing, reducing reconfigurations, temporal variables, memory accesses, eliminating register file and its workload, and eliminating instruction decoder and its pipeline stage. Therefore, total compiler workload is less than in other models. Future work is to develop simulator and to design a vector cache.

References

- Richard C. Holt, Some Deadlock Properties of Computer Systems, Computing Surveys, Vol. 4, No. 3, pp. 179-196, September 1972
- Micheal J. Wirthin and Brad L. Hutchings, DISC: The dynamic instruction set computer, Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing, Proc. SPIE 2607, pp. 92-103, 1995
- R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Trainger, Evaluation techniques for storage hierarchies, IBM Systems Journal, Volume 9, Number 2, pp. 78-117, 1970

Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns

Christopher R. Clark[†] and David E. Schimmel

School of Electrical and Computer Engineering,
Georgia Institute of Technology, Atlanta, GA 30332
`{cclark, schimmel}@ece.gatech.edu`

Abstract. This paper presents techniques for designing pattern matching circuits for complex regular expressions, such as those found in network intrusion detection patterns. We have developed a pattern-matching co-processor that supports all the pattern matching functions of the Snort rule language [3]. In order to achieve maximum pattern capacity and throughput, the design focuses on minimizing circuit area while maintaining high clock speed. Using our approach, we are able to store the entire current Snort rule database consisting of over 1,500 rules and 17,000 characters into a single one-million-gate FPGA while comparing all patterns against traffic at gigabit rates.

1 Introduction

Network intrusion detection systems (NIDS) have become critical network security components due to the rapidly growing rate of reported security incidents [1]. This trend, coupled with the fact that network traffic is increasing faster than computer performance [2], places an overwhelming burden on NIDS. Current NIDS are struggling to keep up with traffic rates above 100 Mb/s, while 1 Gb/s networks are common and 10 Gb/s networks are being deployed. It is clear that the gap between network traffic rates and NIDS analysis rates must be addressed. One of the most computationally-intensive tasks performed by rule-based NIDS, such as Snort [3], is pattern-matching on packet content [4]. Despite improved software pattern matching algorithms [4,5], pattern matching is still the limiting factor in the analysis of high-speed traffic. The goal of our research is to eliminate this bottleneck by offloading all the pattern matching tasks to a reconfigurable FPGA co-processor.

The effectiveness of NIDS can always be improved by adding patterns, therefore simply dedicating more hardware resources will not solve the problem; efficiency must be improved as well. The task of matching a large number of patterns against small data sets (packets) is different from classical research, which uses few patterns and large data sets. A method of generating circuits for regular expressions using nondeterministic finite automata (NFA) was presented in [6]. In [7], JHDL [8] was used to develop an NFA circuit generator that translated Snort rules into a pattern matching circuit. We use a similar approach, but our generator supports additional pattern matching functions and produces more efficient circuits. In addition, our design outputs a bit-vector indicating the Snort rules that match the input.

[†] This work was supported in part by NSF Grant 9876573 and by a grant from Intel Corporation.

2 NFA Circuit Optimization and Extension

Logic and Routing Resource Optimization. A circuit that implements an NFA for pattern matching consists of a pipeline of character match units. On a LUT-based FPGA (eg. Xilinx and Altera), one way to implement the match function using two logic elements is shown in general in [6] and specifically for Xilinx parts in [7]. We will now describe an optimization that allows a character match unit to fit into a single logic element.

The key observation leading to the reduction in area of a character match unit is that a full 8-bit comparison does not need to be performed by each unit. In fact, with a pattern set containing several thousand characters it is likely that there will be hundreds of identical 8-bit comparisons performed. These redundant comparisons waste valuable logic and routing resources and can be eliminated. Each character match unit only needs to know whether or not the input character matches the unit's programmed target character. This can be achieved with 8-to-256 decoder. Rather than broadcasting the 8-bit character to every unit, this approach sends the 1-bit output of the decoder corresponding to the value of the unit's target character. By sharing the character comparison results, it is possible to fit a character match unit into one logic element. Compared to the highest-capacity design found in previous works [7], our approach more than doubles the maximum pattern capacity of a given reconfigurable logic device. Fig. 1 illustrates the differences between the designs.

In addition, our approach uses routing resources more efficiently. While the distributed character approach requires $8*n$ connections from the input character to the character match units, the shared decoder approach only requires n connections, where n is the number of character match units. Since n increases linearly as the number of patterns is increased and the character match distribution is in the critical path, our approach scales much better in terms of capacity *and* operating frequency.

Case Insensitivity. If directly translating a regular expression to an NFA, a case-insensitive comparison performs separate comparisons for each case. However, there is a more efficient method that accomplishes the comparison with a single character match unit by using the fact that the ASCII codes for each case differ by only one binary digit. Using the distributed comparator, this can be implemented by having two 1-values in the LUT for the high-order bits of the character. Using the shared decoder, the OR of the match signals for each case is taken as the match input.

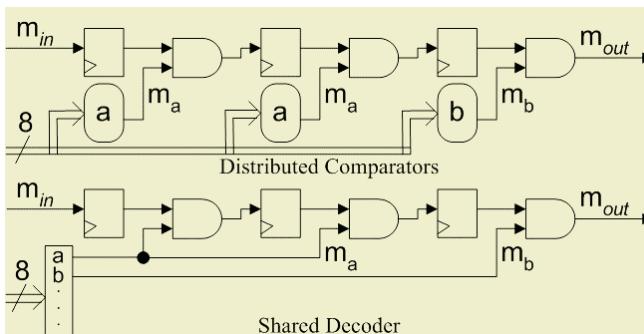


Fig. 1. Comparison of design approaches for the pattern “aab”

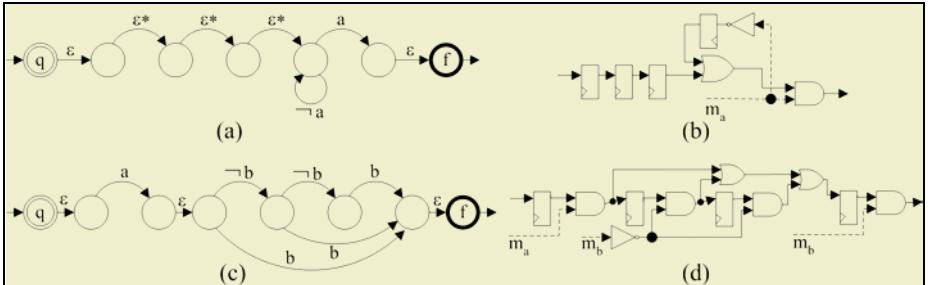


Fig. 2. Bounded-length wildcards. NFA (a) and circuit (b) for $(^* .3)(a)$. NFA (c) and circuit (d) for $(a)(^* .2)(b)$.

Bounded-Length Wildcards. Earlier work has shown how to implement wildcards for zero or more characters [6] and zero or one characters [7] using NFAs. Here we show how to implement wildcards whose length is specified by either a lower bound or an upper bound. In regular expressions, we use $(^* .n)$ to denote any character sequence with a minimum length of n characters and $(^* .n)$ for a sequence with a maximum length of n . In NFA diagrams we use $*$ to represent a character that is equivalent to $,$, but whose transition cannot be eliminated in the conversion to logic, and we use $(\neg c)$ to label a transition for any character other than the character c . Examples of NFAs using this notation and corresponding circuits are shown in Fig. 2

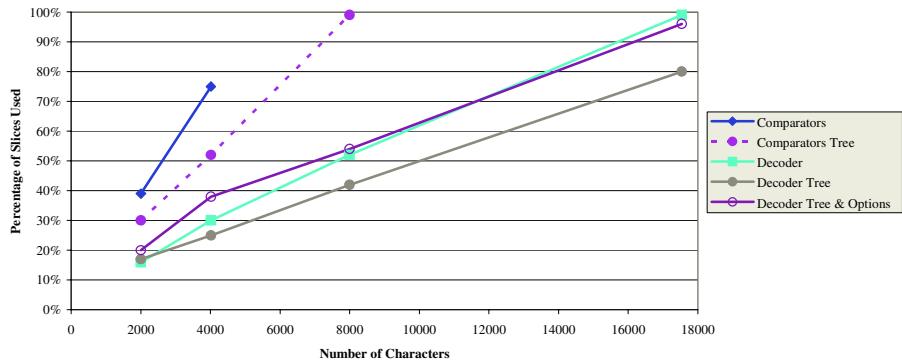
3 Performance

We implemented and tested five designs. The first was based on the distributed comparator approach described in [7]. The second extended the first by adding a prefix tree to eliminate circuitry for patterns with common prefixes. The third and fourth designs were based on our shared character decoder approach, with the latter using the prefix tree optimization. The fifth added support for all Snort content rule options. Each design was tested with sets of rules from Snort 2.0. The full set of default rules contained 17,537 characters. JHDL was used for synthesis and Xilinx Foundation was used for mapping, placement, and routing. The same configuration parameters were used for every case, and included a target clock rate of 100 MHz (the maximum supported by our FPGA platform). The designs were verified to produce correct output on a Xilinx Virtex-1000, a one-million-gate FPGA.

The speed and area results are presented in Table 1. The distinguishing property between the designs is their character capacity. This is clearly illustrated in Fig. 3, which plots logic element usage against the number of characters. All designs have approximately linear increases in logic usage. The important message is the relative slopes of the lines, which indicate the scalability of the designs. A prefix tree improves efficiency somewhat, but it is apparent that a shared character decoder is the key to providing major gains in capacity. As shown by the plot, support for all pattern matching options adds only a small constant overhead. To our knowledge, the fifth design is the first published system that performs all the pattern matching operations of Snort against all the rules. By processing one 8-bit character each cycle at 100 MHz, its throughput is 800 Mb/s, which is sufficient for 1 Gb/s networks.

Table 1. Speed and Area Comparison of Different Design Approaches

Number of Characters	Comparators		Comparators Tree		Decoder		Decoder Tree	
	Area	Freq	Area	Freq	Area	Freq	Area	Freq
2,001	39%	100.8	30%	83.3	16%	100.6	17%	101.7
4,012	75%	100.9	52%	73.9	30%	101.4	25%	102.1
7,996	-	-	99%	69.5	52%	100.2	42%	101.1
17,537	-	-	-	-	99%	100.1	80%	100.1

**Fig. 3.** FPGA slice usage for all designs

4 Conclusion

We have presented techniques that provide significant improvements in the efficiency and capacity of pattern matching circuits for reconfigurable logic. We have also extended the regular expression support of NFA-based designs. A co-processor for NIDS was implemented that is able to completely offload the task of pattern matching and compare the entire Snort rule set against packets at nearly 1 Gb/s. Our ongoing research indicates that speeds of 10 Gb/s and beyond are feasible with our approach.

References

1. J. Allen, et. al., "State of the Practice of Intrusion Detection Technologies," Technical Report CMU/SEI-99-TR-028, 1999.
2. L.G. Roberts, "Beyond Moore's Law: Internet Growth Trends," *IEEE Computer*, pp. 117-119, Jan 2000.
3. Martin Roesch and Chris Green, "Snort User's Manual". <http://www.snort.org>.
4. Mike Fisk and George Varghese, "Fast Content-Based Packet Handling for Intrusion Detection," Technical Report UCSD CS2001-0670, May 2001.
5. C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards Faster String Matching for Intrusion Detection," *DARPA Information Survivability Conference*, June 2001.
6. R. Sidhu and V.K. Prasanna, "Fast Regular Expression Matching using FPGAs," *Proceedings of IEEE FCCM 2001*, Apr. 2001.
7. R. Franklin, D. Carver, and B.L. Hutchings, "Assisting Network Intrusion Detection with Reconfigurable Hardware," *Proceedings of IEEE FCCM 2002*, pp. 111-120, Apr. 2002.
8. P. Bellows and B.L. Hutchings, "JHDL—An HDL for Reconfigurable Systems," *Proceedings of IEEE FCCM 1998*, pp. 175-184, Apr. 1998.

FPGAs for High Accuracy Clock Synchronization over Ethernet Networks

Roland Höller

Vienna University of Technology,
Institute of Computer Technology, ASIC Design,
Viktor-Kaplan Straße 2, 2700 Wiener Neustadt, Austria
roland.hoeller@tuwien.ac.at
<http://agcad.ict.tuwien.ac.at>

Abstract. This article describes the architecture and implementation of two systems on a programmable chip, which support high accuracy clock synchronization over Ethernet networks. The network interface node on one hand provides all necessary hardware support to be flexibly used in a broad range of applications. The switch add-on on the other hand accounts for the packet delay uncertainties of Ethernet switches and is crucial for high accuracy clock synchronization.

1 Introduction

The SynUTC (Synchronized Universal Time Coordinated) technology enables fault tolerant high accuracy distribution of GPS time and time synchronization of network nodes connected via standard Ethernet LANs [1–8]. By means of exchanging data packets in conjunction with hardware support at network nodes and network switches, an overall worst-case accuracy in the range of some 100 ns can be achieved, with negligible communication overhead. Applications can use the high-accuracy global time provided by SynUTC for event timestamping and event generation, both at hardware and software level [2, 3].

Furthermore the IEEE standard 1588 has been approved recently, which specifies a dedicated protocol, the Precision Time Protocol (PTP), for Ethernet networks to synchronize clocks in networked measurement and control systems [10]. Both the IEEE 1588 standard and the SynUTC technology are based upon inserting highly accurate time information into dedicated data packets at the media independent interface (MII) between the physical layer transceiver and the network controller upon packet transmission and reception, respectively. Implementing a clock synchronization service according to the ideas outlined above requires support on every node, which can be provided in a single SoPC (System on a Programmable Chip).

To alleviate the deterioration caused by network switches due to the undefined amount of time a packet stays on the switch before it is forwarded, a mechanism that measures this variable duration on-the-fly is necessary. This measurement is performed by another FPGA, which enhances standard Ethernet Switch functionality with a delay measurement mechanism to achieve highest precision and accuracy also in switched networks [8].

2 The Network Node FPGA

The network nodes support clock synchronization by making use of the network node FPGA. This FPGA provides the following functions [8]:

- A 96-bit adder-based local clock with a mechanism for linear continuous amortization.
- Two additional 64-bit adder-based clocks holding and automatically deteriorating the bounds on accuracy with respect to external reference time.
- An interface to a GPS timing receiver, made up of a 1-pps digital input and a RS232 serial interface.
- Application-level event generation and timestamping capabilities.
- A standardized interface for packet timestamping near the physical layer (IEEE 802.3 MII).
- An Ethernet MAC Controller.
- An interface to the CPU, which runs the synchronization algorithm and protocol stacks according to IEEE 1588.

For the first evaluation prototype the FPGA uses the PCI bus to communicate with the node's CPU, which run the protocol stacks and the synchronization algorithm. In this case the CPU is situated on a PC/104+ computer board.

However if a network node is intended to be used to connect to e.g. a sensor or an actuator in an industrial automation network, space and power requirements have to be taken into account. To be able to develop a system node that contains also the CPU on the chip, reconfigurable hardware is used as the centerpiece of the network interface card. It allows incremental realization of a corresponding system on chip without the need to develop a new printed circuit board by being able to skip the PCI interface and integrate a processor into the FPGA instead. This processor is an Altera NIOS 32-bit microcontroller core, which runs uCLinux as operating system (see Figure 1). In the first step the Ethernet MAC and the PCI interface are common of the shelf IP cores, that are connected via a 32-bit on-chip bus. The AHB bus (Advanced High Speed Bus) is used in this design.

3 The Switch Add-on FPGA

The switch add-on is used in conjunction with a common four port Ethernet switch. The switch add-on board provides eight 10/100-Mbit Ethernet physical layer interfaces of which four are connected to the Ethernet switch and four to the network nodes (see Figure 2). Whenever a dedicated clock synchronization packet passes through one of the four ports, which are connected to the network nodes, a 96-bit timestamp will be inserted on-the-fly into the packet. The packet's frame check sequence will be updated accordingly.

When after having been processed in the Ethernet switch, the clock synchronization packet is again traversing the switch add-on, now entering from one of the four ports connected to the Ethernet switch, the value of the timestamp in

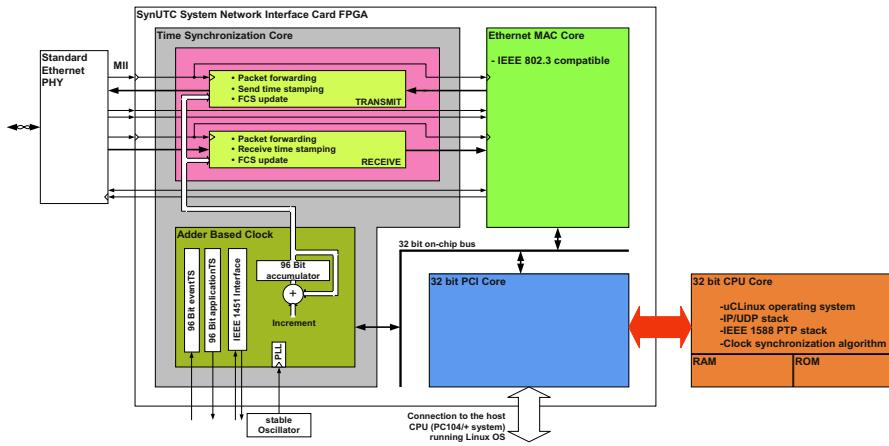


Fig. 1. Overview of the network interface card FPGA

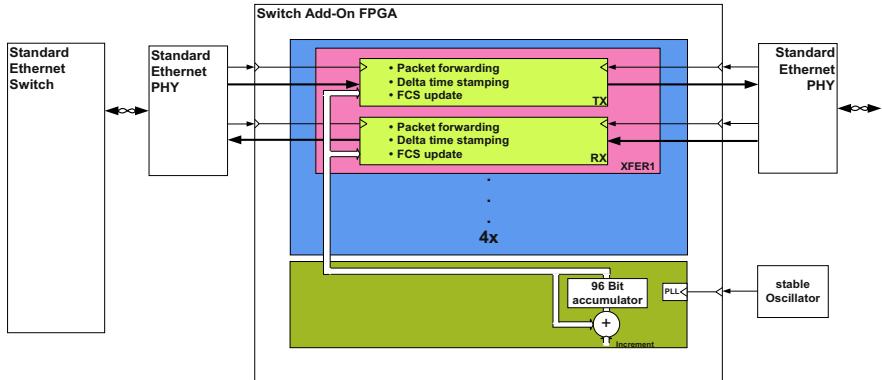


Fig. 2. Overview of the switch add-on FPGA

the packet will be subtracted from the current local time. This allows accurate measurement of the amount of time the packet needed to be forwarded by the switch. This delta time is now stored into the packet as it is sent to the destination node. The clock synchronization algorithm is now able to take this variable delay into account [7, 8].

4 Conclusion and Future Work

This article presented two FPGAs, that support the synchronization of clocks over Ethernet networks. The basic architecture of the designs has been presented and implementation results have been shown. The system containing the two integrated circuits will be presented at the first workshop on IEEE-1588 Standard

for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. Using reconfigurable hardware allows incremental realization of the system and future more complete protocol support (e.g. IEEE 1588 boundary clocks).

References

1. M. Horauer. *Hardware Support for Clock Synchronization in Distributed Systems*. Supplement of the 2001 International Conference on Dependable Systems and Networks, pp. A.10-A.13, Göteborg, July 2001.
2. M. Horauer, U. Schmid, K. Schossmaier. *NTI: A Network Time Interface Module for High-Accuracy Clock Synchronization*. Proceedings of the 6th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), Orlando Florida USA, April 1998.
3. U. Schmid. *Orthogonal Accuracy Clock Synchronization*. Chicago Journal of Theoretical Computer Science 2000(3), 2000, pp. 3-77. IEEE Computer, 1990, Vol. 23 (10), pp. 33-42.
4. U. Schmid, M. Horauer, N. Kerö. *How to Distribute GPS-Time over COTS-based LANs*. 31st Annual Precise Time and Time Interval (PTTI) Systems and Applications Meeting, Dana Point - California, December 7-9 1999.
5. Ulrich Schmid, Klaus Schossmaier. *Interval-based Clock Synchronization*. Journal of Real-Time Systems 12(2), March 1997, pp. 173-228.
6. M. Horauer, R. Höller. *Integration of highly accurate Clock Synchronization into Ethernet-based Distributed Systems*. SSGRR 2002w, 2002, ISBN 88-85280-62-5.
7. R. Hoeller, M. Horauer, G. Griedling, N. Keroe, U. Schmid, K. Schossmaier. *SynUTC - High Precision Time Synchronization over Ethernet Networks*. Proceedings of the 8th Workshop on Electronics for LHC Experiments, 9-13 September 2002, Colmar, France, pages 428-432, ISSN 0007-8328, ISBN 92-9083-202-9.
8. M. Horauer, K. Schossmaier, U. Schmid, R. Höller, N. Kerö. *PSynUTC- Evaluation of a High Precision Time Synchronization Prototype System for Ethernet LANs*. Precise Time and Time Interval (PTTI), December 2002, Washington, USA.
9. IEEE Standard 802.3, 2000 Edition. *Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications*. March 2000, New York, USA.
10. IEEE Standard 1588-2002. *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. November 2002, New York, USA.

Project of IPv6 Router with FPGA Hardware Accelerator

Jiří Novotný¹, Otto Fučík², and David Antoš³

¹ Institute of Computer Science,

Masaryk University Brno,

Botanická 68a, Brno 602 00, Czech Republic

novotny@ics.muni.cz

² Faculty of Information Technology,

Brno University of Technology,

Božetěchova 2, Brno 612 66, Czech Republic

fucik@fit.vutbr.cz

³ Faculty of Informatics,

Masaryk University Brno,

Botanická 68a, Brno 602 00, Czech Republic

xantos@fi.muni.cz

Abstract. This paper deals with a hardware accelerator as a part of the *Liberouter* project which is focused on design and implementation of a PC based IPv6 router. Major part of the Liberouter project is the development of a hardware accelerator – the PCI board called COMBO6 and its FPGA design which allows processing most of the network traffic in hardware.

Keywords. IPv6, router, Liberouter, Virtex II, FPGA.

1 Introduction

The paper describes a hardware accelerator based on FPGA that is designed to speed-up packet processing in a PC-based IPv6/IPv4 router. It is a part of the *Liberouter* project¹, which aims at developing a high-performance router with entirely open design.

Both the reliability and functionality of PC routers has proven to be fully comparable with commercial middle-class products [4]. Two main limitations keep from wider use of PC based routers. First, the PC routers have more difficult configuration than commercial routers. Second, the PC based routers have reached their theoretical throughput due to system resources saturation. Even the fast PCI bus (64 bit, 66 MHz) is not powerful enough.

The goal of the Liberouter project [5] is to develop an IPv6 PC based router, which solves both limitations discussed above. The configuration issues will be worked out by means of a uniform configuration environment based on XML. Performance will be improved using of a hardware accelerator that processes most of the network traffic in hardware.

¹ This research is supported by the FP5 project “6NET” (IST-2001-32603) and CES-NET project “IPv6 implementation in the CESNET2 network” (02/2003).

2 Router Architecture

The hardware accelerator is a PCI card containing network interfaces, FPGA, memories (SSRAMs and DRAM), and necessary logic. The card has several expansion connectors dedicated for interface cards and future extensions [6]. All physical interfaces are mounted on an expansion daughter board which allows to use many different interface standards, either metallic or optical.

Packet switching and filtering itself will be performed by the accelerator. Software can do the rest, providing operations like routing paths calculations, router configuration, and statistics computing. Such operations are not time critical; and PC's resources are suitable for them. Communication through the PCI bus will be limited to board configuration, routing tables and firewall rules initializing and changing, statistics collecting as well as exceptions handling.

Moreover, if the accelerator behaves as usual network adapters from the point of view of the Unix system, we may use ordinary system mechanisms for routing and packet filtering tables maintenance. This way we obtain configurability of a software router and speed of a hardware one. The router configuration is available by means of usual software like `ifconfig` and routing daemons. Development of the PC based router with an accelerator requires the design methodology known as *hardware/software codesign*. We have begun with the PC based router fully implemented in software, moving more and more operations to hardware. We start from input and output stages and continue with more complicated blocks.

3 Packet Processing in Hardware

Packet processing in FPGA is done by a chain of dedicated processors – we call them *nanoprocessors* due the simplicity of their instruction sets. Each nanoprocessor has its own specialized instruction set designed for the particular purpose. The programs of nanoprocessors are stored in both FPGA's on-chip memories as well as in the external SSRAM memories.

Complexity of nanoprocessors lies “between a Finite State Machine (FSM) and RISC processors.” The advantage of the this approach is the possibility to change functionality at runtime, as opposed to the case of FSMs. There is no need to rewrite the source code (e.g., in VHDL), synthesize it, place and route the design, and download the configuration data into FPGA. This differs from partial reconfiguration where all development steps must be done. On the opposite, partial reconfiguration can lead to smaller and more efficient design.

Let us now briefly describe the packet lifecycle in the proposed router. Complete information can be found in [1]. The packet processing is pipelined, the packet flows through the FPGA and memories. An incoming packet is received by the Input Packet Buffer and passed to the Header Field Extractor. The HFE pushes the body of the packet (including original headers) into the dynamic memory. Meanwhile, it parses the packet's headers and creates a *Unified-header* and a structure reflecting the actual arrangement of headers in the packet. The Unified-header is a fixed structure containing relevant information from packet

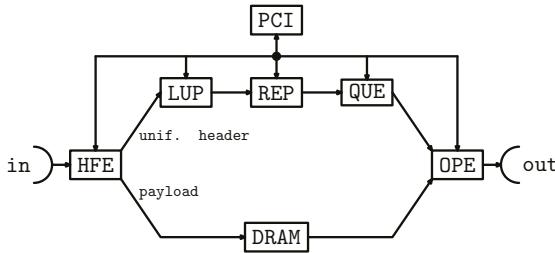


Fig. 1. Lifecycle of a packet traversing the router

headers, allowing to abstract the subsequent lookup operations from the actual header structure.

The Lookup Processor (LUP) uses CAM and SSRAM. LUP processes the Unified-headers by performing the lookup program and puts the result to Output Packet Editor. Using CAM is the fastest possibility when the application is small so that the lookup table fits into the memory. Unfortunately this is not the IPv6 case where we must match more than 440 bits of packet headers to decide what to do with a packet when it provides firewalling services.

Due to reasons discussed above we have developed a novel approach for the lookup machine. The word to be matched is stored in a sequence of registers. Instructions interpreted by the lookup machine are just conditional jumps. The lookup method is based on a (level compressed) search tree structure [1].

The Packet Replicator and Block of Output Queues (RQU) replicates the packet identification as well as the pointers to the editing programs into the dedicated queues. RQU computes the number of replicas of the packet that should be sent out. The Output Packet Editor (OPE) block modifies headers of packets. The OPE can also send a packet to the operating system, in the case when the packet destination is either the host computer itself, or the packet cannot be processed by hardware. This concept allows adding new features, step by step, during the PC based router development and use. The price paid for such approach include slower processing of software processed packets.

4 Software Support

Software drivers are developed first for NetBSD (FreeBSD) and ported to Linux. Driver operations include the FPGA chip configuration, accessing all memories mounted on the card as well as inside FPGA, and other hardware/software interface operations. The other part of router software should also be able to hide the card presence in the PC – the card should perform the same routing and filtering functionality as the operating system itself. The task is to develop a daemon which monitors the changes in both routing and filtering tables and, upon their update, it generates a nanoprogram for the LUP processor. To be able to make both routing and filtering by one searching operation, we have developed

a concept of *routing/firewalling table*. The routing/firewalling table contains filtering rules applied apriori to the routing table rows, it can be represented as a tree structure and converted to a LUP nanoprogram [1].

PC based routers are running under various operating systems with various configuration files. It causes problems to network administrators. To overcome this we are working on a unified configuration environment [3]. We have selected the XML language to store the router configuration. The user interface will be implementing Command Line Interface, web interface, and SNMP. From XML, the router configuration will be generated depending on the current operating system. This approach simplifies the use of the router and the user interface can be compatible with these seen in industry standard routers.

5 Conclusion

The PC based router design is a complex and long time task. It requires cooperation of experts from many areas. Currently, the entire team has more than 35 people from organizations in the Czech Republic including Masaryk University, Brno Technology University, Czech Technical University in Prague, and Camea ltd., as well as several consultants from other countries. The whole Liberouter project is organised by CESNET.

Currently, the accelerator prototype has already been developed, and produced. Processing blocks described are simulated and tested in hardware. The IPv6 PC based router is primarily dedicated but not limited for the use in academic networks, with focus on research in the internet protocols.

The project is fully open and all materials including source codes are available in the Internet [5]. Due to its flexible architecture, the hardware accelerator with low level software utilities can serve as a general purpose hardware platform for computation acceleration as well. There are many research project teams interested in using the proposed system. Possible application include programmable OEO (optic-electric-optic) switch, network monitoring tools, evolvable hardware research, reconfigurable computing, digital signal processing, and encryption.

References

1. David Antoš, Jan Kořenek, Kateřina Minaříková, and Vojtěch Řehák. Packet header matching in Combo6 IPv6 router. Technical Report 1/2003, CESNET, 2003.
2. Jiří Barnat, Tomáš Brázdil, Pavel Krčál, Vojtěch Řehák, and David Šafránek. Model Checking in IPv6 Hardware Router Design. Technical Report 8/2002, CESNET, 2002.
3. Petr Holub. XML Router Configuration Specifications and Architecture Document. Technical Report 7/2002, CESNET, 2002.
4. Ladislav Lhotka. Software tools for router performance testing. Technical Report 10/2001, CESNET, October 2001.
5. Liberouter. Liberouter Project WWW Page. <http://www.liberouter.org>, 2003.
6. Jiří Novotný, Otto Fučík, and Radomír Kokotek. Schematics and PCB of COMBO6 card. Technical Report 14/2002, CESNET, 2002.

A TCP/IP Based Multi-device Programming Circuit

David V. Schuehler, Harvey Ku, and John Lockwood

Applied Research Laboratory, Washington University
One Brookings Drive, Campus Box 1045
St. Louis, MO 63130-4899 USA
`{dvs1,hku,lockwood}@arl.wustl.edu`
<http://www.arl.wustl.edu/projects/fpx>

Abstract. This paper describes a lightweight Field Programmable Gate Array (FPGA) circuit design that supports the simultaneous programming of multiple devices at different locations throughout the Internet. This task is accomplished by a single TCP/IP socket connection. Packets are routed through a series of devices to be programmed. At each location, a hardware circuit extracts reconfiguration information from the TCP/IP byte stream and programs other devices at that location. A novel feature of the Multi-Device Programmer is that it does not use a microprocessor or even a soft-core processor. All of the TCP/IP protocol processing and packet forwarding operations are handled directly in FPGA logic and state machines. This system is robust against lost and reordered packets, and has been successfully demonstrated in the laboratory.

1 Introduction

As large numbers of reconfigurable hardware devices are deployed throughout the Internet, it has been observed that point-to-point configuration mechanisms cannot program multiple devices quickly. The concept of programming a chain of devices located throughout the Internet can be seen in Figure 1. By programming multiple devices using a single TCP/IP data flow, the task of reprogramming large numbers of devices at different locations becomes manageable.

The Multi-Device Programmer is a circuit design which provides a mechanism to extract device programming information from a TCP/IP connection. This information is then utilized to program a FPGA. Because the Multi-Device Programmer circuit does not inhibit the flow of data between the source programmer and the connection end point, multiple devices can be chained together and placed at multiple locations throughout the Internet. Identical copies of the device configuration data is extracted at each location. The development of the Multi-Device FPGA Programmer leverages existing research performed at the Washington University Applied Research Laboratory. More specifically, the Washington University Gigabit Switch (WUGS) and the Field-Programmable Port Extender (FPX) [2] plug-in card are used as the hardware platform on

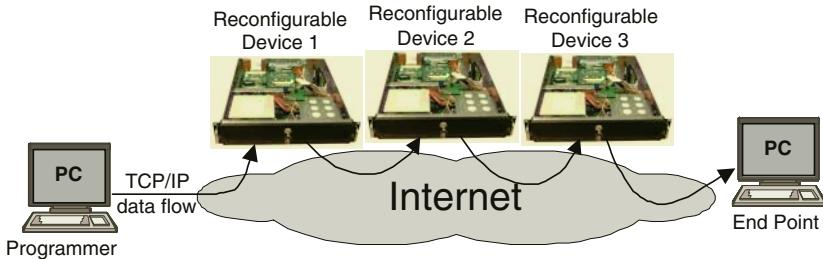


Fig. 1. Multi-Device Programmer

which the Multi-Device FPGA Programmer was developed and tested. The TCP-Splitter [3] and the Layered Protocol Wrapper [1] circuit components provide Internet Protocol packet processing functions. These circuit components provide a logic framework within which the Multi-Device Programmer resides.

2 Design

The Multi-Device Programmer is a circuit implemented in FPGA logic which extracts programming information from the network and programs a targeted FPGA or FPGAs. Specifically, it passes configuration and control data to other reconfigurable hardware devices in a system using information extracted from a TCP/IP byte stream.

In order to program a chain of devices, two software programs are run at the network endpoints. The source programmer application reads a formatted bitfile from disk and sends that data through a standard TCP/IP socket connection. A second program acts as a data sink and terminates the TCP/IP connection. The various components can be seen in Figure 1.

The source programmer can be executed on any workstation connected to a network. A TCP/IP socket connection is established to the connection endpoint. By design, this TCP/IP network connection follows a path through the network which contains the device(s) to be reprogrammed. This can be accomplished by either intelligently choosing the machines for the programmer application and the connection endpoint, or by a network administrator configuring static routes to ensure that the TCP/IP connection between the source programmer and the connection endpoint routes through the device(s) to be updated.

A FPGA on a FPX card is programmed by sending special control cells to the card. Information in these control cells are processed by logic on the card and used to program the target FPGA. The Multi-Device programmer circuit, containing the Protocol Wrappers, the TCP-Splitter, and the Multi-Device Programmer, was initially developed to execute on a FPX device. This circuit extracts device programming information from the TCP data stream and programs a second FPX module. By dropping selected packets, the TCP data

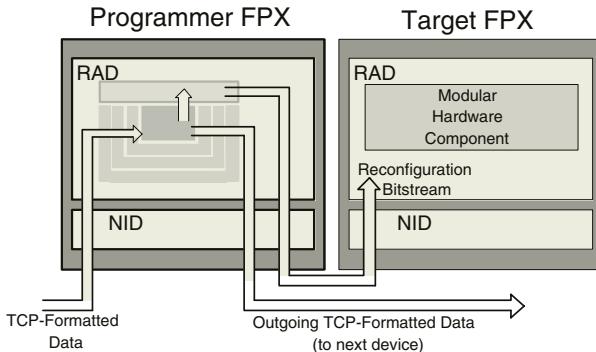


Fig. 2. Target Device Layout

stream can be reconstructed without requiring large reassembly buffers. This ensures that the system is robust against missing or reordered packets.

The layout of the target device, which consists of two components, can be seen in Figure 2. The Programmer FPX contains the Multi-Device Programmer which receives byte streams from the TCP-Splitter and extracts the appropriate device programming information. This FPGA reconfiguration data is then transmitted to one or more Target FPX devices. Upon receiving this reconfiguration data, the Target FPX device is reprogrammed.

The process to program a single device is identical to the process for programming 10, 100, 1000 or any number of devices. The programming application is executed on a node connected to the network and establishes a TCP/IP socket connection to the connection endpoint. This TCP/IP network connection is routed through each of the devices to be programmed. Once the connection is established, the programming application reads configuration information from a disk file and sends the configuration information down the network connection. For FPGAs that support partial reconfiguration, it is also possible to incorporate the Programmer circuit within a region of the target FPGA.

3 Results

The Multi-Device Programmer circuit has been successfully simulated, synthesized and tested in various configurations. The circuit has a post place and route operational frequency of 74 MHz on a Xilinx Virtex XVC2000E-6. All of the circuit components comprising the Multi-Device Programmer, including the TCP-Splitter and IP protocol wrappers, consume 22% of the BLOCKRAMs and 28% of the SLICEs on the target FPGA.

The TCP-Splitter and Multi-Device Programmer are designed to provide a peak theoretical throughput of 2.4 Gigabits/second using a 32-bit interface clocked at 75 MHz. The VirtexE FPGA on the FPX, however, has a reconfig-

uration throughput limited by the 8-bit interface of the SelectMAP interface. This limits the maximum possible reconfiguration to 400 Mbits/second.

The Multi-Device Programmer has successfully reprogrammed three target FPX devices simultaneously in as little as 1.102 seconds. The test configuration consists of a source programmer running on a Windows XP platform, a sink application running on a Linux machine, and a WUGS with three target FPX devices in a stacked configuration. Average programming times for this configuration are on the order of 1.3 seconds. A 2.2MByte configuration file containing command, control, and routing information was used in these tests. This file included a 1.2MByte bitfile. The configuration file was processed at an average bit rate of 16Mbits per second.

4 Conclusion

The Multi-Device Programmer provides an efficient mechanism for delivering FPGA reconfiguration information to a large number of devices over the Internet. This design feature provides a flexible programming solution which can integrate with other FPGA reconfigurations techniques. Partial FPGA reprogramming techniques such as incremental FPGA programming and FPGA plugin modules could easily be integrated with the Multi-Device Programmer.

The two main contributions of this research are (1) the successful demonstration of a FPGA application containing TCP/IP protocol processing in hardware (not a microprocessor or soft core), and (2) the demonstration of an application that can reliably program multiple devices utilizing a single TCP/IP connection. The design of the Multi-Device Programmer could easily be modified to send reprogramming information to any network attached device.

Acknowledgments

The authors of this paper would like to thank Florain Braun and James Moscola for the Layered Protocol Wrappers. We would also like to thank Todd Sproul for his work on NCHARGE and David Lim for his work on the NID.

References

1. F. Braun, J. Lockwood, and M. Waldvogel. Reconfigurable Router Modules Using Network Protocol Wrappers. In *Proceedings of Field-Programmable Logic and Applications*, pages 254–263, Belfast, Northern Ireland, Aug. 2001.
2. J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, Feb. 2001.
3. D. V. Schuehler and J. Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. In *Proceedings of Symposium on High Performance Interconnects (HotI'02)*, pages 127–131, Stanford, CA, USA, Aug. 2002.

Design Flow for Efficient FPGA Reconfiguration

Richard H. Turner and Roger F. Woods

Programmable Systems Laboratory

Queen's University Belfast, Ashby Building, Belfast, BT9 5AH, N. Ireland
rht@salsabelfast.co.uk, r.woods@qub.ac.uk

Abstract. In Run Time Reconfiguration (RTR) systems, the amount of reconfiguration is considerable when compared to the circuit changes implemented. This is because reconfiguration is not considered as part of the design flow. This paper presents a method for reconfigurable circuit design by modeling the underlying FPGA reconfigurable circuitry and taking it into consideration in the system design. This is demonstrated for an image processing example on the Xilinx Virtex FPGA.

1 Introduction

Typically, the amount of configuration data needed bears little relation to the changes being made. This has been exacerbated in the most recent FPGA technologies where reconfiguration comprises fixed size frames which represent the smallest unit of configuration. In many cases, a large part of the configuration data is not required and there are no methods for increasing the density of the useful information in these blocks. Previous studies [1] have indicated that routing accounts for a large portion of this data when compared to that required by the logic.

This paper argues that this high level of reconfiguration results because the designer has no control of the circuit implementation. Previous work has shown that mapping reconfiguration using the reconfiguration mux (RC_MUX) to either the LUTs [1,2] or routing muxes [3] can produce large reductions in reconfiguration times. The main challenge is to develop an approach that allows reconfiguration to be considered as part of the design flow thereby allowing RTR control to be gained. The paper presents a unified approach and demonstrates it with an image processing example.

2 Addition of Redundancy and Control of Reconfiguration

To capture the concept of reconfiguration and treat it as part of the design flow, the well-established method by Shirazi et al. [4] for viewing reconfiguration using a multiplexer, termed RC_MUX, is used. In his original paper, it was envisaged the RC_MUX is used to represent reconfiguration. However, **Fig. 1** highlights a number of multiplexers that can be used to directly implement this mux and achieve a reduction in reconfiguration time. Courtney [3] demonstrated use of the routing multiplexer to reduce reconfiguration and Turner [1] embedded the RC_MUX within the LUT.

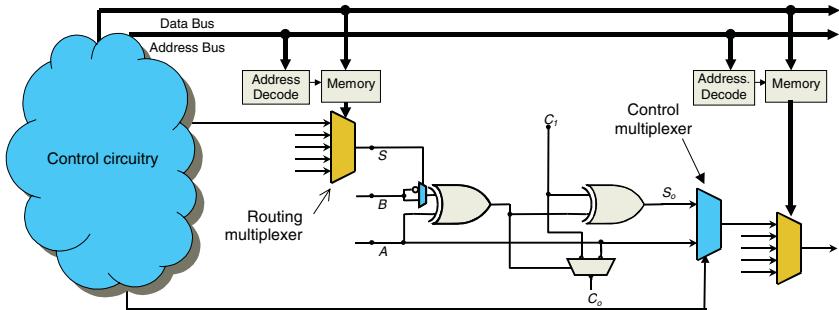


Fig. 1. Alternative view of the FPGA for RTR

A design that shares image filter functions, as illustrated in **Table 1**, originally developed by Shirazi et al. [4] is used to demonstrate the approach. A different implementation to Shirazi et al. [4] is shown in **Fig. 2** which has an extra addition that is not required in the adder branch of circuit **Fig. 2(b)**. This reduces the complexity and therefore size of the resulting circuit when the parts are combined. As the plan is to share the hardware, the circuit architecture has been developed as given by **Fig. 3**. Each of the multiplexers, labeled *A*, *B*, *C*, *D* and *E*, represent reconfiguration. Typically this would not be implemented but in our approach, these are efficiently mapped to the FPGA with the aim of minimizing the number of frames to be changed.

Table 1. Window functions for image processing

$\begin{vmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{vmatrix}$	$\begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}$	$\begin{vmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{vmatrix}$
(a) Gaussian filter	(b) Vertical edge detector	(c) Horizontal edge detector

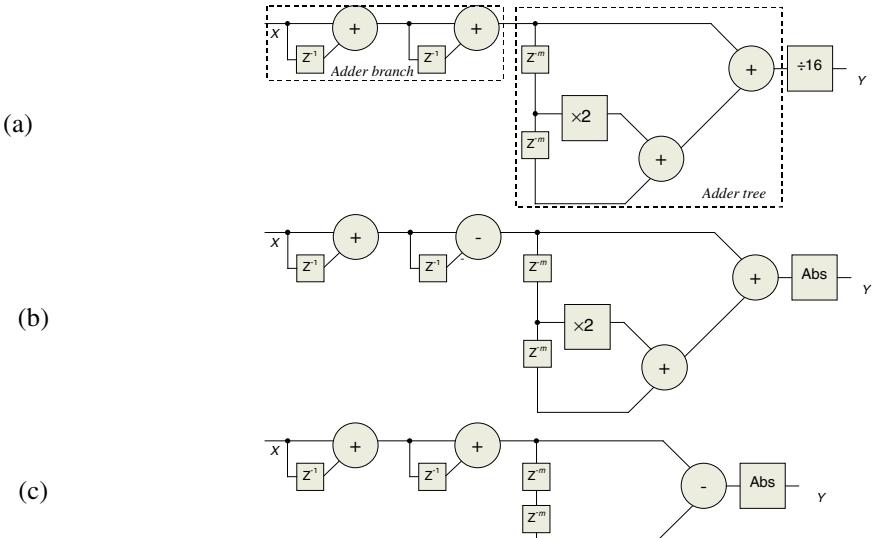


Fig. 2. (a) Gaussian filter, (b) Vertical edge detector and (c) Horizontal edge detector

Multiplexers *A* and *B* in **Fig. 3** will be mapped first and the routing multiplexers which share the same frames, will then be determined. **Table 2** has been drawn up to demonstrate this, giving the multiplexer and the frame affected (The table only includes state transitions that affect one frame and does not include the *Off* state). The proposed mapping is shown in **Fig. 4**. As *A* and *B* are assigned to routing multiplexers *Bx*, *By* of slice 0, this results in only frames 35 and 38 being modified.

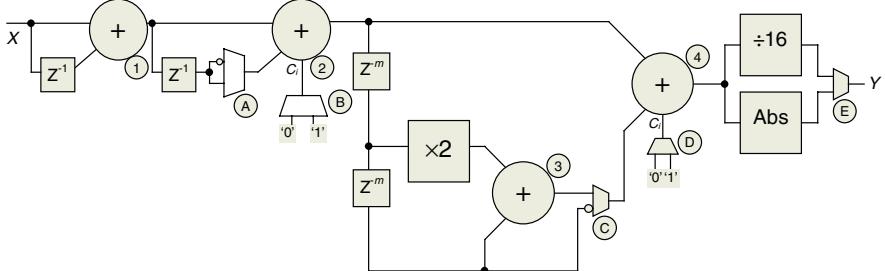


Fig. 3. Revised Combined circuit

Table 2. Frames which will be affected for possible routing multiplexer

Slice 1	Frame	Output	Frame		Frame
I/P Mux F1	38,39,40	MUX 0 (Out-0)	45	Slice 0 Bx	38
I/P Mux F2	26,27,29	MUX 1 (Out-1)	38	Inv MUX	
I/P Mux F3	14,15,16	MUX 2 (Out-2)	33,34	Slice 0 By	35
I/P Mux F4	2,3,5	MUX 3 (Out-3)	26	Inv MUX	
I/P Mux G1	38,39,40	MUX 4 (Out-4)	21	Slice 1 Bx	11
I/P Mux G2	26,27,28	MUX 5 (Out-5)	13, 14	Inv MUX	
I/P Mux G3	13,14,15	MUX 6 (Out-6)	9	Slice 1 By	13
I/P Mux G4	2,3,4	MUX 7 (Out-7)	2	Inv MUX	

This now leaves multiplexer *C*. From **Table 2**, it can be seen that the routing multiplexers, *F1*, *G1* and *Out-1*, share the same frames as *Bx*, *By* of slice 0. For this case, output routing multiplexer 1 (MUX 1 (Out-1)) was chosen. A second output multiplexer is required as each slice performs two bits of the adder, giving two outputs that need to be controlled. Multiplexer *C* is required to switch between an inverted version of one of the inputs to adder 3 and the output of adder 3. Looking at **Fig. 4**, it can be seen that a signal can be routed from *BY* through the inverter to *YB*. Therefore, MUX 1 (Out-1) is used to select between the FU outputs *Y* and *YB* (of slice 1). The other output routing multiplexer, MUX 1 (Out-2) has to be changeable between *X* and *XB* (of slice 1).

The choice of using output routing multiplexer 2 can be seen from the Reconfiguration State Graph (RSG) in **Fig 5** which gives the necessary state transitions to change multiplexer settings. It can be seen to move between states *X* and *XB* (of slice 1) that frames 33 and 35 will be needed, where frame 38 is also used for the inverters. In **Fig. 6**, it can be seen that to change the state of output routing multiplexer 1 from *Y* to *YB* (of slice 1), requires frames 38, 42 and 47 to be changed where frame 35 is also used for one of the inverters.

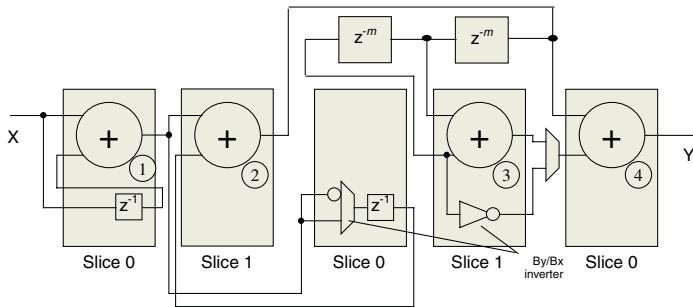


Fig. 4. Proposed mapping of the adders and multiplexers

The end result is that the RC_MUX has been mapped into routing multiplexers, taking account of the frames and minimising the number of frames that require to be changed. This can be extended to the input routing multiplexer. Alternatively, the RC_MUX can be treated as another logic component and mapped to the LUTs as shown in [2] therefore allowing the user to determine whether to implement the RC_MUX in logic or using reconfiguration prior to the place and route stage.

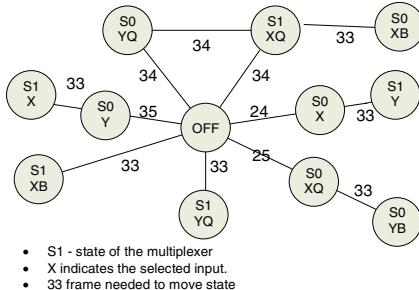


Fig. 5. RSG of o/p routing multiplexer 2

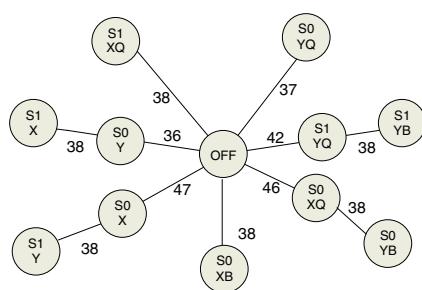


Fig. 6. RSG of o/p routing multiplexer 1

3 Conclusions

It has been shown that the process of mapping reconfiguration into the FPGA can be combined with the logic synthesis flow allowing optimisation, transformation, placement and routing to be done together and therefore minimizing reconfiguration.

References

- [1] R. H. Turner, "Functionally Diverse Programmable Logic Implementations of Digital Signal Processing Algorithms", PhD Diss., Queen's University Belfast, 2002.
- [2] C. N. Ang, R. H. Turner, T. Courtney and R. Woods, "Virtex FPGA Implementation of a Polyphase Filter for Sample Rate Conversion", 34th Asilomar Conf. on Signals, Syst. and Comp., USA, Oct. 2000, IEEE Comp. Soc., pp. 365-369.
- [3] T. Courtney, R. H. Turner, and R. Woods, "An Investigation of Reconfigurable Multipliers for use in Adaptive Signal Processing", IEEE Symp. on FCCM, Napa, USA, May 2000, pp. 341-343.
- [4] N. Shirazi, W. Luk, P. Cheung, "Automating Production of Run-Time Reconfigurable Designs", Proc. IEEE Symp. on FCCM, USA, 1998, pp. 147-156.

High-Level Design Tools for FPGA-Based Combinatorial Accelerators

Valery Sklyarov, Ioulia Skliarova, Pedro Almeida, and Manuel Almeida

University of Aveiro, Department of Electronics and Telecommunications, IEETA,
3810-193 Aveiro, Portugal
`{skl, ioulia}@det.ua.pt`

Abstract. Analysis of different combinatorial search algorithms has shown that they have a set of distinctive features in common. The paper suggests a number of reusable blocks that support these features and provide high-level design of combinatorial accelerators.

1 Introduction

There are many practical problems that can be formulated over such mathematical models as graphs, discrete matrices, sets, Boolean equations, etc. The majority of these models are mutually interchangeable, i.e. any of them can be selected for similar purposes. Many practical problems can be solved by applying various combinatorial search algorithms over a chosen mathematical model. Such algorithms have two distinctive features. Firstly, they require a huge number of different variants to be considered. Secondly, these variants are frequently ordered and examined with the aid of a decision tree that provides an efficient way for handling intermediate solutions. Examples of combinatorial problems that can be solved with the aid of the algorithms mentioned above are Boolean satisfiability; covering of sets and Boolean matrices; graph coloring, mapping and partitioning, etc.

This paper suggests a number of reusable blocks that can be employed for constructing application-specific hardware accelerators (co-processors) that permit combinatorial problems formulated over Boolean and ternary matrices to be solved.

2 Basic Combinatorial Search Algorithm

The considered combinatorial search algorithm is based on such primary operations that involve the application of so-called reduction and selection rules [1]. The *reduction rules* enable an initial matrix and intermediate matrices (that are constructed on each iteration through the algorithm) to be simplified. The *selection rules* allow the problem to be decomposed into several sub-problems that are examined sequentially to determine that either a solution will be found, or that no solution exists. The use of these rules for a particular example of the covering problem was considered in [2].

Search algorithms (such as [2]) for solving different combinatorial problems have similar characteristics. Their distinctive feature is the execution of problem-specific operations, traversing a decision tree starting from its root by involving such procedures as forward search and backtracking (if the forward search fails). Any branch point can be considered to be extracting a sub-tree with a local root. Thus a recursive algorithm, such as [2] can be used very efficiently. Analysis of the basic operations for general search algorithms has shown that it is reasonable to construct a set of functional blocks that enable the design process for combinatorial accelerators (co-processors) to be simplified. These blocks permit the design process to be realized at a high level of abstraction without losing sight of the details of a particular problem, and without increasing the hardware resources required or reducing performance.

The search algorithm contains branch points that can specify two or more alternative branches. The proposed computational model permits a graph to be traversed using the following strategy. All alternative branches of the search tree that might lead to the solution have to be examined. If we can prove that a selected branch does not permit a solution to be found, control is returned to the nearest branch point. All data that are needed to restore the state of any branch point on a path leading to the current step are kept in a stack. In order to store/restore data at branch points, push/pop operations are executed. Operations over vectors (rows and columns of matrices) are executed in a special functional unit that takes into account the uniqueness of combinatorial computations. Analysis of different combinatorial search algorithms has shown that the following four types of application-specific blocks are required: storage for a matrix, stack memory, a functional unit for operations over vectors, and a control circuit that supports recursion.

3 Specification of Reusable Functional Blocks

Matrix storage is the block that affects the timing characteristics of the algorithms most significantly. The proposed architecture includes memory based on RAM, mask registers that permit some rows and columns to be excluded to select a sub-matrix, and the circuits that generate RAM addresses. The latter allow the rows/columns that are not required at the current step to be skipped.

A parameterizable stack memory (that can be used for any type of intermediate data that are needed for backtracking) was constructed as a library component in VHDL and Handel-C. Elementary computations over vectors have been described using bit-manipulation operations and a reprogrammable FSM (examples can be found in [3]).

A control unit is modeled by a hierarchical FSM (HFSM) [4]. An example of a HFSM considered in [4] for executing sorting algorithms demonstrates all the steps that are required for HFSM synthesis. A number of ISE 5.2 projects for FPGAs that implement recursive hierarchical algorithms are available in the tutorial section of [5]. The same method was used for describing combinatorial search algorithms.

All the blocks considered have been designed in Handel-C and in VHDL so that their dimensions can be parameterized and they can be reused for different kinds of combinatorial accelerators (co-processors).

4 Implementation of Combinatorial Accelerators

Three combinatorial accelerators that will find a minimal row cover of a Boolean matrix using the exact algorithm [1] have been constructed based on the Handel-C and VHDL. A combinatorial accelerator that solves the Boolean satisfiability problem has been designed on the basis of VHDL. They are intended to demonstrate how the proposed library can be used. The designed macros permit storage to be allocated either in an external RAM or in FPGA embedded memory blocks (block RAM). The first accelerator was implemented on the prototyping board RC100 of Celoxica [6]. It contains 2 banks of onboard static RAM (256K*36 bit each), FPGA XC2S200 and some other components. A number of auxiliary blocks allowing data input and output have also been designed. They enable us to communicate with the keyboard and mouse, to display matrices, intermediate and final results on a VGA monitor screen, to receive/send data from/to PC, etc. The available drivers of Celoxica [6] have been utilized. However these are just 10-15% of the originally designed circuits, which in addition allow the display and scrolling of matrices, highlighting (or selecting by color) of elements considered at any intermediate step, visualizing the contents of stacks, checking any currently executing operation, and many others.

For debugging purposes the first bank of the onboard RAM was used as a memory for the VGA monitor (this bank can be used as additional storage for matrices in the “release version”). The second bank stores the matrices. The basic operation of the covering algorithm [1] is counting the number of ones in the matrix rows. This operation is only performed at the beginning of the algorithm execution. The results for all the rows are kept in FPGA block RAM and they are corrected at each newly executed step. An auxiliary stack stores these data for each branch point. General-purpose registers permit the result to be accumulated, include a counter that enables us to count the number of ones, and some temporary registers.

The Handel-C project was translated to an EDIF file in the Celoxica DK1 design suite and the bitstream for FPGA was generated from the EDIF file with the aid of Xilinx ISE 5.2.

The second accelerator was implemented on the basis of the ADM-XPL PCI board [7] containing FPGA XC2VP7 of Virtex-II Pro family. Access to the FPGA from the PC is provided through AlphaData API functions [7] that are described in the C language. Communication from the FPGA to the PC is supported by a driver described in Handel-C. Many auxiliary C++ programs that provide support for experiments and establish a user friendly graphical interface have also been implemented. The FPGA clock frequency was set to 60 MHz. An initial matrix is loaded from the PC and the accelerator solves the problem and returns the results to the PC. Finally the results are displayed on the screen. The execution time for randomly generated matrices with dimensions up to 100x100 does not exceed 2 minutes.

The third accelerator was constructed on the basis of VHDL. Two VHDL-based circuits for solving the covering and satisfiability problems were tested in Xilinx FPGAs XC4010XL and XCV812E. The reusable blocks considered above were also implemented as VHDL library modules. They have been tested in FPGA XC2S300E (the prototyping board TE-XC2Se from Trenz Electronic [8]). Many examples of such blocks are available in [5].

The results of experiments have shown that the proposed technique makes it possible to shorten the design of combinatorial accelerators (co-processors). The proposed building blocks take into account specific features of combinatorial search algorithms and they have been optimized for the problems considered. This allows block-based high-level design to be provided, i.e. to concentrate the efforts of the designer on the algorithms being considered, and avoid (or at least minimize) the details of hardware implementation. This permits consideration of either the design flow on the basis of the system-level specification language or the widely used hardware description language. The technique was validated in a number of projects that were implemented for three types of FPGAs and prototyping boards [6-8]. A set of additional blocks that were described in Handel-C and VHDL [5] provide very effective visualization and debugging tools and they can be inserted into a “debug version” and removed from the “release version”. They are also very useful for experimental purposes.

5 Conclusion

It has been shown that many problems formulated over Boolean and ternary matrices can be solved with the aid of search algorithms that have a number of distinctive features. Such algorithms are recursive and they execute periodically operations for simplifying an initial matrix and making a decision for future steps. The paper suggests four primary building blocks for the high-level design of combinatorial accelerators, which provide storage and unique operations on matrices, support stacks, and implement recursive control algorithms. These blocks were described in Handel-C and in VHDL. To validate the method, three combinatorial accelerators were designed, implemented in FPGAs on the basis of stand alone and PCI boards, and tested. This work was supported by the grants FCT-PRAXIS XXI/BD/21353/99 and POSI/43140/CHS/2001.

References

1. Zakrevski, A.D.: Logical Synthesis of Cascade Networks. Moscow: Science (1981)
2. Sklyarov V., Skliarova I.: Architecture of Reconfigurable Processor for Implementing Search Algorithms over Discrete Matrices. In: Proceedings of ERSA'2003 (Engineering of Reconfigurable Systems and Algorithms) (Las Vegas, USA, 2003)
3. Sklyarov V.: Reconfigurable models of finite state machines and their implementation in FPGAs. Journal of Systems Architecture, 47 (2002) 1043-1064
4. Sklyarov, V.: Hierarchical Finite-State Machines and Their Use for Digital Control. IEEE Transactions on VLSI Systems, vol. 7, n. 2 (1999) 222-228
5. <http://webct.ua.pt>, "2 semester", the discipline “Computação Reconfigurável”, public domain is indicated by the letter “i” enclosed in a circle. Login and password for access to the protected section can also be provided (via e-mails: skl@ieeta.pt, ioulia@det.ua.pt)
6. HandelC, DK1, RC100. [Online]. Available: <http://www.celoxica.com>
7. Alpha Data. [Online]. Available: <http://www.alpha-data.com>
8. Spartan-II Development Platform. [Online]. Available: <http://www.trenz-electronic.de>

Using System Generator to Design a Reconfigurable Video Encryption System

Daniel Denning¹, Neil Harold², Malachy Devlin², James Irvine³

¹ Institute of System Level Integration, Alba Campus, Livingston, EH54 7EG, UK
daniel.denning@sli-institute.ac.uk

² Nallatech Ltd, Boolean House, One Napier Park, Cumbernauld, Glasgow, G68 0BH, UK
{n.harold, m.devlin}@nallatech.com

³ EEE Department, University of Strathclyde, 204 George St., Glasgow, G1 1XW, UK
j.m.irvine@strath.ac.uk

Abstract. In this paper, we discuss the use of System Generator to design a reconfigurable video encryption system. It includes the design of the AES (Advanced Encryption System) and Enigma encryption cores. As a result of using this design flow, we are able to efficiently implement our system and algorithms with a significant improvement on traditional design times, without compromise for performance.

1 Introduction

System Generator [1] from Xilinx is an extension of Simulink and provides a block-based high-level schematic tool that generates VHDL. The nature of System Generator makes it ideal for rapid development of data-path algorithms. One of the encryption algorithms used in this system is based on the Enigma machine, which is equivalent to an encrypter/decrypter typewriter whereby pressing a letter of the alphabet reveals a different letter of the alphabet. A much more recent product cipher is AES, which is based on the Rijndael algorithm developed by V.Rijmen and J.Daemen [2]. AES was chosen by NIST (National Institute of Standards and Technology) to replace the highly popular but less-efficient DES (Data Encryption Standard).

Xilinx's Xtreme DSP kit [3], developed in conjunction with Nallatech, contains a BenOne motherboard and a BenAdda module, which are part of the scalable DIME-II™ family.

In this paper we investigate the implementation issues of designing a reconfigurable video encryption system using System Generator. We map our system to a Virtex-II FPGA on a BenAdda module, housed on a BenOne motherboard. The system is fully reconfigurable in that we essentially have two identical designs but with different encryption cores, a modified Enigma algorithm and the AES algorithm. Video is transmitted over a wireless link and fed back into the same FPGA to be decrypted.

2 Modelling with System Generator

System Generator consists of a Simulink library called the Xilinx blockset that maps the Xilinx block elements defined in Simulink into architectures and entities, signals, ports, and attributes. It also produces command files for FPGA synthesis, HDL simulation and implementation tools. The tool keeps the Simulink hierarchy when converted into VHDL.

When designing in System Generator it is possible to access key features in the FPGA such as the high-speed multipliers, and it is also possible to incorporate user-defined VHDL blocks in to the model. For verification and testing the tool can automatically generate testbenches, where by the Simulink input stimuli to the input block can be recorded for the VHDL simulation. The outputs can then be compared with the recorded results from the Simulink simulation in the VHDL simulation.

3 Encryption Cores

The AES algorithm is a symmetrical block cipher. We have chosen the algorithm to use the 128 bit key length. This results in 10 rounds of encryption within the algorithm, with each round, except the last round, having 4 transformations as shown in Fig. 1. For this implementation we have chosen to design the traditional looped feedback for each incremental round without any pipelining. Each round has its own subkey, these are generated when needed and then stored locally in Block-RAM for the decryption core.

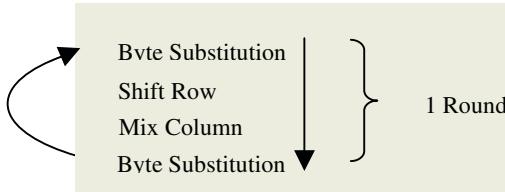


Fig. 1. AES transformations within one round.

Our modified Enigma algorithm is an 8-bit algorithm accepting values between 0 – 255 and uses 3 permanent rotors. For more information on the Enigma see [4].

4 System Implementation

A PAL video stream is captured by an ADC channel on the BenAdda, fed through the FPGA, encrypted, and presented at the DAC. This is then fed back through the second ADC to the FPGA to be decrypted. A high-level System Generator model can be seen in Fig. 2 on the following page. For reconfigurability the system is in two parts. The first part is an encryption system using the Enigma algorithm and the second is the same system except that the Enigma is substituted for the AES algorithm.

The AES algorithm uses the standard looped round design. This takes up the smallest amount of space on the FPGA but affects the throughput by the feedback latency. The S-boxes with in the subByte transformation are implemented using

single port ROM blocks and can be seen as a one-dimensional array. Multiplication with in the mixColumns transformation is a combination of shift, XOR, and multiplexor blocks. The shiftRows transformation uses only slice and concatenate blocks.

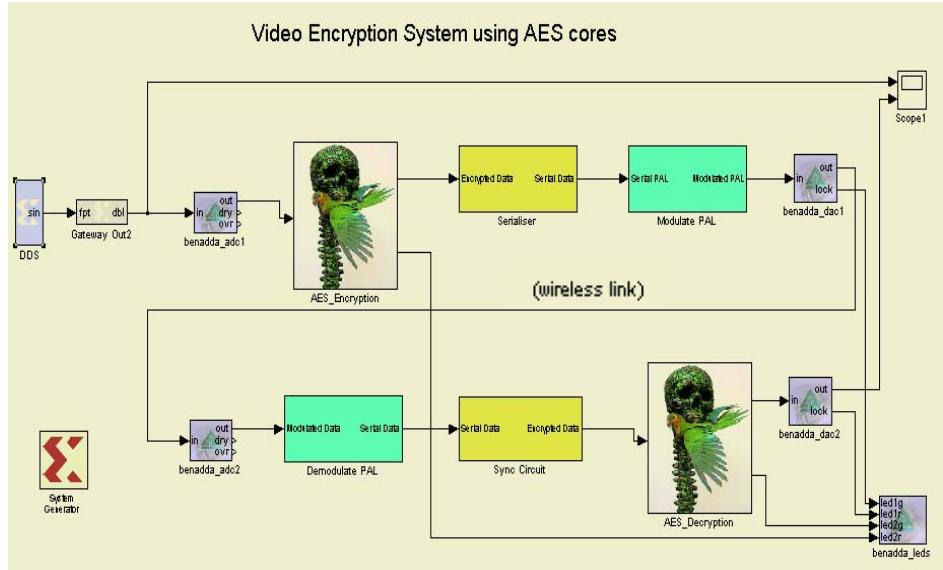


Fig. 2. High Level model of Video System using AES

The decryption core is the inverse of the encryption core [6]. The expansion of the key to produce the subkeys is preformed locally when needed and then stored in block-ROM to be accessed later by the decryption core. A buffering and de-buffering circuit is added before the encryption and after the decryption to buffer up and down 8-bits to 128-bits and vice-versa.

The data having been encrypted, start and stop bits are added prior to transmission. The data is then serialised and modulated at 100MHz (via DAC1) onto a laser diode module that transmits to a pin diode receiver connected to ADC2 on the BenAdda. The modulated data received on ADC2 is captured and passed to a synchronisation circuit, which identifies the boundary of each parallel word and removes the start and stop bits before applying the parallel data to the decryption block.

Following decryption, the reformed 8-bit PAL signal is fed out through DAC2 on the BenAdda to a suitable display. The entire system is bandwidth limited by the ADCs, which are clocked at 100MHz.

5 Results

Each system, encryption and decryption, fits onto one XC2V3000. The Enigma system takes up 4588 slices (32%) of the device while the AES system takes up 1719 slices (12%) of the device. The performance of both systems is limited by the wireless laser link and the ADCs. Without these limitations the performance would go as

high as the performance of the encryption cores. The Enigma encryption core has a throughput of 1.25 Gbit/sec and is fully pipelined. The design of the encryption and decryption cores took one engineer just over one week. The AES encryption core has a throughput of 1.3Gbit/sec and requires 466 (3%) slices of the device. However, due to the use of block-RAMs for s-boxes in the subByte transformation, the core needs 37 out of the 96 block-RAMs available. The design of each encryption core took one engineer just over two weeks. These same statistics can be applied to both of the decryption cores.

6 Conclusions

As can be seen from the results, especially the implementation of the AES core, it is possible with System Generator to design high-speed cores for FPGAs. The standard AES encryption core has also been fully pipelined. The System Generator model took over 2 hours to loop-unroll. A synthesisable core has been produced with a throughput of 16.4Gbit/sec, but has been targeted at a XC2V6000. Although the design times cannot be compared with an equivalent VHDL design time, it is the authors' view that these design times have increased impressively.

The System Generator tool has proved to be very intuitive for datapath and algorithm design. It does produce many VHDL files, for example the AES encryptor/decryptor core has over 2000 VHDL files in its hierarchy yet only takes up 3% of the slices in a XC2V3000. In terms of slices for whole 128-bit processing and on-chip key expansion, this core is one of the most compact AES encryptor/decryptor implementations published. The average simulation time of the system took around 10-15 minutes. When the AES fully pipelined encryption model was simulated the tool took over 4 hours to simulate 15 clock cycles.

Acknowledgements

On behalf of Nallatech Limited the authors would like to thank the Ministry of Defence (UK), and the Engineering and Physical Sciences Research Council. And lastly Dave Shand, Derek Stark, and Eric Lord for their help and support.

References

1. Xilinx Inc., System Generator Reference Guide, www.xilinx.com/ipcenter/dsp/ref_guide.pdf
2. AES, Federal Information Processing Standards Publication 197, Nov 26, 2001.
3. Xilinx Inc. Xtreme DSP Development Kit, www.xilinx.com/ipcenter/dsp/development_kit.htm.
4. Bletchley Park, Enigma, www.bletchleypark.org.uk.

MATLAB/Simulink Based Methodology for Rapid-FPGA-Prototyping

Miroslav Líćko^{1,2}, Jan Schier^{1,2}, Milan Tichý^{1,2}, and Markus Kühl³

¹ Institute of Information Theory and Automation, Department of Signal Processing,
Pod vodárenskou věží 4, Prague 8, Czech Republic

{licko, schier, tichy}@utia.cas.cz

² Center for Applied Cybernetics,
Department of Control Systems, Faculty of Electrical Engineering,
Czech Technical University,
Karlovo nám. 13, Prague 2, Czech Republic

³ FZI Forschungszentrum Informatik, Dept. of Electronic Systems and Microsystems,
Haid-und-Neu-Strasse 10-14, Karlsruhe, Germany

kuehl@fzi.de

Abstract. The paper is focused on rapid prototyping for FPGA using the high-level environment of MATLAB/Simulink. An approach using combination of the Xilinx System Generator (XSG) and Handel-C is reviewed. A design flow to minimize HDL coding is considered.

1 Introduction

For development of embedded applications, the methods of hardware-software co-design are often needed: on one hand, the algorithm design must be solved, on the other hand, proper balance between the flexibility (software implementation) and performance (hardware implementation) must be considered.

At the level of rapid algorithm design, MATLAB is often used for block specifications and for their inner analysis. This environment is characterized by its high-level scripting possibilities, strong support for matrix operations, object oriented approach, extensive graphing possibilities and rich set of application-specific toolboxes.

The application prototyping is typically performed using either some suitable high-level programming language, e.g. C or C++, or the MATLAB scripting language. On contrary, the hardware part of the target implementation has traditionally been designed in a Hardware Description Language (often, VHDL or Verilog). Hence, it was necessary to manually recode the specification. That is itself an error-prone process, leading to two versions of the same code (in different programming languages), which are difficult to keep up-to-date and corresponding to each other. There has been a number of attempts to solve this problem, either by designing languages that can be used for both hardware and software specification (the best-known example is probably the SystemC language) or by automating the HDL code generation by parsing the high-level specifications. In this paper we focus on the second option, namely on using Simulink for the

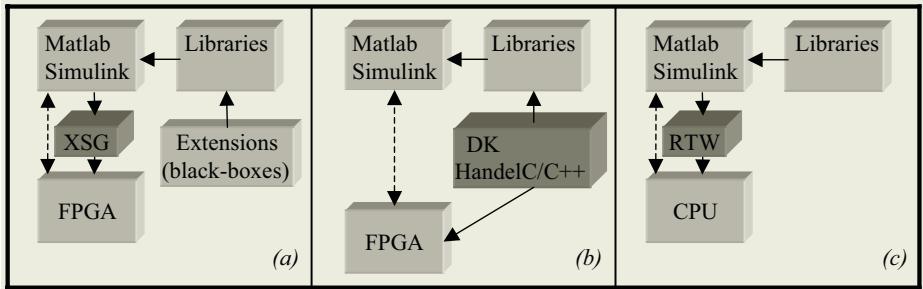


Fig. 1. Some possible high-level design flows for FPGA/CPU designs

initial specification of the system (Simulink is a MATLAB extension for visual programming). Using this tool, the system to be designed can be represented in a form of a block diagrams. In the paper, we revise some possible hardware design methodologies using Simulink for the system description and consider possible extensions. These extensions – aimed to provide support for a hardware-software co-design – are depicted in the diagrams given in Figure 1.

- Xilinx System Generator for DSP* (XSG) is an add-on tool for Simulink. The Core generator produces relationally placed macro components optimized for Xilinx FPGAs. With XSG, it is possible to generate VHDL source code out of Simulink block diagram. It allows to encapsulate user IPs too, using either VHDL or Verilog wrapper [3].
- Handel-C* is a HDL from Celoxica, based on the C language. It can be combined with C/C++ routines (representing parts of the design running in software). The compiler of the Handel-C language allows to generate directly the HDL description or to synthesise the EDIF code out of the C-like description. This design flow addresses HW/SW co-design problems as well [4].
- Real-Time Workshop* (RTW) represents generic methodology for parsing the Simulink block diagrams to high-level languages (primarily to the C-language). Parsers for different CPUs are provided. By adopting the methodology for Handel-C, System-C, or other appropriate language, it is possible to target the FPGA using the Simulink specification.

There are other tools that we mention only briefly:

- The *TargetLink* from dSpace (concept similar to XSG – Figure 1a) allows to use the Simulink block diagram to generate and optimize a C-code.
- The *Processor Expert* tool is a Delphi-like IDE for design of HW components. The concept used by ProcessorExpert is to encapsulate API into interactive IDE with graphical entities.
- *Forge* is a compiler from Xilinx for the FPGA designs written in Java language. It can be used with any Java-based IDE; it is possible to integrate it with the Matlab environment, which is based on the MathWorks proprietary JVM.

2 Designing XSG Extensions Using HW-Oriented HLL

As mentioned earlier, it is possible to develop extensions for the Xilinx System Generator (XSG) using the XSG black-boxes and HDL wrappers. To generalize and speed-up the prototyping and design process, we have combined this methodology with the tools available in the DK-environment from Celoxica (let us recall that it is a toolset around the Celoxica Handel-C language): the toolset was used to generate the VHDL code from the Handel-C description, and this code was then plugged into the System Generator.

The extended design flow is shown in Figure 2 – the right part of this figure is an extension of the flow presented in Figure 1 (a). We plan to investigate the possibilities of RPM support in terms of partially reconfigurable design as well [5].

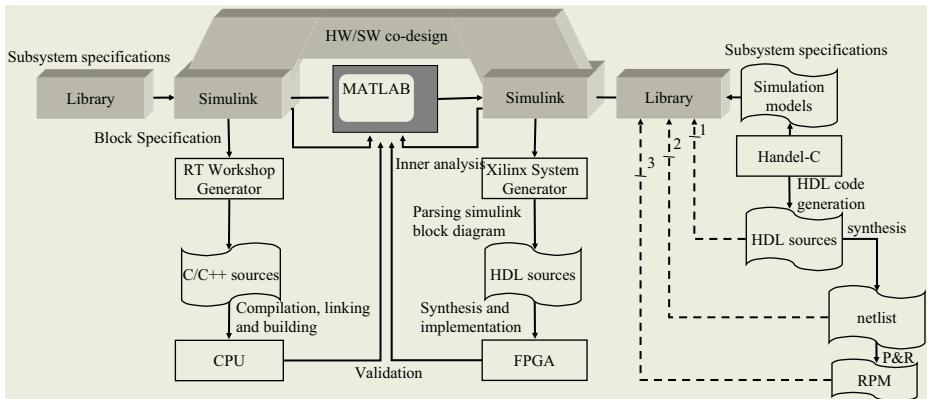


Fig. 2. Rapid-Prototyping design flow for the XSG Black-Boxes

The approach was tested on an example of a High-Speed Logarithmic Arithmetic (HSLA) unit, which has been developed in our group [1]. This unit represents an efficient way for implementation of floating-point arithmetic operations in FPGA. The above specified design flow allowed us to prepare generic specifications using the MATLAB/Simulink IDE and to use them for XSG modeling of applications requiring the floating-point data range precision.

3 HW/SW Co-design Options

The presented design flow could be used also for the HW/SW co-design. An illustration example: in the complex signal processing applications, the HSLA-cores can be used as a floating-point coprocessor. The load of the signal processor can be reduced by allocating parts of the running code into the FPGA circuit.

Using the combination of RTW and XSG, we can go farther: the XSG represents the system-level based implementation on one or more FPGAs, while the RTW is used to co-design the control program for the processor (see left part of the Figure 2). It will be possible to extend this approach so that we can use these high-level tools for the System on Chip (SoC) designs. Finally, we are investigating the possibilities to simulate and analyze the Run Time Reconfiguration schedule (RTR, see [5]). Seamless implementation on the target is the most important factor we are taking into account.

4 Conclusions

A design methodology for rapid prototyping of HW/SW-implemented DSP systems has been presented. The approach presented in the paper uses a combination of the MATLAB/Simulink environment with the Xilinx System Generator for DSP and the Handel-C.

The methodology has been used for preparing a bit-exact and cycle-exact Simulink library of the HSLA functions with corresponding IP cores for FPGA, in a form suitable for DSP engineers. We have started with a sequential implementation using MATLAB with MEX functions. Later, we prepared library for visual programming using Simulink. Using its combination with Handel-C we could better explore algorithmic structure (parallelism). The behavioral simulation in Simulink was used to speed-up our design process.

Acknowledgments

This work has been partially supported by the Ministry of Education of the Czech Republic under the Project LN00B096, by the IST programme under the RECONF2 project (IST-2001-34016) and by the DLR WTZ-project CZE01/19.

References

1. HSLA homepage, Dept. of Signal Processing, ÚTIA AV ČR, Prague, Czech Republic [available online: <http://www.utia.cas.cz/ZS/home.php?ids=hsla>]
2. Coleman, J.N., Chester, E., Softley, C.I. and Kadlec, J. 'Arithmetic of the European Logarithmic Microprocessor', IEEE Trans. Comput. Special Edition on Computer Arithmetic, Vol. 49, No. 7, pp. 702–715 and erratum vol. 49, no. 10, p. 1152 (July 2000)
3. Xilinx Inc., 'Xilinx System Generator for DSP' [available online: http://www.xilinx.com/xlnx/xil_prodat_product.jsp?title=system_generator]
4. Celoxica Ltd., 'Handel-C' [available online: <http://www.celoxica.com>]
5. RECONF Project homepage: IST programme 2002-34016 [available online: www.reconf.org]

DIGIMOD: A Tool to Implement FPGA-Based Digital IF and Baseband Modems

J. Marín-Roig¹, V.Torres¹, M.J.Canet¹, A.Pérez¹, T. Sansaloni¹, F. Cardells³,
F.Angarita¹, F.Vicedo⁴, V.Almenar², and J.Valls¹

¹ Dpto. Ingeniería Electrónica, Universidad Politécnica de Valencia, Gandia, Spain

² Dpto. Comunicaciones, Universidad Politécnica de Valencia, Gandia, Spain

³ Inkjet Commercial Division (ICD) R&D Lab, Hewlett-Packard, Barcelona, Spain

⁴Dpto. Física y Arquitectura Computadores, Universidad Miguel Hernández, Elche, Spain

Abstract. This paper presents a software tool to design intermediate frequency and baseband digital transceivers on FPGA. Main characteristic of this tool is that an *ad-hoc* interpolation or decimation filter chain composed by CIC, polyphase, pulse shaping, matched filters and a CORDIC-based or ROM-based mixer can be selected. The tool allows the software radio designer to develop downconverters and upconverters and, finally, automatically to generate the VHDL code to implement the system on Xilinx FPGAs.

1 Introduction

During the next years it is expected that new communication systems will provide higher mobility and wider bandwidth than present systems. For these reasons those firms that develop future communication systems will have to face up to next challenges: continuous evolving standards, fast deployment of new services, higher spectral efficiency, high capacity and mobility data transmissions, and a demand of a high reliability in those services offered.

In order to meet all theses objectives it will be necessary the use of new methods of design and development. In recent years a new technology called software radio (SWR) has come up. The main idea behind SWR is to move the digital part of any communication system towards the antenna. This means that most of the analogue components from transmitters and receivers have been substituted by digital signal processing under FPGA or DSP devices. Working in this way it is possible to change the system configuration without changing the hardware.

This paper presents a tool that speeds up and makes easy the implementation of wireless communications systems. With this tool one can design a digital transmitter/receiver in intermediate frequency or in base band. By means of its graphical interface the user can choose all the subsystems required, and once the transmitter/receiver is completed, this tool can evaluate the performance of those employed IP-cores. Finally, if the specifications are met, the tool generates the VHDL code for the whole system.

2 Description of DIGIMOD Tool

Graphical user interface of DIGIMOD makes easy and fast the design and implementation of a transceiver for digital communications on FPGA. The transmitter, also called upconverter, is composed of several stages: binary data source, symbol mapping, pulse shaping, interpolation filters and mixer. Meanwhile, the receiver (or downconverter) is composed by a chain of elements that performs the reverse operations: a mixer to bring the signal to base band, decimation filters to cancel the double frequency image and to reduce the sampling rate, a matched filter adapted to the pulse shape, a demapper and a symbol detector. So, the designer of a digital communication modem can use this tool:

- to select the type of modulation between BPSK, QPSK, and QAM, and the pulse shape parameters;
- to evaluate what kind of filters are needed in the interpolation or decimation stages, CIC [1] or polyphase filters can be used;
- to design the selected filters by using MATLAB;
- to carry out a floating point simulation where the system performance is evaluated through two kinds of results: the Error Vector Measurement (EVM) of the generated signal, and the bit error rate (BER);
- to evaluate the finite precision for each block by comparing the use of fixed point operators with the floating point design;
- to generate a VHDL code of the fixed point system, in which each block is an area optimized relatively placed macro (RPM)

2.1 Design Flow

There are several steps during the design process with DIGIMOD. The first step is the selection of those blocks needed in the design: source, modulation mapping, pulse shaping, interpolation filters, (and mixer).

Once the filter chain is specified, a simulation using floating point precision is performed to evaluate if all the specifications are fulfilled. The tool generates the frequency response of the filter chain, as well as of individual filters, and two quality measurements: BER and EVM. If the obtained results do not match with system specifications the user can aggregate or delete blocks, or adjust block parameters until simulation gives the correct results.

After tuning the floating point design, the user can begin with the fixed point evaluation in a similar way. Simulations are performed in each step in order to asses the user to choose the number of bits needed.

If finite precision simulation accomplishes the specifications of the application, the tool is ready to generate the VHDL code of the whole system for a target FPGA device. After DIGIMOD generates the VHDL code, it can be synthesized with Synplify and implemented with ISE Xilinx tool.

2.2 Implementation Technology

Pulse shaping can be designed using two implementation methods: polyphase filter or look-up table [2]. Look-up method makes use of the embedded block select RAM available in a FPGA device. This method allows a higher interpolation factor than polyphase filter at a lower cost and simplifies the interpolation filters that come later in the transmitter chain.

Polyphase filters can be used for pulse shaping, matched filtering and interpolation, they are implemented with bit-serial or digit-serial distributed arithmetic [3].

Mixers can be built using two methods. The first performs a CORDIC-based mixer [4] and the second a ROM-based one [5]. This last method uses compression techniques to reduce the size of the required memories

The VHDL code generated by DIGIMOD tool instantiates different cores that perform the required operations. All the cores have been described in VHDL by using relative placed attributes. Furthermore, they are area efficient with respect to same blocks generated with Xilinx Coregenerator system [6].

3 A Design Example

For a better understanding of DIGIMOD characteristics and performance, in this section we will present a design example of a digital IF QPSK modulator using DIGIMOD tool. In order to benchmark the performance of our tool we will compare it with the results given by Xilinx System Generator [6]. The QPSK modulator parameters are: 2 Mbps of bit rate, QPSK with a roll-off 0.3 root raised cosine pulse shape and a length of 8 symbols, an IF of 10.7 MHz, and a DAC with 8 bits and 40 MHz of sampling rate.

The interpolation chain used consists of: a polyphase structure for pulse shaping with an interpolation factor of 4, a halfband 6th order FIR interpolator, and a 3rd order CIC with an interpolator factor of 5.

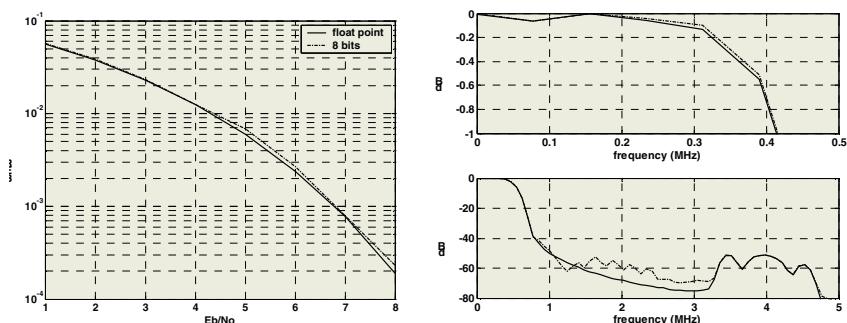


Fig. 1. Floating vs. fixed (dotted) point implementations: BER and signal spectrum

After some simulations fixed point parameters are: 8 bits for filter coefficient, both in pulse shaping and half band, and 8 bits for signal output for every block in the chain system. Figure 1 shows the comparison in BER between floating point design

and 8 bits fixed point design (with a loss of less than 0.1 dB), and the signal spectrum at CIC filter output in both the passband (above) and the stopband.

Table 1 shows the results of three implementations of this design in a Xilinx VirtexE-8 device. The first one has been performed with Xilinx System generator, the second and third ones use DIGIMOD tool, where pulse shaping is performed with a polyphase filter and with the look-up table method. These results show that DIGIMOD implementation is more efficient than system generator one. It is achieved an area saving of 45% with the polyphase filter option. If the look-up table method is used, 2 Block Select RAMs and 27 slices are required. Finally, it can also be made out that DIGIMOD requires less area to implement each block and, furthermore, does not require extra resources to connect those blocks employed, as System generator does.

Table 1. Implementation results

Resource (slices)	System Generator	DIGIMOD	
		Polyphase PS	Look-up PS
Pulse shaping (PS)	201	118	27 (+ 2 BSRAM)
Half band	90	30	
CIC	67	67	67
DDS+MIXER	281	189	189
QPSK modulator	1137	627	350 slices + 2 BSRAM

4 Conclusions

This paper presents a software tool that allows software radio designers to develop digital transceivers: from simulation to VHDL code generation for Xilinx FPGAs. As a design example an IF QPSK modulator has been implemented. Obtained results are compared with those given by Xilinx System Generator, it is shown that our tool leads to an area efficient implementation.

References

1. E. Hogenauer, "An Economical class of Digital Filters for Decimation and Interpolation", IEEE Transactions on Acoustic, Speech and Signal Processing, vol ASSP-29, n°2, April 1981.
2. José Marin-Roig, Javier Valls, Vicenç Almenar, "LUT-based Up-converters for FPGA", Communication Systems, Networks and Digital Signal Processing Conference, July 2002
3. Stanley A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review", IEEE ASSP Magazine, July 1989
4. F. Cardells, J. Valls, "Optimisation of direct digital frequency synthesizers based on CORDIC", Electronic Letters, Vol. 37, no. 21, pp. 1278-1280, October 2001
5. F. Cardells, J. Valls, "Area-Optimized Implementation of Quadrature Digital Direct Synthesizer on LUT-based FPGAs", IEEE Trans. on Circuits and Systems II, Vol. 50, no. 3, pp. 135-138, March 2003
6. www.xilinx.com.

FPGA Implementation of a Maze Routing Accelerator

John A. Nestor

Department of Electrical and Computer Engineering
Lafayette College
Easton, Pennsylvania 18042 USA
nestorj@lafayette.edu

Abstract. This paper describes the implementation of the L3 maze routing accelerator in an FPGA. L3 supports fast single-layer and multi-layer routing, preferential routing, and rip-up-and-reroute. A 16 X 16 single-layer and 4 X 4 multi-layer router that can handle 2-16 layers have been implemented in a low-end Xilinx XC2S300E FPGA. Larger arrays are currently under construction.

1 Introduction

The classic Lee Algorithm for *maze routing* [1] remains popular in electronic design automation because it is guaranteed to find a shortest-path connection if one exists. The algorithm represents the routing surface as a rectangular grid. During the *expansion* phase, it searches outward from the *source* node of a desired connection in breadth-first fashion while labeling each encountered node to indicate the shortest path back to the source. When the *target* node is found, the *backtrace* phase selects a path by following these labels while marking the path as an obstacle. The *cleanup* phase clears the remaining labels before additional connections are routed.

Although popular, the Lee Algorithm is computationally expensive. For single layer routing, expansion is $O(d^2)$ for a connection of distance d , and cleanup is $O(N^2)$ for an $N \times N$ grid. Multilayer routing is even more costly. This has motivated several proposals for hardware accelerators. *Direct grid* accelerators (e.g., [2]) map each node into an array of simple processing elements (PEs). This reduces the expansion time to $O(d)$ and cleanup time to $O(1)$ but requires N^2 processing elements for a single-layer. *Virtual grid* accelerators (e.g. [3]) map more than one node onto each PE. This requires fewer PEs, but each PE must be significantly more complex to handle the multiple mappings. Other acceleration approaches include raster pipelines [4], specialized processors [5], and accelerators intended for routing FPGAs, which have a specialized routing structure [6].

This paper describes a return to the direct grid approach called L3. L3 improves over previous direct grid approaches by providing (1) efficient support for multiple layers; (2) a significant reduction in PE logic; and (3) support for quick initialization and removal of obstacles and connections during rip up and reroute. A preliminary version of L3 was described in [7]; this paper describes a refined version and its implementation in an FPGA.

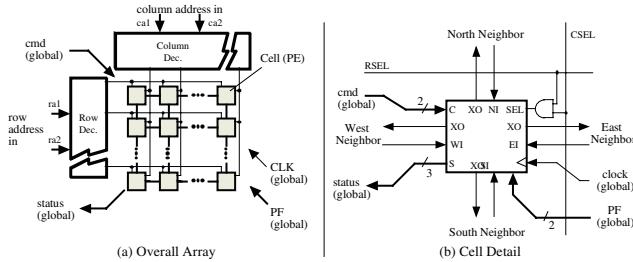


Fig. 1. L3S Organization. The attached control unit is not shown

2 The L3 Architecture

Figure 1 shows the general organization of L3S, the single-layer version of L3. Each *cell* (PE) is connected locally to neighboring cells in the grid using output XO. XO is true when a cell is labeled during expansion. It is connected to the WI, EI, NI, and SI inputs of the cell's west, east, north, and south neighbors, respectively. An attached control unit (not shown) communicates with the array of cells using a global command bus CMD, and a row and column decoder that allow the selection of either individual cells or rectangular regions of cells. Cells pass status information back to the control unit using a global tristate/wired-OR STATUS bus. Global input PF supports preferential routing and will be discussed later.

Each cell is a finite state machine with 6 states: E (empty), BL (blocked), XE (expanded east), XW (expanded west), XN (expanded north) and XS (expanded south). Table 1 describes the function of each cell as it responds to one of four commands: CLEAR, SET, EXPAND, and TRACE.

Table 1. Cell state sequencing in response to CMD. X* indicates any expanded state

CMD	SEL	EW * PF1	WI * PF1	NI * PF0	SI * PF0	PS	NS	S1	S0	
CLEAR	0	—	—	—	—	X*	E	1	1	
	1	—	—	—	—	Any	E	1	1	
SET	1	—	—	—	—	Any	XE	1	1	
TRACE	1	—	—	—	—	Any	BL	D1	D0	
EXPAND	—	1	—	—	—	E	XE	0	SEL'	
	—	0	1	—	—	E	XW	0	SEL'	
	—	0	0	1	—	E	XN	0	SEL'	
	—	0	0	0	1	E	XS	0	SEL'	
	—	—	—	—	—	X*	PS	1	SEL'	
All other conditions								PS	1	1

To perform maze routing, the control unit first selects all cells and broadcasts a CLEAR command to set all cells to the E state. It then selects the source node and uses the SET command to set the source cell to the XE state. It next applies the EXPAND command while selecting the location of the target cell. During each

successive clock cycle, a cell will enter an expanded state if one of its neighbors is in an expanded state and the corresponding preference input (PF1 for east/west, PF0 for north/south) is high. Expansion continues until either the target cell is reached (at which point status bit S0 is pulled low) or expansion has failed (in which case “watchdog” status bit S1 goes high).

If expansion is successful, the control unit starts the backtrace phase by selecting the target cell and broadcasting the TRACE command. In response, the target cell asserts its state code on the STATUS bus and enters the obstacle state (BL). The control unit uses the direction information encoded in the state code to determine the address of the next cell in the path and repeats this process until the source node is reached and the entire path is marked as an obstacle. To route another connection, the CLEAR command is applied with no cells selected to remove expansion labels (but not obstacles) and the process repeats. The entire process takes d clock cycles for expansion, d clock cycles for backtrace, and 1 clock cycle for cleanup.

L3M is the multi-layer version of the L3 architecture. It uses the same array structure as L3S but time-multiplexes cell hardware over multiple layers. Figure 2 shows the organization of a single L3M cell, which uses a shift register to store the states of each layer. The L3M cell processes states from bottom to top on each successive clock cycle. The state sequencer is similar to that of the L3S cell except that it has two additional states XU and XD to support vertical expansion. Expansion information from the layer “above” the current layer is taken from the next shift register stage. Expansion information from the level “below” the current layer is stored in a flip-flop at the end of the preceding cycle. Both calculations are suppressed by the /TOP signal when the top layer is reached. The preferential routing input PFV allows vertical expansion to be suspended; when zero the sequencer recirculates the current layer so that horizontal expansion can continue on successive clock cycles. The L3M array will find a connection in $O(L^*d)$ cycles for L layers.

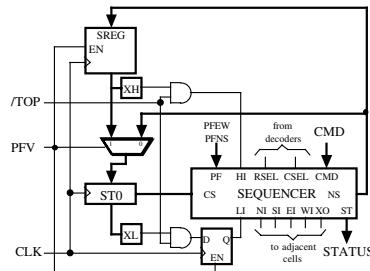


Fig. 2. The L3M Cell design. Layers are processed from bottom to top

3 Implementation Results

As a proof of concept, the modules of the L3S and L3M accelerators have been coded in Verilog HDL and synthesized using the Xilinx ISE 4.2i and Synopsys FPGA Express tools targeting a Xilinx XC2S300E FPGA [8] on a Memec Development Board [9]. The synthesized L3S cell requires 17 4-input lookup tables (LUTs), 3 D flip-flops (DFFs), and 3 tristate buffers (TBUFs). The L3M cell requires 32 LUTs (including 3 SRL16 shift registers), 3 DFFs, and 3 TBUFs.

Table 2. L3S and L3M implementation costs and predicted performance

Array Size	Serial LUTs/FFs	Control LUTs/FFs	Array LUTs/Ffs	Total LUTs/FFs	Clock (ns)
4 X 4	173 / 141	167 / 15	260 / 48	600 / 204	38ns
8 X 8	173 / 141	171 / 19	1083 / 192	1427 / 352	40ns
16 X 16	173 / 141	370 / 23	4067 / 768	4610 / 932	51ns
4 X 4 X 4	173 / 141	238 / 23	425 / 50	836 / 214	41ns

Table 2 shows the results of synthesizing the complete L3 router including cell array, decoders, control unit, and a serial interface to communicate with a host computer. All designs were tested and work properly, although the 16 X 16 single-layer design failed to meet the timing constraint.

The LUT requirements of the L3S and L3M cells can be used to predict the size of router that can be accommodated by a larger FPGA. For example, the Virtex-II Pro XC2VP125 device [8] contains 111,232 LUTs and could accommodate an 82 X 82 array of single-layer L3S cells or a 58 X 58 array of L3M cells.

4 Conclusion

This paper has described a new architecture for a direct-grid hardware routing accelerator and its implementation using FPGAs. The L3 architecture supports multiple layers, preferential routing, and iterative rip-up-and reroute. A working 16 X 16 single-layer router and 4 X 4 multi-layer have been demonstrated in a low-end FPGA. Future work will include implementation of larger routers, further design refinement to improve clock cycle time, and detailed performance comparisons of the hardware router to software implementations.

References

1. Lee, C. Y. "An Algorithm for Path Connections and its Applications," *IRE Transactions on Electronic Computers* vol. EC-10, no. 2, pp. 346-365, 1961
2. Breuer, M., and Shamsa, K. "A Hardware Router," *Journal of Digital Systems*, vol. IV, no. 4, pp. 393-408, 1981
3. Ventkateswaran, R., and Mazumder, P., "Coprocessor Design for Multilayer Surface-Mounted PCB Routing," *IEEE Trans. VLSI Systems*, vol. 1, no. 1, 1993
4. Rutenbar, R. and Mudge, T., "A Class of Cellular Architectures to Support Physical Design Automation", *IEEE Trans. CAD*, Vol. CAD-4, No. 4. October 1984
5. Won, Y., Sahni, S., and El-Ziq, Y., "A Hardware Accelerator for Maze Routing", *Proceedings Design Automation Conference*, June 1987
6. Huang, R., Wawrzynek, J., and DeHon, A., "Stochastic, Spatial Routing for Hypergraphs, Trees, and Meshes", *Proc. International Symposium on FPGAs*, February 2003
7. Nestor, J. "A New Look at Hardware Maze Routing", *Proceedings Great Lakes Symposium on VLSI*, March 2002
8. Xilinx, Inc., *Xilinx Databook*, 2003. Available online at <http://www.xilinx.com>
9. Memec, Inc., *Spartan-II Development Board User's Guide*, 2002

Model Checking Reconfigurable Processor Configurations for Safety Properties*

John Cochran, Deepak Kapur, and Darko Stefanović

University of New Mexico, Albuquerque NM 87131, USA,
cochran@cs.unm.edu

Abstract. Reconfigurable processors pose unique problems for program safety because of their use of computational approaches that are difficult to integrate into traditional program analyses. The combination of proof-carrying code for verification of standard processor machine code and model-checking for array configurations is explored. This approach is shown to be useful in verifying safety properties including the synchronization of memory accesses by the reconfigurable array and memory access bounds checking.

1 Introduction

Reconfigurable computing is a rapidly evolving technology that has great potential for improved processing efficiency for important computational tasks. This improvement, however, comes at the expense of increased risk of problems from faulty or malicious programming. We are exploring model checking combined with proof-carrying code as a method of ensuring safety of reconfigurable processor programs. We show that significant safety properties can be *efficiently* and *automatically* verified by model checking.

2 Approach

The novel approach explored here uses model checking to verify safety properties of the reconfigurable array, and proof-carrying code (PCC) [Nec98] to verify safety properties of the standard machine code. In addition, proof-carrying code provides a context for model checking the reconfigurable array by providing preconditions, postconditions, memory partitions between standard processor executions and array executions, and local safety properties. This is achieved by extending PCC's context mechanism for function calls to deal with indeterminacy from reconfigurable array executions.

The extensions include new instructions for accessing array registers, loading configurations, and starting array execution. The semantics and symbolic execution of old instructions are updated to take into account reconfigurable array execution, including an ϵ -calculus based semantics for the symbolic evaluator to model inaccessible and indeterminate values during array execution. The extensions are documented in [Coc02].

The reconfigurable processor which we use as our example is the Garp processor [Hau00]. This processor has not been physically implemented but it has been thoroughly specified and documented, which is critical for proving safety properties.

* Partially supported by the NSF Grants nos. CCR-9996150 and ITR-CCR-0113611.

3 Model Checking

Model checking is used to verify safety properties of the reconfigurable array. Model checking is a formal method that automatically examines finite models of concurrent systems to verify properties of the models. In this work we use RTCTL [Cam96], a branching-time propositional temporal logic, as a property specification language. RTCTL uses bounded temporal quantifiers to implement bounded model checking.

We use the NuSMV system [CR98] for the verification of array configurations. NuSMV uses the language SMV to describe models. This language is similar to hardware description languages. NuSMV supports RTCTL as a specification language.

We build the model by translating bit-level encodings of configurations to the SMV language. Any information from the surrounding machine code that is needed to verify safety must be used in the safety properties for the configuration as detailed in [Coc02].

4 Properties to Be Model Checked

The generic properties that will be checked for *all* configurations include:

- At most one memory access is initialized per clock cycle
- At most one memory access is scheduled to use the bus for each cycle
- There is a memory item ready to read when a row reads one
- There is a row initiating a memory write when a row transfers to memory

The generic properties all deal with synchronizing memory accesses so that they are defined by the semantics of Garp's reconfigurable array. If any of these properties is false, then there can be undefined behavior from the array. From a list of which control blocks can initiate memory accesses, it is possible to deduce these specifications.

The context dependent safety properties for a *particular* configuration include:

- All memory accesses respect the memory partition
- All memory accesses respect the memory access safety properties
- The postcondition is true after array execution

The context dependent properties all rely on information from the safety policy and the symbolic evaluation of the program where the array configuration is executed.

Both types of properties can also rely on the value of the initial count for the array execution. This count gives the number of cycles that the array executes if it does not halt itself first. The actual or maximum count can be used as a bound for temporal quantifiers so that the behavior of the model after the array would halt is not checked.

5 Performance of NuSMV on Translated Input Files

The performance of the model checker, NuSMV 2.0 running on an AMD Athlon at 1900 MHz with 1024 MB of memory under Debian Gnu/Linux 2.2, is the main factor to be explored for performance. Four example configurations were checked for six properties, the first is a generic property and the rest are context dependent properties:

- Memory control safety (MC)
- Memory alignment (MA)
- Lower bounds for memory reads (LBR)

- Upper bounds for memory reads (UBR)
- Lower bounds for memory writes (LBW)
- Upper bounds for memory writes (UBW)

The four configurations include three that perform the same function, but have different control paths and preconditions. The application is to read 200 word-sized pixels from an array, lighten each color component, and write the results back to the array.

The first configuration (IM1) has the precondition that the register in the fifth row of the reconfigurable array is loaded with a value equal to the value loaded in the first row of the reconfigurable array minus 14. This is because the first row reads the pixel array, the fifth row writes the result back, each of them is incremented by 2 on each cycle, and it takes seven cycles for the computation. The second configuration (IM2) has the precondition that the values loaded into the first and fifth row registers are equal. This is because the fifth row does not start incrementing until it is signaled on cycle 7 by the control path. The third configuration (IM3) does not have any corresponding precondition because it passes addresses from the first row to the fifth alongside the computation so that the value in the fifth row is always correct.

The fourth configuration (HASH) is a simple hash table. It is included to check the effect of *computed addresses* on the model checking. It reads values from an array, computes a 10-bit offset by repeated shifts and exclusive ors, and writes the value to an address plus the offset.

Each of the configurations has preconditions to ensure that the control path is correctly initialized, and that the access locations fit into memory without wraparound. All of the configurations have the trivial postcondition *true* because they do not leave any values in the array registers for later use.

The results are presented in Table 1 for properties that did not require an execution count related bound, and Table 2 for execution count bounded properties. The memory write properties for IM1, IM3, and HASH do not appear in Table 2 because they did not finish model checking in under four days. The reasons for the poor performance on the write boundary properties (LBW, UBW) are varied. For IM1, the precondition requires a very long bit-level specification and therefor a large symbolic representation that makes even trivial specifications practically uncheckable. IM3 and HASH both have a write address that is dependent on several rows of registers and cycles. This seems to be more than the model checker can handle efficiently as there is less possibility for cone-of-influence reduction of the symbolic representation. Even for IM2 it took a significantly longer time to check the write boundary properties as they depend on more of the array than the rest of the properties. Further information is available in [CKS03].

Table 1. Results of Model Checking Example Configurations for Unbounded Properties. Model checking times are given in seconds.

Property	Configuration			
	IM1	IM2	IM3	HASH
MC	6.710	6.140	6.810	17.400
MA	6.740	13.690	14.140	33.670

Table 2. Results of Model Checking Example Configurations for Bounded Properties. Model checking times are given in seconds.

Configuration / Property	Time Step Bound						
	400	1000	1024	4000	4096	16000	16384
IM1 LBR	15.23	16.84	16.92	24.10	24.42	54.00	54.76
IM1 UBR	16.97	20.66	20.26	38.78	36.95	106.72	98.70
IM2 LBR	15.13	16.55	16.65	23.20	23.12	49.38	50.04
IM2 UBR	16.59	19.98	19.99	36.57	36.57	102.92	94.32
IM2 LBW	64.00	76.11	70.52	141.71	88.37	161.54	414.43
IM2 UBW	69.64	87.61	76.30	109.13	197.60	240.14	239.10
IM3 LBR	16.12	18.63	18.91	30.70	31.22	80.50	81.77
IM3 UBR	18.00	23.73	23.22	48.70	47.40	147.10	142.53
HASH LBR	27.27	29.30	29.20	38.84	39.29	78.56	79.93
HASH UBR	28.42	32.99	32.65	58.93	57.52	168.92	169.41

6 Conclusions and Further Work

The main result of this work is that model checking reconfigurable processor configurations is a viable verification method for important safety properties, in particular the memory control properties. Although some memory access boundary properties have been found to be too complex for efficient checking, there may be methods to mitigate this in many cases. As the examples show, configurations having equivalent computations with different control strategies can have very different behavior when model checked. This could be taken into account in a compiler designed to produce efficiently checkable configurations.

Integrating model checking into a synoptic system of program verification in order to solve these problems is being explored. In particular, replacing proof-carrying code based on first-order logic with proof-carrying code based on temporal logic [BL02] is expected to provide a greater range of safety properties that can be checked. This may allow model checking of easy-to-check properties, which are then integrated into the proof of safety attached to the program as a whole.

References

- BL02. A. Bernard and P. Lee. Temporal logic for proof-carrying code. Technical Report CMU-CS-02-130, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 2002.
- Cam96. S.V. Campos. *A Quantitative Approach to the Formal Verification of Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1996.
- CKS03. J. Cochran, D. Kapur, and D. Stefanović. Model checking reconfigurable processor configurations for safety properties. Technical Report TR-CS-2003-18, Computer Science Department, University of New Mexico, 2003.
- Coc02. J. Cochran. Towards provably safe reconfigurable processor code: A model checking and proof-carrying code approach. Master's thesis, University of New Mexico, 2002. Available as Technical Report TR-CS-2002-36.
- CR98. A. Cimatti and M. Roveri. *NuSMV 1.1 User Manual*. ITC-IRST and CMU, 1998.
- Hau00. J.R. Hauser. *Augmenting a Microprocessor with Reconfigurable Hardware*. PhD thesis, University of California, Berkeley, 2000.
- Nec98. G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, 1998.

A Statistical Analysis Tool for FPLD Architectures

Renqiu Huang¹, Tommy Cheung², and Ted Kok²

¹ University of Cincinnati, Cincinnati, OH 45220, USA
huangr@ececs.uc.edu

² Hong Kong University of Science and Technology,
Clear Water Bay, Kowloon, HKSAR, China
{eetommy,eekok}@ust.hk

Abstract. This paper investigates an analysis tool for the routing resources in the FPLD architecture design. The developed tool can assess the performance of a given architecture specified by the physical configuration of logic blocks and the switch boxes topology. Two problems are mainly considered in this paper: given an architecture, the terminal distribution of each switch box is first determined via probabilistic assumptions, then the sizes of required universal switch boxes are evaluated for routing successfully. The estimations are validated by comparing them with the results obtained in the previous published experimental study on FPGA benchmark circuits. Moreover, our result confirms that the universal switch block is a good candidate for FPLD design.

1 Introduction

As design cycle is shortening and design complexity is increasing, system designers respond to these pressures by moving to the more cost-effective block-based designs. Field-Programmable-Logic Devices (FPLDs) and intellectual property (IP) techniques, without designing circuit from scratch, help designers to build the required complexity in a short time-to-market. Being a relatively new technology, FPLDs are constantly undergoing upheaval changes in their architectures to cope with the increasing component density and versatility as demanded by new IP functionality. This paper aims to develop an architectural analysis tool that provides the FPLD architects with the ability to perform trade-offs in designing a new FPLD architecture. The developed tool does not assume a particular architecture, which could be hierarchical and heterogeneous. However, it assumes the switch blocks to be universal [3]. Through a statistical analysis, information will be provided to the architect, identifying the possible deficiencies in the architecture such as routability reaching a low threshold value. This prompts the architect to improve the configuration of design.

2 Model and Analysis

A generic FPLD architecture is illustrated in Fig. 1. It consists of some number of different sizes of nonoverlapping polygons. There are two kinds of polygons.

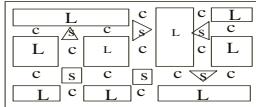


Fig. 1: A FPLD architecture (L for Logic block, S for Switch box, and C for Channel)

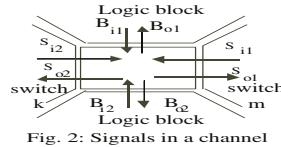


Fig. 2: Signals in a channel

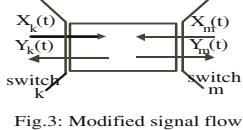
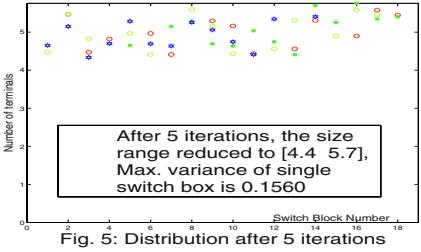
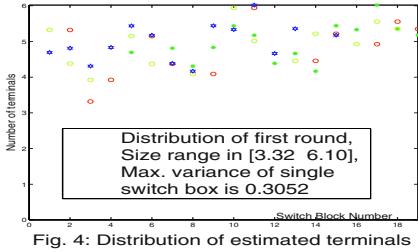


Fig. 3: Modified signal flow model for a channel

The one labeled with “L” is a configurable logic box, which may include look-up tables, flip-flops, and so on. The other labeled polygon with “S” is a switch box, which defines the connections between logic boxes. The white space between two sides of switch boxes is referred to as a routing channel. As for the switch boxes, they are assumed to be universal [2, 3] in this architecture. It has been proved in [3] that the universal switch blocks can accommodate significantly more routing instances than Xilinx XC4000-type switch blocks. Tsu, in his report [8], also concluded that the universal switch blocks require smaller silicon areas. The architectural model presented is generic, possibly hierarchical and heterogeneous.

The typical signals, shown in Fig. 2, may be related to Logic Blocks (B_{i1} , B_{i2} , B_{o1}) or switches (S_{i1} , S_{o1} , S_{i2} , S_{o2}), where the subscript denotes the signal flow directions with “i” for going into the channel, and “o” for getting out of the channel. As a result: $B_{i1} + B_{i2} + S_{i1} + S_{i2} = S_{o1} + S_{o2} + B_{o1} + B_{o2}$. Since our objective is to analyze the routing requirement of a universal switch box, therefore, an equal model as shown in Fig. 3 is considered which only consider signals generated or terminated at the channel under consideration. To analyses the routing resource requirement in a given architecture, there are two basic assumptions: the signals are randomly generated at the side of a logic block according to the Poisson distribution and each signal connection is assumed to have a random path [4–7]. Let $X_k(t)$ denotes the number of non-terminating signals measured at location t in the channel switch k which enter the channel from k . Furthermore, let $Y_k(t)$ denotes the number of signals measured at location t of the channel from k that enter the switch box k . Assumed the signal entering the logic blocks from the channel has a random arrival. Therefore, it can be modeled by Poisson distribution with an arrival rate λ . Further assuming that the probability of a signal to make a connection after traveling a distance t follows the exponential distribution with an average length α . Same as [5], the total number of connections made at location t of the channel as measured from k is given by $Z_k(t)$, where $Z_k(t) = X_k(t) + X_m(t) + Y_k(t) + Y_m(t)$. Noted that $Z_k(t)$ is also Poisson with parameter $\lambda_{Z_k}(t) = E(Z_k(t))$. When $t = 0$, $Z_k(0)$ actually denotes the number of signal emerging from the k -th side of the switch box. If the switch box has N sides, the total number of signal emerged from the switch box to neighboring channels are given by $\Gamma = \sum_{k=1}^N Z_k(0)$. Since $Z_k(0)$ is Poisson with parameter $\lambda_{Z_k(0)}$, Γ is also Poisson with parameter $\lambda_\Gamma = \sum_{k=1}^N \lambda_{Z_k(0)}$. Depending on the connection topology of the switch box, each of the signal in Γ can be connected by one of the K different types of connection. Let $A_i, i = 1, \dots, K$, denotes the random variable for the numbers of i -th type switch used in the switch box. Therefore, the probability of successful connection for all the signals in Γ is given by the multinomial distribution



of the joint probability of A_i with a given required number, γ , of signals to be connected.

$$P\left(A_1 = n_1, \dots, A_K = n_K | \Gamma = (\gamma = \sum_{i=1}^K n_i)\right) = \frac{(\sum_{i=1}^K n_i)!}{n_1! n_2! \dots n_K!} P_1^{n_1} \dots P_K^{n_K} \quad (1)$$

where P_i are the probability of connecting a signal using the i -th switch, and, $\sum_{i=1}^K P_i = 1$. The above conditional probability can be simplified by observing that condition $\Gamma = \gamma$ is independent of the A_i process. Therefore, by sampling the Poisson process

$$P\left(A_1 = n_1, A_2 = n_2, \dots, A_K = n_K\right) = \prod_{i=1}^K \frac{(P_i \lambda_\Gamma)^{n_i} e^{-P_i \lambda_\Gamma}}{n_i!} \quad (2)$$

As a result, the above equations can be used to estimate the routability of FPLDs with universal switch blocks.

3 Experimental Results and Conclusions

The first experiment demonstrates the application of the developed analysis tools for optimizing FPLD architecture to minimize the number of switches requirement. The switch requirement can be minimized by equalizing the number of signals emerging from each side of the switch box. This is important for universal switch design. The optimization begins by choosing the switch box that has the largest difference between the number of signals emerged from two sides of the switch box. The positions of the neighboring logic blocks are then swapped. After swapping the logic blocks, the numbers and types of connections at each box are computed again, followed by another round of logic block swaps. Several FPLD architectures that follow Fig. 1 are developed by randomly choosing the positions, sizes, and the number of logic blocks. The total numbers of switches required are reduced, observations are shown in Fig. 4 and Fig. 5 respectively. Further, the number of switch blocks is also reduced (18 verse 19). The parameters λ and α used in our simulation are predicted as in [4].

The second experiment estimates the switch resource requirement (listed in Table 1.) for a Xilinx XC4000 alike architecture but with the Xilinx clique switch replaced by the universal switch box. Noted that the number, N , is chosen such

Table 1. Switch box size for successful routing of a given circuit and logic blocks

Circuit	logic blocks	Altor [10]		Splace [9]		Channel Width		Switch sizes	
		Sroute [9]	VPR	Sroute	VPR	Max.	Min.	Max	Min
9symml	70	7	6	7	5	8	4	10	8
apex7	77	9	9	6	4	8	4	10	8
example2	120	11	10	7	5	9	5	10	8
Alu2	143	9	8	8	6	9	5	10	8
templ	54	8	7	5	4	7	3	10	8

that the total numbers of logic blocks, $N \times N$, is large enough to implement the logic of the given circuits. Comparing the channel width requirement obtained after placement and routing (first and second row of 3-5th column) in Table 1 with the estimated minimum number of tracks in each channel, it is observed that the results are comparable except for the case of VPR [1] routing and placement. This is because VPR used simulated annealing in the placement and routing algorithm, thus archiving a global minimum in the switch size. Another observation is that all the requirements of switch size are same, there are two reasons: first, the algorithm only consider different switch box configuration, i.e. switch box with different number of switches of each type; second, the rounded values hide the tendency of increase. On the other hand, the relative steadiness of switch size indicates that the universal switch block is a good candidate topology for FPLD design. Listed in Table 1 is the bounds for the switch box size that can achieve higher than 99% of successful routing as estimated by eq.(2).

References

1. V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for deep-submicron FPGAs*. Kluwer Academic publishers, 1999.
2. Y. Chang, D. F. Wong, and C. K. Wong, *Universal Switch Modules for FPGA Design*. ACM Trans. on Design Automation of Electronic Systems, Jan. 1996.
3. M. Shyu, G. Wu, Y. D. Chang, and Y. W. Chang, *Generic universal switch blocks*. IEEE Trans. Comput., pp.348-359 Apr. 2000.
4. S. D. Brown, J. Rose, and Z. G. Vrabesic, *A stochastic model to predict the routability of field programmable gate arrays*. IEEE Trans. CAD, Dec. 1993.
5. A. A. El. Gamal, *Two-dimensional stochastic model for interconnections in master slice integrated circuits*. IEEE Trans. CAS-I, no. 2, pp. 127-138, 1981.
6. A. A. El. Gamal, *A stochastic model for interconnections in custom integrated circuits*. IEEE Trans. CAS-I, pp.888-894, Sept. 1981.
7. S. Sastry and A. C. Parker, *Stochastic models for wirability analysis of gate arrays*. IEEE Trans. Computer-Aided Design, pp.52-65, Jan. 1986.
8. W. Tsu, *A comparison of universal and Xilinx switches*. CS294-7 project report, Univ. of California-Berkeley, Spring 1997.
9. S. Wilton, *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. Ph.D. Dissertation, University of Toronto, 1997.
10. J. Rose, W. Snelgrove and Z. Vranesic, *Altor: an automatic standard cell layout program*. Proc. of Canadian Conf. on VLSI, 1985, pp. 169-173.

FPGA-Implementation of Signal Processing Algorithms for Video Based Industrial Safety Applications

Jörg Velten and Anton Kummert

Communication Theory,
University of Wuppertal,
42119 Wuppertal, Germany
`{velten,kummert}@uni-wuppertal.de`

Abstract. Conventional protective devices as light curtains allow safe but often inconvenient flow of work. Unfortunately uncomfortable safety devices are often bypassed or simply switched off. Consequently, the design of a video based protective device, which avoids inconvenient processing steps is of special interest. The present paper describes favorable combinations of FPGA-hardware and algorithms, which allow safeguarding of work places if several constraints are met. The methods were originally developed for surveillance of press brakes, but it is easily adaptable to different types of machines or work places.

1 Introduction

Development of video based industrial safety devices is still in its beginnings. It is discussed intensively by responsible authorities for the aim of safeguarding complex situations and enable convenient work flows. Problems arise from the combination of demands for high recognition reliability and low device prices, i.e. ability to implement detection algorithms in low cost hardware. Especially a high-speed implementation of video data processing at convenient frame rates represents a challenging barrier. The present paper describes the implementation of suitable combinations of algorithms and hardware for a video based protective device at press brakes in FPGA architecture. A camera takes images from a bird's eye view perspective and delivers the usual R- G- and B- signals with 8-bit resolution each. Fig. 1 shows at the lefthand-side the intensity information of an example image. Two independent algorithms, which follow different principles for ensuring an inviolated "region of danger" are described below. The first algorithm follows the principle of detecting any object intruding a buffer zone between worker and machine, while the second one verifies hands at risk-less positions, as described in [2].

Selection of algorithms has been performed with respect to usual FPGA-elements, mainly 4:1 LUTs with subsequent latch, in this paper referred to as Logic Element or short LE. Consequently, realizations are not restricted to a specific FPGA-family.

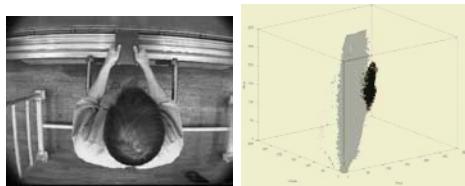


Fig. 1. Example image of a press brake (grayscale) at the left-hand side and the representation of the colour-image in RGB color space at the right-hand side

2 Surveillance of a Danger Zone

Surveillance of a danger zone can be established by comparison of camera images with a previously acquired image of the empty workplace. For reduction of illumination influences, differentiation between edge-images instead of intensity images is performed. The Sobel operator (cf. [4]) uses a 3×3 pixel filter mask containing the values ± 2 , ± 1 and 0, and is consequently suitable for an FPGA implementation. Detected differences are subsequently compared to predefined image masks that represent warning and danger areas. Further independence from illumination conditions is obtained by additional binary morphological operations, namely dilation of the edge image and erosion of the difference image, each performed by a 3×3 pixel structuring element (cf. [3]). The whole processing cycle is shown in Fig. 2.

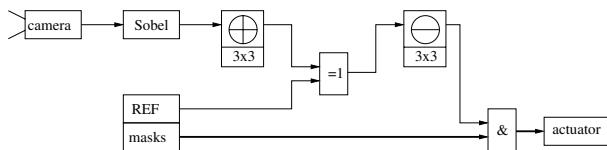


Fig. 2. The processing cycle of "surveillance of a danger zone"

The Sobel operator filter mask can be transferred directly to an FPGA implementation by using a usual 2D-FIR filter for serial image data, where delays in horizontal direction are realized by time delays T_1 , equal to the reciprocal of the sampling frequency, and delays in vertical direction are realized by $T_2 = NT_1$, where N denotes the number of pixels in a row. Size of actual FPGAs allow a direct implementation of the filter. Multiplications are realized by shift operations, which are realized by fixed by wiring. The delays T_1 and T_2 are either implemented by using LE-latches, or external shift registers. Several FPGAs allow the conversion of one LE into a 16-bit shift register, which leads to the possibility of combining several LEs to achieve a long delay T_2 . By exploiting these abilities, the number of necessary delays for implementation of the 3×3 pixel filter mask and 8 bit grayscale image data of size 284×288 pixel, amounts

to 423 LEs. Furthermore, the Sobel operation can be simplified by replacing the square root expression of the gradient calculation by an absolute sum and the search for a maximum by a threshold operation. The full implementation of the simplified Sobel operator for the mentioned input signal needs only 532 LEs (synthesis by LeonardoSpectrum). Morphological operations also can be implemented in a similar 2D-FIR filter structure (cf. [5]) to act in the same way as the Sobel operator, except that signal types are binary, which drastically reduces the necessary hardware effort. Only 100 LEs are used to implement both morphological operations. Comparison with filter masks is done by a bit-wise AND concatenation of the output signal with suitably synchronized read out mask information. The masks as well as the reference image have to be stored in an external memory. The length of each mask word is determined by the necessary mask configuration, i.e. how many different areas have to be distinguished. Timing simulations of this implementation can be performed at frequencies of up to 86 MHz (ModelSim). The number of latency clock pulses is 1184, which leads for 50 Hz cameras and the above mentioned image size to a total recognition time of only 20,22 ms.

3 Detection of Endangered Extremities

The most endangered extremities at press brakes are hands and fingers, which can be recognized by skin color detection. For assessing minimal hardware effort in the present paper, application conditions are limited to a non-varying lighting spectrum and favorable background colors. Fig. 1 shows a color image (in grayscale), as well as its representation in RGB color-space. Furthermore, significance can be enhanced by coercing workers to wear colored gloves. Rough distinction between skin and other colors can be applied in the presented example in RGB space by using a linear plane for separation, which can be established by a very simple ANN consisting of only one perceptron (cf. [1]). This leads to detection of skin color in a non-optimal, but "hardware efficient" way. Fig. 3 shows at the top left hand side the result of a skin color detection performed in the above presented way. Several misclassified pixels have to be eliminated by considering regional information using a binary morphological erosion operation.

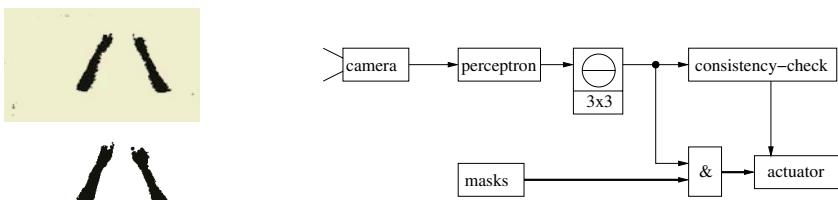


Fig. 3. Respective result images before (top) and after the erosion (bottom) are shown at the left-hand side. The block diagram at the right illustrates the processing cycle of the hand detection

The respective result image is shown at the lower left-hand side. The latter is compared to image masks that represent danger and warning zones as proposed in section 2. Further processing is necessary to consider detectability of hands and foresighted danger recognition. Evaluation of these properties however, doesn't need to take place at video frame rate, since respective attributes don't change very fast. For this reason, detectability of hands can be measured evaluating size and number of detected skin regions. The whole processing flow is shown in Fig. 3, at the right-hand side.

The implementation of the perceptron requires at least a resolution of 11 bits for the input weights. Synthesis of a VHDL coded perceptron with three 8-bit input signals leads to the requirement of 444 LEs.

The morphological operation is realised in a filter structure and requires 50 additional logic elements. The comparison with appropriate filter masks is done by bit-wise AND concatenation of the output signal with suitably synchronized read out mask information. Like in the implementation of the first algorithms, masks have to be stored in an external memory, using 8-bit words for each mask pixel. Detection and mask-comparison functionality can be implemented using 564 logic elements. The timing simulation allows frequencies of up to 100 MHz. The number of latency clock pulses is 397, which for 50 Hz cameras and the above mentioned image size leads to a total recognition time of 20,08 ms

4 Conclusion

The development of video based safety devices based on the proposed algorithms leads to extended possibilities for safeguarding of dangerous workflow. Improvements in reliability are necessary for industrial application. A first possible improvement in this respect, forms the combination of both implementations. Although development of video based protective devices is still in its beginnings, the proposed combinations of algorithms and hardware represent promising results for further development. The latter includes adaptive learning of skin color in the FPGA and evaluation of motion tracking algorithms for the purpose of preemptive danger avoidance.

References

1. S. Haykin. *Neural Networks*. Prentice Hall, New Jersey, USA, 1999.
2. W. Lehner, A. Kummert, and J. Velten. Device and method for monitoring a detection range on an operating tool. Application for US patent, 2001.
3. W. K. Pratt. *Digital Image Processing*. Wiley-Interscience, New York, USA, second edition, 1991.
4. I. Sobel. *Camera Models and Machine Perception*. PhD thesis, Stanford University, Stanford, USA, 1970.
5. J. Velten and A. Kummert. FPGA-Implementation of high speed n-D binary morphological operations. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP'03)*, Bangkok, Thailand, 2003.

Configurable Hardware Architecture for Real-Time Window-Based Image Processing

Cesar Torres-Huitzil and Miguel Arias-Estrada

Computer Science Department, INAOE, Apdo. Postal 51 & 216, México
ctorres@ccc.inaoep.mx, ariasm@inaoep.mx

Abstract. In this work, a configurable hardware architecture for window-based image operations for real-time applications is presented. The architecture is based on an array of elemental processors under a systolic and pipeline approach to achieve a high rate of processing. A configurable window processor has been developed to cover a broad class of image processing algorithms and operators. The system is modeled in a Hardware Description Language and has been prototyped on an FPGA device. Some implementation and performance results are presented and discussed.

1 Introduction

Window-based image processing requires a large amount of low level processing on massive amounts of data in order to analyze the image contents and recover useful information [1]. The processing is characterized by a large amount of data processed at small neighborhoods by relatively simple operations [2]. In window-based operations the input is an image, and the output, is some symbolic quantity denoting features or location of objects in the image.

The window-based image processing complexity is expressed in terms of the elementary arithmetic operations required to process an image. A window-based operator has a computational complexity of $O(w^2MN)$ for an $M \times N$ image with a $w \times w$ mask. The complexity is further increased since most vision systems are intended for real time operation [3]. Over 50 million operations are required to process an image of 640×480 pixels, and a computational power of tens of giga-operations per second is required to achieve real-time performance. The time requirements and the inherent limitations on the computational power of conventional processors have motivated the development of dedicated coprocessors for image processing. The nature of window-based operations is inherently local and suitable for data parallel processing. Combining data overlapping and parallel processing, it is possible to achieve high performance in window-based image processing.

The rest of the paper is organized as follows. Section 2 presents the hardware architecture developed for window-based image processing. Section 3 presents some implementation and performance results. Finally, section 4 presents some conclusions and further work derived from this research.

2 Configurable Hardware Architecture

A simplified block diagram of the proposed hardware architecture is shown in figure 1. The architecture reads data from the input memory where input image pixels and mask window pixels are stored, denoted as P and W, respectively, in figure 1. The pixels are read in a column-based scan and transmitted to the array of processors to compute, in parallel, several window operations [4]. The processed data is captured by the global data collector and stored in the output memory bank.

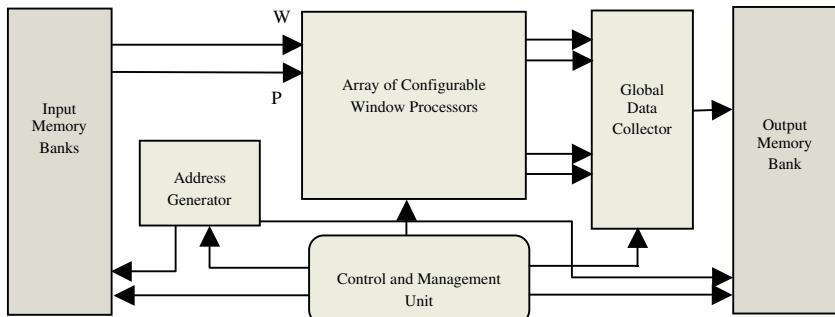


Fig. 1. Proposed hardware architecture for window-based image processing

The architecture contains four separate functional units: *control and management unit*, *address generator*, *global data collector*, and an *array of configurable window processors*. The *control unit* manages the data flow and synchronizes the different operations performed in the architecture. The *address generator* module generates the addresses for accessing image data. The input image memory and the window memory are read in a column-based scan, but a row-based scan is possible. Also, the address generator generates the addresses for the output memory to store the results produced by the architecture. The *global data collector* module collects the processed image data over the entire array and the result is sent to the output image memory.

The *array of processors* is the computation core of the architecture. The processors are arranged under a 2-D systolic approach. A closer view of the array of processors is shown in figure 2. The modules labeled by CWP and D denotes a processing element and a delay line, respectively. A linear systolic array of CWPs is interconnected to another through a delay line D for transmission of the mask window coefficients [5][6]. The delay line size is dependent on the number of rows processed in parallel. For an array of $w \times w$ CWPs the delay line is composed of $(w-1)$ registers.

Figure 2 also shows a block diagram of the CWP designed to cover most window-based operations. It is called Configurable Window Processor (CWP). An accumulator, a register and an ALU-like processor (AP) integrate a CWP. Each CWP in the array computes a window operation progressively. At each cycle, one CWP receives a different value of the window mask and all the CWPs receive the same pixel value from the input image. After a latency period, the CWPs deliver their results progressively. Once a result is produced and captured by the data collector, the CWP is ready to start a new computation. The architecture works progressively in the same manner until the end of the data in the input image.

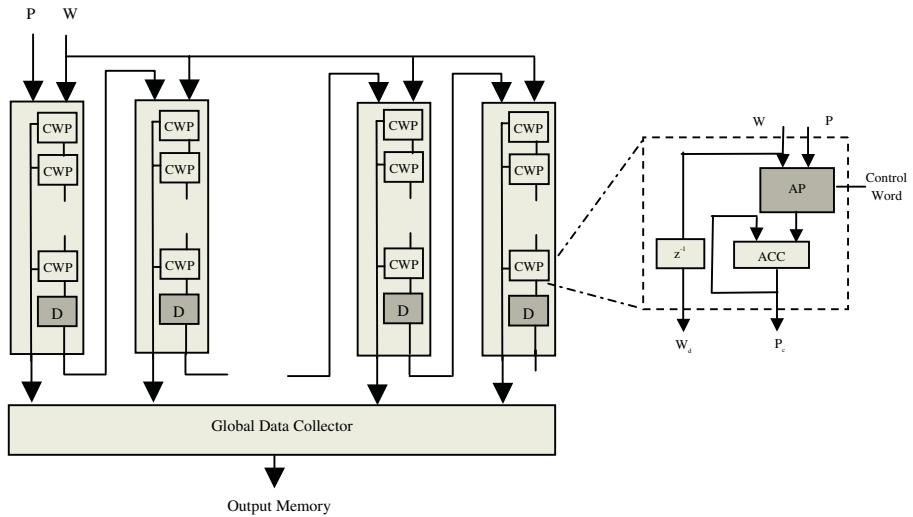


Fig. 2. Array of processing elements for fast speed computing of window-based operations and a block diagram of the processing element

3 Implementation and Results

The architecture was modeled in VHDL and synthesized for a VirtexE FPGA device using an array of 7×7 CWP modules. The architecture was prototyped on an RC1000PP board from AlphaData [7]. At a frequency of 40 MHz, the processing time for a 512x512 image is 12.6 milliseconds, so the architecture processes around 80 images per second. Table 1 shows a summary of the hardware resources and performance characteristics. Table 2 compares the architecture to other systems reported in [8][9]. The architecture achieves similar performances with less hardware resources.

Table 1. Summary of hardware resources utilization and performance characteristics of the architecture when mapped to a VirtexE XCV2000E-6 BGA560 device

Complete Architecture (7×7 CWP)	2604 slices
Single CWP	54 slices
FPGA percentage	13 %
Maximum frequency	40 Mhz

4 Conclusions and Further Work

In this work a configurable hardware architecture for window-based algorithms has been presented. The regular structure of the architecture allows the array of CWP modules to be expanded and configured to adapt to different mask sizes and performance

requirements. The use of a large number of CWP_s in the architecture to increase the number of rows processed in parallel, can reduce the processing time of an image to hundred of microseconds. The architecture has been tested on algorithms such as convolution, correlation, 2-D filtering, feature detection, template matching, and gray-level morphology. The architecture functionality can be extended to applications such as motion estimation, and stereo disparity. As a further work the integration of the architecture on a smart camera will be addressed and aspects of dynamic reconfiguration, at the processing element level, will be explored for a more efficient use of hardware resources

Table 2. Comparison performance of the proposed architectures with other systems using image convolution on a 512×512 gray-level image

System	Architecture	Window size	Timing
PDSP16488, 40 MHz	ASIC	8x8	6.56 ms
LSI Logic's L64240, 20 MHz	ASIC	8x8	13.11 ms
DECchip 21064, 200MHz	Multiprocessor	5x5	220 ms
MAP1000, 200 MHz	VLIW	7x7	7.9 ms
Proposed architecture, 40 MHz	FPGA-based	7x7	12.6 ms

References

- 1 N. Ranganathan, *VLSI & Parallel Computing for Pattern Recognition & Artificial Intelligence*, Series in Machine Perception and Artificial Intelligence, Volume 18, World Scientific Publishing, 1995.
- 2 R. Jain, R. Kasturi and B. S. Shunck, *Machine Vision*, McGraw-Hill, International Edition, 1995.
- 3 Phillip A. Laplante and Alexander D. Stoyenko, *Real-time Imaging: Theory, Techniques, and Applications*, IEEE Press, 1996.
- 4 Miguel Arias Estrada, and César Torres Huitzil, “*Real-time Field Programmable Gate Array Architecture for Computer Vision*”, Journal of Electronic Imaging, Volume 10, number 1, January 2001, pp. 289-296
- 5 H. T. Kung, “*Why systolic architectures?*”, IEEE Computer, pp. 37-46, January 1982.
- 6 Peter Pirsh, and Hans-Joaching Stolberg, “*VLSI Implementations of Image and Video Multimedia Processing Systems*”, Transactions on Circuits and Systems for Video Technology, Vol. 8, No. 7, November 1998, pp. 878-891
- 7 Charles Sweeney, and Bill Blyth, “*RC1000-PP Hardware Reference Manual*”, Celoxica Limited, Version 1.22, 2000
- 8 M. Y. Siyal, M. Fathy, “*A Programmable Image Processor for Real-time Image Processing Applications*”, Microprocessors and Microsystems 23, 1999, pp. 35-41
- 9 N. Ratha and A. Jain, “*FPGA-based computing in Computer Vision*”, Fourth IEEE International Workshop CAMP, p. 128-137, IEEE Computer Society, 1997.

An FPGA-Based Image Connected Component Labeller

K. Benkrid, S. Sukhsawas, D. Crookes, and A. Benkrid

School of Computer Science, The Queen's University of Belfast, Belfast, BT7 1NN, UK
([k.benkrid](mailto:k.benkrid@sussex.ac.uk), [s.sukhsawas](mailto:s.sukhsawas@sussex.ac.uk), [d.crookes](mailto:d.crookes@sussex.ac.uk), [a.benkrid](mailto:a.benkrid@qub.ac.uk))@qub.ac.uk

Abstract. This paper describes an FPGA implementation of a Connected Component Labelling algorithm (CCL), developed at Queen's University Belfast. The algorithm iteratively scans the input image, performing a non-zero maximum neighbourhood operation. It has been coded in Handel C language and targeted Celoxica RC1000-PP PCI board. The whole design was fully implemented and tested on real hardware in less than 24 man-hour. It uses a Virtex-E FPGA and two banks of off-chip memory. For 1024x1024 input images, the whole circuit consumes 583 FPGA slices and 5 Block RAMs and can run at 72 MHz, leading to a 68 pass/sec performance. The FPGA implementation outperforms, easily, an equivalent software implementation running on a 1.6 GHz Pentium-IV PC. A 10-fold speed up has been realised in many instances.

1 Introduction

Connected Component Labelling is an important task in intermediate image processing with a large number of applications [1][2]. The problem is to assign a unique label to each connected component in the image while ensuring a different label for each distinct object as illustrated in Figure 1.

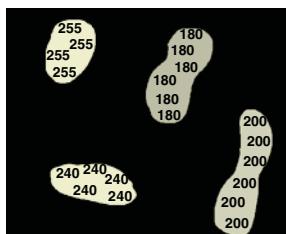


Fig. 1. A labelled Image example

To date, many algorithms have been developed to handle this problem [3][4]. This paper describes an FPGA implementation of an architecture based on a serial iterative algorithm for CCL developed at Queen's University Belfast [5].

In the following section, we describe the algorithm used. We then describe the hardware architecture, after which we give details of the FPGA implementation and its performance. Finally conclusions are drawn.

2 The Proposed CCL Algorithm

Given an arbitrary input grey-scale image, the CCL algorithm which we use is an iterative one and consists of the following steps:

- Step 1: Threshold the input image to obtain a binary image. This will make all pixels in the objects equal to 1 and all other pixels equal to 0.
- Step 2: The thresholded binary image is initially labelled. The initial labelling technique we adopt is: firstly to give the first non-zero pixel the highest label, and decrease the label value for subsequent non-zero pixels. Secondly, and in order to reduce the number of bits per pixel, we give adjacent non-zero pixels, within the same column (assuming a vertical scan), the same label as we know they are connected.
- Step 3: Apply an iterative ‘non-zero maximum’ neighbourhood operation on the image, using the window given in Figure 2. During this operation, each result pixel is stored back in the source image. A complete forward pass is followed by an inverse pass in which the image is scanned in reverse order.
- Step 4: Repeat Step 3 until there is no change in the image.

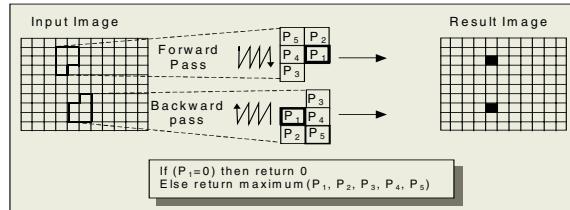


Fig. 2. CCL neighbourhood operation

3 The Hardware Architecture

The hardware architecture for the algorithm proposed above is illustrated in Figure 4. This architecture is a serial one, where pixels are fed to the FPGA one at a time. During the first pass, the input image is first thresholded, to produce the binary image, then initially labelled. In subsequent passes, these operations are bypassed. In order to eliminate the propagation of the pixel label from the bottom of an image column to the top of the following one, a column counter is provided to inhibit this propagation. To cater for the forward/backward multi-pass scheme, the image is read in the reverse order from which it has been stored during a backward pass.

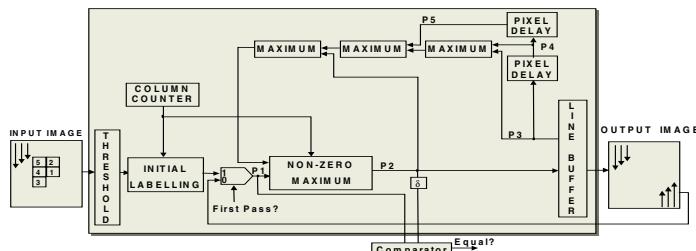


Fig. 4. Architecture of the Labelling Unit

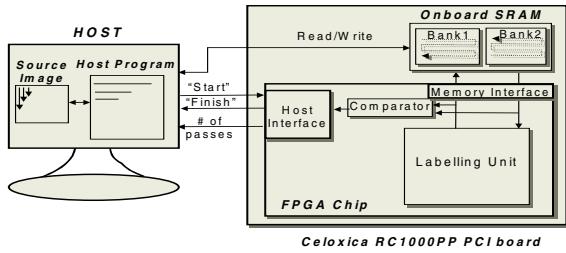


Fig. 5. Hardware organisation of the CCL implementation on Celoxica RC1000-PP PCI board

Detecting if a pass has resulted is done on the fly, to avoid a separate pass for this purpose. A flag is maintained during processing, and is set to 1 if and when any result pixel differs from its original value. To test for termination of the whole algorithm, this flag must be 0 at the end of a pass (either forward or backward).

The organisation of the proposed architecture on Celoxica's RC1000-PP PCI board is given in Figure 5. The architecture makes use of the onboard XCV2000E-6 Virtex chip and two 4MB SRAM banks. The FPGA reads the input image alternatively from one bank, performs a non-zero maximum pass on it and writes the result onto the other bank. The input image is first downloaded into one SRAM bank on the board by the host software (written in C). The latter then starts the CCL algorithm. After that, the FPGA operates autonomously. A forward pass is followed by an inverse one. The process is repeated until no change in the image occurs.

The memory interface block generates the proper off-chip RAM address sequences, whereas the Host interface block interfaces between the Host program (written in C) and the FPGA hardware through a number of registers.

4 Real Hardware Implementation

The CCL algorithm presented above has been fully coded in Handel C language and implemented and fully tested using Celoxica DK1 suite and RC1000-PP PCI board in less than 24 man-hour. The circuit description is scaleable and parameterisable in terms of the: the image size, the input image pixel word length, the processing word length and the threshold level (for binarisation).

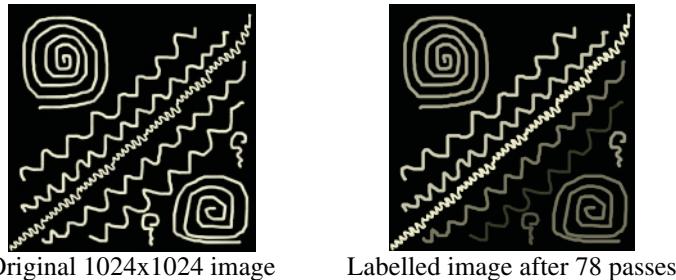
The whole algorithm needed ~250 lines of Handel C code. The **par** instruction for expressing parallel flows, as well as the **if** statement for expressing conditions and selecting alternative flows based on these conditions, have been used extensively. The circuit was simulated conveniently at the Handel C source level. Hardware, in the form of EDIF netlist, is generated automatically from Handel C by Celoxica tools.

We have tested the algorithm using many combinations. Table 1 gives a sample of the results for different image sizes using 8-bit/pixel input images, a user-tunable threshold level and a processing word length of 32 bits.

Figure 6 gives an example of an input image and the corresponding labelled image after the application of the CCL algorithm, along with the number of passes required. It takes ~1.57 sec to label this image on FPGA. An equivalent software implementation (written in C), on a 1.6 GHz Pentium-IV PC, takes ~18.17 sec. Thus, a 10-fold speed up is possible with the FPGA-based implementation.

Table 1. Implementation results of CCL algorithm on the RC1000-PP PCI board

Image size	Slices (out of 19200)	BRAMs (out of 160)	Speed (MHz)	Number of passes/sec
256x256	526	1	78	1190
512x512	553	3	73	278
1024x1024	583	5	72	68

**Fig. 6.** Sample result of the implementation of our CCL algorithm

5 Conclusion

In this paper, we have presented an FPGA implementation of an architecture based on a serial iterative algorithm for Connected Component Labelling (CCL). The circuit has been coded in Handel C language and fully tested on Celoxica Virtex-E based RC1000-PP PCI board in less than 24 man-hour. Handel-C proved extremely convenient for this kind of control-intensive algorithms.

The paper presented implementation results for different image sizes. For instance, the implementation of the proposed algorithm for 1024x1024 input images consumes 583 slices and 5 Block RAMs on the FPGA and can run at 72 MHz. This results in a maximum number of passes equal to 68 passes/sec. The FPGA implementation, easily, outperformed a 1.6 GHz Pentium-IV PC implementation. A 10-fold speed up has been realised in many instances.

References

- [1] D. L. Milgram, "Region extraction using convergent evidence", *Computer Graphics and Image Processing*, vol. 5, no. 2, pp. 561-572, 1988.
- [2] L. C. Sanz and D. Petkovic, "Machine vision algorithms for automated inspection of thin-film disk heads", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 10, no. 6, pp. 830-848, 1988.
- [3] H. M. Alnuweiri, and V. K. Prasanna, "Parallel architectures and algorithms for image component labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 14, No. 10, pp. 1014-1034, Oct. 1992.
- [4] R. Klette, P. Zemperoni, "Image Processing Operators", John Wiley & Sons, New York, 1995.
- [5] D. Crookes, K. Benkrid, "An FPGA Implementation of Image Component Labelling", SPIE, Vol. 3844, USA, 1999.
- [6] Celoxica Handel-C and RC1000-PP PCI board Production Information, Celoxica Ltd. <http://www.celoxica.com>

FPGA Implementation of Adaptive Non-linear Predictors for Video Compression

Rafael Gadea-Girones¹, Agustín Ramírez-Agundis²,
Joaquín Cerdá-Boluda¹, and Ricardo Colom-Palero¹

¹ Department of Electronic Engineering, Universidad Politécnica de Valencia, Camino de Vera
s/n, 46020, Spain

rgadea@eln.upv.es, rcolom@eln.upv.es

² Department of Electronic Engineering, Instituto Tecnológico de Celaya, Av. Tecnológico s/n,
38010, México
aagundis@itc.mx

Abstract. The paper describes the implementation of a systolic array for a non-linear predictor for image compression. We can implement very large interconnection layers by using large Xilinx and Altera devices with embedded memories and multipliers alongside the projection used in the systolic architecture. These physical and architectural features create a reusable, flexible, and fast method of designing a complete ANN (Artificial Neural Networks) on FPGAs. Our predictor, a MLP (Multilayer Perceptron) with the topology 12-10-1 and with training on the fly, works, both in recall and learning modes, with a throughput of 50 MHz, reaching the necessary speed for real-time training in video applications.

Introduction

In recent years, it has been shown that neural networks can provide solutions to many problems in the area of image compression. Software simulations are useful for investigating the capabilities of ANN models and creating new algorithms; but hardware implementations remain essential for taking full advantage of the inherent parallelism of ANN.

Traditionally, ANNs have been implemented directly on special-purpose digital and analogue hardware. More recently, ANNs have been implemented with reconfigurable FPGAs. Although do not achieve the power, clock rate, or gate density, of custom chips; they are much faster than software simulations [1]. Until now, a principal restriction to this approach has been the limited logic density of FPGAs.

This paper offers advances in two basic respects to previously reported neural implementations on FPGAs. The first is the use of an aspect of backpropagation and stems from the fact that forward and backward passes of different training patterns can be processed in parallel [2]. The second point we contribute is to produce a completed ANN with on-chip training, and good throughput for the recall phase – on a single FPGA. This is necessary, for example, in industrial machine vision, and for the training phase, with continual online training (COT) [3]. In our research, we need this properties in order to design a adaptive non linear predictor for video compressionNon-linear Predictor Application

Non-linear Predictor Application

In this section we present results obtained by experimenting with the adaptive non linear prediction approach proposed by Marusic and Deng [4] together with the encoder method proposed by Howard and Vitter [5] in order to obtain a complete lossless compression system (Fig. 1).

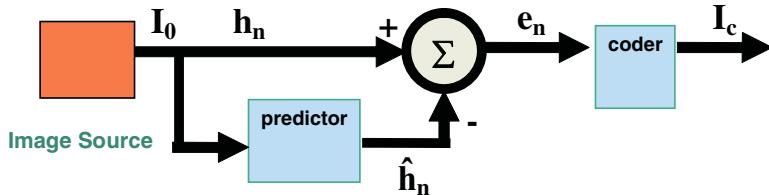


Fig. 1. Structure of the Predictor System

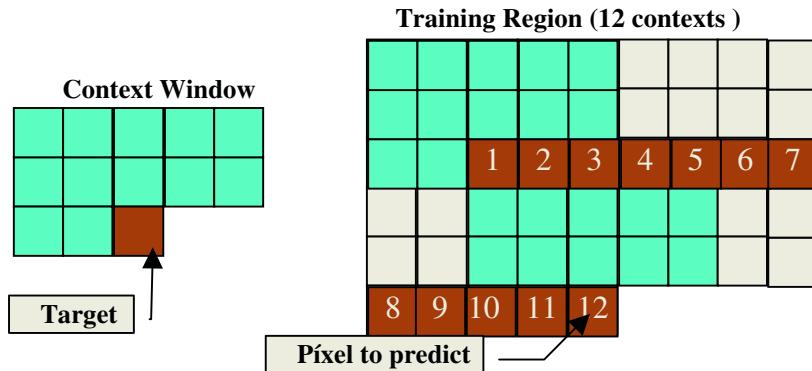


Fig. 2. Area for adaptive training.

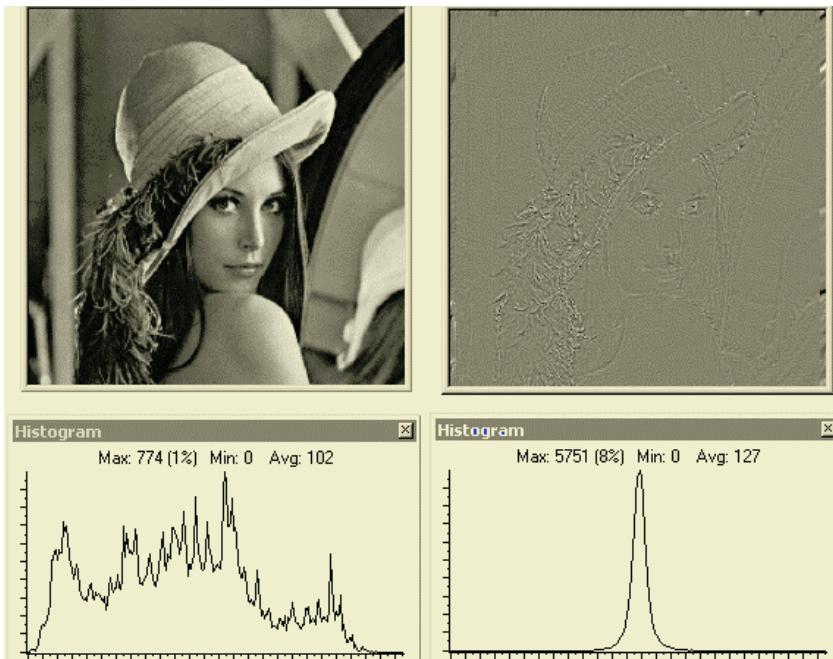
The prediction stage is an adaptive predictor that uses a dynamic training region for each pixel to be estimated. The training region has twelve context windows, each one with twelve pixels (Fig. 2).

The image is scanned, pixel by pixel, from left to right and from top to bottom. On each pixel, the training region is represented by twelve training vector; these are used to train a 12-10-1 two layer perceptron. Once the network has been trained, the pixel is predicted and the current weight matrix is used as the initial one for the next pixel (Fig. 2).

We can evaluate a predictor method by its ability to reduce the correlation between pixels (entropy) as well as the PSNR for the predicted image (e_n). We can observe a summary of these parameters for different topologies of the MLP of the Adaptive Non-linear Predictor in Table 1. We obtain, with the on line version of the Backpropagation algorithm, better results than with batch line version and we show that the number of hidden layers (20 or 10) is not relevant for the prediction performance. These two ideas are very important for our hardware implementation, both in throughput and in area.

Table 1. Results of prediction with different topologies with Lena256x256 Image

Hidden Neurons	10	20	10	10	20
Context (inputs)	12	12	4	12	12
PSNR	27.8575	27.9511	27.4993	28.5662	28.5537
Entropy of residue	4.9858	4.9763	5.0409	4.855	4.8648
Entropy of original	7.5888	7.5888	7.5888	7.5888	7.5888
Version Learning	Batch line	Batch line	Batch line	On line	On line

**Fig. 3.** Original (h_n) and residue (e_n) images, and its histograms.**Table 2.** Design summary for MLP(12-10-1) .

X2V3000BF957-6 (VIRTEX2)			
Number of Slices:	3,601	out of	14,336 25%
Number of Flip-Flops:	2,114	out of	28,672 18%
Number of Mult18x18s:	75	out of	96 78%
Number of RAMB16s:	48	out of	96 50 %

The results of the implementation of the MLP(12-10-1) are shown in Table 2 and 3. This implementation works, both in recall and learning modes, with a throughput of 50 MHz (only possible with our pipeline version), reaching the necessary speed for real-time training in video applications and enabling more typical applications (wavelet transform and run-length coder) to be added to the image compression.

Table 3. Design summary for MLP(12-10-1) .

EP1S60B856C7 (STRATIX)			
Number of Logic Cells:	7,362 out of	57,120	12%
Number of Flip-Flops:	2,381 out of	59,193	4%
Number of DSP blocks:	133 out of	144	92%
Number of m512s:	47 out of	574	8%

Conclusions

Until now, mask ASICs offered the preferred method for obtaining large, fast, and complete ANN for designers who implement neural networks with full and semi-custom ASICs. Now, we can exploit all the embedded resources of the new programmable ASICs (FPGA) and the enormous quantities of logic cells to obtain complete applications of neural networks, with real-time training on the fly, and with topologies (size) that were impossible to achieve just two years ago.

Of course other certain drawbacks must be overcome in addition to size when we face up to the implementation of a real application. Decrease the throughput of the training is very important for the MLP in the transmit part of the prediction system and we attain it with the pipeline version of the algorithm; but also we must resolve the reception part, and for this, the latency of the training must be improved. Therefore we have to increase the parallelism of our implementation in the future.

References

-
- [1] S. Hauck, "The Roles of FPGAs in Reprogrammable Systems" *Proceedings of the IEEE*, 86(4), April 1998, pp. 615-638.
 - [2] R. Gadea, A. Mocholí, "Forward-backward parallelism in on-line backpropagation", *Lecture Notes in Computer Science* , Vol. 1607, June 1999, pp. 157-165.
 - [3] B. Burton, R.G. Harley, G. Diana, and J.R. Rodgerson, "Reducing the computational demands of continually online-trained artificial neural networks for system identification and control of fast processes" *IEEE Transaction on Industry Applications*, Vol 34. no.3, May/June 1998, pp. 589-596
 - [4] Marusic, S., Deng, G. "A Neural Network based Adaptive Non-Linear Lossless Predictive Coding Technique". *Signal Processing and Its Applications, 1999. ISSPA '99*.
 - [5] Howard, P.G., Vitter, J.S, "Fast Progressive Lossless Image Compression". *Image and Video Compression Conf. Symp. On Electronic Imaging*, SPIE-2186, San Jose, C. A., 1994

Reconfigurable Systems in Education

Valery Sklyarov and Iouliia Skliarova

University of Aveiro, Department of Electronics and Telecommunications, IEETA,
3810-193 Aveiro, Portugal
`{skl, ioulia}@det.ua.pt`

Abstract. This paper describes methods and tools that have been used for teaching disciplines dedicated to the design of reconfigurable digital systems. It demonstrates students' projects, disseminates experience in the integration of different disciplines, and gives examples of stimulating student activity. A set of animated tutorials for students that are available on WebCT with a number of practical projects that cover a variety of topics in FPGA-based design can be seen as the most valuable contribution to the area considered.

1 Introduction

Today, FPLDs are considered to be an alternative to ASICs, and they have already been very efficiently used in a large number of practical applications, such as co-processors for general-purpose computers, problem-oriented digital systems, embedded controllers, and so on. Since the scope of potential applications is growing rapidly, a large number of well-prepared engineers are needed in the relevant areas. Thus, new trends must be reflected in the respective pedagogical activity of universities concerned with these subjects. Note that the domain of reconfigurable systems design is very dynamic and many-sided. Periodic upgrading of the pedagogical plans is essential in order to mirror recent advances in FPLD technology, design methods and CAD tools. Consequently, it is very important to provide an exchange of experience in pedagogical activity. This paper describes a methodology that has been used for more than 5 years for teaching reconfigurable systems at the Department of Electronics and Telecommunications of Aveiro University.

2 Basic Directions in Teaching Reconfigurable Systems

The major topics that have been considered as a base for the disciplines within the scope of reconfigurable systems design include design tools, prototyping boards with FPLDs and methods that provide an understanding of how FPLD-based circuits communicate with the external world, i.e. with peripheral electronic devices.

Design tools have been selected so that it is possible to learn both system-level specification languages and traditional HDL design flows. Two system-level specification languages, SystemC [1] and Handel-C [2], have been taught to students. SystemC is a library that permits hardware components to be modeled using a

standard C/C++ compiler and it was considered just at a description level. Handel-C permits digital circuits to be described in a C-based style and synthesized using Celoxica tools [2]. The latter together with Xilinx ISE 5.2 software [3] were used for the design, modeling, and implementation of digital systems based on FPGAs.

A number of *prototyping boards* have been employed for testing circuits in hardware. Recently used boards contain FPGAs from two Xilinx series; the Spartan-II/Spartan-IIIE (the boards TE-XC2Se [4] and RC100 [2]) and the Virtex-EM/Virtex-IIPro (the PCI boards ADM-XRC and ADM-XPL [5]).

For the majority of practical applications, FPGA-based circuits have to interact with external devices. Three groups of *interfaces* have been studied. They are widely-used standard protocols, such as parallel, RS232, USB, etc; PCI; and those that provide interactions between FPGAs and external microchips, such as static memory, LCD controllers, microprocessors, etc.

Three kinds of applications have been proposed to students, combinatorial accelerators, hardware/software co-simulation, and processors with customized sets of instructions. During practical classes, students have to implement individual blocks for the systems mentioned above. Complete systems have to be constructed within semester projects.

The classes given to the students have a number of distinct features. The lectures are well-covered by a set of animated tutorials and examples of FPGA-targeted projects. All the required supplementary materials are available on the WebCT.

3 Tutorials and WebCT

In order to maximize the effectiveness of the classes, the students should have access to all the required materials. The materials can be divided into the following basic groups: manuals about FPGAs and the corresponding computer-aided design systems (ISE 5.2, ModelSim and DK1 in our case); supplementary documents (manuals on peripheral microchips, specification of interfaces, etc.); descriptions of auxiliary equipment that is required, such as logic analyzers; methods and tools that increase the productivity of education through facilities such as the extensive use of animated tutorials and providing materials for distance learning. For example, the section *Tutorials* in the public domain of [6] contains 10 examples. Each of them includes an animated tutorial in PowerPoint and examples of ISE 5.2 projects for VHDL-based design flow. They address the following topics:

- A sequence of steps for beginners to design, implement, and test in an FPGA, a trivial circuit based on a very simple VHDL code;
- The interaction of an FPGA with components such as LEDs, push buttons and DIP switchers through a CPLD;
- Design and functionality of a simple arithmetical circuit, which displays the results on a LCD (2 lines with 16 characters each). This tutorial explains how to use the ISE 5.2 schematic editor, the Core Generator, Xilinx libraries, hierarchical design, combining different components in the same project, interaction with an LCD controller, etc.;
- Synthesis of finite state machines (FSM) and the use of Xilinx StateCAD;

- Synthesizable VHDL in alphabetical order. It allows any letter (for instance, *G - Generic*) to be chosen to learn the corresponding topic, to run a relevant project in ISE 5.2, to load the generated bitstream into the FPGA, and to test the circuit in hardware;
- Simulation of VHDL descriptions in ModelSim;
- Interacting with a touch panel [7] through an RS232 interface;
- Parameterizable (generic) VHDL code for a reprogrammable FSM and examples of the FSM interacting with a datapath;
- FPGAs interacting with two LCDs (4 lines 20 characters in each and 2 lines 16 characters in each). The examples explain a number of control sequences for scrolling, editing, etc.;
- A very detailed and relatively complex example that explains how to use recursive hierarchical control. It shows a C++ program that illustrates algorithms, describes two recursive sub-algorithms, demonstrates how to construct a recursive hierarchical FSM that builds a special binary tree from a sequence of arbitrary integers and sorts integers on the basis of this binary tree. The results (i.e. the sorted data) are displayed on an LCD.

All the tutorials make use of different animation effects available in PowerPoint (Windows XP). This enables many processes to be demonstrated in a step by step manner, such as all the events appearing in each clock cycle; how VHDL code activates these events and reacts; how various bits in interface lines are changed, etc.

Similar tutorials have been prepared for explaining the functionality of FPGA DLLs, demonstrating an interaction with a mouse, a keyboard, a VGA monitor, and for a number of Handel-C topics. They are available in English and in Portuguese.

4 Stimulation of Student Activity and Integration with Other Disciplines

Two types of evaluation have been proposed to the students. The first is a traditional examination. The examination can be replaced by the second type of evaluation through an individual project that is suggested in the middle of a semester. According to the requirements, students have to design, implement, and test a digital system based on commercially available FPGAs. Potential projects are discussed with the students. This allows a task to be chosen from the area that is of the most interest to a particular student. The results of the projects have to be demonstrated in a working FPGA-based device and presented in a written report before the end of the examination period. The best projects are recommended for publications in the magazine “*Electrónica e Telecomunicações*” that is issued by the Department. All these publications can be accessed through the WebCT [6].

The methodology provides a very important opportunity, especially for final year students. It permits integration between different disciplines to be established. The group of disciplines considered in this paper suggests methods and tools; the other groups of disciplines offer applications. This approach provides additional motivation to the students because reconfigurable systems can be linked with practical work in other disciplines that particular students are interested in.

Our experience has shown that there are some auxiliary methods that stimulate the work of students. First of all, the result of the work should be visible and touchable especially at the beginning. That is why it is reasonable to use stand-alone boards that are cheap and provide a number of interactions such as communication between an FPGA and a mouse, a keyboard, a VGA monitor, LCD panels, etc. Only after some period of time hidden PCI-based prototyping boards can be used. As a rule they are much more expensive and do not permit the results to be appreciated visually. The work is organized through a set of API functions and it looks like programming. On the other hand, PCI-based boards contain much more powerful FPGAs and they are recommended for experienced students, especially those in Ph.D. and M.Sc. scholarships.

For example, in 2002/2003 a combinatorial processor that implements an exact algorithm for solving the covering problem has been proposed as a project for final year students. Initially, a stand-alone RC100 prototyping board with an FPGA from the Spartan-II family was used. All the individual components of the project were described in Handel-C and carefully tested using available peripheral devices and drivers supplied by Celoxica. Finally the entire circuit was implemented. After that the same circuit was constructed using the ADM-XPL PCI board [5] containing FPGA XC2VP7 of Virtex-II Pro family. This circuit allows much more complicated problems to be resolved [8]. The previous experience gained with a relatively cheap stand-alone board provides a basis for achieving results rapidly, and very similar methods and tools can be used for the most advanced FPGAs available on the market.

5 Conclusion

This paper disseminates experience in teaching reconfigurable systems, summarizes the pedagogical methods that have been adopted, the organization of classes, the basic directions of student's projects, and many other aspects. This work was supported by the grants FCT-PRAXIS XXI/BD/21353/99 and POSI/43140/CHS/2001.

References

1. SystemC. [Online]. Available: <http://www.systemc.org/>
2. Handel-C, DK1, RC100. [Online]. Available: <http://www.celoxica.com/>
3. ISE 5.2, Xilinx series FPGA. [Online]. Available: <http://www.xilinx.com/>
4. Spartan-II Development Platform. [Online]. Available: www.trenz-electronic.de
5. Alpha Data. [Online]. Available: <http://www.alpha-data.com>
6. <http://webct.ua.pt>, "2 semester", the discipline "Computação Reconfigurável", public domain is indicated by the letter "*i*" enclosed in a circle. Login and password for access to the protected section can also be provided (via e-mails: skl@ieeta.pt, ioulia@det.ua.pt)
7. EA KIT240-7, EA DIP204-4. Electronic Assembly. Available: <http://www.lcd-module.de>
8. Sklyarov, V., Skliarova, I., Almeida, P., Almeida, M.: High-Level Design Tools for FPGA-based Combinatorial Accelerators. Proceedings of FPL'2003 (Lisbon, 2003)

Data Dependent Circuit Design: A Case Study

Shoji Yamamoto, Shuichi Ichikawa, and Hiroshi Yamamoto

Department of Knowledge-based Information Engineering

Toyohashi University of Technology

1-1 Hibarigaoka, Tempaku, Toyohashi, Aichi 441-8580, JAPAN

{shoji, ichikawa, i8hyama}@ich.tutkie.tut.ac.jp

<http://ich.tutkie.tut.ac.jp/en/>

Abstract. Data dependent circuits are logic circuits specialized to specific input data. They are smaller and faster than the original circuits, although they are not reusable and require circuit generation for each input instance. This study examines data dependent designs for subgraph isomorphism problems, and shows that a simple algorithm is faster than an elaborate algorithm. An algorithm that requires many hardware resources consumes an accordingly longer circuit generation time, which outweighs the performance advantage in execution.

1 Data Dependent Hardware

If any input of a logic circuit turns out to be constant, the circuit can be reduced. For example, if any inputs of an AND gate turn out to be zero, the output becomes zero (*constant propagation*). This reduction can be applied recursively, consequently reducing the logic scale of the circuit. The derived circuit would operate at a higher frequency than the original, because the logic depth and wiring delay would also be reduced by this reduction.

Since the consequent circuit becomes dependent on the input data instance, such a circuit is called a *data dependent circuit* in the following discussion. The obvious drawback of a data dependent approach is that the derived circuit is *not reusable*. This naturally means that (1) the circuit must be generated for each input instance, and (2) reconfigurable devices such as FPGA must be used.

The total execution time T of a data dependent circuit is given by the sum of the circuit generation time T_{gen} and the execution time T_{exec} . T_{gen} consists of the time for HDL source code generation, logic synthesis, technology mapping, placement, routing, and FPGA configuration. T_{gen} depends on the logic scale, since a larger circuit usually requires accordingly larger generation time. T_{exec} is the product of cycle count and cycle time. Here, the cycle count depends on the algorithm, and the cycle time depends on the implementation. Fast algorithms can make T_{exec} smaller, but they often require more hardware resources and make T_{gen} larger. The total execution time T is thus not so obvious without empirical studies.

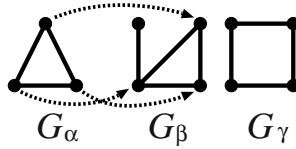


Fig. 1. Subgraph Isomorphism

2 A Case Study: Subgraph Isomorphism Problem

Hereafter, we examine a problem called a *subgraph isomorphism problem* as an example application. A subgraph isomorphism problem is a simple decision problem. Given two graphs G_α and G_β , it is determined whether G_α is isomorphic to any subgraph of G_β (Fig. 1). In Fig. 1, G_β has a subgraph that is isomorphic to G_α , while G_γ does not.

Ullmann [1] proposed a depth first search algorithm with a smart pruning procedure (*refinement procedure*) for subgraph isomorphism problems. He pointed out that his procedure can be implemented with parallel hardware, but Ichikawa et al. [2] later revealed that his circuit is too large to handle practical problems with the state-of-the-art FPGAs. Ichikawa, Udorn, and Konishi [3] proposed a new algorithm (Konishi's algorithm), which has a simpler pruning procedure than Ullmann's. Konishi's algorithm is generally slower than Ullmann's, but it can be implemented in a much smaller logic circuit than Ullmann's.

Ichikawa et al. [4] [5] previously suggested that data dependent implementations of Ullmann's circuit can be much smaller than the original circuit. The present study confirms this by showing the evaluation results with a Xilinx Virtex-II FPGA.

A data dependent Konishi circuit has not yet been investigated. This study also shows the implementation results of data dependent Konishi circuits, and compares them with data dependent Ullmann circuits. As the original Konishi circuit [3] was not suited for data dependent implementation, we designed a brand-new logic circuit with Konishi's algorithm in this study. In this design, the adjacency check circuits are implemented by parallel hardware. Although this design is an interesting example of a data dependent circuit, we do not have the space to detail it here.

3 Evaluation

This section describes the evaluation results for data dependent circuits. Each result in this section is the average of 100 pairs of G_α and G_β , which are randomly generated. Let p_α and p_β be the number of vertices of G_α and G_β , respectively. We only deal with the cases of $p_\alpha = p_\beta$ in this study. We implemented data dependent circuits for various graph sizes, and measured the execution time on a XC2V1000 FPGA. Our evaluation environment is summarized in Table 1.

Table 1. Evaluation Environment

Item	Note
Circuit Generation	Athlon XP 1800+, Memory 1GB, Windows2000 SP3 Synopsys FPGA Compiler II (2001.08-FC3.7) Xilinx ISE 4.2i (Target device: Virtex-II XC2V1000)
FPGA platform	Insight MicroBlaze Development Kit (XC2V1000, 24MHz)
Software	Celeron 1.2GHz, Memory 512MB, Red Hat Linux 7.2
Implementation	Written in C, compiled with gcc-2.95.3

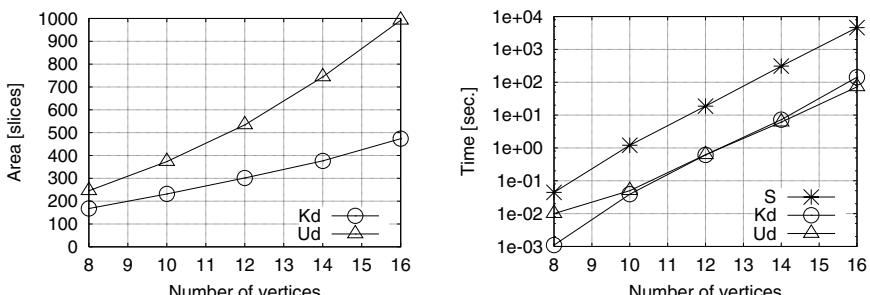
We examined the cases of $(ed_\alpha, ed_\beta) = (0.3, 0.6)$, where ed_α and ed_β indicate the edge density of G_α and G_β , respectively. Edge density ed is defined by the equation $ed = (2q)/(p(p - 1))$, assuming that p is the number of vertices and q is the number of edges. In other words, ed is the ratio of the number of edges to that of the perfect graph K_p . It is clear that $0 \leq ed \leq 1$ holds.

In this study, we examine 4 designs. **Ko** and **Uo** designate the original Konishi circuit and the original Ullmann circuit, respectively. **Kd** and **Ud** designate the data dependent versions of a Konishi circuit and Ullmann circuit for the above-mentioned input graph set.

Figure 2 (left) displays the average logic scale of Ud and Kd, shown by the number of *slices* of Virtex-II FPGA. For the same number of vertices ($8 \leq p_\alpha = p_\beta \leq 16$), the logic scale of Uo is estimated to be 2.4–3.6 times larger than Ud. Meanwhile, Ko is 1.7–1.9 times larger than Kd.

Figure 2 (right) displays the average execution time on XC2V1000 FPGA with a 24 MHz system clock. For comparison, the software implementation of Ullmann's algorithm was also evaluated. The evaluation environment is summarized in Table 1. The execution time of software is denoted by **S** in Fig. 2 (right). For $p_\alpha = p_\beta = 16$, Ud and Kd is 63 and 32 times faster than S, respectively. This performance gain becomes larger in larger graphs.

Figure 3 (left) displays the circuit generation time. It is readily seen that the circuit generation time of Ud is far larger than that of Kd. This comes from the difference of logic scale. Figure 3 (right) shows the average total execution time

**Fig. 2.** Logic Scale (left) and Execution Time (right)

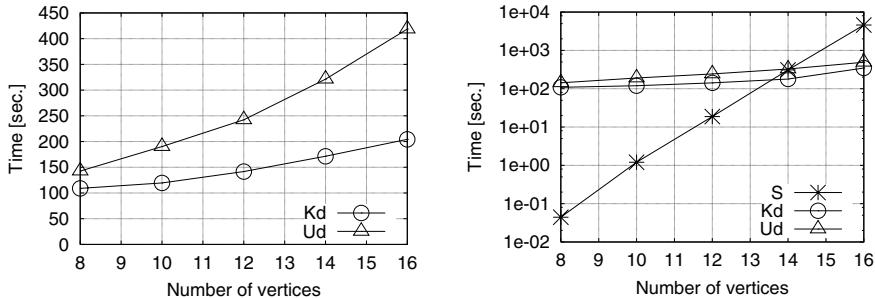


Fig. 3. Circuit Generation Time (left) and Total Execution Time (right)

of U_d and K_d , which is the sum of the circuit generation time and the execution time. The software execution time (S) is also shown for comparison.

U_d and K_d are faster than the software for $p_\alpha = p_\beta > 14$, even reckoning the circuit generation time. When $p_\alpha = p_\beta = 16$, K_d and U_d are 13.3 and 9.4 times faster than the software, respectively. As is readily seen, this performance advantage becomes larger when p_α and p_β are larger. It is also worth noting that K_d is faster than U_d after all, because the long circuit generation time of U_d outweighs its performance advantage over K_d .

Acknowledgment

This work was partially supported by a grant from the Okawa Foundation for Information and Telecommunications. The custom circuits in this study were designed with Synopsys CAD tools through the chip fabrication program of VDEC (the University of Tokyo).

References

1. Ullmann, J.R.: An algorithm for subgraph isomorphism. *J. ACM* **23(1)** (1976) 31–42
2. Ichikawa, S., Saito, H., Udorn, L., Konishi, K.: Evaluation of accelerator designs for subgraph isomorphism problem. In: Proc. FPL2000. LNCS1896, Springer (2000) 729–738
3. Ichikawa, S., Udorn, L., Konishi, K.: An FPGA-based implementation of subgraph isomorphism algorithm. *IPSJ Trans. High Performance Computing Systems* **41(SIG5)** (2000) 39–49 (in Japanese).
4. Ichikawa, S., Yamamoto, S.: Data dependent circuit for subgraph isomorphism problem. In: Proc. FPL2002. LNCS2438, Springer (2002) 1068–1071
5. Ichikawa, S., Yamamoto, S.: Data dependent circuit for subgraph isomorphism problem. *IEICE Trans. Information and Systems* **E86-D(5)** (2003) 796–802

Design of a Power Conscious, Customizable CDMA Receiver

Maurizio Martina, Andrea Molino, Mario Nicola, and Fabrizio Vacca

Dipartimento di Elettronica, Politecnico di Torino,
Corso Duca degli Abruzzi 24, 100129 Torino, Italy,
{maurizio.martina, andrea.molino}@polito.it,
{mario.nicola, fabrizio.vacca}@polito.it

Abstract. 2G wireless systems have gained a widespread diffusion. Due to this fact, the transition to 3G ones can be critical. A possible solution to the interoperability problem can come from the Software Defined Radio paradigm. In this paper a complete, reconfigurable CDMA receiver implementation over a Xilinx XCV300E FPGA is described.

1 Introduction

Software Defined Radio (SDR) paradigm [1] is one of the most interesting topics in wireless communications, since it can help the transition from 2G to 3G. SDR main idea is to employ the reconfigurability as a key feature for implementation of wireless terminals. Existing works [2] have demonstrated that actually a DSP implementation isn't feasible, due to high processing demand. On the other hand, FPGAs grant good performances and reconfigurability, but waste a great amount of static power [3]. So FPGAs are an interesting platform for SDR: in this paper a reconfigurable CDMA receiver architecture based on FPGAs is described, and some strategies to dynamically manage the power consumption are advised.

2 CDMA Fundamentals

CDMA is a technique to access a shared channel, spreading transmitted signals with orthogonal codes [4]. Spreading can be performed multiplying the signal $x[n]$ and the code $w[n]$. The $x[n]$ rate is the *bit-rate*, while $w[n]$ rate is the *chip-rate*: their ratio is called *spreading factor* (G). The receiver can recover the information from k -th user correlating over G chips the received bit stream r and the despreading code w_k . CDMA poses many design challenges, including the need of synchronization between received signal and despreading sequence. Now an implementation of a BPSK-CDMA receiver is presented (figure 1(a)).

3 Receiver Architecture

3.1 Digital Down Conversion

The DDC is usually a critical block since input signal is at the ADC rate. To perform filtering operation at this rate, Cascaded Integrator Comb (CIC) filters [5]

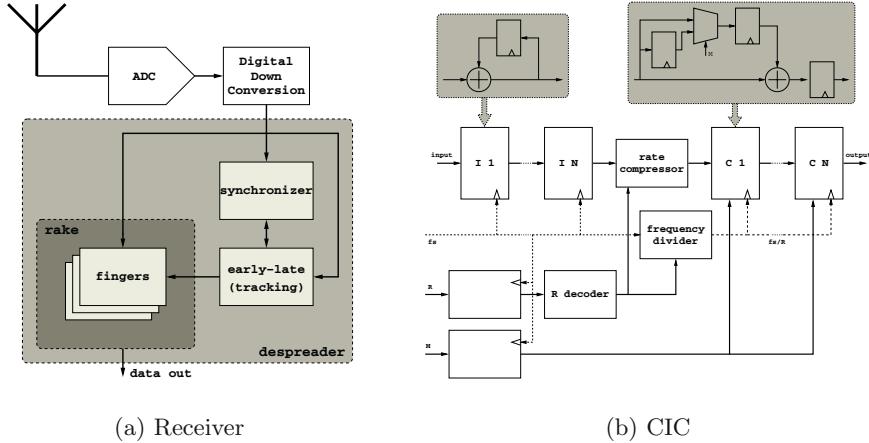


Fig. 1. System architecture with detailed view of downconverter

have been used (figure 1(b)). A CIC filter is made of integrators and combs connected in cascade. Integrator can be implemented using an accumulator, while the comb stage consists in a digital programmable delay chain followed by a subtractor. In our architecture both downsampling ratio (R), and length of comb's delay chain (M) can be programmed. Some previous works concerning hardware implementations of CIC filters can be found (e.g. [6]), but these architectures are optimized to be implemented in ASIC.

3.2 Synchronizer

An exhaustive search over all possible synchronization schemes is used in synchronizer (figure 2(a)): it consists in evaluating energy of correlation, since it shows a peak when synchronization is reached [4]. Correlation is evaluated for both quadrature and in-phase branches in a proper number of chip cycles (N_{test}) to achieve reliable results, then the contributes are squared and added together. After an appropriate number of measures (L), the final result is compared with a programmable threshold. In order to increase system's flexibility, both N_{test} and L can be programmed by the control unit. Code generator is implemented using reconfigurable Linear Feedback Shift Registers (LFSR) that can be enabled/disabled by a control unit: this approach is different from existing works suggesting use of clock tree structures [7].

3.3 Early-Late DLL

After synchronization is reached, an Early-Late DLL is needed to keep the receiver tracked. This block (figure 2(b)) is made of four main functional blocks: the correlation detector, the loop filter, the Numerically Controlled Oscillator

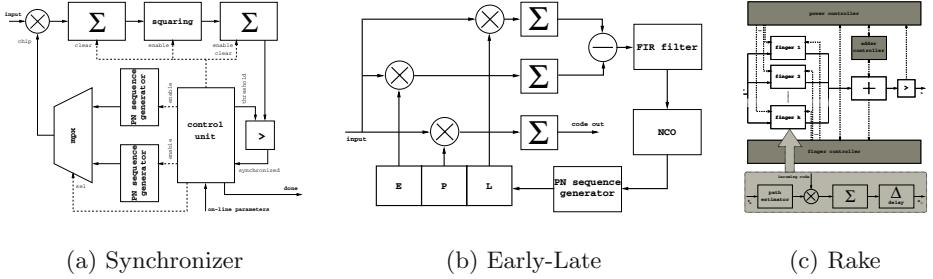


Fig. 2. Synchronization units and rake receiver

(NCO) and the Early-Punctual-Late code generator. DLL loop filter has been implemented as an FIR filter with a reconfigurable number of taps and a programmable number of computing resources. The filter output controls the NCO block, an accumulator with programmable increment, which is the block devoted to drive the sequence generator. The tracking loop is closed over two correlators, evaluating correlation of signal respectively with early and late replica of local code. The difference between these two values is the filter input signal.

3.4 Rake Receiver

The rake receiver is devoted to increase the received signal to noise ratio in a multipath environment. In particular it estimates both the attenuation and the delay parameters for a set of given paths: equations can be found in literature [4]. It's worth noting that estimation is obtained through an average over N_p chip: this parameter, crucial to satisfy both high performance and precision demand, is programmable in this architecture. In the block scheme (figure 2(c)) three main parts can be identified: the rake receiver core, composed by a programmable number of *fingers*, the adder, and the programmable comparator. In each finger phase delay and attenuation are estimated, then outputs from distinct fingers are added together, obtaining a value that can be used to recognize received bit. To reduce power consumption, fingers are turned off and on by a power controller, accordingly to the energy of received signal.

4 Results and Conclusions

This architecture supports both “on-line” and “off-line” reconfigurability: the former is the most valuable one and requires more resources, while the latter needs an FPGA reconfiguration to change parameters. In this work the “off-line” reconfigurability is used to pose some well known bounds, given a particular scenario (e.g. UMTS, CDMA2000, ...), while the “on-line” parameters enable the support for different operative profiles (table 1).

Table 1. Reconfigurable parameters description

Block	Parameter	Description	Type on-line off-line
all blocks	#PIPE DATA_W G CODE_LENGTH	Pipeline levels in data path unit Width of elaborating data Spreading/Despread factor (B_{chip}/B_{bit}) Despread code length	* * * *
Digital Down Converter	R M	Output data rate (rate downsample factor) Differential Delay	* *
Synchronizer	THR $N_{test,L}$	Threshold for synchronization decision Length of energy measure	* *
Early-Late	#TAP N_MAC COEFF[] NCO_INC	Number of taps in loop FIR filter Number of MAC units for the loop filter Coefficient values of loop FIR filter Increment value in NCO block	* * * *
Rake Receiver	#FGR #AFGR N_p	Number of fingers Number of turn-on (active) fingers Number of chips used for the estimation	* * *

Starting from this parametric description, the whole CDMA receiver architecture has been tested and validated using the CoCentric System Studio environment from Synopsys. The logical synthesis has been carried out with Synplify Pro v7.0 by Synplicity. The complete flow over the FPGA has been performed with Xilinx ISE tools. Whole receiver occupies roughly 88% of a Xilinx XCV300E, achieving a maximum 96 MHz clock frequency and an estimated power consumption of 410 mW plus 546 mW of static power dissipated by FPGA. These results have been obtained using a 4 stages DDC, 8 tap FIR for tracking, and 4 fingers rake. Data are in 16 bit, fixed point format.

The experimental results prove the feasibility of a 3G wireless receiver on a medium-sized FPGA. This can open the possibility of a systematic reconfigurable fabrics exploitation on wireless communications terminals.

References

1. Buracchini, E.: The software radio concept. *IEEE Communications Magazine* **38** (2000) 138–143
2. Dent, P.: W-CDMA reception with a DSP based software radio. *International Conference on 3G Mobile Communication Technologies* (2000) 311–315
3. Martina, M., Maserà, G., Piccinini, G., Vacca, F., Zamboni, M.: Energy Evaluation on a Reconfigurable Multimedia-Oriented Wireless Sensor. *International Conference on Field Programmable Logic and Application* (2002)
4. Viterbi, A.: CDMA: principles of spread spectrum communications. Addison-Wesley Publishing Company (1995)
5. Hogenauer, E.: An economical class of digital filters for decimation and interpolation. *IEEE Trans. Acoustic, Speech and Signal Processing* **29** (1981) 155–162
6. Key-Yong, K., et al.: Efficient High Speed CIC Decimator Filter. In: *IEEE Asic Conference*. (1998) 251–254
7. Kitsos, P., et al.: A reconfigurable linear feedback shift register (LFSR) for the Bluetooth system. *IEEE International Conference on Electronics, Circuits and Systems* **2** (2001) 991–994

Power-Efficient Implementations of Multimedia Applications on Reconfigurable Platforms¹

K. Tatas, K. Siozios, D. Soudris, and A. Thanailakis

VLSI Design and Testing Center, Department of Electrical and Computer Engineering,
Democritus University of Thrace, 67100, Xanthi, Greece
[{ktatas,ksiop,dsoudris,thanail}](mailto:{ktatas,ksiop,dsoudris,thanail}@ee.duth.gr)@ee.duth.gr

Abstract. The power-efficient implementation of motion estimation algorithms on a system comprised by an FPGA and an external memory is presented. Low power consumption is achieved by implementing an optimum on-chip memory hierarchy inside the FPGA, and moving the bulk of required memory transfers from the internal memory hierarchy instead of the external memory. Comparisons among implementations with and without this optimization, prove that great power efficiency is achieved while satisfying performance constraints.

1 Introduction

In this paper data memory hierarchy power exploration at the high levels of design and register transfer (RT) level implementation are combined in order to achieve the optimum implementation in terms of power consumption while meeting performance constraints. An FPGA has been used for the implementation of two common motion estimation algorithms with and without memory power optimization.

2 Target Architecture

The architecture we have considered is illustrated in Fig. 1. It consists of an FPGA which implements the required logic and an external (off-chip) memory, which stores the data of the application.

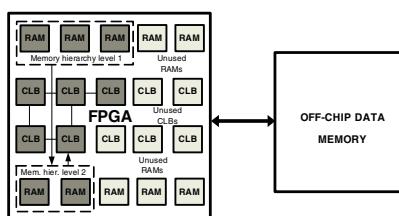


Fig. 1. Target architecture

¹ This work was partially supported by the project IST-34793-AMDREL which is funded by the E.C.

3 Target Application and Optimization Methodology

We have chosen two popular multimedia kernels for implementation, namely the full-search (FS) and hierarchical search (HS) motion estimation algorithms [1], [2], [3].

The optimization methodology is based on the fact that a large, off-chip memory consumes greater power per memory transfer, than a small, on-chip one, therefore the methodology attempts to move the greatest number of transfers from the off-chip memory to on-chip ones, by locating data that are used often.

It has been proven [4], [5] that a total of 21 possible data memory hierarchies for our applications exist. These candidate hierarchies are implied by the 21 possible data-reuse transformations, which are applied on the high-level description of the kernels. The power consumption of each memory hierarchy was estimated using Landman's memory model [6]. We have selected the one that exhibited the lowest power consumption. It implies two levels of on-chip memory hierarchy: A line of candidate blocks which is loaded from the external memory and a single candidate block is loaded from the line of candidate blocks.

4 Experimental Results

For illustration purposes, greyscale frames of 144×176 pixels were used, therefore an external memory of $2 \times 25344 \times 8$ bit is necessary in order to contain two such greyscale type of frames. The selected low-power transformation, was described in VHDL and implemented in various Xilinx devices. For the on-chip memory hierarchies plus the motion vector storing and the subsampled frames storing in the hierarchical search motion estimation kernel, an appropriate number of Xilinx BlockSelectRAMs were used in their 512×8 configuration [7].

Power consumption for the FPGA was estimated using Xpower [8]. The power consumption of the external memory was estimated using Landman's model [6].

Table 1 provides the total power consumption of all implementations in detail, at a frequency of 25 MHz. The last column presents the percentage gains of a specific optimized FPGA implementation in comparison to the corresponding non-optimized implementation on the same device.

Table 2 shows the device utilization for all implementations in number of slices and number of BlockRAMs used. The number of required resources (slices/BlockRAMs), the number of total available resources and the percentage of utilization is given.

Performance measurements can be seen in Table 3. Let us consider the performance measures for slow-motion video (10 FPS), video conference (15 FPS), video file transfer (15 FPS) and digital video (30 FPS) according to [5]. The original FS algorithm can meet all applications except full-motion digital video in most devices.

The power-optimized FS scheme cannot meet the real-time constraint in any application due to the performance overhead. The original HS motion estimation, on the other hand, more than adequately satisfies the performance constraints in all devices.

The power-optimized version barely satisfies the performance required for full-motion video, but it is more than adequate for the remaining applications.

Table 1. Total power consumption comparison of alternative implementations

Application	Device	FPGA power (mW)	External memory power (mW)	Total power (mW)	Power gain (%)
FS (original)	xc2s15	160.88	703.76	864.64	
	xc2s200	275.81	703.76	979.57	
	xcv50	217.25	703.76	921.01	
	xcv1000	343.62	703.76	1047.38	
	xcv400e	586.49	703.76	1290.25	
	xc2v10000	593.02	703.76	1296.78	
FS (optimized)	xc2s15	120.38	211.93	332.31	61.56
	xc2s200	239.70	211.93	451.63	53.89
	xcv50	172.22	211.93	384.13	58.29
	xcv1000	295.2	211.93	507.13	51.58
	xcv400e	555.66	211.93	767.59	40.50
	xc2v10000	590.98	211.93	802.91	38.08
HS (original)	xcv400e	637.61	1093.88	1731.49	
	xc2v10000	592.85	1093.88	1686.73	
HS (opt.)	xc2v10000	1170.53	110.39	1280.92	24.05

Table 2. Area comparison of alternative implementations

Application	Device	Slice count (% utilization)	BlockRAM count (% utilization)
FS (original)	xc2s15	164/192 (85%)	2/4 (50%)
	xc2s200	164/2,352 (6%)	2/14 (14%)
	xcv50	164/768 (21%)	2/8 (25%)
	xcv1000	164/12288(1%)	2/32 (6%)
	xcv400e	164/4800 (3%)	2/40 (5%)
	xc2v10000	164/61440 (0.002%)	2/192 (0.01%)
FS (optimized)	xc2s15	166/192 (86%)	4/4 (100%)
	xc2s200	166/2352 (7%)	4/14 (21%)
	xcv50	166/768(22%)	4/8 (50%)
	xcv1000	166/12288 (1%)	4/32 (12%)
	xcv400e	166/4800 (3%)	4/40 (10%)
	xc2v10000	166/61440	4/192 (0.02%)
HS (original)	xcv400e	761/768 (99%)	36/40 (90%)
	xc2v10000	761/61440 (1%)	36/192 (18%)
HS (opt.)	xc2v10000	16817/61440 (27%)	161/192 (84%)

5 Conclusions

A power-efficient implementation of two popular multimedia applications based on exhaustive high-level data memory power exploration and RTL design on FPGAs was presented. Simulation results indicated significant power gains at the expense of performance due to larger hardware complexity.

Table 3. Performance comparison of alternative implementations

Application	Device	# cycles	Clock freq. (MHz)	FPS
FS(original)		5702400	25	4.38
	xc2s50		Max (87.030)	15.24
	xc2s200		Max(80.97)	14.19
	xcv50		Max(69.152)	12.11
	xcv1000		Max(91)	15.94
	xcv400e		Max(95)	16.64
	xc2v10000		Max(103)	18.0
FS (optimized)		9554688	25	2.61
	xc2s50		Max (54.44)	5.68
	xc2s200		Max(57.47)	6
	xcv50		Max(54.8)	5.72
	xcv1000		Max(51.16)	5.34
	xcv400e		Max(66.29)	6.92
	xc2v10000		Max(81)	8.45
HS (original)		320544	25	77.99
	xcv400e		Max(44.08)	137.53
	xc2v10000		Max(78.58)	245.13
HS (opt.)	xc2v10000	950994	Max(25.24)	26.44

References

1. V. Bhaskaran and K. Kostantinides: Image and Video Compression Standards. Kluwer Academic Publishers (1998)
2. K. M. Nam, J.-S. Kim, Rae-Hong Park, and Y. S. Shim: A fast hierarchical motion vector estimation algorithm using mean pyramid. IEEE Transactions on Circuits and Systems for Video Technology, vol. 5, no. 4 (1995) 344–351
3. Peter Kuhn: Algorithms, Complexity Analysis and VLSI, Architectures for MPEG-4 Motion Estimation. Kluwer Academic Publishers (1999)
4. D. Soudris, N. D. Zervas, A. Argyriou, M. Dasycgenis, K. Tatas, C. Goutis, A. Thanailakis: Data-Reuse and Parallel Embedded Architectures for Low-Power, Real-Time Multimedia Applications. IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), Göttingen, Germany (2000) 243- 254
5. N. D. Zervas, K. Masselos, C.E. Goutis: Data-reuse exploration for low-power realization of multimedia applications on embedded cores. Proc. Of 9th Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS'99) (1999) 71-80
6. P. Landman: Low-power architectural design methodologies. Doctoral Dissertation, U.C. Berkeley (1994)
7. <http://direct.xilinx.com/bvdocs/publications/ds003.pdf>
8. http://support.xilinx.com/support/sw_manuals/xilinx4/manuals.pdf

A VHDL Library to Analyse Fault Tolerant Techniques*

P.M. Ortigosa, O. López, R. Estrada, I. García, and E.M. Garzón

Dpt. of Computer Architecture and Electronics, University of Almería,
Almería 04120, Spain. {pilar,inma,ester}@ace.ual.es

Abstract. This work presents an initiative to teach the basis of fault tolerance in digital systems design in undergraduate and graduate courses in electrical and computer engineering. The approach is based on a library of characteristic circuits related to fault tolerance techniques which has been implemented using a Hardware Description Language (VHDL). Due to the properties of the design tools associated to these languages, this approach allows with ease: (1) to implement faults tolerant digital systems; (2) to determine the behaviour of system when faults are presented; (3) to evaluate the additional resources and response time linked to any fault tolerance technique in the laboratory.

1 Introduction

Due to the great importance of faults tolerance techniques, current electrical and computer engineers have to know the fundamental topics related to this field. Therefore, Testing and Fault Tolerance belong to the set of areas associated with the body of knowledge of Computer Engineering defined by the Joint IEEE Computer Society and ACM Task Force on Computing Curricula [1].

Most fault tolerance techniques are based on the addition of not necessary resources for a common system function. These additional resources can be defined as a collection of characteristic circuits only used in this context. Our approach is based on the design of this specific library by a hardware description (VHDL in particular) to facilitate: (1) the design of systems to which fault tolerant techniques will be applied; (2) the insertion of redundant hardware systems; (3) the design, instantiation and use of generic systems; (4) the analysis of the area or logic cells needed to implement a determined system, and consequently, the analysis of the additional resources when fault tolerant techniques are used; (5) the analysis of the changes in the system response time when applying any of the fault tolerant technique; (6) the testing of the different fault tolerant techniques by using test bench included in the library and specifically designed to be used in the educational context; and (7) the development of the laboratory exercises due to the availability in a library of all basic components needed for the design of fault tolerant systems.

* This work was supported by the Ministry of Education of Spain (TIC2002-00228)

Table 1. Main hardware redundancy techniques and the circuits for implementing

<i>Passive hardware redundancy</i>	
Technique	Circuits
N-Modular Redundancy (NMR)	Majority Voters Mid-value selectors Flux-summer
<i>Active hardware redundancy</i>	
Technique	Circuits
Duplication with Comparison	Comparators
Standby Replacement or Splicing	Faults Detectors Reconfiguration Switches
Pair and a Spare	Comparators Faults Detectors Reconfiguration Switches
<i>Hybrid hardware redundancy</i>	
Technique	Circuits
N-Modular Redundancy with Spares	Comparators Faults Detectors Reconfiguration Switches Majority Voters
Triple-duplex Redundancy	Reconfiguration Switches Flux-summer
Sift-Out Modular Redundancy	Comparator/Faults-Detectors Collector

In this work, we present our own initiative to support the teaching of faults tolerance techniques. This initiative is based on a VHDL library of circuits related to fault tolerance techniques, and a set of exercises to develop in the laboratory. Section 2 describes the circuits included in the library. An illustrative example of fault tolerant circuits implemented with the library is described. Finally, the main conclusions are described in Section 3.

2 Circuits to Implement Fault Tolerance

One common approach to reduce the probability of a system failure is based on including redundancy. Most references [2–4] distinguish four kinds of redundancy (1) hardware, (2) information, (3) software and (4) time. Therefore, it must be emphasised that the redundancy usually has a relevant impact on system qualities, such as performance, size, weight, power consumption, reliability and so on. Thus, a measure of this impact must be included in the evaluation of systems with redundancy. This work is focused on hardware and information redundancy, as well as, a framework which provides measures of additional resources and response times related to fault tolerance techniques. Moreover, the framework provides a collection of characteristic circuits to implement fault tolerance.

Table 1 enumerates the main fault tolerance techniques based on *hardware redundancy* and the modules related to these techniques [2–4]. It must be em-

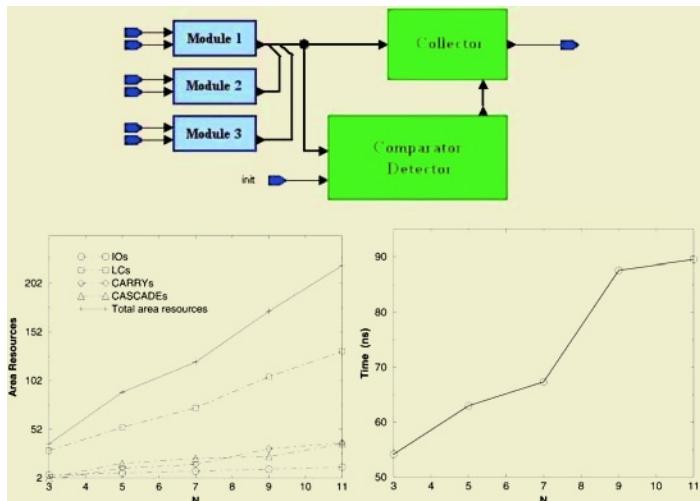


Fig. 1. Block diagram (top) and experimental results for with sift-out modular redundancy

phasised that some characteristic circuits can be related to several techniques, such as voters and comparators. So, a library that includes these characteristic circuits can help students to develop exercises related to this kind of fault tolerance techniques.

An approach to decrease the data corruption possibility is related to the addition of redundant information for data. This kind of redundancy is defined in [2–4] as *information redundancy*. Wide diversity of error detecting and/or correcting codes has been developed for specific applications. The following codes are the most extended [2–5]: parity codes, m-of-n codes, duplication codes, checksums, cyclic codes, arithmetic codes, Berger codes. These codes require length or code word greater than that of the original data. Thus, the systems must include additional resources to carry out the following stages (1) data encoding, (2) data processing, storage or transmission and (3) data decoding. Stages 1 and 2 are carried out by circuits (encoders and decoders) specified by the used code. Consequently, these specific modules to data encode and decode can be included in the proposed library to develop exercises related to these redundancy techniques.

Next, an example related to a hybrid hardware redundancy method which is called *sift-out modular redundancy* is described. This technique was proposed by de Sousa and Mathur [6], and it is described in classical references as [2]. Once the students know the theoretical model, they must design the fault tolerant system using both the components from fault tolerant library and the modules that are able to fault. These modules can either be instantiated from a basic digital circuit library or be designed by the students. As the specific components of the fault tolerant system (i.e. collector and comparator detector) have defined the number

of possible inputs as a generic parameter, the students can easily design systems with different redundancy levels using the same specific components (modifying the generic parameter) and just replicating new input modules. When a system is designed, it can be simulated in order to analyse its behaviour and performance and, later on, it can be synthesised for a particular FPLD. With the synthesis tool, analysis of consumed area and required times can be extracted.

Illustrating this kind of comparisons, Figure 1 shows block diagram and area and time results for five couples of collector and comparator/detector modules whose number of input modules (N) range from 3 to 11. Specifically, the consumed area resources for the FLEX10K70 board are shown. These resources can be classified by the number of Input/Output Ports (IOs), number of Combinatorial Logic Blocks (LCs), Carry bits (CARRYs) and Cascade bits (CASCADEs). It can be seen that the area resources and the response time increase as the number of redundant modules does.

3 Conclusions

In this article we have described an educational approach to analyse fundamental fault tolerance topics. This initiative mainly consists of a library of characteristic circuits which are present in specific fault tolerant designs, and the guidelines to develop several kinds of exercises in the laboratory. The library circuits and the systems designed by the students have been implemented using a Hardware Description Language and a specific design CAD tool. We consider the approach introduced as the core of a specific framework which facilitates the comprehension and the analysis of the main issues in fault tolerant design. It could be included in undergraduate and graduate courses related to Digital Systems Design and Fault Tolerant Systems in Electrical and Computer Engineering.

References

1. The Joint Task Force on Computing Curricula IEEE Computer Society ACM "Computing Curricula 2001. Computer Engineering Main Report". October 25, (2002) URL: www.eng.auburn.edu/ece/CCCE/MainReport/MainReport.PDF
2. Barry W. Johnson. Design and Analysis of Fault-Tolerant Digital Systems. Addison-Wesley Publishing Company. (1989)
3. Pralag K. Lala. Self-Checking and Fault-Tolerant Digital Design. Morgan Kaufmann Publishers. (2001)
4. Dhiraj K. Pradhan. Fault-Tolerant Computer Systems Design. Prentice Hall. (1996)
5. Martin L. Shooman. Reliability of Computer Systems and Networks: Fault Tolerance, Analysis and Design. John Wiley & Sons. (2001)
6. P.T. de Sousa and F. P. Mathur. Sift-out modular redundancy IEEE Transactions on Computers C-27 7 (1978) 624–627

Hardware Design with a Scripting Language

Per Haglund, Oskar Mencer, Wayne Luk, and Benjamin Tai

Department of Computing, Imperial College, London SW7 2BZ, UK

Abstract. The Python Hardware Description Language (PyHDL) provides a scripting interface to object-oriented hardware design in C++. PyHDL uses the PamDC and PAM-Blox libraries to generate FPGA circuits. The main advantage of scripting languages is a reduction in development time for high-level designs. We propose a two-step approach: first, use scripting to explore effects of composition and parameterisation; second, convert the scripted designs into compiled components for performance. Our results show that, for small designs, our method offers 5 to 7 times improvement in turnaround time. For a large 10x10 matrix vector multiplier, our method offers respectively 365% and 19% improvement in turnaround time over purely scripting and purely compiled methods.

1 Introduction

Existing HDLs based on Java, C, or C++, such as JHDL [1], Handel-C [2], or PAM-Blox [6], are compiled languages for hardware design. The compilation stage of compiled HDLs delays the design process. Software programmers frequently use scripting languages for rapid application prototyping.

Previous work highlights how designers can benefit from scripting common hardware design tasks. For example, Luk [5] uses scripting to automate core testing, and Ho [4] links together stages in the tool-chain with scripts. However, we are not aware of methods that involve scripting a high-level structural description of circuits, such as proposed in this paper.

We propose a two-step methodology for capturing the advantages of scripting at the hardware design stage. First, designers use a scripting language to prototype designs using existing compiled components. Scripting removes the compilation stage and substantially accelerates the exploration process. Second, on completion of the exploration process, designers convert their scripted designs into new compiled components.

We implement our scripting methodology by extending the Python scripting language [7, 8] with the features of the C++ HDL PamDC/PAM-Blox [6]. The contributions of our hardware scripting language PyHDL are:

- Extending a scripting language with existing hardware design libraries.
- Evaluating the impact of hardware scripting on design time.

The remainder of this paper is organised as follows. Section 2 briefly describes our implementation. Section 3 evaluates PyHDL through the presentation and analysis of experimental results from several circuit designs. Section 4 offers conclusions.

2 Hardware Scripting

We develop a scripting HDL by *extending* a scripting language with the facilities of existing compiled hardware design libraries. The extension method requires a compiled HDL, a scripting language and an API to allow interaction between the scripting language and the HDL. We use the Python/C API to extend the scripting language Python with the C++ hardware design libraries of PamDC and PAM-Blox. PamDC provides primitives such as wires and registers. PAM-Blox is a collection of parameterisable, object-oriented hardware components built from PamDC primitives.

PyHDL consists of five components: mapping classes, interface, PAM-Blox library, design scripts, and output scripts. *Mapping classes* are a set of Python classes that replicate the class hierarchy of PAM-Blox. The mapping classes type-check and propagate inheritance before forwarding control to the interface. The *interface* manages the flow of control between Python and PAM-Blox. The interface associates instances of PyHDL mapping classes with PAM-Blox hardware objects. The *PAM-Blox library* contains the primitives and compiled hardware modules found in PamDC and PAM-Blox. *Design scripts* contain circuit designs written in Python. Python circuit designs typically contain both primitives and PAM-Blox components. An *Output script* defines the parameters for a particular design and drives the simulator or EDIF generator.

3 Evaluation of PyHDL

We use the following metrics to evaluate the effectiveness of our approach: (a) *turnaround time* is the time taken to generate new simulation output after a design change; (b) *hardware performance* results obtained from FPGA-vendor tools; (c) *lines of code* to indicate potential syntactical differences.

We implement three circuits in Python and C++: A greatest common denominator finder (GCD), a credit card validator (CARD), and a scalable matrix vector multiplier (MATMUL) using 16-bit combinational multipliers. The GCD and CARD circuits are small and allow us to evaluate the effectiveness of hardware scripting for testing and prototyping components. The MATMUL circuit is scalable and allows us to evaluate how the performance of hardware scripting changes with circuit size.

We present three implementations of the MATMUL circuit. Each implementation uses a serial shift-add multiplier. We design the multiplier using either primitives within Python or PAM-Blox components.

The first MATMUL implementation uses pure Python and illustrates how the performance of a scripting-only approach scales poorly with circuit size. Porting the Python multiplier to C++ produces a new compiled PAM-Blox multiplier. The second MATMUL uses the new PAM-Blox multiplier and thus illustrates the effectiveness of extending a scripting language with compiled components. The third MATMUL implementation is written entirely in C++ and shows the base case performance resulting from using only compiled C++ code.

Table 1. The implementation results of three circuits: Greatest common denominator (GCD), credit card validation (CARD), and a 10x10 matrix vector multiplier (MATMUL). The three implementations of MATMUL are: pure Python (multiplier made from primitives), Python using C++ components (PAM-Blox multiplier), and pure C++ (PAM-Blox multiplier). The hardware results are based on targeting Xilinx XCV1000E and XC2V6000 devices

	GCD		CARD		MATMUL (10x10)		
Circuit design language	Python	C++	Python	C++	Python		
Component design language	C++	C++	C++	C++	C++		
Turnaround (seconds)	0.80	5.60	1.02	5.84	69.1	18.9	22.5
Slices	38	40	283	285	25170	25170	20702
LUTs	71	72	482	482	43570	43569	34051
FFs	21	22	25	26	3781	3781	4660
Clock speed (MHz)	71.5	69.1	4.3	4.1	18.2	18.3	29.8
Gate count	843	865	3469	3485	387981	387981	339252
Lines of code	86	84	128	135	169	104	109

We use GNU Cygwin on a 1.7 GHz Pentium 4 and Xilinx ISE 5.1 to produce our results. We target a Xilinx XCV1000E device for the GCD and CARD circuits, and a XC2V6000 device for the MATMUL circuit.

Table 1 presents our results for the GCD, CARD, and MATMUL circuits. For the two small circuits, Python reduces the turnaround time by over 80%. The results for the three implementations of the matrix vector multiplier show that scripting performance depends on both the design language and the component implementation. Pure C++ is faster than pure Python, but slower than Python combined with compiled components. Figure 1 shows how the turnaround time scales with the size of the matrix vector multiplier. Scripting-only becomes inefficient for vectors of length 4 or greater. However, using compiled components in a scripted circuit design is consistently faster than using either pure Python or pure C++.

The hardware results show that PyHDL produces hardware similar to PAM-Blox. The discrepancy is mostly due to differences in buffers added between the logic and technology mapping, which does not yet occur identically in all setups.

The experimental results confirm that the proposed approach, described in Section 2, can provide significant benefits. First, hardware scripting eliminates most of the compilation overhead, as seen in Table 1 and Figure 1. Second, converting a Python component into a compiled component preserves the performance advantage of hardware scripting for larger designs. The curve *Pure Python* in Figure 1 illustrates how the performance of a design approach based only on scripting deteriorates with increasing circuit size. The curve *Python with C++ components* in Figure 1 shows how scripting performance improves after converting the pure Python multiplier into a compiled PAM-Blox component.

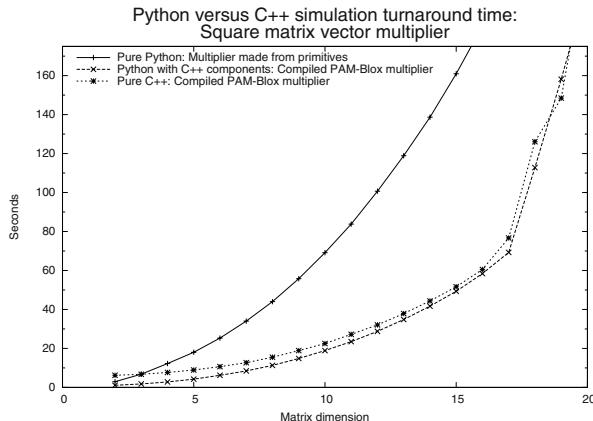


Fig. 1. The turnaround times of three implementations of a scalable matrix vector multiplier using Python and C++. Each design uses a structurally identical multiplier implemented in either Python or C++.

4 Conclusions

This paper shows how hardware scripting facilitates rapid prototyping and exploration of component characteristics. We evaluate our hardware scripting language PyHDL, an extension of the popular scripting language Python with the compiled PAM-Blox framework. We find that hardware scripting reduces the design time for small circuits, and that satisfactory circuit descriptions can be ported to C++ and compiled to reduce the design time for large circuits also.

References

1. P. Bellows and B. Hutchings, “JHDL - an HDL for reconfigurable systems,” in *IEEE Symposium on FPGAs for Custom Computing Machines*, K. L. Pocek and J. Arnold, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 175–184.
2. Celoxica, “Celoxica homepage,” <http://www.celoxica.com/>.
3. R. Goering, “Engineer creates open-source hdl in Ruby language,” <http://www.eetimes.com/story/OEG20020807S0019>.
4. C. Ho, P. Leong, K. Lee, K. Tsoi, R. Ludewig, P. Zipf, A. Ortiz, and M. Glesner, “Fly - a modifiable hardware compiler,” in *Field-Programmable Logic and Applications*. Springer Verlag, 2002, pp. 381–390, LNCS 2438.
5. W. Luk, D. Siganos, and T. Fowler, “Automating qualification of reconfigurable cores,” in *Reconfigurable Systems*. IEE Digest, 1999, pp. 4/1–4/6.
6. O. Mencer, M. Morf, and M. J. Flynn, “PAM-Blox: High performance FPGA design for adaptive computing,” in *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Press, 1998, pp. 167–174.
7. Python Software Foundation, “Python homepage,” <http://www.python.org>.
8. G. van Rossum, “An introduction to python for UNIX/C programmers,” *Proc. NLUUG - Dutch Unix User Group Conference*, 1993.

Testable Clock Routing Architecture for Field Programmable Gate Arrays

L. Kalyan Kumar, Amol J. Mupid, Aditya S. Ramani, and V. Kamakoti

Department of Computer Science and Engineering, Indian Institute of Technology
Madras, Chennai 600036, Tamilnadu, India.

{kalyan,amol,ramani}@peacock.iitm.ernet.in, kama@iitm.ernet.in

Abstract. This paper describes an efficient methodology for testing dedicated clock lines in Field Programmable Gate Arrays (FPGAs). A H-tree based clocking architecture is proposed along with a test scheme. The H-tree architecture provides optimal clock skew characteristics. The H-tree architecture consumes at least 25% less of the routing resources when compared to conventional clock routing schemes. A testing scheme, which utilizes the partial reconfiguration capabilities of FPGAs through selective re-programming of the Complex Logic Blocks, to detect and locate faults in the clock lines is proposed

1 Introduction and Previous Work

An FPGA, like any other semiconductor device is affected by faults occurring during the components lifetime. Though most faults occur at the time of fabrication, occasional operational faults can result after extended usage. Unlike manufacturing defects that can be avoided by using spare routing wires and programmable fuses, operational failures need to be addressed by generating a new programming configuration [3]. *Online checkers* are used in the design for detecting such operational/run-time errors [1]. FPGA testing is divided into testing of logic blocks, interconnects, embedded memory, I/O block and miscellaneous testing which includes clock and powerlines [1]. To the best of our knowledge, there is no result reported in the literature, which addresses the testing of clock lines. The objective of any clock design is to minimize clock skew, clock delay, clock area, power, and noise while maximizing clock reliability. A clock routing architecture for ASICs called H-tree is described in [2]. *This paper proposes a H-tree based clock routing structure for symmetric FPGA architectures and derives bounds to prove its efficiency. A BIST based testing algorithm is also presented to test the proposed H-tree architecture.*

2 H-Tree Clock Routing Architecture for FPGA

This architecture essentially consists of an H shaped routing structure that is repeated with successively halving edge lengths. The starting point of the clock routing architecture, called the *root node*, is located in the geometric center point

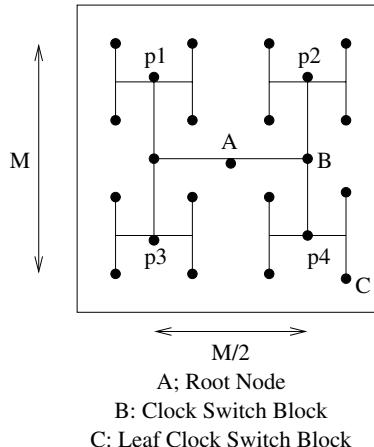


Fig. 1. 2-level H-Tree Clock Routing Structure

of the FPGA chip and the H-tree structure distributes it across to all CLBs. This distribution needs to be done till each CLB has an adjacent clock line from which it can tap a clock input. The external clock can be tapped either from the periphery of the FPGA chip (Flip I/O Technology) or from the geometric center point in the area of the FPGA chip (Area I/O technology) and connected to the root node. If clocks are predetermined, then, they may be embedded (*in-core* clock) inside the FPGA chip at the geometric center (root node). Additionally we assume that each CLB has an output that can be tapped to aid the testing procedure. The H-tree constructed has 4 end points p_1, p_2, p_3 and p_4 as shown in Figure 1. The algorithm recursively proceeds to construct H structures as in the figure with p_1, p_2, p_3 and p_4 as the center points and the lines of these structures, each covering $M/4$ CLBs. This is done till each of the CLBs receive a clock input.

The following bounds hold good for a (MXM) CLB matrix.

1. The maximum Clock-to-leaf switch length of the clock line is $\leq M$.
2. Cumulative Length of Clock Routing Lines = $\frac{3}{2}(\frac{M}{2} - 1)M$

3 BIST for Testing H-Tree Structure

The BIST for testing the H-tree structure uses some of the CLBs available in the FPGA by configuring the Look-Up Tables (LUTs) in them in one of the following modes.

Normal Pass Mode: In this mode the LUT is loaded with a function, which passes one of its specified inputs to the output.

Inverse Pass Mode: In this mode the LUT is loaded to output the complement of one of its inputs.

The following definitions are used in the testing algorithms that follow.

Clock Switch Block Cover: A clock switch block C covers a CLB A if A is one of the 4 CLBs nearest to it and hence can take a clock input from it.

Leaf CLB: A leaf CLB is one that is covered by a leaf clock switch block

Algorithm 1 (Test All H-tree Clock Routing Lines)

1. Unmark all leaf clock switch blocks.
2. **while** (*there exists an untested leaf clock switch block*)
 - Select an unmarked leaf clock switch block L.
 - Apply Algorithm 2 to test L if the path from clock to L is fault free.
 - **if** (*a faulty clock line was reported*) locate clock segment using Algorithm 4.
 - Mark L.

Algorithm 2 (Testing of path P from Root node to any clock switch block L)

1. Select a CLB M covered by L.
2. Connect the output of M, by programming the interconnect structure, to an I/O port of the chip denoted by Tapped Output.
3. Let Clock be one of the inputs to CLB M.
4. Configure M in pass mode so as to pass clock to its output.
5. Apply the logical signal 1 at the Root node and observe the Tapped Output.
6. **if** (*Tapped Output = 0*) Fault is in the Path P or Tapped Output.
7. Find region of fault using Algorithm 3 with input as (M,0).
8. **if** (*fault is in Tapped Output*)
 - Mark M as un-useable.
 - Select an unmarked/usable CLB M covered by L and go back to Step 2.
 - **if** (*no such CLB exists*)
 - report clock switch block to be un-testable and **return**.
 - **if** (*fault is in path P*) report that (P is faulty) and **return**.
9. Repeat steps 5 to 8 by applying the logical signal 0 at the Root node.
(Parallel case replacing 0 → 1 and 1 → 0.)
10. **if** (*no faulty outputs have been obtained*), mark M as usable, report P is fault-free and **return**.

Algorithm 3 (Locating the fault region through a leaf CLB M)

1. **if** (*stuck value = 0*)
 - Let Clock be one of the inputs to CLB M.
 - Configure M in inverse pass mode so as to pass inverted clock to its output.
 - Apply logical signal 0 at Root node.
 - **if** (*Tapped Output = 1*) fault is in the path P.
else fault is in the Tapped Output Line.
2. Repeat Step 1 checking for stuck value = 1
(Parallel case replacing 0 → 1 and 1 → 0).

It is easy to see that the inverse pass mode detects the position of the fault as the output is dependant on whether the fault was encountered before or after inversion. The Algorithm 4 is similar to a binary search procedure to locate the faulty segment.

Algorithm 4 : Lookup(P) (Locate the faulty segment in a faulty path P)

1. Let $l_1 - l_2 - \dots - l_r$ be the segments making up P, where l_1 is the end-point closest to the root node.
2. Note that every l_i connects two clock switch blocks, c_{i-1} and c_i where c_0 is the Root node.
3. **if** ($r = 1$) report faulty segment is l_r .
else
 - Let $c_{r/2}$ be the clock switch block corresponding to segment $l_{r/2}$ in the path P.
 - Identify a usable/untested CLB M covered by $c_{r/2}$ and perform the testing procedure in Algorithm 2, which will report either the path from - Root Node to CLB M is faulty or not.
 - **if** (*faulty clock path was reported by Algorithm 2*) $P = l_1 - l_2 - \dots - l_{r/2}$
 - **if** (*fault-free clock path was reported by Algorithm 2*)
 $P = l_{r/2} - l_{r/2+1}, \dots - l_r$.
 - **if** (*clock path was reported to be un-testable by Algorithm 2*) **exit**.
4. **LookUp(P)**

4 Conclusion

In this paper we presented a novel method to detect and locate faults in clock lines of FPGA based systems based on an efficient in-built dedicated routing architecture that we proposed. The number of clock switch blocks in any path from a clock Root node to a CLB, in a MXM CLB array, which contributes to the major portion of the propagation delay is upper bound by $\log_2 M$. We exploited the selective re-programming capabilities of FPGAs to arrive at an efficient fault detection and location procedure. The time required to locate the faulty clock segment, after identification of a faulty clock path is upper bound by $\log_2 \log_2 M$.

References

1. Subhasish Mitra, Philip P. Shirvani and McCluskey, J. E.: Fault Location in FPGA Based Reconfigurable Systems, BMDO/IST 1999.
2. Ullman, D. J.: Computational aspects of VLSI Design, Computer Science Press., pp84.
3. Vijay Lakamraju and Tessier, R.: Tolerating Operational Faults in Cluster-based FPGAs., FPGA2000, Monterey, CA, USA.

FPGA Implementation of Multi-layer Perceptrons for Speech Recognition

E.M. Ortigosa¹, P.M. Ortigosa², A. Cañas¹, E. Ros¹, R. Agís¹, and J. Ortega¹

¹ Dept. of Computer Architecture and Technology,
ETS Ingeniería Informática. University of Granada, E-18071 Granada, Spain
{eva, acanas, eros, ragis, jortega}@atc.ugr.es

² Dept. of Computer Architecture and Electronics,
University of Almería, E-04120 Almería, Spain, ortigosa@ual.es

Abstract. In this work we present different hardware implementations of a multi-layer perceptron for speech recognition. The designs have been defined using two different abstraction levels: register transfer level (VHDL) and a higher algorithmic-like level (Handel-C). The implementations have been developed and tested into a reconfigurable hardware (FPGA) for embedded systems. A study of the two considered approaches costs (silicon area), speed and required computational resources is presented.

1 Introduction

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems process information. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. The work presented in this paper addresses the study of the implementation viability and efficiency of ANNs into reconfigurable hardware (FPGA) for embedded systems, such as portable real-time Automatic Speech Recognition (ASR) systems for consumer applications. Let us focus on *a voice controlled phone dial system* (this can be of interest for drivers that should keep their attention in driving). This particularizes the Multi-Layer Perceptron (MLP) parameters to be implemented and the word set of interest (numbers from 0 to 9).

The multi-layer perceptron neural network model consists of a network of processing elements or nodes arranged in layers. The principle of the network is that when data from an input pattern is presented at the input layer the network nodes perform calculations in the successive layers until an output value is computed at each of the output nodes. The weights of the connections define the behavior of the network and are adjusted during training through a supervised training algorithm called back-propagation [1]. In the “forward pass” an input pattern vector is presented to the input layer. For successive layers, the input to each node is the sum of the scalar products of the incoming vector components with their respective weights (1),

$$sum_i = \sum_j w_{ij} out_j , \quad (1)$$

$$f(sum_i) = \frac{1}{1 + e^{-sum_i}} \quad (2)$$

where w_{ij} is the weight connecting node j to node i and out_j is the output from node j . The output of a node i is $out_i = f(sum_i)$, which is then sent to all nodes in the following layer. The function f denotes the activation function of each node. A sigmoid activation function is frequently used, eq. (2).

2 Hardware Implementations

For our test bed application we need a MLP with 220 data inputs (10 vectors of 22 features extracted from speech analysis) and 10 output nodes in the output layer (corresponding to the 10 recognizable words). After testing different architectures, the best results (96.83% correct classification) have been obtained with 24 nodes in the hidden layer.

For the MLP implementation we have chosen fixed point computations with two's complement representation and different bit depths for the stored data (inputs, weights, activation function, outputs, etc). It is necessary to limit the range of different variables: inputs to the MLP (8 bits), output of the activation function (8 bits), weights (8 bits), and inputs to the activation function. Apparently we need 23 bits for the input of the activation function, but because we are using the sigmoid waveform, most of the values are repeated and only a small transition zone ($\approx 1\%$ of values) needs to be stored. After taking all these discretization simplifications the model achieves similar classification results. The results of the hardware system differ in less than 1% from the software full resolution results.

2.1 Register Transfer Level (VHDL)

The Register Transfer Level design of MLP has been defined using standard VHDL as hardware description language. As it can be seen in equation (1), the basic computations of a single neuron are the multiplication of the outputs from the connected neurons (synaptic signals) by their associated weights, and the summation of these multiplied terms. The Functional Unit (processing element) is composed by an 8-bit multiplier and a 24-bit accumulative adder.

The serial version of the MLP consists on a single functional unit that carries out all computations, for all neurons. The inputs of the functional unit are both the synaptic signals and their associated weights, which are stored in separate RAM modules. The output of the functional unit is connected to the activation function module. The activation function output is stored either in the hidden or in the final neuron RAMs, depending on the layer of the computed neuron.

The proposed parallel architecture describes a kind of node parallelism, in the sense that requires one functional unit per neuron when working at a determined layer. With this strategy, all neurons of a layer work in parallel and therefore produce

their outputs simultaneously. This is not a fully parallel strategy because the outputs for different layers are obtained in a serial way. For our particular MLP where 24 neurons exist at the hidden layer and 10 ones at the output layer, 24 functional units are required. All of them will work in parallel when computing the outputs of the hidden layer and only 10 of them will work when the output layer is computed.

2.2 High Level Description (Handel-C)

The high level design of MLP has been defined using Handel-C [2] as a system-level specification language. Based on ANSI-C, Handel-C includes a simple set of extensions required for hardware development. The whole design processes have been defined with DK1 Design Suite tool from Celoxica [2]. Sequential and parallel designs have been finally compiled using the development environment *Xilinx Foundation 3.5i* [3].

In the sequential version, the MLP computes the synaptic signals for each neuron in the hidden layer by processing the inputs sequentially; and later on, the obtained outputs are similarly processed by the output neurons. In the parallel version, all neurons belonging to the same layer compute their results simultaneously, in parallel, except for accessing to the activation function that is done in a serial way.

3 Comparative Results

The systems have been designed using the development environments FPGA advantage and DK1.1 to extract the EDIF files. All designs have been finally placed and routed in a VirtexE 2000 FPGA, using the development environment *Xilinx Foundation 3.5i* [3]. Table 1 shows the implementation results obtained after synthesizing both sequential and parallel versions of the MLP defined using VHDL. These results are characterized by the following parameters: number of slices, number of EMB RAMs, minimum clock period, number of clock cycles required for each input vector (220 input components) evaluation, and total time consumed for each input vector evaluation. The computing time for each input vector (last column) is much shorter (20 times) in the parallel version. In this way we are taking advantage of the inherent parallelism of the ANN computation scheme.

Table 1. Implementation characteristics of the designs with VHDL

MLP design	# slices	% slices	# EMB RAMs	% EMB RAMs	Clock (ns)	# Cycles	Evaluation time (μs)
Serial	379	1.5	19	11	49.024	5630	276.005
Parallel	1614	8.5	26	16	53.142	258	13.710

Table 2 presents the results obtained after synthesizing, the sequential and parallel versions of the MLP defined using Handel-C. When defining the MLP we have to decide how to store the data in RAM. Different strategies are considered: (a) only

distributed RAM for the whole designs have been utilized, (b) the weights associated to synaptic signals (large array) make use of EMB RAM modules, while the remaining data are stored in a distributed mode; and finally, (c) only EMB RAM modules are used. Results in Table 2 show that independently of the distribution memory option, the parallel version requires more area resources than its corresponding serial one. As happened in VHDL description, the computing time for each input vector is much shorter (about 20 times) in the parallel version. The choice of a determined option in the memory implementation will depend on the area and time constraints.

Table 2. Implementation characteristics of the designs with Handel-C. (a) Only distributed RAM. (b) Both EMB RAMs and distributed RAM. (c) Only EMB RAM

MLP design	# Slices	% slices	# EMB RAMs	% EMB RAMs	Clock (ns)	# cycles	Evaluation time (μs)
(a) Serial	2582	13	0	0	50.620	5588	282.864
	Parallel	6321	32	0	58.162	282	16.402
(b) Serial	710	3	24	15	62.148	5588	347.283
	Parallel	4411	22	24	59.774	282	16.856
(c) Serial	547	2	36	22	65.456	5588	365.768
	Parallel	4270	22	36	64.838	282	18.284

When comparing Tables 1 and 2, where results for a RTL and a higher level description are shown respectively, it can be seen that the RTL implementation leads to a more optimized approach for the final system. However, one of the main advantages of the high level description is the design time of a system. So, for our MLP designs, it must be known that the design time for the serial case when using high level description has been about 10 times shorter than the RTL one. For the parallel case, the design time for the high level it has been relatively short (just to introduce the “*par*” directives) while for the RTL description a new architecture and control unit have been designed; this fact implies a larger difference in the designing time.

4 Conclusions

We have presented the FPGA implementation of MLP for speech recognition applications. Both sequential and parallel versions of the MLP have been described using two different abstraction levels: register transfer level (using VHDL) and a higher algorithmic-like level (using Handel-C). Results show that RTL implementation produces more optimized system; however, one of the main advantages of the high level description is the time consumed to design a system. For the speech recognition application we obtain a correct classification rate of 96.83% with a computation time around 14-16 microseconds per sample, which fulfills by far the time restrictions imposed by the application.

Acknowledgement

The work has been supported by CICYT TIC2002-00228 and SpikeFORCE (IST2001-35271).

References

1. Widrow, B., Lehr, M.: 30 years of adaptive neural networks: Perceptron, Madaline and Backpropagation. *Proceedings of the IEEE*, vol. 78, no. 9, pp.1415-1442 (1990)
2. Celoxica, <http://www.celoxica.com/>
3. Xilinx, <http://www.xilinx.com/>

FPGA Based High Density Spiking Neural Network Array

Juan M. Xicotencatl and Miguel Arias-Estrada

Computer Science Department, INAOE, Apdo. Postal 51 & 216, Mexico
jperez@inaoep.mx, ariasm@inaoep.mx

Abstract. Pulsed neural networks can be applied to the design of dense arrays using minimum hardware resources in the interconnection among neurons. Using statistical saturation in pulse frequency coded neurons, a minimum size hardware neuron can be implemented. The proposed neuron is compact enough to be included in large arrays. The presented architecture has additional interesting characteristics like unrestricted topology and scalability. In this paper, the design and implementation of a high density spiking neural array is presented.

1 Introduction

ANNs are parallel processing structures that have a large number of processors and a large numbers of interconnections among them. In an ANN each processor is connected to its neighbors, so there are more interconnections than processors. The ANN computational power lies in the large number of interconnections. There is an associated synaptic strength or a weight with each connection. ANNs digital implementation focuses on the architecture design. A digital implementation has some important characteristics like flexibility, high accuracy and repeatability [2].

This paper presents an FPGA based high-density neural network array, which is a generic platform for applications that require over 1000 neurons in hardware. The rest of the paper is organized as follows: section 2, 3 and 4 discuss the architectural concepts proposed in our approach at the synapse, soma and interconnections respectively. Next section presents the architecture simulation and a discussion is given. Finally, some conclusions and future work are presented.

2 Synapse Design and Weight Storage

Practical ANNs dense implementations are possible only if the circuitry devoted to multiplication involved in the synapse is reduced. In this work, the instantaneous value of the neuron activity is represented as the instantaneous frequency of digital pulses (PFM, Pulse Frequency Modulation). Since the signal level is expressed by the frequency, a simple frequency converter replaces the multiplier in the synapse. Figure 1 shows the implemented synapse design.

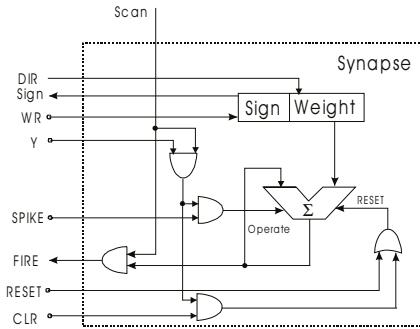


Fig. 1. The synapse module is implemented with a register and an accumulator.

The designed module loads a signed weight in serial form. The weight is stored in a serial input parallel output (SIPO) register to be accessible by the accumulator. The accumulator increases its value with each input pulse (SPIKE). If the accumulator reaches a maximum, it produces an overflow. Finally, the generated overflow is present in the FIRE terminal to be externally processed when the SCAN line is one.

3 Soma Design, Summation of Synaptic Contribution and Non-linear Function Activation

In the designed soma, only the synapses that are fired contribute to the output. In this way, the proposed soma avoids to access or to route synapses with a zero contribution. To obtain the non-linearity in the output, the statistical saturation present in the counter is used. This element and the necessary logic to update and to access its content are shown in figure 2.

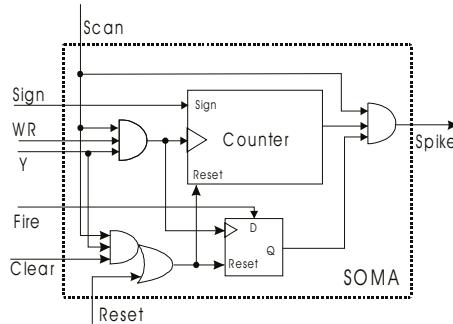


Fig. 2. Soma design. The counter adds the contributions from the previous layer.

The proposed soma counts the signed pulses from an input pulse train. The soma fires when the number of positive pulses exceeds the number of negative pulses and a internal flip-flop contains a one. This flip-flop is set to one when the last synapse associated to the soma fires.

4 Routing of the Activity among Neurons

In the proposed design, the topology can be changed online or offline if the application requires it. Since, there are not direct interconnections among neurons, the topology is mapped to an external memory. This scheme is shown in figure 3.

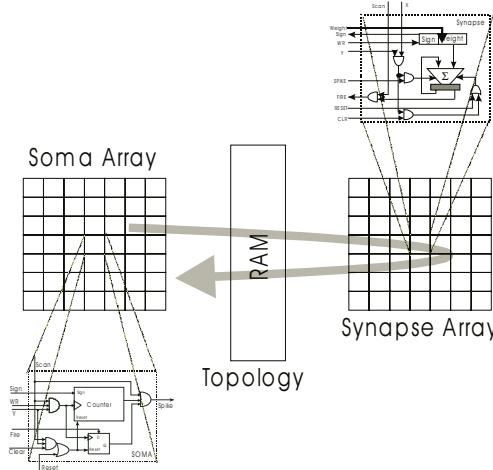


Fig. 3. The proposed neural network design, an external memory is used to map the topology.

In figure 3, two large arrays are defined: the soma and the synapse arrays. Both arrays read the RAM memory to access the corresponding elements in the other array. The RAM memory contains all the module locations, i.e., it contains the *address* from each element inside the arrays.

5 Implementation and Results

The architecture has been modeled in VHDL and synthesized using XILINX ISE 5.1i for an XCV2000BGA560-6. The whole architecture has been tested using FPGA-based board from AlphaData. The statistics of the design are presented in table 1.

Table 1. FPGA Statistics from the proposed design using a XCV2000BGA560-6

	Slices	Flip-Flops	LUTs	FPGA percentage	Frequency
1000 Synapses	13340	21340	25346	70%	-
1120 Somas	5329	6600	9028	27%	-
Total	18669		97%		35.6Mhz

The transfer function from the architecture is shown in figure 4. Its transfer function is approximated from the stochastic response as the equation:

$$f(\text{weight}) = \frac{1}{(1 + e^{-1.3\text{weight} + 1})}. \quad (1)$$

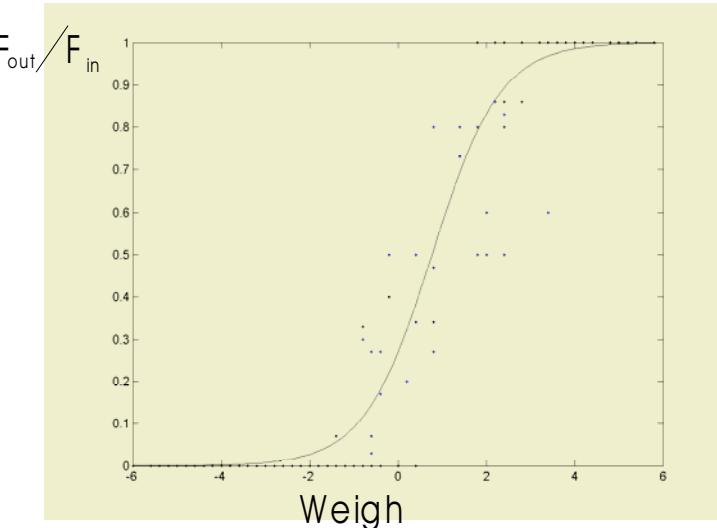


Fig. 4. Transference function. A single soma and a 5x5 synapse array were used. Six synapses are mapped in the external RAM.

Using the PFM technique, the performance in small and large arrays is the same. The bottleneck in the design is the memory, which is used to map the topology. The total number of somas and synapses is a function of the available resources in the FPGA. In small arrays, the memory can be implemented directly in the FPGA.

6 Conclusions and Future Work

An FPGA based high-density neural array was implemented and its functionality has been described. The neuronal arrays are related with an external topology memory. The proposed architecture can implement a massive number of neurons in a single FPGA. Additionally, the proposed design is scalable and flexible. Future work will focus on implementing a learning module and a hierarchy model to avoid the memory bottleneck.

References

1. T. Schoenauer, S. Atasoy, N. Mehrtash and H. Klar, Simulation of a Digital Neuro-Chip for Spiking Neural Networks, International Joint Conference on Neural Networks (IJCNN), Como, Italy, July 2000.
2. M. Schaefer, T. Schoenauer, C. Wolff, G. Hartmann, H. Klar and U. Rueckert, Simulation of Spiking Neural Networks - Architectures and Implementations, Neurocomputing, Elsevier, 2001.

FPGA-Based Computation of Free-Form Deformations

Jun Jiang, Wayne Luk, and Daniel Rueckert

Department of Computing, Imperial College London
180 Queen's Gate, London SW7 2BZ, England

Abstract. This paper describes techniques for producing FPGA-based designs that support free-form deformation in medical image processing. The free-form deformation method is based on a B-spline algorithm for modelling three-dimensional deformable objects. We transform the nested loop in this algorithm to eliminate conditional statements, enabling the development of a fully pipelined design. Further optimisations include precalculation of the B-spline model using lookup tables, and deployment of multiple pipelines so that each covers a different image. Our design description, captured in the Handel-C language, is parameterisable at compile time to support a range of image resolutions and output precisions. An implementation on a Xilinx XC2V6000 device at 67MHz has a throughput which is 12.8 times faster than an Athlon based PC at 1400 MHz.

1 Introduction

This paper describes techniques for producing FPGA-based designs that support free-form deformation (FFD) in medical image processing. Free-form deformations, based on B-splines, are a powerful tool for modelling three-dimensional deformable objects.

The image registration algorithm has been applied in several areas such as remote sensing and three-dimensional computer vision. In medical image processing such as contrast-enhanced breast Magnetic Resonance Imaging (MRI), the FFD method is adopted as an important part of non-rigid registration, a method for analyzing deformable objects. However, there is one disadvantage of this image registration implementation which adopts FFD as the local motion model: the processing time of a three-dimensional image with a resolution of 256 by 256 by 64 voxels takes between 15-30 minutes of processor time on a Sun Ultra 10 workstation.

Previously, we have presented a method for eliminating conditional statements in a nested loop for FFD computation by narrowing the range of the input [2]. This paper presents another method to achieve the same effect by transforming input data. This method has the advantage that all data can be processed by the hardware compared to our previous method.

2 B-Spline Based FFD

In medical image processing, B-spline based free-form deformations (FFD) are frequently used in non-rigid registration, such as 3D contrast-enhanced MRI, to model local deformations. For instance, the motion of the breast is non-rigid so that rigid or

affine transformations alone are not sufficient for the motion correction of breast MRI. Therefore a combined transformation T , which consists of a global transformation and a local transformation, is defined as follows [4]:

$$T(x, y, z) = T_{global}(x, y, z) + T_{local}(x, y, z)$$

To define a B-spline based FFD, the domain of the image volume is defined as $\Omega = \{(x, y, z) | 0 \leq x < X, 0 \leq y < Y, 0 \leq z < Z\}$. Let Φ denote a $n_x \times n_y \times n_z$ mesh of control points $\phi_{i,j,k}$ with uniform spacing. The FFD can be written as the 3D tensor product of the familiar 1D cubic B-splines:

$$T_{local}(x, y, z) = \sum_{i=0}^3 \sum_{j=0}^3 \sum_{k=0}^3 B_i(u) B_j(v) B_k(w) \phi_{i+l, j+m, k+n} \quad (1)$$

where

$$\begin{aligned} l &= \lfloor \frac{x}{n_x} \rfloor - 1, & m &= \lfloor \frac{y}{n_y} \rfloor - 1, & n &= \lfloor \frac{z}{n_z} \rfloor - 1, \\ u &= \frac{x}{n_x} - \lfloor \frac{x}{n_x} \rfloor, & v &= \frac{y}{n_y} - \lfloor \frac{y}{n_y} \rfloor, & w &= \frac{z}{n_z} - \lfloor \frac{z}{n_z} \rfloor \end{aligned}$$

and B_i represents the i -th basis function of the B-spline, and $u \in [0, 1]$.

$$\begin{aligned} B_0(u) &= (1-u)^3/6 \\ B_1(u) &= (3u^3 - 6u^2 + 4)/6 \\ B_2(u) &= (-3u^3 + 3u^2 + 3u + 1)/6 \\ B_3(u) &= u^3/6 \end{aligned} \quad (2)$$

3 Conditional Loop Transformation

In the three nested for-loop of the B-spline based FFD local deformation, there are three conditions which determine whether the loop body is executed or not. All these conditions depend not only on the for-loop variables, but also on the input values of the deformation.

Figure 1 shows the pseudo code of the nested for-loop for the FFD computation. The inner loop body would be executed only when conditions for K , J and I are all satisfied. The variables n , m and l are the fraction part of the input fixed-point numbers. Although one can implement a sequential hardware implementation using the Handel-C language [1], the performance is predictably low.

We use a transformed loop structure (Figure 2) to eliminate the conditional statements by assigning the first and the last elements of the transformed data array to zero. This corresponds to ignoring the effects of those transformed control lattice points outside the grey box shown in Figure 3, which do not have impact on the calculation result. With this method, a pipelined implementation of B-spline based FFD algorithm for processing a 2D image has been successfully compiled for an FPGA.

```

For k=0 to k=3
Begin
  K = k + n - 1
  if (0 <= K < CP_Z)
    For j=0 to j=3
      Begin
        J = j + m - 1
        if (0 <= J < CP_Y)
          For i=0 to i=3
            Begin
              I = i + l - 1
              if (0 <= I < CP_X)
                x = x + Bi(u) * Bj(v) * Bk(w) * Phi_X[K][J][I]
                y = y + Bi(u) * Bj(v) * Bk(w) * Phi_Y[K][J][I]
                z = z + Bi(u) * Bj(v) * Bk(w) * Phi_Z[K][J][I]
            End
        End
      End
    End
End

```

Fig. 1. Pseudo code of B-spline based free-form deformation, where $l = \lfloor x/n_x \rfloor$, $m = \lfloor y/n_y \rfloor$, $n = \lfloor z/n_z \rfloor$.

```

For k=0 to k=3 Begin
  K = k + n
  For j=0 to j=3 Begin
    J = j + m
    For i=0 to i=3 Begin
      I = i + l
      x = x + Bi(u) * Bj(v) * Bk(w) * Phi_X[K][J][I]
      y = y + Bi(u) * Bj(v) * Bk(w) * Phi_Y[K][J][I]
      z = z + Bi(u) * Bj(v) * Bk(w) * Phi_Z[K][J][I]
    End
  End
End

```

Fig. 2. Pseudo code of transformed loop, where $l = \lfloor x/n_x \rfloor$, $m = \lfloor y/n_y \rfloor$, $n = \lfloor z/n_z \rfloor$. When either (a) $K = 0$, $J = 0$ or $I = 0$, or (b) $K = CP_Z$, $J = CP_Y$ or $I = CP_X$, we assign $\Phi_X[K][J][I]$, $\Phi_Y[X][J][I]$ and $\Phi_Z[X][Y][I]$ to zero.

4 Pipelined Design for FFD

Our pipelined design for FFD involves three steps. The first step is to precalculate the four basis functions of a third-order B-spline, as shown in Equation 2, and store the values in four lookup tables. The second step is to design a fully pipelined FFD core. The third step is to deploy multiple pipelines.

Currently our pipelined design uses one input channel. The total number of execution cycles is around $N^d \times (i+1)^d \times M$, where N denotes the resolution, i denotes the order of B-spline, d represents the dimension of an image, and M represents the number of pipeline stages of the floating-point adder (Figure 4).

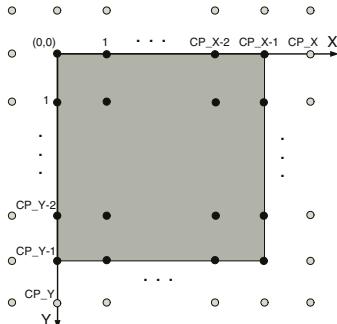


Fig. 3. The arrangement of control lattice on a 2D image.

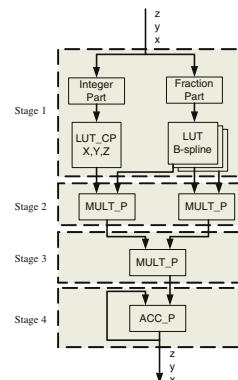


Fig. 4. Pipelined hardware for free-form deformation computation. MULT_P denotes a pipelined floating-point multiplier. ACC_P denotes a pipelined floating-point adder.

The input data have to be interleaved so that the whole pipeline could be fully used. In the pseudo code (Figure 2), we can see that the values of $Bl(u)$, $Bm(v)$ and $Bn(w)$ need to be accessed simultaneously in order to make a more efficient pipeline. Therefore, we replicate the lookup tables twice to meet the requirement of accessing three independent lookup tables concurrently (Equation 1).

We have also implemented two pipelines in XC2V6000 using our customisable representation with 12-bit mantissa and 8-bit exponent. It costs around 6 percent of Slice resources and 50 percent of Block RAM resources on the XC2V6000.

5 Performance

For a 2D image of resolution 256 by 256, the clock speed after place and route of our current two-pipeline implementation on a Xilinx Virtex II XC2V6000 device (Table 1) with 12-bit mantissa and 8-bit exponent is 67 MHz. Hence the estimated execution time for the XC2V6000 device is $(256 \times 256 \times 16 \times 3)/(67 \times 10^6 \times 2) \simeq 0.023$ second.

Table 1. Performance comparison of FFD processor for images of 256 by 256.

Processor	Area (slices)	Block RAMs (kbit)	Clock Speed (MHz)	Exec.Time (sec)	Throughput (data/second)
XC2V6000 (two pipelines)	2127	504	67	0.023	8375000
XC2V1000 (one pipeline)	1149	252	89	0.035	5562500
XC2V6000 (one pipeline)	1156	252	76	0.041	4750000
AMD Athlon	-	-	1400	0.10	655360
Pentium 4	-	-	1800	0.11	595782

Compared with the software version which runs on an AMD Athlon 1.4 GHz PC, the execution time of our customised system is around 4.3 times faster. Moreover, the throughput of this system is 12.8 times faster than the PC.

6 Summary

We have described hardware techniques for medical image processing. The key elements of our approach include precalculating the B-spline basis function, adopting custom number representation, transforming a nested loop to avoid conditional calculation, and developing fully-pipelined circuits and multiple pipeline designs. Current and future work includes integrating our design with a hardware image warper [3], exploring run-time reconfiguration to reduce the amount of FPGA resources required [5], and automating conditional loop transformations [6].

References

1. <http://www.celoxica.com>, Celoxica Limited.
2. J. Jiang, W. Luk and D. Rueckert, “FPGA-based computation of free-form deformations”, *IEEE International Conference on Field-Programmable Technology*, pp. 407–410, 2002.

3. J. Jiang, S. Schmidt, W. Luk and D. Rueckert, "Parameterizing designs for image warping", *Reconfigurable Technology: FPGAs and Reconfigurable Processors for Computing and Communications, Proc. SPIE*, vol. 4867, 2002.
4. D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach and D. J. Hawkes, "Non-rigid registration using free-form deformations: Application to breast MR images", *IEEE Transactions on Medical Imaging*, vol.18, no.8, pp. 712–721, 1999.
5. N. Shirazi, W. Luk and P.Y.K. Cheung, "Framework and tools for run-time reconfigurable designs", *IEE Proc. Comput. Digit. Tech.*, pp. 147–152, May 2000.
6. M. Weinhardt and W. Luk, "Pipeline vectorization", *IEEE Trans. on Computer-Aided Design*, pp. 234–248, February 2001.

FPGA Implementations of Neural Networks - A Survey of a Decade of Progress

Jihan Zhu and Peter Sutton

School of Information Technology and Electrical Engineering, The University of Queensland,
Brisbane, Queensland 4072, Australia
{jihan, p.sutton}@itee.uq.edu.au

Abstract. The first successful FPGA implementation [1] of artificial neural networks (ANNs) was published a little over a decade ago. It is timely to review the progress that has been made in this research area. This brief survey provides a taxonomy for classifying FPGA implementations of ANNs. Different implementation techniques and design issues are discussed. Future research trends are also presented.

1 Introduction

An artificial neural network (ANN) is a parallel and distributed network of simple nonlinear processing units interconnected in a layered arrangement. *Parallelism*, *modularity* and *dynamic adaptation* are three computational characteristics typically associated with ANNs. FPGA-based reconfigurable computing architectures are well suited to implement ANNs as one can exploit concurrency and rapidly reconfigure to adapt the weights and topologies of an ANN.

FPGA realisation of ANNs with a large number of neurons is still a challenging task because ANN algorithms are "multiplication-rech" and it is relatively expensive to implement multipliers on fine-grained FPGAs. By utilizing FPGA reconfigurability, there are strategies to implement ANNs on FPGAs cheaply and efficiently. It is the goal of this paper to: 1) provide a brief survey of existing ANN implementations in FPGA hardware; 2) highlight and discuss issues that are important for such implementations; and 3) provide analysis on how to best exploit FPGA reconfigurability for implementing ANNs. Due to space constraints, this paper can not be comprehensive; only a selected set of papers are referenced.

2 A Taxonomy of FPGA Implementations of ANNs

2.1 Purpose of Reconfiguration

All FPGA implementations of ANNs attempt to exploit the reconfigurability of FPGA hardware in one way or another. Identifying the purpose of reconfiguration sheds light on the motivation behind different implementation approaches.

Prototyping and Simulation exploits the fact that FPGA-based hardware can be rapidly reconfigured an unlimited number of times. This apparent hardware flexibility allows rapid prototyping of different ANN implementation strategies and learning algorithms for initial simulation or proof of concept. The GANGLION project [1] is a good example of rapid prototyping.

Density enhancement refers to methods which increase the amount of effective functionality per unit circuit area through FPGA reconfiguration. This is achieved by exploiting FPGA run-time / partial reconfigurability in one of two ways. Firstly, it is possible to time-multiplex an FPGA chip for each of the sequential steps in an ANN algorithm. For example, in work by Eldredge et al. [2], a back-propagation learning algorithm is divided into a sequence of feedforward presentation and back-propagation stages, and the implementation of each stage is executed on the same FPGA resource. Secondly, it is possible to time-multiplex an FPGA chip for each of the ANN circuits that is specialized with a set of constant operands at different stages during execution. This technique is also known as *dynamic constantfolding*. James-Roxby [3] implemented a 4-8-8-4 MLP on a Xilinx Virtex chip by using constant coefficient multipliers with the weight of each synapse as the constant. All constant weights can be changed through dynamic reconfiguration in under 69 μ s. This implementation is useful for exploiting training-level parallelism with batch-updating of weights at the end of each training epoch. The same idea was also previously explored in work by Zhu et al. [4].

As both methods for density enhancement incur reconfiguration overhead, good performance can only be achieved if the reconfiguration time is small compared to the computation time. There exists a break-even point q beyond which density enhancement is no longer profitable: $q = r / (s - 1)$ where r is the time (in cycles) taken to reconfigure the FPGA and s is the total computation time after each reconfiguration [5].

Topology Adaptation refers to the fact that dynamically configurable FPGA devices permit the implementation of ANNs with modifiable topologies. Hence, iterative construction of ANNs [6] can be realized through topology adaptation. During training, the topology and the required computational precision for an ANN can be adjusted according to some learning criteria. de Garis et al. [7] used genetic algorithms to dynamically grow and evolve cellular automata based ANNs. Zhu et al. [8] implemented the Kak algorithm which supports on-line pruning and construction of network models.

2.2 Data Representation

A body of research exists to show that it is possible to train ANNs with **integer** weights. The interest in using integer weights stems from the fact that integer multipliers can be implemented more efficiently than floating-point ones. There are also special learning algorithms [9] which use powers-of-two integers as weights. The advantage of powers-of-two integer weight learning algorithms is that the required multiplications in an ANN can be reduced to a series of shift operations. A few attempts have been made to implement ANNs in FPGA hardware with **floating-point** weights. However, no successful implementation has been reported to date. Recent work by Nichols et al. [10] showed that despite continuing advances in FPGA

technology, it is still impractical to implement ANNs on FPGAs with floating-point precision weights.

Bit-Stream arithmetic is a method which uses a stream of randomly generated bits to represent a real number, that is, the probability of the number of bits that are "on" is the value of the real number. The advantage with this approach is that the required synaptic multiplications can be reduced to simple logic operations. A comprehensive survey of this method can be found in Reyneri [11] while most recent work can be found in Hikawa [12]. The disadvantage of bit-stream arithmetic is the lack of precision. This can severely limit an ANN's ability to learn and solve a problem. In addition, the multiplication between two bit-streams is only correct if the bit-streams are uncorrelated. Producing independent random sources for bit-streams requires large resources.

3 Implementation Issues

3.1 Weight Precision

Selecting weight precision is one of the important choices when implementing ANNs on FPGAs. Weight precision is used to trade-off the capabilities of the realized ANNs against the implementation cost. A higher weight precision means fewer quantization errors in the final implementations, while a lower precision leads to simpler designs, greater speed and reductions in area requirements and power consumption. One way of resolving the trade-off is to determine the "minimum precision" required to solve a given problem. Traditionally, the minimum precision is found through "trial and error" by simulating the solution in software before implementation. Holt and Baker [13] studied the minimum precision required for a class of benchmark classification problems and found that 16-bit fixed-point is the minimum allowable precision without diminishing an ANN's capability to learn these benchmark problems.

Recently, more tangible progress has been made from a theoretical approach to weight precision selection. Draghici [14] relates the "difficulty" of a given classification problem (i.e. how difficult it is to solve) to the required number of weights and the necessary precision of the weights to solve the problem. He proved that, in the worst case, the required weight range $[-p, p]$ is estimated through the minimum distance between patterns of difference classes $d = (\sqrt{n})/(2p)$ to guarantee a solution, where p is an integer and n is the dimension of the input. This serves as an important guide for choosing data precision.

3.2 Transfer Function Implementation

Direct implementation for non-linear sigmoid transfer functions is very expensive. There are two practical approaches to approximate sigmoid functions with simple FPGA designs. **Piece-wise linear approximation** describes a combination of lines in the form of $y = ax + b$ which is used to approximate the sigmoid function. Note

that if the coefficients for the lines are chosen to be powers of two, the sigmoid functions can be realized by a series of shift and add operations. Many implementations of neuron transfer functions use such piece-wise linear approximations [15]. The second method is **lookup tables**, in which uniform samples taken from the centre of a sigmoid function can be stored in a table for look up. The regions outside the centre of the sigmoid function are still approximated in a piece-wise linear fashion.

4 Conclusion and Future Research Directions

When implementing ANNs on FPGAs one must be clear on the purpose reconfiguration plays and develop strategies to exploit it effectively. The weight and input precision should not be set arbitrarily as the precision required is problem dependent. Future research areas will include: **benchmarks**, which should be created to compare and analyse the performance of different implementations; **Software tools**, which are needed to facilitate the exchange of IP blocks and libraries of FPGA ANN implementations; **FPGA friendly learning algorithms**, which will continue to be developed as faithful realizations of ANN learning algorithms are still too complex and expensive; and, **topology adaptation** approaches, which take advantage of the features of the latest FPGAs such as specialized multipliers and MAC units.

References

1. Cox, C.E. and E. Blanz, *GangLion - a fast field Programmable gate array implementation of a connectionist classifier*. IEEE Journal of Solid-State Circuits, 1992. **28**(3): p. 288-299.
2. Eldredge, J.G. and B.L. Hutchings. *Density enhancement of a neural network using FPGAs and run-time reconfiguration*. in *Proceedings of IEEE Workshop on Field-Programmable Custom Computing Machines*, 1994. pp 180-188.
3. James-Roxby, P. and B.A. Blodget. *Adapting constant multipliers in a neural network implementation*. in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000. pp 335-336.
4. Zhu, J.M., G.J.; Gunther, B.K., *Towards an FPGA based reconfigurable computing environment for neural network implementations*, in *Proceedings of Ninth International Conference on Artificial Neural Networks*, 1999. pp. 661-666, vol.2.
5. Guccione, S.A. and M. Gonzalez. *Classification and Performance of reconfigurable architectures*. in *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*. 1995, pp 439-448, Springer-Verlag, Berlin.
6. Perez-Uribe, A. and E. Sanchez. *FPGA Implementation of an Adaptable-Size Neural Network*. in *Proceedings of the Sixth International Conference on Artificial Neural Networks*. 1996. pp 382-388, Springer-Verlag.
7. de Garis, H., et al. *Initial evolvability experiments on the CAM-brain machines (CBMs)*. in *Proceedings of the 2001 Congress on Evolutionary Computation*, 2001. pp 635-641, vol. 1.
8. Zhu, J. and G. Milne. *Implementing Kak Neural Networks on a Reconfigurable Computing Platform*. in *Proceedings of the 10th International Workshop on Field-*

- Programmable Logic and Applications - Roadmap to Reconfigurable Computing.* 2000. pp 260- 269. Springer Verlag.
- 9. Marchesi, M., et al., *Fast neural Networks without multipliers.* IEEE Transactions on Neural Networks, 1993. **4**(1): p. 53-62.
 - 10. Nichols, K., M. Moussa, and S. Areibi. *Feasibility of Floating-Point Arithmetic in FPGA based Artificial Neural Networks.* in *Proceedings of the 15th International Conference on Computer Applications in Industry and Engineering.* 2002. pp San Diego, California.
 - 11. Reyneri, L.M. *Theoretical and implementation aspects of pulse streams: an overview.* in *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems.* 1999. pp 78-89.
 - 12. Hikawa, H., *A new digital pulse-mode neuron with adjustable activation function.* IEEE Transactions on Neural Networks, 2003. **14**(1045-9227): p. 236-242.
 - 13. Holt, J.L., T.E. Baker. *Back propagation simulations using limited precision calculations,* in *Proceedings of International Joint Conference on Neural Networks.* 1991. pp 121-126 vol. 2.
 - 14. Draghici, S., *On the capabilities of neural networks using limited precision weights.* Neural Networks, 2002. **15**: p. 395-414.
 - 15. Wolf, D.F., Romero, R. A. F., Marques, E. *Using Embedded Processors in Hardware Models of Artificial Neural Networks.* in *In proceedings of SBAI - Simpósio Brasileiro de Automação Inteligente.* 2001. pp 78-83.

FPGA-Based Hardware/Software CoDesign of an Expert System Shell

Aurel Nețin, Dumitru Roman, Octavian Creț, Kalman Puszta, and Lucia Văcariu

Technical University of Cluj-Napoca, Computer Science Department, ROMANIA
{netin, cret, pusztai, lucia}@bavaria.utcluj.ro,
titiroroman@email.ro

Abstract. This paper presents a new method for implementing in hardware expert systems based on belief revision concepts. The expert system's knowledge base is first automatically translated to an equivalent network representation where nodes are facts and links stand for relationships. Then, changes are propagated throughout the network. The conclusions are extracted after no more changes occur in the state of the nodes. The automatic generation of the hardware network structure is described. Finally, the results obtained in this FPGA-based implementation are compared to those yielded by a Java-based implementation, the system's efficiency being thus demonstrated.

1 Introduction

The past years have witnessed a noticeable research effort towards a theory of reasoning under uncertainty. Probability theory was introduced in this area with the emergence of Bayesian belief network [3]. An alternative approach to the probability theory as a tool for modeling uncertainty is the use of belief functions. The research effort has been directed towards the specification of a knowledge representation framework that combines the merits of classical logic and Bayesian belief networks.

We represent uncertainty as a set $E = \{E_i \mid i \in \{1, 2, \dots, 9\}\}$ of nine ordered linguistic variables. The natural order induced among the variables holds true: E_1 stands for *impossible*, E_9 - for *certain*. Uncertainty is represented by a set of two parameters varying on E : *support* – the positive evidence for the assertion, and *plausibility* – the difference between the absolute certainty and the support of the negation of the assertion. Support and plausibility are independently updated (they are defined as different kind of information associated to a proposition, separately acquired and conceptually unrelated). The *belief states* partition in six areas the 9×9 table combining all the possible values (see Table 1). Two belief intervals are different only if they belong to different belief states.

Table 1. The belief states intervals (values for support and plausibility)

S \ PL	$E_1 \dots E_4$	$E_5 \dots E_8$	E_9
E_1	Disbelieved (D)		Unknown (U)
$E_2 \dots E_5$	Rather Disbelieved (RD)		
$E_6 \dots E_9$	Contradictory (C)	Rather Believed (RB)	Believed (B)

2 Operators Defined on Belief Intervals

To compute the belief interval associated to a compound expression, *operators* have been defined. The notation $|<\text{exp}>|$ is a shorthand for $[s(\text{exp}), p(\text{exp})]$.

- *Negation*: $N(E_i) = E_i - E_i = E_{i(i+1)}$ (the negation of a variable)
- *NOT*($|a|$)= $[N(p(a)), N(s(a))]=[s(\bullet a), p(\bullet a)]$ (the negation of a belief interval)
- *Conjunction*
 $[E_1, E_2], \text{if } |a| \in U, |b| \in C; \text{AND}(|a|, |b|) = [min(s(a), s(b)), min(p(a), p(b))], \text{otherwise}$
- *Disjunction*
 $[E_1, E_2], \text{if } |a| \in U, |b| \in C; \text{OR}(|a|, |b|) = [max(s(a), s(b)), max(p(a), p(b))], \text{otherwise}$
- *Aggregation*: Denoted by \odot ; it aggregates the evidence coming from different sources to a single assertion (a_i stands for the evidence pertaining to a and coming from source i). $AGGR(|a_1|, \dots, |a_n|) = [max(s(a_1), \dots, s(a_n)), min(p(a_1), \dots, p(a_n))]$
- *Detachment*: Denoted by \rightarrow , it propagates the evidence pertaining of an inference rule to its conclusions. The definition of the DET operator, with the belief interval pertaining to the rule's *premises* $|h|$ (hypothesis) and the rule's *strength* $|h \rightarrow t|$, expressing the deduction, is given below:
 $\text{AND}(|h|, |h \rightarrow t|), \text{if } h \in B, |h| \in RB; DET(|h|, |h \rightarrow t|) = [E_p, E_q], \text{otherwise}.$

3 Translating Rules to Network Representation

For a set of rules, if at least one proposition is inferentially related to itself we face a *circular dependency problem*. In such cases, the network representation contains loops and the belief interval propagation could suffer from termination problems.

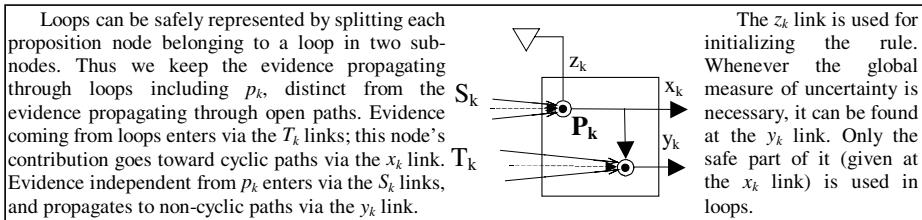


Fig. 1. The node splitting algorithm: the p_k node split example

The best solution is to make the system automatically modify the network's topology without altering the connections' semantic. We represent knowledge by rules in a dependency network, having three types of nodes: *proposition*, *rule* and *operator*. Each *rule* node receives as input the belief intervals of its premises and produces the belief interval of its conclusion. Each *operator* node receives as inputs the belief intervals of its operands and produces the belief interval of its result.

Propositions are modeled as predicate-value pairs. The rule's format is: *name*, used in the explanatory process, *premise*, a compound expression, *conclusion*, and *strength*, a measure of the uncertainty used as a belief interval. By parsing a rule we obtain a node with a *detachment* (\rightarrow) operator having the *strength* as parameter.

4 Hardware Implementation Experimental Results

Due to its specific nature, the knowledge base (KB) can be efficiently implemented in hardware (HW): for *propositions*, 8-bits data registers are sufficient to store belief intervals, while the *rules* (*DET operator*) and the other *operators* are implemented by combinational logic. The communication between the system and the external environment is done by means of a Data, an Address and a Control Bus.

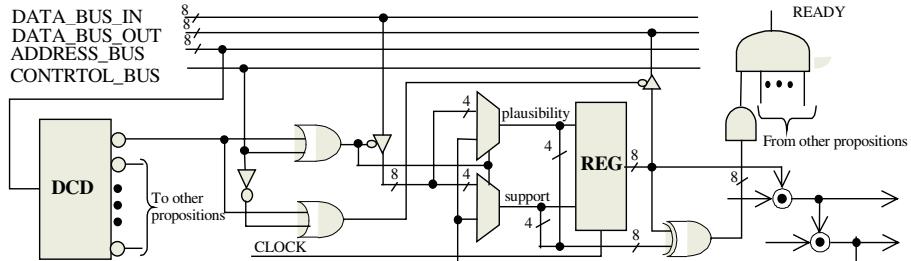


Fig. 2. The hardware structure of a proposition

The KB is given as a text file. This file is parsed and the network representation is automatically generated in a VHDL source code file, by eliminating loops. The operators and the propositions are already described in *predefined* VHDL code (this part is independent of the provided KB), but the actual network's structure is *dynamically generated* at this stage by the software (SW) part of the application.

In the execution phase, information is supplied to the system with evidences as external assumptions for propositions. The observations can be expressed in natural language, then translated in belief intervals and encoded for the network propagation.

The system was tested in two diagnosis fields: the medical and the HW technical support and debugging. A SW (Java-based) and a HW (VHDL and Xilinx FPGA-based) implementations were realized. The HW solution's advantages appeared to be obvious: apart of the intrinsic higher speed of the HW implementation, the *parallel processing* and *parallel propagation* of the belief intervals in the network yielded an increased performance.

The system's intrinsic pipeline-like structure significantly increases its performance; this depends mainly on the number of operators in the most complex rule of the KB, which gives the number of levels of combinational logic between the data buffers. For example, for a rule with 4 levels of logic operators in the KB, the maximal working frequency reported was 109.7 MHz, while for a simple KB (with only one level of operators), it was 120 MHz. For a KB containing 30 rules, with 13 external assumptions, the SW results were obtained after 550 ms, while the HW results were obtained in 142.6 ns (17 clock cycles * 8.39 ns). For a KB containing 60 rules, with 24 external assumptions, the SW results were obtained after 2140 ms, while the HW results were obtained in 234.92 ns (28 clock cycles * 8.39 ns).

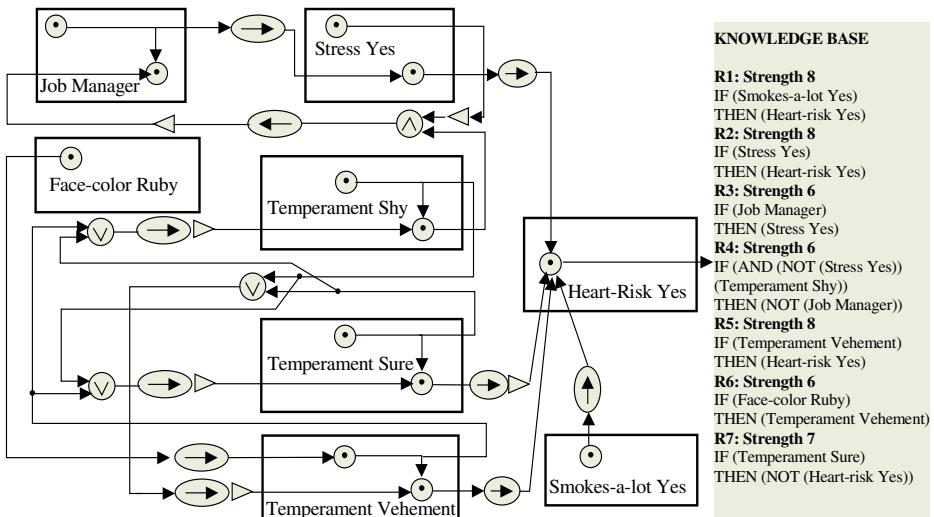


Fig. 3. The medical knowledge base and its network representation

5 Conclusions

A process of belief revision based on uncertainty propagation was proposed. Uncertainty was represented using a theory based on belief intervals defined in terms of subjective linguistic estimates. The KB is translated into an equivalent network representation. The HW architecture is automatically generated by SW, resulting in a VHDL description of the network that corresponds to the KB. After the VHDL synthesis process, the design is downloaded in a Xilinx Virtex FPGA for execution.

The advantages of the HW over the SW implementation, were underlined. They consist mainly of a greater speed, obtained by exploiting the intrinsic parallel features: *parallel processing and propagation* of belief intervals values through the network.

References

- [1] A. Bonarini, et. al., "Belief Revision and Uncertainty: a proposal accepting cyclic dependencies", Dipartimento de Elettronica, Politecnico di Milano, Report n.90-067, 1990.
- [2] C. Cenan, "An expert system shell based on belief revision concepts". ACAM scientific journal, p.35-45, Cluj-Napoca, 1996.
- [3] G. Kleiter, "Bayesian diagnosis in expert systems", Artificial Intelligence, 1992, No. 54, p.1
- [4] O. Creț, et.al., "A HW Implementation of an Expert System Shell Based on Belief Revision Concepts", 4th International Conference on Technical Informatics CONTI' 2000, Timisoara, Romania, October, 2000.

Cluster-Driven Hardware/Software Partitioning and Scheduling Approach for a Reconfigurable Computer System

Theerayod Wiangtong¹, Peter Y.K Cheung¹, and Wayne Luk²

¹ Department of Electrical & Electronic Engineering,
Imperial College, London, UK
{tw1,p.cheung}@ic.ac.uk

² Department of Computing, Imperial College, London, UK
w1@doc.ic.ac.uk

Abstract. To achieve a good performance when implementing applications in codesign systems, partitioning and scheduling are important steps. In this paper, a two-phase clustering algorithm is introduced as a preprocessing step to an existing hardware/software partitioning and scheduling system. This preprocessing step increases the granularity in the partition design, resulting in a higher degree of parallelism and a better mapping to the reconfigurable resource. This cluster-driven approach shows improvements in both the makespan of the implementation, and the CPU runtime.

1 Introduction

Coarse grain partitioning can improve the performance of an implementation by increasing parallelism as reported in [1]. It is therefore not surprising that clustering methods, which tend to increase the granularity of tasks, can be applied to the partitioning problem with good effects. Furthermore, after clustering, the size of the task graph (or problem size) is reduced, and this benefits the runtime of the synthesis process.

In this paper, we introduce an algorithm to solve a multiple-objectives clustering problem, called the *two-phase algorithm*. Our clustering algorithm is further applied as a pre-processing step to the partitioning and scheduling algorithm, which maps and schedules tasks to the target system. System constraints including FPGA resources, shared resources conflicts (such as bus or memory contention), as well as reconfiguration time, communication overhead, processing overhead are all taken into account during the partitioning and scheduling process. We employed the tabu search algorithm for partitioning and list scheduling in the scheduler as previously report in [3]. The results of clustering, mapping, and scheduling are then implemented on the UltraSONIC reconfigurable computing platform [4]. In summary, the contributions of this paper are: 1) two-phase clustering algorithm designed for multi-objectives optimization and 2) integration and evaluation of the two-phase clustering algorithm with partitioning and scheduling in codesign systems.

2 Two-Phase Clustering Algorithm

In our clustering algorithm, the objectives are set to 1) minimize total communication time and execution time of all the tasks in the DAG¹, and 2) minimize critical path of the DAG. These are subjected to constraints including a maximum cluster size, a maximum number of edges on the new cluster, and the resultant graph must be a DAG.

For such a multiple-objective problem, we need to find a method that can achieve both objectives. Based on preliminary experiments, which reveal that 1) in order to minimize delay on the critical path, tasks on the critical path itself have the most impact, and 2) once a minimum critical path is found, further clustering to reduce system cost will increase the critical path delay from its minimum value, the two-phase clustering algorithm is introduced. During the *first phase*, the delay on the critical path is minimized as much as possible, while the *second phase* is responsible for reducing the system cost without increasing the delay on the critical path.

This algorithm clusters tasks in a hierarchical manner. The first phase combines pairs of task nodes for the best fitness value while shortening the critical path delay on each refinement step. In order to prevent being trapped in a local minimum, the delay on the critical path is allowed to hill-climb. This allowance is limited by a threshold value which is decreasing in each iteration step to continuously force critical path to reduce. Two questions arise at this stage, 1) what is the decrement scheme for the threshold value, and 2) how fast is the decrement rate. Too fast a decrement leads to being trapped in a local optimum; too slow a decrement results in long search time and possible failure to find the minimum point. This is essential the idea of employing a *cooling schedule* found in simulated annealing optimization.

In the second phase, we attempt to reduce the system cost (overall communication time and computation time) without increasing the critical path value obtained in the first phase. This is achieved by only considering nodes and clusters outside the critical path as candidates for merging. The second phase is terminated when no further merging satisfies all the constraints or a maximum number of successful clustering attempts have been met.

3 Experimental Results

3.1 Comparing with Greedy Algorithm

We compare the effectiveness of the two-phase clustering algorithm described above with a straight forward greedy algorithm by applying them to randomly generated task graphs with 100, 200 and 400 tasks nodes with different granularity of tasks varying from 0.1 to 1.0. (Task granularity can be defined in many different ways. In this paper, we use the ratio of the average computation time and communication time as defined in [2].)

¹ This is referred to as the *system cost* in the rest of the paper.

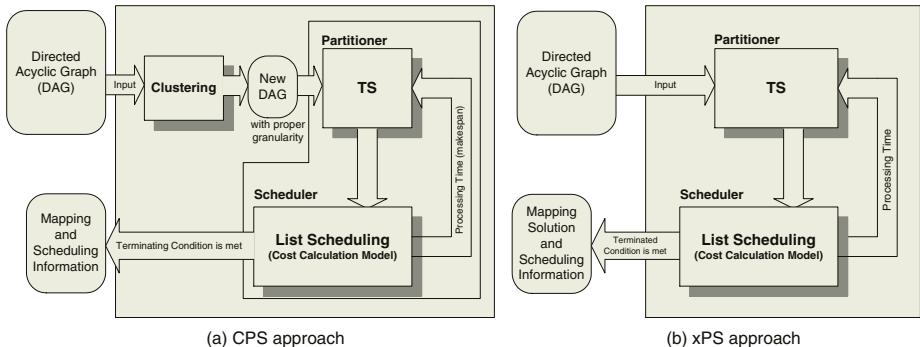
Table 1. Comparisons between two-phase approach and greedy approach

Reduction of	Two-Phase algorithm	Greedy algorithm
Critical path delay	30.2%	8.6%
The system cost	14.4%	20.6%
Number of nodes	21.3%	28.1%
Number of edges	16.2%	21.3%
CPU time (PIII 866MHz)	166 sec	362 sec

Table 1 summaries the overall improvement as a result of applying the two clustering methods. From the table, the two-phase method reduces the critical path delay by an average of 30.2%, which is significantly larger than the 8.6% obtained using the greedy algorithm. Since critical path delay has a direct impact on the makespan produced after partitioning and scheduling, this is a significant and useful improvement.

3.2 Combining with the Existing Partitioning and Scheduling Program

The cluster-partition-schedule approach (called CPS) shown in Fig. 1(a) is compared with the one without clustering step (called xPS) shown in Fig. 1(b). As can be seen in Fig. 2, the CPS strategy yields better makespan (between 13% and 17% improvement) and faster runtime (by around 16%) than that without clustering.

**Fig. 1.** Algorithm structuring of two different approaches

For a real application, the FFT algorithm is selected as a case study. For each task, the software execution time is obtained by profiling tasks on the PC, while the hardware running time and area are obtained using Xilinx development tools. Parameters such as reconfiguration time, bus speed, FPGA size, are all based on the UltraSONIC platform [4] with two reconfigurable processing elements.

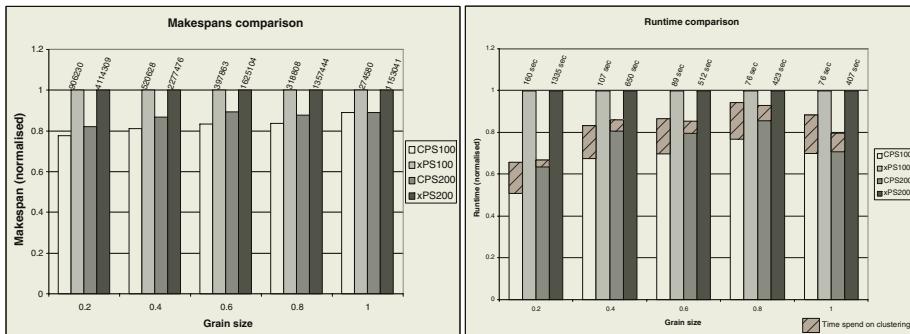


Fig. 2. The comparisons demonstrated in column graphs

Results show that our clustering algorithm can help partitioning and scheduling to reduce the overall makespan of the FFT implementation by around 14%~15% in 8-point and 16-point FFT implementations respectively. This reduction is comparable to average values of improvement getting from random graphs as described earlier.

4 Conclusions

The two-phase clustering algorithm modifies the granularity of the tasks in the DAG in order to improve the critical path delay and the overall communication time and node computation time. The clustering algorithm presented in this paper is successfully combined with our partitioning and scheduling algorithm for mapping abstract task graphs onto a realistic reconfigurable computing system. Using our algorithm to implement the FFT algorithm onto the UltraSONIC reconfigurable computer shows promising results.

References

1. Srinivasan, V.; Govindarajan, S.; Vemuri, R., “Fine-grained and coarse-grained behavioral partitioning with effective utilization of memory and design space exploration for multi-FPGA architectures”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, pp. 140 -158, 2001.
2. Palis, M.A.; Liou, J.-C.; Wei, D.S.L., “A greedy task clustering heuristic that is provably good”, *Parallel Architectures, Algorithms and Networks*, 1994.
3. Wiangtong, T.; Cheung, P.Y.K.; Luk, W., “Comparing Three Heuristic Search Methods for Functional Partitioning in HW-SW Codesign”, *International Journal on Design Automation for Embedded Systems*, vol. 6, pp. 425-449, July 2002.
4. Haynes, S.D.; others, a., “UltraSONIC: A Reconfigurable Architecture for Video Image Processing”, *Field-Programmable Logic and Applications (FPL)*, 2002.

Hardware-Software Codesign in Embedded Asymmetric Cryptography Application – A Case Study

Martin Šimka¹, Viktor Fischer², and Miloš Drutarovský¹

¹ Department of Electronics and Multimedia Communications,
Technical University of Košice,
Park Komenského 13, 04120 Košice, Slovakia
`{Martin.Simka,Milos.Drutarovsky}@tuke.sk`

² Laboratoire Traitement du Signal et Instrumentation,
Unité Mixte de Recherche CNRS 5516, Université Jean Monnet,
Saint-Etienne, France
`fischer@univ-st-etienne.fr`

Abstract. This paper presents a case study of a hardware-software codesign of the RSA cipher embedded in reconfigurable hardware. The soft cores of Altera's Nios RISC processor are used as the basic building block of the proposed complete embedded solutions. The effect of moving computationally intensive parts of RSA into an optimized parameterized scalable Montgomery coprocessor(s) is analyzed and compared with a pure software solution. The impact of the tasks distribution between the hardware and the software on the occupation of logic resources as well as the speed of the algorithm is demonstrated and generalized.

1 Introduction

The protocols in public key cryptography (e. g. RSA [1]) are an excellent example for studying hardware-software codesign concept: the protocol and the key generation have a strong sequential feature, while the algorithm itself can be better realized in parallel and pipelined structures. System on a chip (SOC) offers the best solution: it can consist of an embedded processor and one or more coprocessors. However, the reconfigurable SOC has an extra aspect to be taken into account: both hardware and software part of the system are embedded in the (same) chip. So even entirely software solution occupies hardware resources inside the chip – logic elements for processor implementation and, above all, embedded memory for data and program storing.

2 RSA Algorithm

RSA was proposed by Rivest, Shamir, and Adleman in 1978 [1]. Basic mathematical operation used to encrypt a message X is modular exponentiation [1]:

$$Y = X^E \bmod M \tag{1}$$

that a binary or general m -nary methods can break into a series of modular multiplications. All of these computations have to be performed with large k -bit integers (typical $k \in \{1024, 2048, \dots\}$).

To speed-up modular multiplication required in (1) the well-known Montgomery Multiplication (MM) algorithm [1] is used. It computes the MM product for k -bit integers $X, Y: MM(X, Y) = XYR^{-1} \bmod M$. While the algorithm is simple and can be controlled by software, the MM is an expensive operation suitable for implementation in an algebraic coprocessor.

3 Scalable Montgomery Multiplication Coprocessor and Its Interfacing with the Embedded Processor

We have tested two different approaches to implement scalable processing element (PE) (see Table 1): the first one (called MWR2MM_CSA) is based on a redundant form with Carry-save adders [2], the second one (called MWR2-MM_CPA) has a FPLD-optimized architecture based on Carry-propagated structure present practically in all kinds of modern FPLDs. The core of both approaches is a modified Multiple Word Radix-2 Montgomery Multiplication algorithm [2], which imposes no constraints to the precision of operands.

Table 1. Comparison of the PE size and speed for APEX Altera FPLDs

Family	Carry Propagate Adders			Carry Save Adders		
	Length w (bits)	Size (LEs)	Speed (MHz)	Length w (bits)	Size (LEs)	Speed (MHz)
APEX	8	59	161	8	81	232
	16	115	129	16	161	202
	32	229	99	32	321	170

The data path is organized as a cascade chain of PEs (stages) connected to the data memory (implemented in EMBs - Embedded Memory Blocks) (see Figure 1). The maximum degree of parallelism for this organization is found as: $n_{max} = \lfloor \frac{e}{2} \rfloor$. The coprocessor has 3 main parameters (word length w , number of words e , and number of stages n) that can be changed according to the required area of the implemented coprocessor and the required timings for MM computations (n, w) or the security level (e). This approach gives an unusual flexibility to the processor-coprocessor codesign.

As the embedded processor is used a Nios soft-core processor from Altera [4], that includes a CPU optimized for SOC integration. This configurable, general-purpose RISC processor can be combined with user-defined logic and programmed into Altera FPLDs. Nios supports both 16- and 32-bit variants with 16-bit instruction set. Features of an parameterized Avalon bus included in the Nios are used for a flexible connection of the processor and the MM coprocessor(s).

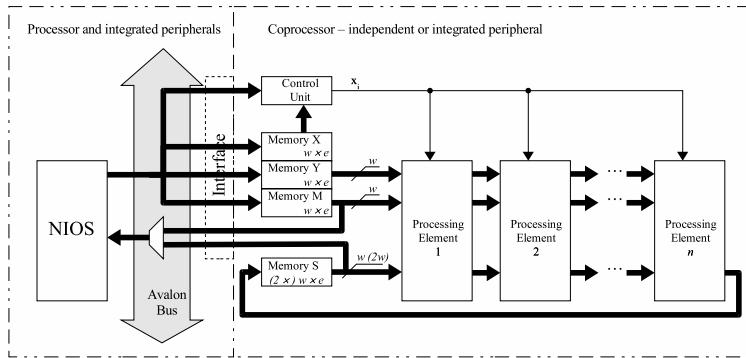


Fig. 1. Block diagram of the Nios processor and the MM coprocessor interconnection

4 Analysis and Discussion of Selected Solutions

To evaluate software/hardware proportion in the solution and its impact on the size and the speed of the system, we have assumed four different representative architectures implemented on Altera Nios development board with APEX EP20K200EFC484-2X [4].

Fully Software Solution. Time-critical part of the software implementation (MM operation) has been programmed in the Nios assembly language using all known optimization techniques. The 32-bit Nios processor has taken 2583 logic elements (LEs) and 45 EMBs including a hardware integer multiplier (used by MUL instruction) occupying 446 LEs. The execution times of the RSA operation ($k = 1024$) in Nios clocked by 50 MHz are: 46 ms for encryption with $E = F_4$, and 845 ms for decryption with CRT algorithm.

Processor with One Pipelined MM Coprocessor. In this version a 16-bit Nios version (occupying only 1275 LEs and 27 EMBs) has been used. Table 2 presents the area occupations and the RSA timings based on the use of the 16-bit MM coprocessor (clocked by 100 MHz) with implemented MWR2MM_CPA algorithm. Note that times includes also pre-computation performed by the Nios processor. Therefore the overall speed is not decreasing linearly with the number of stages. The MM coprocessor requires extra memory resources to store sub-results S , but on the other hand the program code and the program memory is smaller.

Processor with Two Pipelined MM Coprocessors. For typical decryption exponents there are about $k/2$ non-zero bits. Parallel execution on two separate coprocessors can decrease the average execution time to about 66% of the execution time with one coprocessor of the same size. Similarly, during the decryption process based on the CRT algorithm, the computations can be executed in parallel, and thus decrease the execution time to about 50%. However, two coprocessors require two times more hardware resources (LEs and EMBs). When these resources are available, better solution is to add two times more stages to the one coprocessor solution.

Table 2. Execution times of RSA operations with the MM coprocessor

Length $e \times w$	# of stages n	Encr (ms)	Decr (ms)	Size (LEs)	Length $e \times w$	# of stages n	Encr (ms)	Decr (ms)
1024	2	14	155	429	2048	2	55	1146
1024	8	9	56	1462	2048	8	35	354
1024	16	8	39	2837	2048	16	31	222

Fully Hardware Solution. The implementations realizing the whole system as a parallel hardware architecture are the fastest solutions. The disadvantage of this kind of solutions is that all input data are expected to be already stored in a memory before the computation. And in that case even small changes in the implemented protocol may require the remake of the whole design. The software control of the process can allow the user to obtain very flexible and reusable solution. Therefore we do not see the fully hardware solution as a suitable way to implement asymmetric encryption algorithm in FPLDs.

5 Conclusions

Parameterized processors embedded in reconfigurable hardware are becoming a standard building block in complex SOC designs. It was demonstrated that execution of carefully selected parts of the algorithm in properly optimized co-processors increases considerably the speed of the complete RSA algorithm. Even more, it was shown that hardware resources used in this combined hardware-software design are not more significant than in a pure software solution, because the combined design can use simpler embedded processor.

Acknowledgements

This work has been done as a part of the project CryptArchi (project number CR/02 2 0041) of the French national program ACI Cryptologie.

References

1. J. A. Menezes, P. C. Oorschot, and S. A. Vanstone. *Applied Cryptography*. CRC Press, New York, 1997.
2. A. F. Tenca and C. K. Koc. A scalable architecture for Montgomery multiplication. In C.K. Koc and C. Paar, *Cryptographic Hardware and Embedded Systems*, number 1717 in Computer Science, pages 94–108, Berlin, Germany, 1999. Springer Verlag.
3. M. Šimka and V. Fischer. Montgomery Multiplication Coprocessor for Altera Nios Embedded Processor. *Proceedings of the 5th International Scientific Conference on Electronic Computers and Informatics 2002*, pages 206–211, Kosice, Slovakia, October 2002.
4. Nios Soft Core Embedded processor, www.altera.com/nios

On-chip and Off-chip Real-Time Debugging for Remotely-Accessed Embedded Programmable Systems

Jim Harkin, Michael Callaghan, Chris Peters,
Thomas M. McGinnity, and Liam Maguire

Intelligent Systems Engineering Laboratory,
Faculty of Informatics, University of Ulster, Magee Campus
Northland Rd, Co. Derry, N. Ireland, BT48 7JL, UK
jg.harkin@ulster.ac.uk

Abstract. Embedded programmable systems are becoming common in system designs, resulting in the need for educational institutions to teach advanced embedded systems design and develop debugging competence in students. Remote laboratory experimentation provided as part of a web-based distance learning allows flexible access to on-campus resources free of time or geographical constraints. However, adapting and redeveloping existing software and hardware resources to this purpose is both time consuming and expensive. This paper introduces a remote-access laboratory architecture, which extends current e-learning strategies to provide real-time debugging for embedded programmable systems via the web. An example experiment illustrates the on-chip and off-chip real-time debugging capabilities of the laboratory.

1 Introduction

Electronic engineers are experiencing the need to specialise and re-train in advanced fields, particularly in the area of embedded systems. To provide appropriate education in this field requires practical experience in laboratory experimentation. Distance learning via the web offers professional engineers and students the flexibility to access educational material to suit their lifestyles. Remote experimentation offered as part of a web-based distance learning approach allows remotely located users to develop skills which deal with real systems and instrumentation.

Several remote-access labs are currently available via the web. For example, the University of Zagreb [1] and the Open University [2] provide on-line access experiments in embedded system design. However, the main disadvantage of current laboratories is the inability to provide users with access to debugging instrumentation (logic analysers) and modern embedded systems components (FPGAs).

The Engineering and Physical Sciences Research Council (EPSRC) funded project at the University of Ulster, aims to develop a remote-access laboratory for an Embedded Systems module on a distance learning Masters Degree course [3]. The project aims is to provide experience in embedded systems experiments to supplement the lecture material presented within the distance learning module. This paper presents a remote-access architecture and programming model, which addresses the deficiencies of current remote-access labs and illustrates the integration of modern debugging instrumentation and embedded systems technology. In particular, the

ability to perform on-chip an off-chip real-time debugging is demonstrated. Section 2 presents the architecture and programming model, section 3 demonstrates the real-time debugging capabilities of the laboratory and section 4 provides a conclusion.

2 Remote-Access Embedded Systems Laboratory

This section presents the architecture and programming model for the remote-access laboratory. The remote-access lab consists of workstations coupled to various instrumentation and embedded systems. Access to the workstations is via a web-client application [4]. The switching architecture provides a method of inter-connecting the embedded systems boards and the various instrumentation to the workstation. Fig. 1 illustrates the architecture for connecting analysis points on multiple experimental embedded systems boards to the suite of embedded debugging instrumentation. The architecture illustrates 5 main data/control buses: 1) board control, 2) configuration data, 3) diagnostic data, 4) instrument data and 5) instrument SCADA (supervisory control & data acquisition). The board control, configuration data and instrument SCADA buses provide communication with the experimental workstation via parallel, serial and GPIB channels, respectively. Board control provides control/setup information for the embedded system boards' multiplexer devices.

The instrumentation SCADA bus uses the GPIB protocol to send and retrieve information between the workstation and the matrix and instruments. The configuration data bus provides download monitors for program code and FPGA configuration information to target boards. All three buses are connected to the workstation. The diagnostic bus contains 120 analysis points from the target experiment board. Twenty-four of these are fed directly to the logic analyser; the other ninety-six are fed to the switching matrix. The main switching component of the architecture is the 96x16 line switching matrix [4]. The switching connections to be

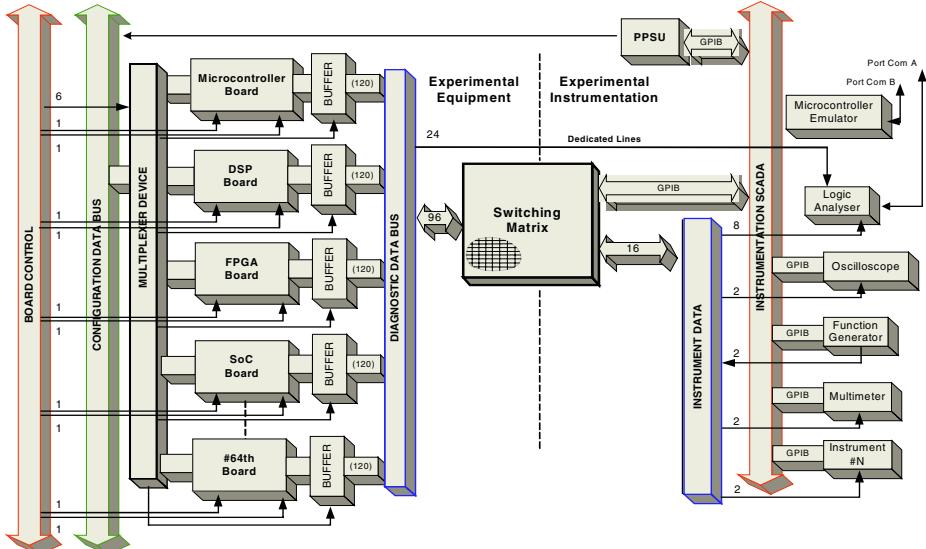
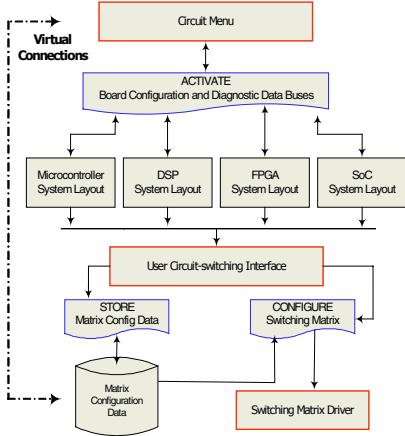
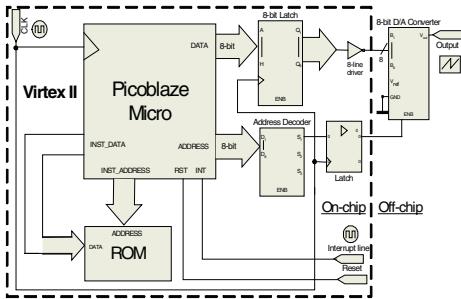


Fig. 1. Remote-access laboratory: Switching architecture

**Fig. 3.** Programming model**Fig. 3.** Embedded programmable system

closed, making a link between one of the 96 lines on the left of the matrix with one of the 16 lines on the right, are configured using the instrument SCADA bus (GPIB). Users configure the matrix to connect each of the instrument channels to one of the 96 available analysis points on the embedded systems. Configuration of the architecture is achieved using the programming model illustrated in Fig. 2. Programming can be divided into two tasks: 1) activating the embedded systems board access to the configuration/diagnostic buses and 2) configuring the switching matrix component.

Connecting instruments to the analysis points on the embedded systems is achieved using a circuit-connections window (Flash Macromedia) in the remote-access client application. This provides a visual display of the instrumentation and the points available for analysing. Selecting an experiment from a list of categories produces a layout of the desired Micro/DSP/FPGA embedded system in the circuit window. From the layout, the highlighted analysis points can be connected/wired to the instruments by dragging instrument channel icons to the particular points on the system and using the *Configure* icon. The user's instrument displays are provided in separate individual windows. In the model, the ACTIVATE program is called when an embedded systems experiment is selected. This activates and selects the associated board's multiplexer on the buses. Users highlight analysis points to be connected to the instrumentation by using the drag and drop facility. Clicking on *Configure* calls the CONFIGURE program which reads the matrix configuration data file and configures the switching matrix through the GPIB bus. The STORE program copies the configuration data of the switch matrix to a file. All low-level programming is achieved using C.

3 Real-Time Debugging

Fig 3 illustrates an example embedded programmable system experiment for debugging. The system includes the Virtex II FPGA with the Picoblaze 8-bit microcontroller IP core [5] and an 8-bit digital-analog converter (DAC) external to the FPGA. Users are required to create a program that will execute on the Picoblaze

and produce a waveform at the output of the DAC. Interrupting the Picoblaze will cause the program to temporally change the DAC output pattern. Users can debug the hardware-software experiment using the instrumentation in the lab. Laboratory users are able to program the Picoblaze, place and route the VHDL design and download the configuration to the Virtex II FPGA through the software available in the remote-access client [4]. Off-chip debugging (external to the FPGA) is achieved using the switching architecture and programming model, whereby signals external to the FPGA are connected to the laboratory instrumentation. Users connect signals to the logic analyser and oscilloscope using the circuit-connections window. In addition, reset and trigger buttons are also available in the window. The trigger-condition for the logic analyser is configured using software available in the remote-access client. By selecting the trigger, the program calls its interrupt service routine and prompts the logic analyser to commence sampling. From the signals displayed in the logic analyser software, users can identify any real-time anomalies in the system's functionality from the changing 8-bit bus pattern and toggling trigger line displayed in the logic analyser.

On-chip debugging is provided through Xilinx's ChipScope Pro [6]. This allows users to sample signals that are on-chip (internal to the FPGA) in real-time. In this debugging mode, a logic analyser (LA) core is placed inside the FPGA. The switching architecture's configuration data bus enables the transfer of the on-chip data samples from the core to the remote-access client for analysis.

This example experiment illustrates how users can remotely debug systems in real-time. The combination of the architecture and programming model details how users can sample signals in real-time, visualise the information and route it back to their remote desktop. In particular, the remote-access lab demonstrates how commercially available FPGA systems can be programmed, analysed and verified over the Internet.

4 Conclusion

A remote-access architecture and programming model have been presented which address the integration of modern embedded systems experimental equipment and debugging instrumentation for remote access over the web. The laboratory architecture illustrated the method to designing, re-configuring and debugging embedded programmable systems remotely.

References

- [1] Muzak, G, Cavrak, I: The Virtual Laboratory Project, Information Technology Interfaces Conference (2000) 241-246
- [2] Open University. <http://www.open.ac.uk/>
- [3] Intelligent Systems Engineering Laboratory. <http://isel.infm.ulst.ac.uk/>
- [4] Callaghan, M.J, Harkin, J, McGinnity, T.M, Maguire, L.P: An Internet-based Methodology for Remotely Accessed Embedded Systems, IEEE Conference on Systems, Man and Cybernetics (2002) 157 -162
- [5] Xilinx Picoblaze 8-Bit Microcontroller for Virtex-II Devices V1.0 (2002)
- [6] Xilinx ChipScope Pro 5.1i User Manual (2002)

Fast Region Labeling on the Reconfigurable Platform ACE-V

Christian Schmidt and Andreas Koch

Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23, D-38106 Braunschweig, Germany
schmidt, koch@eis.cs.tu-bs.de

1 Introduction

This work will revisit the computer vision application of labeling connected regions in images, and compare results achievable on current configurable architectures with previous work both by our group [1] [2] as well as one of the first attempts targeting the pioneering Splash-2 custom computing machine [3].

2 Algorithm Basics

The algorithm used in our new implementation is fundamentally similar to that of the previous version [1] [2]. For brevity, we will just highlight the changes here.

The algorithm expects as input a black and white image organized as a two-dimensional bit array. The output is a two-dimensional array of 16b words having the same dimensions as the input array, but now labeling all pixels within the same individual object with a unique integer identifier.

In addition to the image itself, we employ two data structures: The *adjacency map* represents an *is-adjacent-to* relation between two object labels. In some cases, data already entered into adjacency map may become invalid when additional adjacency relations are discovered later in the image. To reduce pressure on the memory holding the map, such updates are deferred until the core algorithm itself does not require access to the adjacency map. In the meantime, the adjacency corrections are stored in a separate data structure, called the *adjacency list*, as pairs of 16b integer labels. When the map data structure does become available again, entries in the list are removed and used to correct the corresponding map entries.

As in [1], the labeling procedure itself consists of three distinct phases: 1.) Pixel labeling and adjacency handling, 2.) transitive flattening, and 3.) adjacency-based object merging.

3 Platform Architectures

The algorithm presented in [1] was intended for execution on the Xilinx XC4010-based SPARXIL co-processor [4]. For a valid comparison with current device technology, we ported it to the ACE-V platform.

The configurable computer ACE-V [5] combines a reconfigurable compute unit (RCU, realized by a Xilinx Virtex 1000 FPGA) with a conventional RISC processor

(SUN microSPARC-IIep). The RCU has access to four independent 256Kx36b ZBT SSRAM memory banks and via a bus interface unit (BIU) to 64 MB of DRAM shared with the CPU.

The well-known Splash-2 architecture is described in detail in [6]. For our purposes, it consists of 17 XC4010 FPGAs that are connected both in a fixed manner as a systolic array and additionally in a variable manner using a configurable crossbar interconnect.

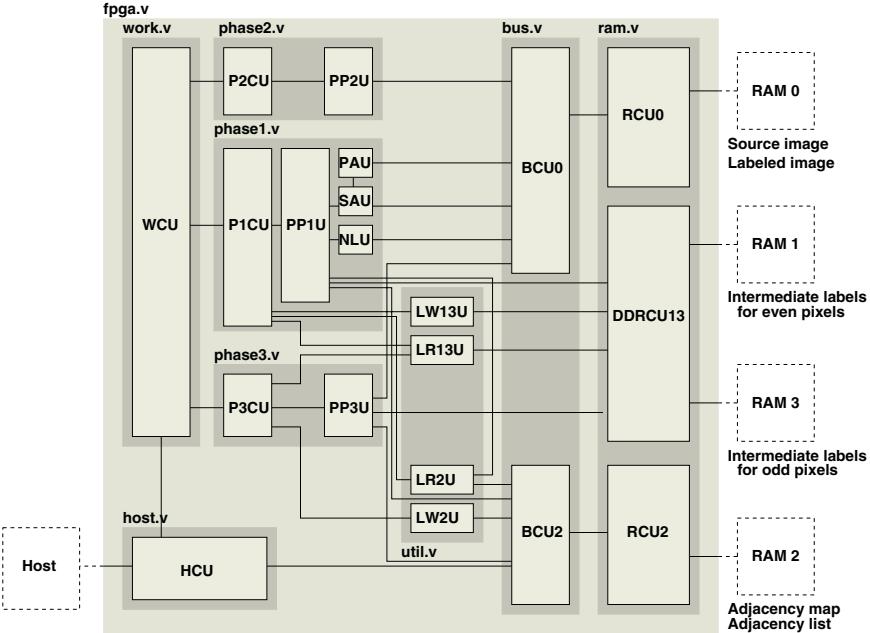


Fig. 1. Hardware architecture

4 Hardware Implementation

Our revised hardware is implemented according to the architecture shown in Figure 1. The Work Control Unit WCU controls the global execution of the algorithm. It accepts a start command from the host CPU and then hands control to the local controllers P1CU, P2CU and P3CU for each processing phase. The WCU also supervises the Host Control Unit HCU, which in turn accepts slave-mode data transfers initiated by the CPU from and to local memories (this is suspended during RCU execution). The HCU also accepts parameters from the host such as the image dimensions and whether to renumber labels consecutively. Furthermore, it also makes per-phase profiling data available to the host for benchmarking (see Section 5).

Each of the three processing phases relies on a highly optimized pipeline unit (PP1U, PP2U and PP3U) to actually perform the computations. In Phase 1, the background

task for managing and evaluating the adjacency list is implemented in parallel to the pipeline: The **NLU** provides the next label to use and enters it in the adjacency map, the **SAU** stores a adjacency correction in the adjacency list, and the **PAU** processes list entries for correcting the adjacency map.

Phase 1 and 3 (which actually operate on the image) rely on the dedicated address generators **LW13U**, **LR13U**, **LR2U** and **LW2U** for storing and retrieving image data. These units also handle special cases such as clipping on the edges of the image. The memory accesses themselves are processed in two bus control units **BCU0** and **BCU2** that multiplex address and data lines. The associated RAM control units **RCU0** (source image, labeled image) and **RCU2** (adjacency map and list) handle bidirectional to unidirectional bus conversion, and delay write data according to the ZBT SSRAM access protocol. **DDRCU13**, the RAM control unit responsible for accesses to the intermediate label array in RAM banks 1 and 3, is a special case: It uses both banks to present an external view of a single bank capable of *simultaneously* reading and writing to alternating even/odd addresses. This access pattern is always used when scanning over the image in Phase 1 and 3.

5 Evaluation

Table 1 compares the various realizations described for processing a 512x512 image. Note that the current implementation can handle any image size of less than 2^{18} total pixels.

Table 1. Area and performance comparison

Algorithm	Platform	RCU	RAMs	Area[LUTs]	Clock[MHz]	Time[ms]
[3]	Splash-2	9x XC4010	9	< 7200	10.0	33+
[1]	Emulated SPARXIL	1x XCV1000	2	2000	33.5	31
New	ACE-V	1x XCV1000	4	2334	36.0	15

The Splash-2 version [3] is not fully comparable with the other solutions, since it is optimized for real-time processing of a stream of video frames at a fixed 30 fps. To this end, it uses *duplicated* hardware for Phase 2 and 3 of the algorithm to process two video frames in parallel. Furthermore, this frame rate is not guaranteed as the hardware can begin to drop frames when they become too complicated to process in the allotted time slot.

Figure 2 shows the execution time in relation to the image size processed. This compares current RISC (SUN UltraSPARC+ 900MHz) and CISC CPUs (AMD Athlon XP 1533MHz) to the ACE-V reconfigurable computer running at 36 MHz. The ACE-V easily beats the RISC even when a highly optimizing compiler is being used and is only slower by 1.16 compared to the AMD CISC (which has 42.6x the clock speed). On average, the improved algorithm requires only 2.5 clocks per pixel. This is a considerable improvement over the design in [1], which also had a higher degree of data-dependence and required up to 6 clocks per pixel.

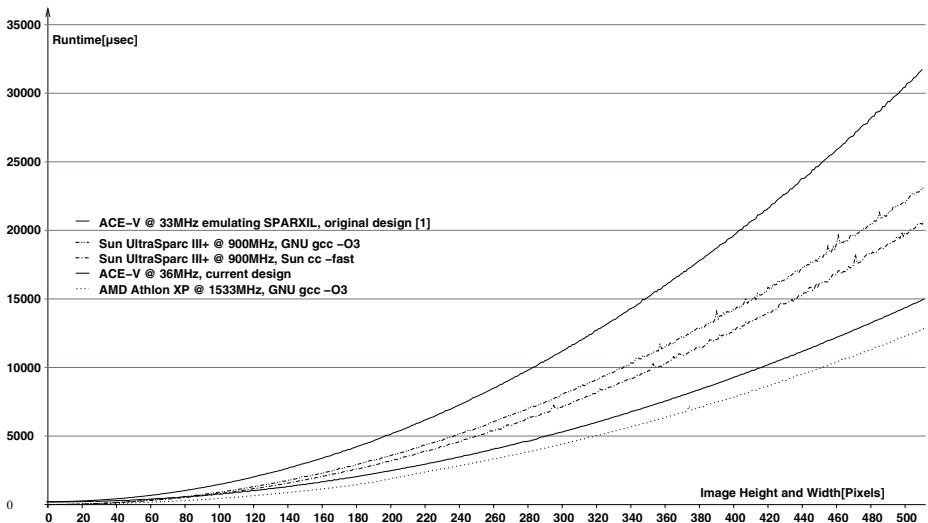


Fig. 2. Performance comparison with RISC and CISC CPUs

6 Potential Improvements

The performance of the new ACE-V based implementation could be improved even further: The factor limiting the clock speed is currently the speed of the HCU and its connection to the *external* BIU. After the redesign presented here, the *internal* computation pipelines PP1U, PP2U and PP3U have much shorter critical paths than in [1], allowing a potential double-clocking of these parts in the 50-60 MHz range even on the slow speed grade -4 Virtex device currently used.

References

1. Koch, A., Golze, U., “Practical Experiences with the SPARXIL Co-Processor”, *Proc. 31st Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove (CA), 1997
2. Meyer, K., “Entwurf eines FPGA-basierten Co-Prozessors zur Objekt-Etikettierung in der Bilderkennung”, *diploma thesis*, Tech. Univ. Braunschweig (E.I.S.), Germany, 1997
3. Rachakonda, R.V., Athanas, P.M., Abbott, A.L., “High-Speed Region Detection and Labeling using an FPGA-based Custom Computing Platform”, *Proc. Field Programmable Logic and Applications (FPL)*, Springer, 1995
4. Koch, A., “A Universal Co-Processor for Workstations”, in *More FPGAs*, eds. Moore, W., Luk, W., Oxford 1994
5. Koch, A., Golze, U., “A Comprehensive Prototyping Platform for Hardware-Software Code-design”, *Proc. Workshop on Rapid Systems Prototyping*, Paris, 2000
6. Arnold, J.M, Buell, D.A., Davis, E.G., “Splash 2,” *Proc. Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1992

Modified Fuzzy C-Means Clustering Algorithm for Real-Time Applications

Jesús Lázaro, Jagoba Arias, José L. Martín, and Carlos Cuadrado

Escuela Superior de Ingenieros,
University of the Basque Country
Alameda Urquijo s/n
48013 Bilbao, Spain

{jtplaarj,jtparpej,jtpmagoj,jtpcuvic}@bi.ehu.es
<http://www.ingenierosbilbao.com>

Abstract. The fuzzy approach in image processing is taking each day greater importance. It is greatly due to the fact that every new application of artificial vision is closer to human vision. This means that tightly knot algorithms are not always a good solution and a more “imprecise” and fuzzy approach is desirable. This paper describes a modified Fuzzy C-Means algorithm intended to be implemented in hardware. The original algorithm was modified to match the desired level of parallelism, speed and to simplify the hardware implementations.

1 Introduction

The Fuzzy C-Mean (FCM) algorithm is used in a great variety of image processing designs. It is used from satellite image analysis to OCR systems. Since the fuzzy C-Mean algorithm is very time consuming, special attention to performance must be taken in most applications. This is the reason why, since its introduction, several particular implementations have been developed to boost its efficiency [1][2][3][4].

In this paper, a real time C-Mean algorithm is described. The boost of performance of this circuit is far beyond other implementations but, to achieve it, several particularities have been added. One of them is the definition of real-time. The circuit that is proposed in this paper is capable of clustering grey scale video stream with a resolution up to 256x256 pixels per image and 50 images per second (both fields).

2 Fuzzy C-Mean

A clustering approach that involves minimization of some objective function, or error criterion, belongs to a family of objective function clustering algorithms [5]. A common goal of these algorithms is to find an “optimal” partitioning of feature space given a collection of data samples. The algorithms that, in addition to minimizing an error function, estimate the prototypes of resulting

classes within a partition, are often referred to as C-Means clustering algorithms, where the integer c stands for the number of classes. If the classes, for which the prototypes are estimated, are allowed to be fuzzy, the Fuzzy C-Means (FCM) clustering algorithm may be used [6].

The fuzzy C-Means algorithm minimizes the least-squares functional that is given by a generalized within-groups sum of square errors function:

$$J_m(U, z) = \sum_{k=1}^n \sum_{i=1}^c u_{i,k}^m \cdot d_{i,k}^2, \quad (1)$$

where $U \in M_{fcn}$ is a fuzzy c -partition of X ; $z = (z_1, z_2, \dots, z_c) \in R^{cp}$, with $z_i \in R_p$ as the cluster center or prototype of the i^{th} class; $d_{ik}^2 = \|x_k - z_i\|$, with $\|\cdot\|$ being any inner product induced norm on R^p ; and *weighting* or *fuzzy exponent* $m \in (1, \infty)$. Clearly, $J_m : M_{fcn} \times R^{cp} \rightarrow R^+$. The optimum is reached when the fuzzy partition matrix U^* and a collection of prototypes z^* are found such that J_m is minimized. That is, when the weighted within-groups sum of distances between the samples and the prototypes is the smallest possible.

The solutions of minimization are least-squared error stationary points of J_m . The necessary conditions for minimization of J_m are derived in [5]. The necessary conditions for minimization of J_m can be written as:

$$u_{i,k} = \left(\sum_{j=1}^c (d_{i,k}/d_{j,k})^{\frac{2}{m-1}} \right)^{-1}, \quad \forall i, k, \quad (2)$$

$$Z_i = \frac{\sum_{k=1}^n u_{i,k}^m \cdot x_k}{\sum_{k=1}^n u_{i,k}^m} \quad (3)$$

The convergence theory of the FCM algorithm was initially studied in [5][7] and later improved in [8][9].

3 Modified Fuzzy C-Means

The FCM has several problems to be implemented n hardware. One of the first problems is the number of clusters. This number cannot be left as a variable since the amount of memory and circuits depend heavily on it. In fact, the division into two clusters needs the smallest amount of memory.

Moreover, the fuzziness factor m appears as exponent in several points of the algorithm: in equation (2) as the denominator of an exponent and in equation (3) as the exponent itself. The implementations of fractional exponents is such a difficult task that from equation (2) we obtain an “optimum” m of 2. This election makes equation (2) easier to calculate since obtaining the square of a number is a feasible matter.

A second problem is the initialization of the $U \in M_{fcn}$ matrix. This is normally done by using a random number generator (this is the method used by the Matlab algorithm). Such a circuit would increase the size of the resulting circuit

and would only be used once every running time. This problem was solved using the input image as the initialized U matrix.

The third problem lies in the loop section. A close analysis shows that it is necessary to iterate through all the input data and the U matrix, in order to obtain the fuzzy centers. In addition, the new U matrix is calculated from the old U matrix, the data and the fuzzy centers. This means that, for each picture, two iterations through the input data are needed or, in other words, the input data should be stored and read twice in the period of time between images. This is practically impossible in a real time application. To solve this problem, the old centers can be used to obtain the new U matrix. This means that the new U matrix and centers can be obtained in the same iteration without needing to store the input image. Another implication of this particularity is that only the value of a single pixel is needed to obtain the corresponding element in the U matrix. Thus, the clustering can be performed as the image arrives, not being necessary to store the whole image to start the processing. This allows us to use a different frame in each iteration instead of storing in memory each image and iterate through it.

As it can be seen, the algorithm performs the same operations over to different data at the same time, one over $U_{i,1}$ and one over $U_{i,2}$. To efficiently use the silicon area, both operations have been performed with the same hardware. To do so, those pipelined parts are clocked at double rate and the special registers (such as the ones in accumulators) have been doubled as well. The ends of these pipes are two registers, one for each different input data.

4 Results

In this section, the results of the proposed algorithm can be seen. The algorithm has been implemented in Matlab, both using floats and integers (any rounding done towards zero). These two algorithms have been tested against the Matlab FCM algorithm with the root mean square as evaluation function.

The *Video 1* is a low motion picture of moving robots. *Video 2* is a video from a crowded corridor with people entering and leaving the scene. The *Video 3* is a medium motion picture with fish swimming in a fish tank.

In table 1 can easily be seen that the lowest motion the better, as it would be expected from an algorithm that instead of iterating over the same data, it uses the new data. Another interesting effect is the initial transitory.

5 Conclusions

This paper presents a modified Fuzzy C-Means algorithm primary intented for real time video applications. This algorithm reduces the needs of memory space and of information movements leading to a highly parallelizable code. This code modifies several important points of the original FCM such as the way of obtaining the U matrix. The algorithm does not iterate over the same data set

Table 1. Root mean square error

Frame	FCM ÷ Float			FCM ÷ Int		
	Video 1	Video 2	Video 3	Video 1	Video 2	Video 3
1	78.3397	208.4966	20.6442	76.1842	209.3727	20.1699
2	36.4761	11.5990	7.0792	31.6272	14.1593	5.7707
3	9.2124	2.9654	1.5493	6.1617	6.3049	1.4090
4	1.6521	0.9975	1.4782	1.2300	4.6446	2.3872
5	0.3508	1.1299	1.7246	1.2843	4.5752	2.3628
6	0.2063	2.301	2.2093	1.2872	4.0168	3.3583
7	0.0235	9.1179	2.1011	1.2875	6.0489	2.9013

but uses a new image for each iteration. This code has been implemented in hardware by means of a programmable device.

References

- Richard J. Hathaway and James C. Bezdek. "Optimization of Clustering Criteria by Reformulation". IEEE Transactions on Fuzzy Systems, 3(2): 241-245, 1995.
- M.S. Kamel and S.Z. Selim. "New algorithm for solving the fuzzy clustering problem", Pattern Recognition, 27(3): 421-428, 1994.
- T.W. Cheng, D-B. Goldgof and L.O. Hall. "Fast clustering with application to fuzzy rule generation" Proc. IEEE Int. Conf. Fuzzy Syst. 2289-2295, 1995.
- J.F. Kolen and T. Hutcheson. "Reducing the Time Complexity of the Fuzzy C-Means Algorithm". IEEE Transactions on Fuzzy Systems, 10(2):263-267, 2002.
- J.C. Bezdek. "Pattern Recognition with Fuzzy Objective Function Algorithms", Plenum Press, New York, 1981.
- J.C. Bezdek, Robert Ehrlich and William Full, "FCM: The Fuzzy C-Means Clustering Algorithm", Computers and Geosciences, 10:191,203, 1984.
- J.C. Bezdek. "A Convergence Theorem for the Fuzzy ISODATA Clustering Algorithms", IEEE Transactions on Pattern Analysis and Machine Intelligence, 2(1):1-8,1980
- J.C. Bezdek, R.J. Hathaway, M.J. Sabin and W.T. Tucker. "Convergence Theory for Fuzzy c-Means: Counterexamples and Repairs". IEEE Transactions on Systems, Man, and Cybernetics, 17(5):873-877, 1987.
- M.J. Sabin. "Convergence and Consistency of Fuzzy c-means/ISODATA Algorithms". IEEE Transactions on Pattern Analysis and Machine Intelligence, 9(5):661-668, 1987.

Reconfigurable Hybrid Architecture for Web Applications

David Rodríguez Lozano, Juan M. Sánchez Pérez, and Juan A. Gómez Pulido

Department of Computer Science, University of Extremadura
Campus Universitario s/n. 10071 Cáceres (Spain)
{drlozano, sanperez, jangomez}@unex.es

Abstract. This paper describes a Reconfigurable Hybrid Architecture for the developing, distribution and execution of web applications with high computational requirements. The Architecture is a layered model based on a hybrid device (standard microprocessor and FPGA), for which has been designed and implemented a component as a web browser plug-in. Web applications are divided into two parts: an standard part and a reconfigurable part. The plug-in links the software and hardware applications, implementing an API for the management and access to the FPGA. A real implementation of the proposed architecture has been developed using Handel-C, the RC1000-PP platform, a compatible Intel CPU, and a Visual C++ ActiveX control plug-in.

1 Introduction

Internet has become a great platform for the developing, distributing and exploiting of software applications. Many companies are porting its corporative applications using web architectures and technologies that can be executed on network computers. Many works have demonstrated the viability of developing standalone [2] terminals based on reconfigurable hardware such as Field Programmable Gate Arrays (FPGAs) for the design of reconfigurable Internet platforms [4] with fully networking and multi-media capabilities [3].

Most of desktop and corporate software applications can be distributed and run using web technologies, so the Web Browser may become as the Universal Client of personal and network computers. However, because of limitation in performance of scripting languages and Java virtual machines, applications that require complex computations are actually outside of the web applications scope.

According to these considerations, it seems reasonable the use of a Reconfigurable Hybrid Architecture [1] that will allow a client web browser to request on demand the FPGA as an specific purpose co-processor. In this work we identify the different elements of the proposed reconfigurable hybrid architecture, develop a software component that implements an application programming interface (API) and prototype the hybrid architecture using a standard PC and a FPGA platform.

2 Reconfigurable Hybrid Architecture for Web Applications

The proposed architecture is a layered approach Fig. 1. Hybrid web applications are on the top layer. Next layer corresponds to the Reconfigurable Hardware Manager (RHM), which provides to client applications an API for easy access to hardware acceleration. The operating system (OS) is the next layer, it hosts the device drivers for the communication between the RHM and the FPGA. Finally, a standard CPU and a FPGA compose the hybrid processor layer.

Web applications designed for this architecture are divided into two parts: a *soft* part with the program code and a *hard* part with the co-processors descriptions. The program code are the web pages that support the application and user interface, HTML, JavaScript, VBScript or Java may be used. The hardware resources are the FPGA configuration files that describe specific co-processors required by the hybrid web application, Hardware Description Language (HDL) can be used.

In the described architecture the key component is the RHM plug-in, which provides an API that allows the easy development of accelerated web applications. The main functions of the RHM are: to control the correct download of the hardware resources files required by the hybrid web application and to manage the FPGA.

The RHM component can be downloaded as a software plug-in from a web server, different technologies may be used for this plug-in implementation, at this time Microsoft ActiveX, Netscape LiveConect and Sun Java Applets are the most widely used. ActiveX and LiveConect are platform and browser dependent, while Java Applets are limited by the efficiency of the Java Virtual Machine (JVM).

3 Architecture Implementation

An Intelx86 compatible processor is used for the fixed part, and the Celoxica RC1000PP with a Xilinx Virtex for the reconfigurable part. Windows 2000 is the hosting OS. Finally, we use a Microsoft ActiveX control to implement the RHM plug-in functions, and Microsoft Internet Explorer to host the hybrid application.

The Celoxica RC1000 board is a complete platform for the development of reconfigurable computing systems with full support in Handel-C [8]. The RC1000 platform has been used with success in previous works for developing reconfigurable data processing applications [6].

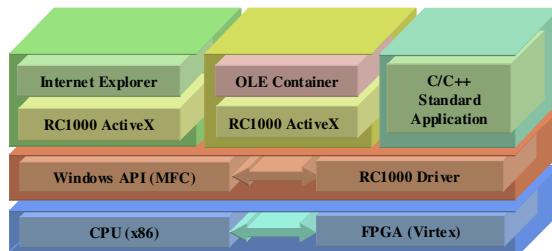


Fig. 1. Components and technologies of the Reconfigurable Hybrid Architecture prototype

Based on the IE components architecture and ActiveX technology [7], the RC1000AX control has been designed and programmed. The RC1000AX control implements all the functions required by the RHM plug-in. The control has been built using Microsoft Visual C++ 6.0, and the Microsoft Foundation Classes (MFC).

The RC1000AX control can be embedded as an OBJECT tag into the HTML pages that implements the hybrid web applications. Any other application that acts as a control container can host the RC1000AX control. For example, Microsoft Access or Excel may use the RC1000AX control for accelerating computational operations related with large databases or complex data sheets. The RC1000AX control can be used into an ASP (Active Server Pages) page running on a Windows Internet Information Server, this would allow hardware accelerating of complex server processes.

4 Application Design

In this section we describe the three phases involved in the development, distribution and execution of hybrid web applications.

Development: The design and implementation of FPGA configuration files is performed using Handel-C and DK1. The RC1000AX is a MFC ActiveX control.

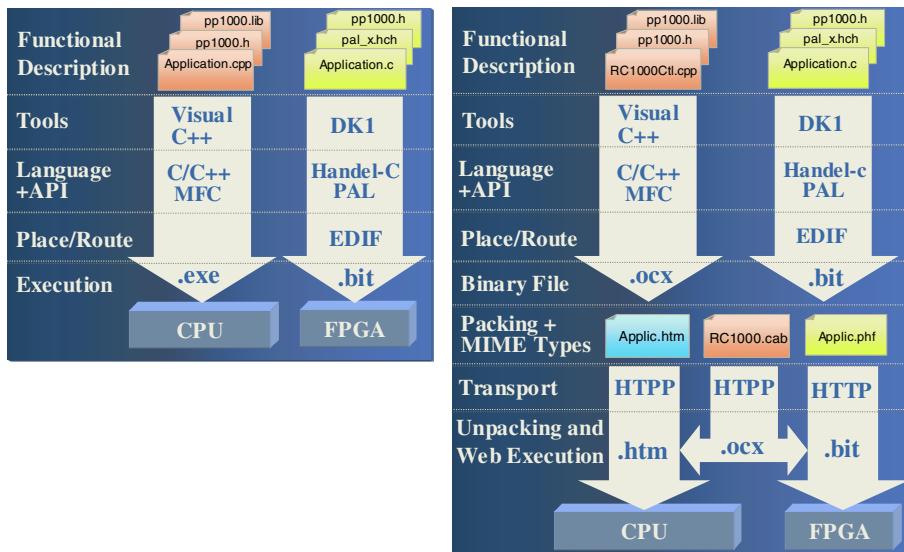


Fig. 5. Tools, languages, protocols and files required in a standard reconfigurable co-processing applications (left) and in a hybrid web application (right)

File Packing: The RC1000AX control and the FPGA description file are prepared for transport over the Internet/Intranet. The control is distributed inside cabinet file, containing all the information necessary to install, and register the control. The binary

FPGA configuration file and additional information is distributed inside a compressed proprietary format with .phf (Programmable Hardware File) extension.

Distribution and Execution: The client web browser is the target application for the web pages that support the skeleton and user interface of the hybrid web application, Internet Explorer also acts as container for the RC1000AX control, which is the target application of the new proprietary Programmable Hardware File format. A new MIME Media Type has been defined for indicating to web servers and client web browser how to deal with the files with .phf extension. The new MIME type and sub-type are: **application/x-rhmplugin**, and the associated file extension is **phf**.

5 Conclusions and Acknowledgements

In this work we have described the design and implementation a Hybrid Reconfigurable Architecture that allows transparent acceleration of web applications. We have used standard Internet and Web technologies for the programming and distribution of the hybrid architecture components. Web application developers can easily make use of hardware acceleration including the RHM plug-in into HTML pages. The future work includes the design and programming of the Java version of the RHM plug-in, a Java Applets will allow cross-platform and cross-browser hardware acceleration. Also we are working in the distributed version of the control using DCOM (Distributed Component Object Model) that will allow to web clients the use under demand of the reconfigurable device located in a DCOM server, reducing the cost of dedicated reconfigurable hardware.

This work has been partially supported by the TRACER project (TIC- 2002-04498-C05-01) of the Spanish Technology and Science Ministry.

References

1. Estrin, G.: Organization of Computer Systems: The Fixed-plus Variable Structure Computer, Proceedings of the Western Joint Computer Conference (1960) 33-40
2. Haenni, J.O., Beuchat, J.L., Sanchez, E.: RENCO: A Reconfigurable Network Computer. Proceedings IEEE Symposium FCCM'98, IEEE Computer Society Press, (1998) 288-289
3. Rodríguez, D., Zarallo, F., Conejo, I.: "Labograph": Graphic Information System over Labomat3 Platform. Proceedings JCRA'01 (2001) 102-110
4. Fallside, H., Smith, M.J.: Internet Connected FPGAs. Proceedings IEEE Symposium FCCM'00, IEEE Computer Society Press (2000) 289-290
5. Chappell, S., Sullivan, C.: Handel-C for co-processing & co-design of Field Programmable System on Chip. Proceedings JCRA'02 (2002) 65-70
6. Styles, H., Luk, W.: Customising Graphics Applications: Techniques and Programming Interface. Proceedings IEEE Symposium FCCM'00 77-88
7. Sankar, K.: Internet Explorer Plug-In and ActiveX Companion. MacMillan (1997)
8. Celoxica Limited: Handel-C Language Reference Manual (v3.1) <http://www.celoxica.com>

FPGA Implementation of the Adaptive Lattice Filter

Antonín Heřmánek, Zdeněk Pohl, and Jiří Kadlec*

Department of Signal Processing
Institute of Information Theory and Automation, CAS
Pod vodárenskou věží 4, 182 08 Prague 8, CZE
Tel. +420 266 052 432, xpohl@utia.cas.cz

Abstract. The paper presents the FPGA implementation of a noise canceler with an adaptive RLS-Lattice filter in the Xilinx devices. Since this algorithm requires floating-point computations, Logarithmic Numbering System (LNS) has been used. The pipelined lattice filter macro and input/output conversion routines has been designed. The implementation results are compared with an implementation on 32-bit IEEE floating point signal processor.

1 Introduction

The adaptive filtering has been in focus of DSP research since many years. With the growing computation capacity of modern digital devices, also the requirements on the properties of adaptive algorithms increase, namely on their convergence speed. Algorithms with suitable convergence typically use floating point arithmetic and often do not exist in fixed point version. One of such as algorithm was used in our implementation - an adaptive lattice filter.

The architecture of contemporary digital signal processors is highly optimized for vector operations. However not all DSP algorithms can make efficient use of this operation. In that case the performance of digital signal procesors degrades and the flexibility of FPGAs can provide a promising alternative.

Our team has recently investigated the use of an innovative approach to floating-point arithmetic, so called logarithmic number system (LNS) arithmetic [5, 6]. The proposed solution can provide a very fast floating-point-like multiplications, divisions and square-roots. Additions and subtractions are rather more complicated, but it was shown [3, 5] that it could be very efficient for complicated algorithms with a long critical path.

We have implemented an adaptive noise canceller. At present, a standard LMS filter or one of its variants is often employd at present. In our implementation the lattice filter [1, 2] was selected for the following reasons:

* This work has been partially suported by the Ministry of Education of the Czech Republic under Project LN00B096 and from EU Project RECONF2 (IST-2001-34016).

- very good numerical stability,
- faster convergence than LMS algorithm,
- better theoretical analysis in comparison with some modifications of standard LMS.

On the other hand, the adaptive lattice algorithm is an example of a modern DSP algorithm with high computational requirements and long critical path where the vector processing can not be used effectively. Moreover it requires all basic arithmetic operations – including two divisions on the critical path which degrade the implementation performance (in floating point as well as in fixed point versions). Fortunately, the LNS arithmetic is very suitable for such tasks.

The echo canceller implementation will be presented in this paper . The following section presents an overview of LNS arithmetic and its implementation for FPGA. Next, the hardware implementation of a real-time adaptive noise canceller is represented and implementation results are compared with our implementation on digital signal processor TI TMS320C6711. Finally, the last section will conclude the work.

2 LNS Arithmetic

The design employs the LNS arithmetic as an alternative approach to floating-point. A real number is represented in LNS as the fixed-point value of base two logarithm of its absolute value with a special arrangement to indicate zero and NaN. An additional bit indicates the sign.

Multiplication, division and square-root are implemented as simple as fixed-point addition, subtraction and right shift, but, unfortunately, addition and subtraction require more complicated evaluation. The algorithm of add/sub interpolation has been described in [3, 5, 6].

The resulting 19-bit LNS arithmetic library targeted in Xilinx Virtex devices is presented in Table 1. The mul, div and sqrt macros are very small and compact. They finish in two clock cycles, but they are not pipelined due to usage comfort. The add/sub macro is implemented as a dual-issue pipelined 10 stage macro where two add/sub units share the look-up tables.

3 Lattice Adaptive Noise Canceller

A lattice structure of the algorithm is shown on a signal-flow graph on Figure 1. The higher filter order, the more lattice bars (order stages) required. Full description of the update mechanism is beyond the scope of this paper and for a

Table 1. 19 bit LNS arithmetic implementation for Xilinx Virtex-1000

LNS Operation	Latency [Clock cycles]	Number of units in one macro	Pipelined	Macro size [Slices]	BRAMs
add/sub	10	2	Yes	720	6
mul,div,sqrt	2	1	No	≈ 160	0

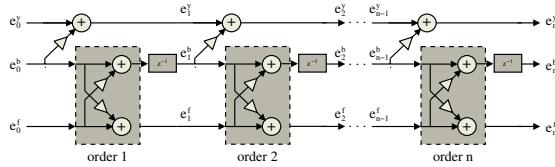


Fig. 1. Lattice computation data flow

reader can found its description on any DSP textbook. The signal-flow graph is shown in Figure 1. The update of the amplifier gain , which is computationally expensive, lies on the critical path of the algorithm. The complete critical path includes one division, four multiplications and two additions. As presented above, the add/sub unit is the most resource consuming part of the LNS arithmetic, therefore it is required to optimize its utilization. We have used pipelining as is shown on the Figure 2. There are four pipeline stages running in parallel. Each stage updates n^{th} order lattice filter sequentially (see Figure 1). Stages 1 and 3 and stages 2 adn 4 share one add/sub unit. To avoid conflicts, the stage 3(4) is started 4 clock cycles later than stage 1(2). The mul and div units are relatively small macros they are not shared between the stages.

The algorithm arrangement provides the lattice filter of order $4n$ with a latency equal to four input data periods . In other words, four data samples have to be acquired before the first output value will appear.

The noise canceller implementation is composed of the pipelined lattice filter in arrangement shown in the Figure 2 and of the data conversion modules. The conversion modules share the dual add/sub unit with the lattice filter to achieve its higher utilization.

Performance comparison of FPGA echo canceller implementations and our implementation on TI TMS320C6711 (IEEE 32-bit floating point device) of 100^{th} order are shown in Table 2. The noise canceller algorithm is optimized for each platform in a different way, therefore the maximal input data throughput was used as a comparison measure.

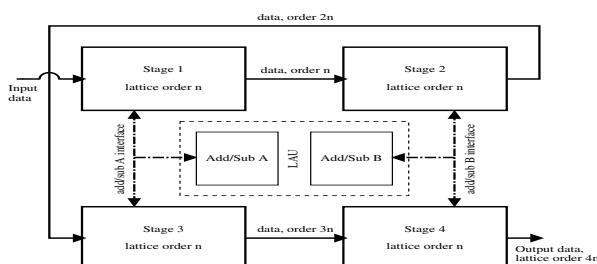


Fig. 2. Structure of the lattice filter implementation

Table 2. Implementation results of 100th order lattice

Arithmetic	Device	Clock Frequency [MHz]	Sampling clock [kHz]
LNS 19 bit	XC2V6000-6	83	48
LNS 19 bit	XCV1000-6	46	27
LNS 19 bit	XCV800-4	36	21
FP 20 bit	XC2V6000-6	140	32
FP 32 bit	TI C6711	225	1

It can be seen that the LNS arithmetic use lower clock frequency for the same sampling frequency of the input signal than floating point FPGA implementation and thus it consumes less power. Finally, the architecture of the TI TMS320C6711 device is not able to handle complicated data dependencies and parallel data access and therefore it can not be compared with FPGAs.

4 Conclusions

The lattice noise cancellation application has shown that FPGA can be a suitable platform for complicated DSP algorithms. The presented noise canceller is running at the XSV800 prototyping board in real-time with 16 kHz data frequency for the filter order 160. Thanks to the small overall latency of LNS arithmetic it proposes better solution for the algorithms with difficult data dependencies than floating point solution. In future, we would like to optimize the design for the Virtex-II device family and compare the results with modern FP arithmetic for FPGA.

References

1. B. Friedlander: *Lattice Filters for Adaptive Processing*, Proceedings IEEE, vol. 70, no. 8, August 1982, pp. 829-867.
2. J. Kadlec: *Lattice feedback regularised identification*. In Proc. of 10th IFAC Symposium on System Identification. Preprints. IFAC, Copenhagen 1994, pp. 277-282.
3. R. Matouek, M. Tichý, Z. Pohl, J. Kadlec, C. Softley: *Logarithmic number system and floating-point arithmetics on FPGA*. In Proc. FPL2002, Springer, Berlin 2002, pp. 627-636.
4. E.E. Swartzlander, and A.G. Alexopoulos: *The Sign/Logarithm Number System*. In: IEEE Trans. Computers, vol. 24, 1975, pp. 1,238-1,242.
5. J.N. Coleman, E.I. Chester, C.I. Softley, and J. Kadlec: *Arithmetic on the European Logarithmic Microprocessor*. In: IEEE Trans. Computers, vol. 49, 2000, pp. 702-715.
6. J.N. Coleman and E.I. Chester: *A 32b Logarithmic Number System Processor and Its Performance Compared to Floating Point*. In: Proc. 14th IEEE Symposium on Computer Arithmetic, Adelaide, April 1999, pp. 142-152.

Specifying Control Logic for DSP Applications in FPGAs

J. Ballagh, J. Hwang, H. Ma, B. Milne, N. Shirazi, V. Singh, and J. Stroomer

Xilinx Inc. 2100 Logic Drive, San Jose, CA 95124 (USA)

{Jonathan.Ballagh, Jim.Hwang, Haibing.Ma, Brent.Milne,
Nabeel.Shirazi, Vinay.Singh, Jeff.Stroomer}@xilinx.com

Abstract. New non-HDL programming models for signal processing in FPGAs have focused primarily on building high-performance data paths. Along with the ability to construct sophisticated custom signal processors comes increased requirements for creating complex control circuitry. Recent enhancements to System Generator for DSP begin to address this need by providing mechanisms that include co-simulation interfaces to extend Simulink with HDL semantics, automatic compilation from Matlab m-code into Simulink and VHDL, and embedded microcontrollers. In this paper, we describe how such mechanisms can be used in a QAM receiver designed for a CCSDS standard.

1 Introduction

Field-programmable gate arrays (FPGAs) are widely used in modern digital communication systems in part because of their ability to implement highly parallel custom signal processors. FPGAs are commonly employed for up/down conversion, forward error correction (FEC), adaptive equalization and synchronization, spectral analysis and digital filtering [4].

Recent design tool efforts have focused on providing new programming models and design methodologies for DSP in FPGAs, moving away from the traditional FPGA design flows that begin with hardware description languages (HDLs) and mirror approaches to ASIC design. System Generator for DSP [5] provides a programming model based on Simulink® [6] that allows the signal processing engineer to efficiently target an FPGA without requiring HDL expertise. For example, Dick and Harris have used System Generator to construct a 50 Megabit/s QAM demodulator that includes digital mixing, adaptive equalization, and carrier recovery [1]. We have extended this system to include concatenated FEC and packet framing according to the CCSDS standard for telemetry channels [2]. In addition, we augmented the system with logic that performs carrier quadrant correction and symbol demapping. These extensions require real-time control (e.g. for flow control between the Viterbi and Reed-Solomon decoders) and asynchronous control suitable for a microcontroller (quadrant correction in the QAM demapper).

In this paper, we describe one aspect of the receiver's control circuitry, the symbol demapper used for quadrant correction, implemented using the System Generator *m-code* block. This block compiles Matlab code into Simulink for simulation and VHDL during code generation. This bridging of an imperative model of computation (i.e. Matlab semantics) with discrete-time simulation (i.e. Simulink), and discrete-event RTL (which models hardware behavior), is representative of System Generator (which in fact preserves bit and cycle-accuracy) and other heterogeneous computing frameworks [3].

2 Symbol Demapping and Quadrant Adjustments

The QAM demodulator computes the phase error between the received quadrature carrier signals and the locally generated carriers, and adjusts the phase of the local carrier to drive the average error to zero. Although this approach suffices when the phase error is between $\pm 90^\circ$, it fails when the error falls outside this range. To correct for this, the receiver provides additional logic to rotate the QAM hard-decisions by 90° , 180° and 270° under the control of a quadrant select signal generated by an embedded microcontroller, also modelled in System Generator and implemented in the FPGA. Figure 1 shows how the demapped symbols are rotated between quadrants in the receiver.

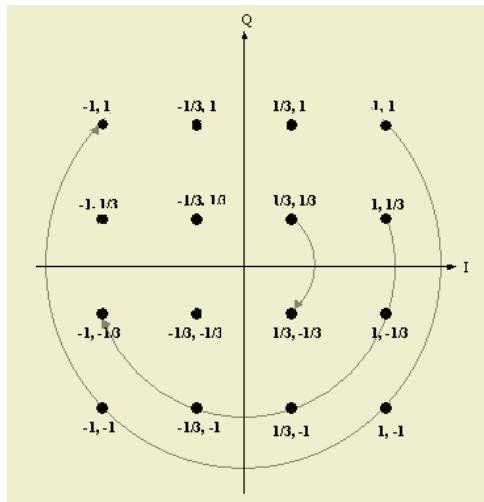


Fig. 1. 16-QAM constellation plot showing 90° , 180° and 270° mapping adjustments

The symbol demapping and rotation logic is shown in Figure 2. A multiplexer selects between phase-shifted hard-decision symbols for the I and Q channels. The multiplexer outputs are converted into 3-bit words that drive the address lines of two ROM blocks that translate hard-decisions into 2-bit output words.

An alternative approach creates a simpler and easier to understand design. The System Generator *m-code block* supports a hardware-centric subset of the Matlab language, including conditional expressions and branching statements, variable assignment, and fixed-point arithmetic. The user provides a Matlab function that the block interprets during Simulink simulation. During code generation, System Generator provides a faithful VHDL translation of the Matlab m-code. Figure 3 shows the Matlab function for symbol demapping and rotation logic.

The m-code block allows us to handle the problem addressed by the circuit shown in Figure 2 using simple, easy to understand Matlab assignments, *switch*, and *if* statements. The m-code block automatically sets its port labels to match the names of the function's input and output parameters (Figure 3).

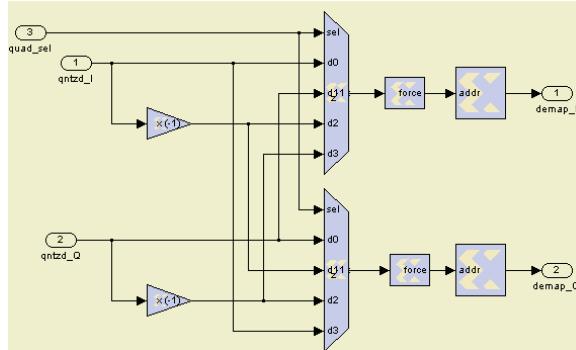


Fig. 2. System Generator block implementation of demapping circuitry

```

function [demapI, demapQ] = demap(quadSelect, qntzdI, qntzdQ)

plusHalf = xfix((x1Signed, 3, 1), .5);
minusHalf = xfix((x1Signed, 3, 1), -.5);

switch quadSelect
    case 0,   ISel = qntzdI;  QSel = qntzdQ;
    case 1,   ISel = qntzdQ;  QSel = -qntzdI;
    case 2,   ISel = -qntzdI; QSel = -qntzdQ;
    case 3,   ISel = -qntzdQ; QSel = qntzdI;
    otherwise, ISel = 0;      QSel = 0;
end

if (ISel == plusHalf),
    demapI = 0;
elseif (ISel == 1),
    demapI = 1;
elseif (ISel == minusHalf),
    demapI = 2;
else,
    demapI = 3;
end

if (QSel == plusHalf),
    demapQ = 0;
elseif (QSel == 1),
    demapQ = 1;
elseif (QSel == minusHalf),
    demapQ = 2;
else,
    demapQ = 3;
end

```

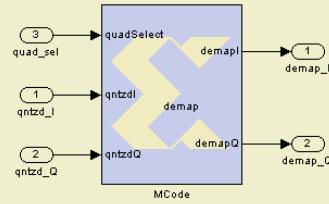


Fig. 3. Matlab m-code demapping function (left) with corresponding m-code block (right)

Note that the function inputs $qntzdI$ and $qntzdQ$ are quantized representations of the hard-decisions computed by the demodulator's slicer. By translating hard-decision symbols from the values 1, $1/3$, -1 , and $-1/3$ to 1, $1/2$, -1 and $-1/2$, respectively the circuit requires fewer bits for the comparisons.

Although the two simulation models of the demapping circuit are functionally equivalent, their hardware implementations differ. System Generator was used to translate the two circuits into hardware. The hardware was synthesized using XST and run through the Xilinx ISE 5.2i tools to produce the resource costs shown in Table 1. The table shows that the m-code solution is slightly less expensive than the traditional block approach. The difference in size is a consequence of synthesis tool trimming of unused logic from the m-code HDL. The block-based approach uses EDIF-based ROM cores whose logic cannot be further optimized.

Table 1. Virtex-II™ resource costs for the block-based and m-code solutions to the symbol demapping and rotation problem.

Design Approach	Slices	LUTs
Traditional Blocks	22	12
M-Code Block	17	12

3 Conclusions

As modern FPGA-based communication systems become increasingly complex, the control logic required to manage these systems is becoming equally sophisticated. These requirements have impacted FPGA DSP design tools by encouraging design environments in which control can be specified with as much flexibility and abstraction as the data paths. While providing this convenience, the design tool must also ensure an efficient implementation in hardware. We have demonstrated an example where a portion of the control logic in a System Generator QAM receiver design was specified using the Matlab m-code programming language.

References

1. C. H. Dick and F. Harris, "FPGA QAM Demodulator Design," *Field Programmable Logic Conference (FPL)*, Sept. 2-4 Montpellier, France, 2002.
2. Consultative Committee for Space Data Systems, "Telemetry Channel Coding : CCSDS 101.0-B-5 Blue Book," Recommendation for Space Data Systems Standards, June, 2001.
3. J. Buck, S. Ha, E.A. Lee, D.G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal on Computer Simulation*, Jan 1994.
4. C. H. Dick and F. Harris, "Configurable Logic for Digital Communications: Some Signal Processing Perspectives," *IEEE Comm. Magazine*, vol. 2, pp 107-111, Aug, 1999.
5. J. Hwang, B. Milne, N. Shirazi, and J. Stroomer, "System Level Tools for DSP in FPGAs," *FPL 2001*, Lecture Notes in Computer Science, pp 534-543, 2001.
6. The Mathworks Inc., "Simulink, Dynamic System Simulation for Matlab, Using Simulink," Natick, Massachusetts, 1999.

FPGA Processor for Real-Time Optical Flow Computation

Selene Maya-Rueda and Miguel Arias-Estrada

Computer Science Department, INAOE, Apdo. Postal 51 & 216, Mexico
selene@ccc.inaoep.mx, ariasm@inaoep.mx

Abstract. In this work an FPGA-based architecture for optical flow computation in real-time is presented. The architecture is based on an algorithm providing a dense and accurate optical flow at an affordable computational cost. The architecture is composed of an array of processors interconnected under a systolic approach. The array of processors is mainly focused in performing matrix operations to speed up the computations of optical flow. The architecture is being prototyped on an FPGA device. Results are presented and discussed.

1 Introduction

Motion is one of the most important characteristics of an image sequence since it contains the dynamics of a scene through the relationship between spatial characteristics of an image and the temporal changes in a scene [1]. In spite of the research on motion, there are two major limitations to applying motion computation to vision systems in real tasks: robustness in the real world, and the computational resources required for real-time operation [2]. Many current motion detection algorithms and systems require highly specialized hardware or up to several minutes of computing time to process medium resolution images. As a consequence, there exists an inability to obtain reliable motion estimation under real-world conditions, and applications for optical flow algorithms remain scarce [3].

However, the motion estimation problem can be transformed into regular computations to apply parallel processing in a hardware architecture. A systolic FPGA-hardware architecture is proposed for optical flow computation in real-time. The use of FPGA technology allows the design and implementation of parallel and pipelined structures, since FPGA devices are fast enough and provide high internal storage and density for digital logic design.

The structure of the paper is as follows. In section two we describe the problem of motion estimation in general terms and the algorithm used to compute optical flow. In section three the hardware architecture for optical flow computation is presented and discussed in some detail. Implementation results and evaluation of the architecture are presented in section four. Finally, in section five the conclusions and future work are presented.

2 Optical Flow Algorithm

Several computational methods have been developed for optical flow computation. A comprehensive survey can be found in [4][5]. The algorithm used in this research was recently developed by Srinivasan [6], and it was selected since it provides a good accuracy-complexity tradeoff at an affordable computational cost. In Srinivasan's algorithm, the optical flow is modeled as a weighted sum of overlapped basis functions. In this approach the motion field is force-fitted to the model and derives the smoothness properties from those of the model basis functions. Equation 1(a) expresses the *gradient constraint*, which forms the basis of gradient-based methods, and equation 1(b) shows the restrictions imposed to 1(a) in the algorithm.

$$\frac{\partial \psi}{\partial t} + u \frac{\partial \psi}{\partial x} + v \frac{\partial \psi}{\partial y} = 0 \quad (a)$$

$$u = \sum_{k=0}^{K-1} u_k \phi_k, \quad v = \sum_{k=0}^{K-1} v_k \phi_k \quad (b)$$

where u, v are the velocity vectors, ψ is the image intensity, and ϕ is a family of basis functions.

The basis functions ϕ_k are typically cosine functions. Combining the expressions in equation 1, the problem can be reformulated as a set of linear equations:

$$\begin{aligned} \int \phi_l \frac{\partial \psi}{\partial x} \frac{\partial \psi}{\partial t} + \sum_k u_k \int \phi_k \phi_l \left(\frac{\partial \psi}{\partial x} \right)^2 + \sum_k v_k \int \phi_k \phi_l \frac{\partial \psi}{\partial x} \frac{\partial \psi}{\partial y} &= 0 \\ \int \phi_l \frac{\partial \psi}{\partial y} \frac{\partial \psi}{\partial t} + \sum_k u_k \int \phi_k \phi_l \frac{\partial \psi}{\partial x} \frac{\partial \psi}{\partial y} + \sum_k v_k \int \phi_k \phi_l \left(\frac{\partial \psi}{\partial y} \right)^2 &= 0 \end{aligned} \quad (2)$$

Each pair of equations characterizes the solution around the image area covered by the basis functions. The unknowns are the weights of the basis functions, u_l, v_l . To obtain the optical flow vectors it is required to solve equation 2. The iterative method of Preconditioned Biconjugate Gradients (PBG) is employed [7] to solve linear systems of the form $Ax = b$. The matrix size to be inverted is dependent on the image size and the spacing between the basis functions so, there is an accuracy-performance tradeoff. The method involves the computation of matrix operations, such as matrix addition and matrix multiplication in the matrix inversion process.

3 Architecture

A block diagram with a general overview of the proposed architecture for optical flow computation, based on a systolic approach [8], is shown in the figure 1. The main modules of the architecture and a brief description are given in the following.

The *address generator* module provides the addressing sequence for storing pixels in the input memory and reading them when required by the data dispatch module. The external memory organization is based on a Harvard-like scheme. The *data dispatch* module distributes data to the systolic processors for parallel processing, exploiting the data-level parallelism found in image and matrix operators. The *management unit* provides the control signals to synchronize the module operation and data exchange among modules inside the architecture. The *sequence generator*

controls the iteration number in the matrix inversion process, to reduce the loop overhead. The *data collector* extracts the computed optical flow vectors from each processor and sends them to the output memory. The *derivatives computation* module computes the temporal and spatial derivatives of the images. The module is composed of a set of registers and adders/subtractors. The derivatives are required to setup the equation system for the matrix inversion.

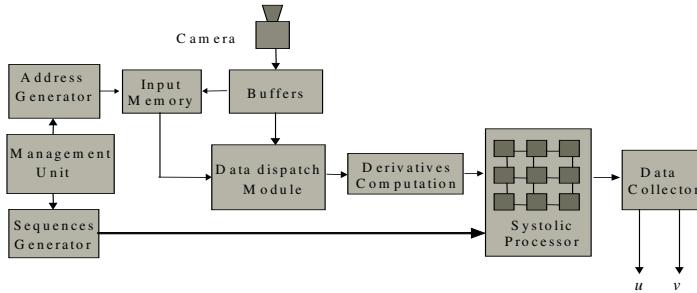


Fig. 1. A block diagram of the hardware architecture for optical flow computation

The *systolic processor* computes the matrix inversion. The systolic processors are arranged in a 2D mesh with local interconnections. Under this approach, matrix multiplication complexity is reduced to a linear order. A closer view of the module is shown in figure 2(a). A set of 8x8 processing elements is employed. The array receives data from previous computed derivatives and the basis functions. A control word from the sequence generator controls the number of iterations. Once the iterative process has converged to a solution, the array transmits the results to the data collector to send them to the output memory bank.

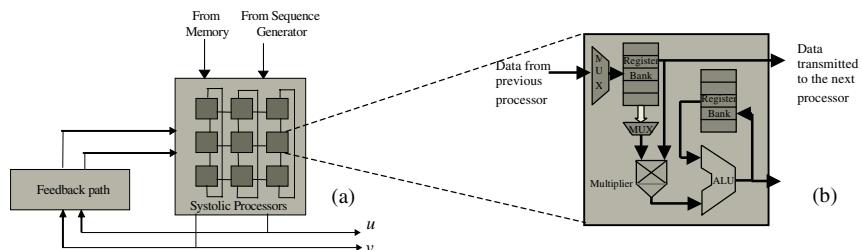


Fig. 2. (a) Systolic processing module and (b) Block diagram of the processing element

The Processing Element (PE) shown in figure 2(b) is based on a specialized Arithmetic Logic Unit (ALU). The PE contains two register banks with four 32-bit registers for temporal storage and pipeline stages. A multiplexer selects data to transmit partial results to neighbor processors. The numerical representation of data is single precision floating point. The PEs perform operations to setup the equation system for optical flow computation, combining information of spatial and temporal derivatives and the basis functions according to equation 2. Then, the PEs establish the equation system (equation 2), perform the matrix inversion and finally calculate the optical flow vectors (equation 1(b)).

4 Implementation and Results

A Handel-C model has been developed and the functional validation of the system has been done. The PE model has been synthesized for a VirtexE2000-6 device. A summary of hardware resources and timing reported for a single PE is shown in table 1. The implementation of an array of 8x8 PEs requires over 50,000 slices, which exceeds the current FPGA capacity. The rest of the architecture requires about 2,000 slices. With the reported frequency, it is possible to process about 22 images per second. The architecture throughput is close to ten giga-operations per second.

Table 1. Hardware resources utilization and timing for the PE on a VirtexE2000-6

Number of Slices	795
FPGA percentage	4%
Maximum Frequency	25.79 MHz

5 Conclusions and Further Work

The proposed architecture is able to compute near real-time the optical flow but it is expected to improve its performance. The design takes advantage of parallel structures and arithmetic logic in the FPGA. The architecture is based on a systolic array of processors interconnected by a 2D mesh and it is modular and scalable. The architecture constitutes a platform for developing motion applications in computer vision, especially for compact and mobile robotic applications. Further goals of this research will be focused on the use of the architecture in applications such as image stabilization, mosaicking, and 3D information recovering.

References

1. B. Jähne and H. Haubecker, “Computer Vision and Applications”, Academic Press, 2000
2. M. Arias-Estrada, C. Torres-Huitzil. “A Real-time FPGA Architecture for Computer Vision”, Journal of Electronic Imaging, January 2001, pp. 289-296, 2001.
3. T. Camus. “Calculating time-to-contact using real-time quantized optical flow”. Max-Planck-Institut für biologische kybernetik. Tech. Report No. 14. Feb. 1995.
4. J. L. Barron et al. “Performance of optical flow techniques”, Revised tech. Report RPLTR-9107, Queen’s University, Jul. 1993.
5. B. McCane, K. Novins, D. Crannich and B. Galvin. “On Benchmarking Optical Flow” Computer Vision and Image Understanding, Vol. 84, pp. 126-143, 2001.
6. S. Srinivasan. “Image sequence analysis: estimation of optical flow and focus of expansion, with applications”. PhD Thesis, University of Maryland, 1999.
7. W. H. Press, S. A. Teukolsky, et al. “Numerical Recipes in C”, 2nd Edition. Cambridge University Press, Cambridge, UK, 1992.
8. J. H. Moreno and T. Lang. “Matrix computation on systolic-type meshes”, High-performance VLSI Signal Processing Innovative Architectures and Algorithms, Vol. 1. Edited by H. J. Ray Liu and Hung Yao, IEEE Press, 1998.

A Data Acquisition Reconfigurable Coprocessor for Virtual Instrumentation Applications

M. Dolores Valdés¹, María J. Moure¹, Camilo Quintáns², and Enrique Mandado¹

¹ Instituto de Electrónica Aplicada “Pedro Barrié de la Maza”,

Universidad de Vigo, España

{mvaldes, mjmoure, emandado} uvigo.es

<http://www.dte.uvigo.es>

² Departamento de Tecnología Electrónica,

Universidad de Vigo, España

{quintans}@uvigo.es

Abstract. Virtual instruments intended for electronic circuits verification arose from the combination of computers supporting advanced graphical interfaces with data acquisition systems providing input/output capabilities. In order to increase the versatility and the operation rate of virtual instruments, we have designed several data acquisition/generation modules based on reconfigurable hardware. By this way, not only the software modules but also the hardware functions are dynamically changed according to the requirements of each specific instrument. The main basis of the software and hardware levels of reconfigurable virtual instruments are described in this paper. This methodology summarize our experience in the design of virtual instrumentation platforms oriented to different measurement applications. Finally, a new data acquisition/generation coprocessor based on FPGAs and optimized for the implementation of portable instruments is described.

1 Introduction

Most of the current virtual instruments are based on a general-purpose data acquisition board, so the data processing and analysis are software implemented. In this way, a wide range of instruments can be programmed using the same hardware resources and the function of the instrument is not completely defined by the manufacturer [1]. Nevertheless, these virtual instruments cannot operate at the same frequency as traditional ones whose hardware is specially oriented to their specific functionality.

Several years ago, we have proposed a technique for electronic instruments design based on a reconfigurable data acquisition hardware [2], [3]. By this way, not only the virtual instrument software modules but also the hardware functions are dynamically changed according to the requirements of each application. The reconfigurable hardware increases versatility of the virtual instruments as well as their bandwidth.

2 Hardware Level Design

2.1 FPGA Selection

An FPGA intended for virtual instrumentation applications must support the following features:

Standard configuration port. The FPGA must be reconfigured in real time from the computer.

Embedded memory blocks. Memory units in the hardware level make the data acquisition/generation frequency independent from the delays associated with the software levels and the computer interface.

Bidirectional input/output blocks.

Low-skew paths. Fast-capture latches and low skew paths available in some FPGAs contribute to reduce the different delays associated with the capture and propagation of related signals.

Integrated PLLs /DLLs (“Phase Locked Loops”/ “Digital Locked Loops”).

2.2 Computer Interface

Standard interfaces commonly used in instrumentation systems were analyzed:

Parallel Port. This protocol is very simple and can be implemented directly in any FPGA using a minimum number of logic blocks.

GPIB. This interface is commonly used in measurement instruments. Its implementation and low speed requirements are similar to the parallel port and both are used for the implementation of low speed measurement instruments.

USB serial bus. This is a more recent standard which allows hot connection, plug&play, and high transfer rates using version 2.0. This is the best approach to portable instruments with medium performance. Nevertheless, the implementation of the USB bus in the FPGA has a great cost in logic resources.

PCI bus. The PCI interface has been chosen as the best way to achieve a high-speed link between the acquisition/generation hardware and the computer. Nevertheless its implementation have also a great cost in logic blocks and requires a high speed programmable device.

2.3 Analog Inputs/Outputs

An advantage of using FPGAs to implement the logic control of an analog data input/output system is the availability of a great number of registered I/O pins. In contrast with microcontrollers or DSP based data acquisition systems, characterized by a reduced number of I/O ports, FPGAs allows the simultaneous sample of many analog inputs channels avoiding the use of additional devices like sample&hold circuits, analog or digital multiplexors and the logic control associated with them.

2.4 Hardware/Software Codesign

The following decisions must be taken previously to any implementation:

Hardware/Software functions. At least, the reconfigurable hardware must provide data acquisition/generation services to the software. By the same way, the software application must implement the human interface or control panel of each instrument. The analysis of the basic architecture and functions of the measurement instruments permits to define which parameters must be programmable and which configurable. For example, the reconfiguration must be used when a new instrument is created by the application software or when the main parameters of the current instruments must be modified. By the contrary, the trigger detection logic and timing functions must be programmable because they are prone to be changed during the operation of any instrument.

Hardware/Software synchronization. The tasks previously assigned to the hardware and software levels must carry out in parallel at the maximum possible frequency. Specially, the delays generated in the software levels must not be propagated to the acquisition/generation hardware.

3 Software Level Design

A main process is created when the instrumentation application program starts. This process is called Virtual Instrumentation Manager (VI Manager) and waits for the events generated by the user. These events define the active instruments of the system and their parameters.

When an instrument becomes active, the VI Manager generates a new process (instrument 1). Each instrument process is split into two threads, one manages the user interface and the other controls the data acquisition/generation process. By this way, the delays associated with the user interface (for example due to the graphical representation of data) never stop or delay the acquisition/generation process. When a new instrument becomes active, a new process is created (instrument #). This process/thread based approach guarantees the minimum latency in the active acquisition/generation tasks.

Services provided by the VI Manager are used by the threads in order to communicate with the reconfigurable hardware. The VI Manager encapsulate the particularities of the hardware so the physical system can be reconfigured without software changes at user level. When an application thread from one instrument requests a service, it blocks itself until a callback from hardware is generated. By this way, an optimal resources sharing among active instruments is achieved.

A message from a thread controlling the acquisition/generation process to the thread managing the user interface is generated when new data must be displayed or when some event must be notified to the user.

When a data acquisition/generation is finished, the associated thread or process is killed.

4 A Reconfigurable Data Acquisition/Generation Coprocessor

This section is focused on a reconfigurable data acquisition/generation coprocessor especially oriented to the implementation of portable instruments for electronic measurement. We have chosen the USB 2.0 as the standard interface with the computer and we use the Cypress EZ-USB FX2 controller that includes memory blocks (4KB) and allows four end-points simultaneously [4]. This memory capacity avoids the need of specific memory blocks in the data acquisition hardware. The data transfers can be selected to 8/16 bits and the transfer rate can achieve 48 MHZ using master, slave, synchronous or asynchronous modes. Finally, this chip includes a 8051 core that can be used for some specific tasks; for example we use this microcontroller for the implementation of the JTAG interface intended to the FPGA reconfiguration. Figure 1 shows one of the implementations (patent pending) using an Altera FPGA EP20K100EQC240-2X. Also we have developed additional cards can be combined with this one using the expansion ports in order to implement for example analog/digital and digital/analog conversion.



Fig. 1. Hardware implementation of a reconfigurable virtual coprocessor base on a USB serial interface

References

1. Virtual Instruments in <http://zone.ni.com/devzone/>
2. Moure, M.J., Valdés, M.D., Mandado, E.: Virtual Instruments based on Reconfigurable Logic. In: Hartenstein, R.W., Keevallik, A. (eds.): Field-Programmable Logic and Applications. Lecture Notes in Computer Science, Vol. 1482. Springer-Verlag, Berlin Heidelberg New York (1998) 505–509.
3. Moure, M.J., Valdés, M.D., Mandado, E.: Virtual Instruments based on Reconfigurable Logic: Design Methodology and Applications. In: Proceedings of the International Workshop on Virtual and Intelligent Instrumentation. IEEE, Annapolis (2000) 44–51
4. In <http://www.cypress.com/>

Evaluation and Run-Time Optimization of On-chip Communication Structures in Reconfigurable Architectures

T. Murgan, M. Petrov, A. García Ortiz, R. Ludewig, P. Zipf,
T. Hollstein, M. Glesner¹, B. Oelkrug, and J. Brakensiek²

¹ Institute of Microelectronic Systems,
Darmstadt University of Technology, Germany
murgan@mes.tu-darmstadt.de

² Nokia Research Center, Bochum, Germany

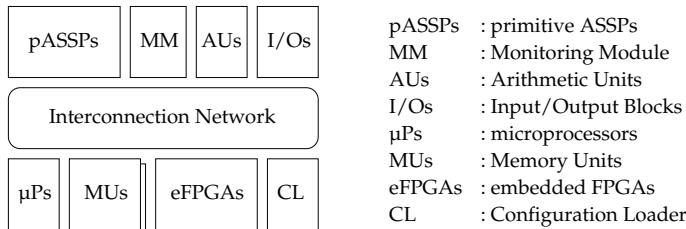
Abstract. With technology improvements, the main bottleneck in terms of performance, power consumption, and design reuse in single chip systems is proving to be generated by the on-chip communication architecture. Benefiting from the non-uniformity of the workload in various signal processing applications, several dynamic power management policies can be envisaged. Nevertheless, the integration of on-line power, performance and information-flow management strategies based on traffic monitoring in (dynamically) reconfigurable templates has yet to be explicitly tackled. The main objective of this work¹ is to define the concept of *run-time functional optimization* of application specific standard products, and show the importance of integrating such techniques in reconfigurable platforms and especially their communication architectures.

1 Introduction

Forthcoming wireless communication standards and ubiquitous computing technologies, together with strict time-to-market, low costs, high performance, and design efficiency requirements pose tremendous pressure on the design process. Driven by those demands reconfigurable architectures are getting an increasing interest in recent times [5]. Thus, a remarkable variety of different platforms have been proposed to trade energy-efficiency, cost and performance [4]. However, as technology improves and the integrable die size enlarges, one of the most important limiting factors for system performance, die area, and power consumption will be generated by the on-chip interconnect networks [2, 7].

This paper addresses the necessity of integrating run-time functional optimization policies into reconfigurable architectures. Additionally, several possible dynamical optimization strategies for on-chip communication architectures are discussed and underlined.

¹ The presented work was carried out within the German funded BMBF project IP2, i.e. *Intellectual Property Prinzipien für konfigurierbare Basisband SoCs in Internet-Protokoll basierten Mobilfunknetzen*, No. 01M3059B.

**Fig. 1.** ASSP Architecture Template

2 Dynamic Optimization of Reconfigurable Architectures

A reconfigurable architecture template or platform as in figure 1 is build around a reconfigurable interconnection network and may consist of processing units, memory blocks, specialized arithmetic units, reconfigurable and dedicated hardware, I/O blocks, monitoring modules, and configuration loaders. The so-called *Application Specific Standard Products (ASSPs)* are obtained through different instantiations of such modules and their application class is influenced by the design of what we call *primitive ASSPs (pASSPs)*.

The majority of battery powered electronic devices, like portable computers and mobile phones, exhibit a non-uniform workload. Peak or very high performance levels are generally needed only during particular operation states. In an attempt to extend battery life and reduce the cost and noise of cooling systems, *Dynamic Power Management (DPM)* techniques reduce the power consumption of electronic systems by dynamically adjusting performance levels to the workload through shut-down or at least slow-down of active components [1].

The decisive drawbacks of reconfigurable architectures in comparison to ASICs are the required die area, performance, and power consumption. Power consumption and dissipation become critical issues and the extension of the inter-charging operations is of increasing interest. Consequently, we believe that dynamic power management strategies emerge as an inextricable part of dynamically reconfigurable architectures, especially with the advent of VDSM (very deep sub-micron) technologies.

We call *run-time functional optimization policies* the set of those dynamic power and performance management strategies that can be efficiently applied in primitive ASSPs and especially in interconnection structures of reconfigurable architectures during operation time in order to dynamically reduce the power consumption under performance constraints, or vice-versa.

3 On-chip Communication Architectures Requirements

In order to interconnect the processing and memory blocks in a complex System-on-Chip, various interconnection networks can be employed. In both fine- and coarse-grained reconfigurable templates, the programmable interconnect architecture has a vital influence in the total area, performance and power consumption of the system. To achieve high computational performance and flexibility, the different modules have to be richly interconnected.

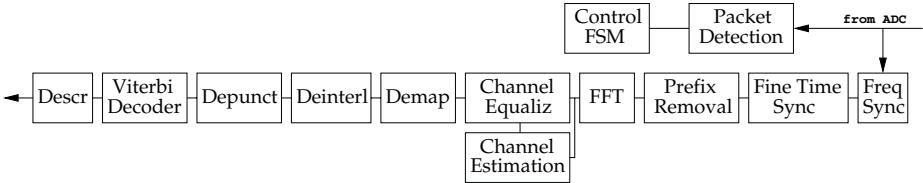


Fig. 2. Basic block scheme of a baseband OFDM receiver

Several interconnection strategies for coarse-grained reconfigurable architectures have been reported. Taking into account the locality of the data in several applications, local architectures are generally composed by point-to-point like structures devoted to provide high performance communication of closed units. They used to be restricted to the first level of neighbors (e.g. *DReAM*, *KressArray*) or they can include a second level as in *MATRIX*. Other employed structures can be divided into three categories: multi-bus (e.g. *RaPID*), crossbar (e.g. *PADDI-2*), and mesh; either regular (e.g. *DReAM*, *KressArray*, *Matrix*, *Garp*) or irregular (*Pleiades*).

We investigate dynamically reconfigurable architectures tailored for communication systems envisaging mainly OFDM-based wireless applications. The aforementioned project focuses on run-time functional optimization for OFDM-based wireless systems. Since the IEEE 802.11a wireless LAN standard allows for multiple data rates - from 6MB/s to 54MB/s - the data flow between modules will be variable, depending on the selected data rate. Several studied modules can be found in figure 2 representing a basic OFDM receiver.

However, the effective sample rate $f_s = 20\text{ MHz}$ at the output of the sender is specified by the standard as a function of the symbol duration and the number of samples, thus not depending on the selected data rate. Consequently, the sample rates after and before the cyclic prefix extension are 20 Ms/s and 16 Ms/s , respectively. The IFFT also increases the data rate, because only 48 of the 64 samples are data samples, the others being either pilots or zero signals. Knowing that the data rate at the output of the modulator is constant, i.e. 12 Ms/s , the possible data rates at the output of the sender as functions of the type of modulation and puncturing can be computed.

For the receiver, the data rate requirements can be determined in a similar manner. The FFT block operates at 16 Ms/s , exactly like the IFFT on the transmitter side. Assuming that 8 bits are used for sample quantization and knowing that each sample is a complex value, the effective bit rate will then be 256 Mbps . After the FFT, the data rate slightly decreases with the ratio 52/64, resulting in an effective data rate of 13 Ms/s . Thus, for 8-bit samples, the channel correction blocks operate at a data rate of 208 Mbps . The same data rate can also be found at the input of the phase tracker, which uses the 4 pilot signals to perform further adjustments of the constellation. Thereafter, the pilot signals are removed, leading thus to a decrease of the data rate to 192 Mbps . Furthermore, the traffic that has to be supported in a reconfigurable template increases due to the activity of the monitoring module and the configuration loader.

4 System Level Communication Evaluation

Preliminary results of a SystemC framework allowing for high level performance evaluation of communication architectures have already been reported in [6]. In order to ameliorate the overall latency of communication architectures with a high number of routers, a technique called *wormhole routing* is used [3].

In order to develop an application independent evaluation platform, we use a parameterizable hierarchical Markov model for both producer and consumer processes. Basically, the producer and the consumer may switch in three basic states: an idle one, and two active ones. While in the idle one no data can be send or received, during the other two active states data can be produced or consumed with a different probability. At their turn, both active states are hierarchical, and the probabilities can be slightly modified. Hence, both abrupt and slight data traffic modifications can be modeled stochastically.

5 Optimization Opportunities and Future Work

Several optimization strategies can be envisaged. In a hardware module that monitors the transition activity of signals in a digital system for estimating the power consumption has been developed. Such a task may be included together with a traffic controller in a DPM observer and used at run-time for monitoring and functional optimization. For example, the queuing strategy can be dynamically adapted. The buffers can be either clock gated or completely shut down whenever registering sparse communications. Thus, a dynamic trade-off between performance and power is realized. Additionally, the routing algorithm can also be dynamically changed or adjusted under the same circumstances.

References

1. L. Benini, A. Bogliolo, and G. De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Trans. on VLSI Systems*, 8(3):299–316, June 2000.
2. L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *IEEE Computer*, pages 70–78, January 2002.
3. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks. An Engineering Approach*. Morgan Kaufmann Publishers, 2003.
4. R. Hartenstein. A Decade of Reconfigurable Computing: A Visionary Retrospective. In *Proc. of DATE*, pages 642–649, 2001.
5. K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni Vincentelli. System-Level Design: Orthogonalization of Concerns and Platform-Based Design. *IEEE Trans. on CAD of Int. Circuits and Systems*, 19(12):1523–1543, December 2000.
6. T. Murgan, A. García Ortiz, M. Petrov, and M. Glesner. A Stochastic Framework for Communication Architecture Evaluation in Networks-on-Chip. In *IEEE Intl. Symposium on Signals, Circuits and Systems*, July 2003.
7. C. Seitz. Let's Route Packets Instead of Wires. In *Advanced Research in VLSI: Proc. of the 6th MIT Conf.*, pages 133–138, 1990.

A Controlled Data-Path Allocation Model for Dynamic Run-Time Reconfiguration of FPGA Devices

Dylan Carline and Paul Coulton

Department of Communication Systems, Lancaster University, Lancaster,

Lancashire, U.K., LA1 4YR

d.carline@lancaster.ac.uk, p.coulton@lancaster.ac.uk

Abstract. Although methods for dynamic run-time FPGA reconfiguration have been proposed, few address the problems associated with increasing data-path delays due to a full or partial reconfigurations. In this paper, a method is proposed that enables specific timing requirements to be maintained within a reconfigurable architecture, by using logic-module partitioning and *known-delay* interconnection modules. This system allows data-paths of varying widths to be routed effectively between device modules along paths that are fixed in both length and position. Further, the technique may be regarded as extending the Xilinx Modular Design tools methodology to support run-time scenarios.

1 Introduction

Reconfigurable computing is currently enjoying immense popularity in both communications and computing fields, through systems capable of near-software flexibility with a near-hardware performance. This coupled with the *dynamic* handling of device functionality, makes reconfigurable systems extremely attractive. Although new device architectures are being developed for this role [1,2], the Field Programmable Gate Array (FPGA) remains the most applicable target currently available. Using the FPGAs for this research also allows the concepts described to be applied *generically* to other emergent architectures. The research is based on the natural partitioning of internal FPGA logic and the sending of data though paths of set position and length in order to maintain synchronicity in a *run-time* reconfigurable environment. Serious problems can be encountered when elements are reconfigured in terms of their specific timing requirements and using static routing elements can maintain the module's interconnectivity. In this paper, Section 2 describes the benefits gained from device partitioning, whilst in Section 3 the module interconnections are described. Section 4 details the Xilinx Partial Reconfiguration model and an implementation of this idea is given in section 5, before our conclusions are finally drawn.

2 Device Partitioning

As FPGA devices have increased in functional size, the algorithms used in their design and operational control have become increasingly complex. In an environment

where dynamic reconfiguration may be responsible for the movement of individual logical elements, it is imperative that such movement does not prevent signals maintaining synchronicity across the device. Traditionally, device partitioning has been used in the placement of logic in FPGA devices [3,4]. Rather than using the partitioning to define the initial placement however, it is useful to first consider the *logical content* of the design. As System on Chip (SoC) implementations begin to proliferate and their target devices become commonplace, a natural separation occurs between specific logical groups or ‘objects’ within a design. Furthermore, as the functional size of devices increase, individual modules may take the form of complete systems (e.g. modulators or coders). The use of modular techniques is comparable to the way in which software-programming benefits from the use of object oriented techniques. These separate objects form the basis of the natural partitioning utilised in our study.

Not only are the specific architectures of FPGAs evolving in terms of reconfiguration support, new software tools are also emerging. Among these are the now well-known JBits Application Program Interface (API) [5] developed by Xilinx, which offers advantages to reconfigurable computing by providing direct manipulation of the Xilinx Virtex FPGA bitstream [6] thus reducing the normally considerable re-design times and enabling a finer granularity of device control [7].

3 Module Interconnections

In a practical environment, where a reconfigurable unit is adapting both its functionality *and* connectivity at high-speed, serious problems can occur when logical units are placed far from its connecting logic, ultimately preventing clock synchronicity. One method of overcoming this problem is to maintain a *fixed* route with *fixed* port locations between the modules with fixed elements remaining largely static over time. This coupled with a pre-defined boundary for logic-element placement within the modules, reduces complexity for any placement algorithms used thereby helping decrease the required time between possible reconfiguration events.

To create this system, register-modules are defined that allow data to pass between the functional elements of the design with registers having *fixed* port positions *within* each partition of the design. The delay time associated with these interconnections can be used to ensure that synchronicity between logical objects is maintained. The initial register modules were designed and implemented using the Xilinx Modular Design tools [8].

4 Partial Reconfiguration

The Partial Reconfiguration (PR) model by Xilinx [9] provides support for altering the functionality of parts of a design whilst the remainder maintains operation. Two modes are provided: ‘*Multi-Column PR, Independent Designs*’ and ‘*Multi-Column PR, Communication Between Designs*’, both allowing individual *column-based* reconfiguration through the device’s SelectMAP [10] interface. Currently, the macro will only support *one* bit communication between modules, per Tri-State Buffer

(TBUF) longline, of which four can be supported per CLB row. Although this system provides a means of inter-module communication, the routing resources themselves are largely static. In terms of a reconfiguration event, this is a serious limitation as it is likely that the routing will need to be altered when the structure of the modules change or another module is added to the design. The system previously proposed avoids this problem by designing the routing resources as individual modules *themselves*, giving them freedom to be placed as per particular requirement. Furthermore, if CLB resources are remaining, routing resources can be added without overburdening the more scarce resource of tri-state buffers.

5 Implementation

The system proposed takes in a serial input to the module ‘framer’ which separates the input stream into 8-bit words. These 8-bit words are subsequently passed to the ‘detector’ module, which searches for a particular 8-bit synchronization sequence. Upon receiving this word, the module sends a copy of the synchronization word to both the ‘inverter’ and ‘concatenate’ modules. The ‘inverter’ module sends an inverted copy of the synchronisation word to the ‘concatenate’ module, which finally concatenates the two inputs before outputting the 16-bit word. The placed and routed version of the design can be seen in figure 4.

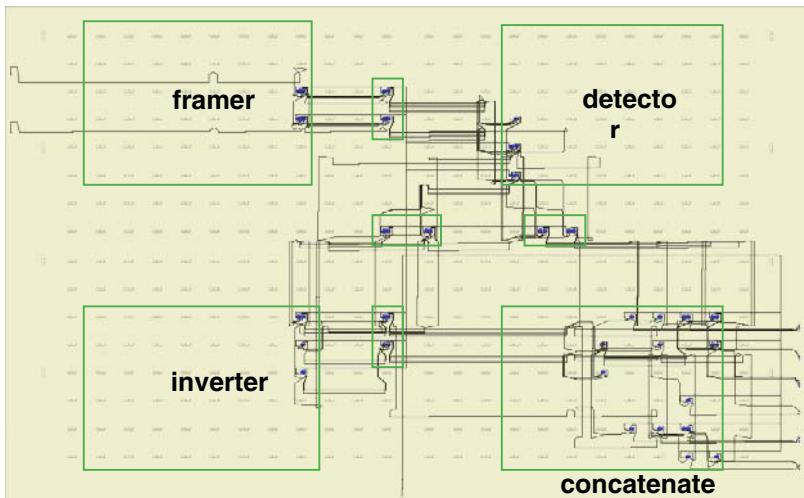


Fig. 1. The placed-and-routed layout of the four-module implementation

6 Conclusion

As the operating environments in modern communications and computing become increasingly demanding in terms of both protocol diversity and algorithm complexity, reconfigurable computing using devices capable of *run-time* reconfiguration is becoming increasingly desirable.

As previously stated, a new breed of device family is in development, which will offer enhanced reconfigurability with low power and memory requirements. Currently, these devices seem far from commercial availability and in the interim period other technologies must be considered. FPGA's cannot be overlooked in this regard, as the recent evolution of these devices has given further possibilities for their long-term integration. This is demonstrated by their increased support for partial reconfiguration and modular design techniques and ever-increasing functional sizes and operating bandwidths. One hardware aspect that must be improved upon however is the FPGA's large power requirements, which negate it from use in many portable implementations. If run-time FPGA reconfiguration is to be achievable, provision must be made for increased support of placement and routing processes. A method has been proposed that provides a structure enabling major logical groups to be considered independently thus reducing the time taken by optimisation processes and directly aiding the placement of logic. Further, by providing a mechanism of data transfer along paths of known delay the management of routing within a design is simplified. We believe the techniques suggested in this paper contribute to the evolution of the field of run-time reconfigurable computing and its eventual implementation.

Acknowledgement

The authors would like to thank H W Communications for their continued support of this work.

References

1. Tredennick, N.: The Death of the DSP. Digital Infrastructures: Megaflops and Microwonders. Dublin, Ireland. 2001
2. PACT Informationstechnologie GmbH.: The XPP White Paper. Release 2.1. 2002
3. Kernighan, B. W., Lin, S.: An Efficient Heuristic Procedure for Partitioning Graphs. Bell Systems Tech. J. 49, 2. pp. 291-308
4. Fiduccia, C. M., Mattheyses, R. M.: A Linear Time Heuristic for Improving Network Partitions. In Proceedings of the 9th Design Automation Conference. pp. 175-181. 1982
5. Xilinx Inc.: JBits Documentation. JBits 2.8. June 2002
6. Xilinx Inc.: Virtex Series Configuration Architecture User Guide. September 2000
7. Carline, D., Coulton, P.: Reconfigurable Computing Using FPGAs: is JBits the Answer. Proc. 9th International Workshop on Systems, Signals and Image Processing. UK. November 2002
8. Xilinx Inc.: Development System Reference Guide – ISE 5. February 2003
9. Xilinx Inc.: Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations. May 2002
10. Xilinx Inc.: Using a Microprocessor to Configure FPGAs Slave Serial or SelectMAP Modes. November 2002

Architecture Template and Design Flow to Support Application Parallelism on Reconfigurable Platforms

Sergei Sawitzki¹ and Rainer G. Spallek²

¹ Philips Research Laboratories

Prof. Holstlaan 4

5656AA Eindhoven, The Netherlands

Sergei.Sawitzki@philips.com

² Institute of Computer Engineering

Department of Computer Science

Dresden University of Technology

01062 Dresden, Germany

rgs@ite.inf.tu-dresden.de

Abstract. This paper introduces the ReSArT (Reconfigurable Scalable ATemplate). Based on a suitable design space model, ReSArT is parametrizable, scalable, and able to support all levels of parallelism. To derive architecture instances from the template, a design environment called DEFInE (Design Environment for ReSArT Instance Generation) is used, which integrates some existing academic and industrial tools with ReSArT-specific components, developed as a part of this work. Different architecture instances were tested with a set of 10 benchmark applications as a proof of concept, achieving a maximum degree of parallelism of 30 and an average degree of parallelism of nearly 20 16-bit operations per cycle.

1 Introduction

Parallel processing and reconfigurable computing are two very powerful techniques of computation, which can be advantageously combined, as has been proven by dozens of successful projects [1, 2]. Some of the proposed architectures are introduced in the form of templates, which can be parameterized to suit the needs of the particular application. The question, however, remains, if it is possible to combine a design space model, a universal architecture template, and a tooling environment to support both instruction and data parallelism at different grain sizes.

This paper introduces the ReSArT (Reconfigurable Scalable ATemplate). Based on a suitable design space model, ReSArT is parameterizable, scalable and able to support all levels of parallelism. To derive architecture instances from the template, a design environment called DEFInE (Design Environment for ReSArT Instance Generation) is used, which integrates some existing academic and industrial tools with ReSArT-specific components, developed as a part of this work.

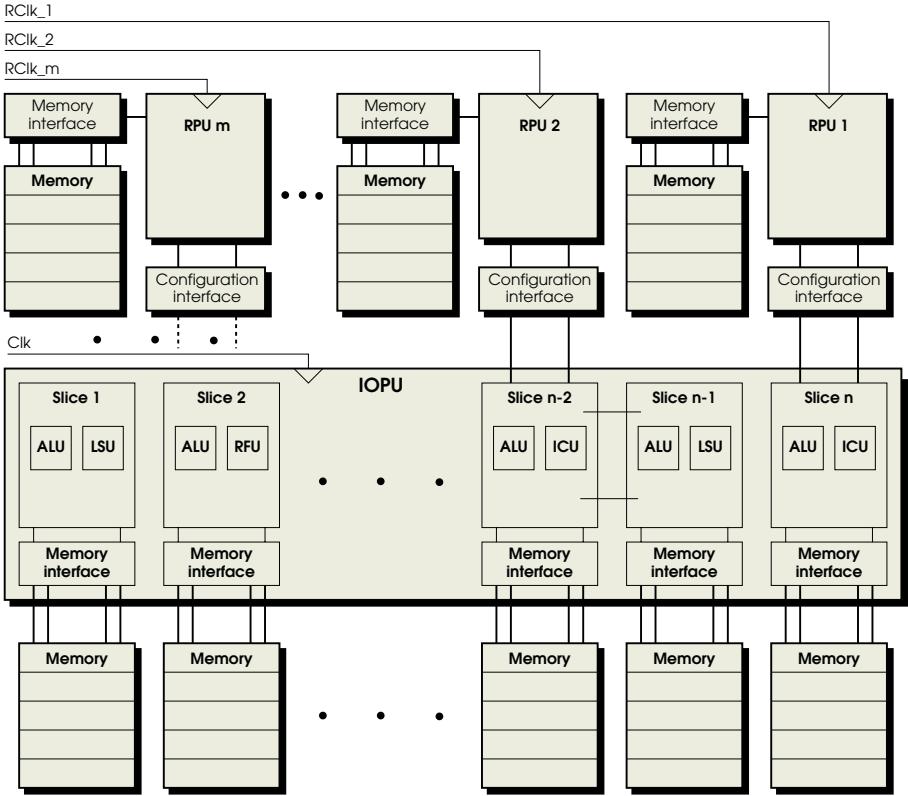


Fig. 1. ReSArT Top Level View

2 ReSArT and DEflnE

The detailed description of the design space can be found in [3]. According to this model, ReSArT leaves most dimensions open at the template level. In general, ReSArT can be split up in three views: a template, describing a family of reconfigurable architectures, a set of microarchitectures of reconfigurable resources (RPU), and a configuration context (as well as program code) to derive instances from these general descriptions and program them to complete particular tasks.

ReSArT consists of an instruction oriented processing unit (IOPU) and N_{RPU} reconfigurable processing units ($N_{RPU} \geq 0$), see Fig. 1. The IOPU executes RISC-like instructions. It is constructed out of slices, each one being a simple double-issue processor core. Every slice includes one instruction fetch unit (IFU), one or two register files, one decode unit (DU) and two execution units. One execution unit is always an ALU, whereby the second one could be also an ALU or load-store unit (LSU), reconfigurable functional unit (RFU) or interface control unit (ICU). RFU is a small reconfigurable unit with much less logic capacity than RPU. Still, it can be used to implement simple single-cycle instructions which

are not covered by the other hardwired units. The ICU is used to implement the communication between IOPU and RPU via memory-mapped I/O.

The RPU is used to implement computationally intensive data-flow-oriented parts with less synchronization. Each RPU has its own memory interface to further reduce the communication overhead. For each application, the RPU can be generated in one of three different flavors: pipelined, static with shared I/O ports, and dynamically reconfigurable. The hardware structure of the RPU fits into the model for SRAM-based, clustered, island-style FPGAs [4]. For every application, a different LUT and cluster size can be used. The optimal values are determined by the **DEflnE** (Design Environment for ReSArT Instance Generation).

The input language of **DEflnE** is a special class of directed acyclic graphs, so called \mathcal{F} -DAG. Such graphs can describe both fine and coarse-granular kernels and can be easily produced based on other formalisms. Each node of the \mathcal{F} -DAG represents an operation whereby edges represent the data dependencies.

In the first step, the **flow_opt** module reads the \mathcal{F} -DAG description in and generates an initial schedule according to a “as soon as possible” (ASAP) principle. In the second step, this initial schedule is optimized according to one of the four different strategies (minimize area function of the stages, minimize delay difference of the nodes in the same stage, minimize input or output data stream of the stages). After the optimization, the critical path of the schedule is computed and (based on this value) the stage numbers are mapped to the pipeline stages (or subcircuits, in case of the dynamically reconfigurable PU). Finally, the statistics of the **flow_opt** run are created and stored.

The best \mathcal{F} -DAG schedule is converted to the BLIF format [5] and transferred to the **rec_map** module, which consists of the SIS, TVpack, VPR [4] and **par_extract** steps. SIS and TVpack are responsible for technology mapping for a set of 36 different RPU microarchitectures (featuring different LUT and cluster sizes). VPR produces the place and route statistics, which are used by the **par_extract** tool to find the best microarchitecture for the given application. The **par_check** tool compares the circuit parameters with the requirements and (in case they are conform) transfers the processing to the **flow2vhdl** converter, which produces the appropriate VHDL code for the RPU. This code is forwarded to the **arch_gen** module, which adds the IOPU description and produces one design library for the selected ReSArT instance. This library is synthesizable and can be further processed by traditional back-end tools.

3 Experiments and Results

For the proof of concept some experiments were carried out with ReSArT and **DEflnE** using the Xilinx Virtex II architecture as a prototyping platform. The scalability of the IOPU was proven with respect to bitwidth and the number of slices (as well as some other parameters). The area scales linearly with all parameters. A prototype with 8 IOPU slices on the XC2V10000 device is able to execute $6 \cdot 10^8$ 16 bit operations per second, consuming only about 35% of the logic resources of the chip (thus, leaving a lot of space for RPU implementation).

For the RPU tests, a set of 10 benchmark applications was translated into \mathcal{F} -DAG format. This set includes a simple FFT module (8 points transform as a building block for bigger transforms), a soft-in soft-out decision subblock of a turbo decoder, a linear predictive codec for speech compression, as well as a couple of other applications. The DEflnE performance was tested for all optimization strategies. The improvement of 15.38–100% in the schedule quality in 6 of 10 cases results in the clock frequency improvement of 3.2–17.6% after mapping to the Virtex II chip (the difference is due to the fact that Virtex II was not selected by DEflnE as an optimal architecture for all benchmarks).

Some results can be compared with Xilinx benchmarks documented in [6]. The 1024pt 16 bit FFT requires about 1 μ s on Xilinx Virtex II. Using ReSArT and DEflnE, the same transform runs in about 1.2 μ s, given the programmability of the architecture and possibility to run additional tasks on the same device. A TMS340C64 family DSP needs slightly less than 8 μ s for the same benchmark.

4 Conclusions

This paper introduced the ReSArT architecture template developed to support instruction and data parallelism at various levels and the DEflnE design environment to create ReSArT architecture instances. Tests with a set of 10 benchmark applications prove that ReSArT is scalable with respect to the bitwidth and the number of slices. Different RPUs running the benchmarks achieved the maximum degree of parallelism of 30 with an average of nearly 20 operations.

ReSArT and DEflnE build a powerful platform for the exploration of reconfigurable parallel processing systems. Considering tighter time-to-market margins in the modern SoC world resulting in the requirements for shorter design cycles, template-based architectures with embedded reconfigurable resources and integrated software environment will become increasingly important in the future.

References

1. Hartenstein, R.W., Hirschbiel, A., Weber, M.: A Novel Paradigm of Parallel Computation and its Use to Implement Simple High Performance Hardware. Int. Conf. memorizing the 30th Ann. of Computer Society Japan, Tokyo (1990)
2. Gokhale, M.B., Holmes, W. et.al.: Building and Using a Highly Parallel Programmable Logic Array. IEEE Computer **24**(1) (1991) 81–89
3. Sawitzki, S., Spallek, R.G.: A Concept for an Evaluation Framework for Reconfigurable Systems. In: Lysaght, P., Irvine, J., Hartenstein, R.W. (eds.): Field-Programmable Logic and Applications: 9th Intl. Workshop. LNCS, Vol. 1673. Springer-Verlag, Berlin Heidelberg New York (1999) 475–480
4. Betz, V., Rose, J., Marquardt, A.: Architecture and CAD for Deep-Submicron FPGAs. Kluwer Academic Publishers, Boston (1999)
5. Sentovich, E., Singh, K.J. et.al.: SIS: A System for Sequential Circuit Synthesis. Electronics Research Laboratory, Department of Electrical Engineering and Computer Science, University of California, Memorandum UCB/ERL M92/41 (1992)
6. R.T. Olay III: Xilinx XtremeDSP Initiative Meets the Demand for Extreme Performance and Flexibility, Xcell 40 (2001) 30–33

Efficient Implementation of the Singular Value Decomposition on a Reconfigurable System

Christophe Bobda, Klaus Danne, and André Linarth

Heinz Nixdorf Institute / Paderborn University
Fuerstenallee 11,D-33102 Paderborn, Germany
`{bobda,danne,linarth}@upb.de`

Abstract. We present a new implementation of the singular value decomposition (SVD) on a reconfigurable system made upon a Pentium processor and a FPGA-board plugged on a PCI slot of the PC. A maximum performance of the SVD is obtained by an efficient distribution of the data and the computation across the FPGA resource. Using the reconfiguration capability of the FPGA help us implement many operators on the same device.

1 Introduction

The Singular Value Decomposition of a matrix has many important scientific and engineering applications [1, 4, 5]. The SVD implementation on parallel processors is usually done using the Brent and Luk[4] parallelization of the Hestenes method explained in section 2. Using hardware to compute the SVD of a matrix have been done by Cavallaro et al [10, 11]. A CORDIC processor have been used here to compute the SVD of a 2x2 matrix. No indication have been given on the application of the method to solve larger matrices. In this paper we present an efficient implementation of the SVD on a reconfigurable system made upon a general purpose processor and a FPGA board.

Section 2 introduces the SVD problem and explain how a solution due to Hestenes is computed. It also presents the Brent and Luk parallel implementation of the Hestenes method. Section 3 explains the SVD implementation on our taget architecture while the performance of the system is given in section 4. Finally, we give an overview of our work in section 5.

2 The Singular Value Decomposition

The Singular Value Decomposition of an $m \times n$ real matrix A is its factorisation into a product $A = U \times \Sigma \times V^T$, where U is an $(m \times n)$ matrix with orthogonal columns, $\Sigma = diag(\sigma_1, \dots, \sigma_n)$ and V is an $(n \times n)$ orthogonal matrix. The values σ_i of the diagonal matrix Σ are the singular values of the matrix A , the columns of the matrices U and V are the left and right singular vectors of A . The SVD can be computed using the Hestenes method based on the classical one side Jacobi iteration for digitalization of real symmetric matrices. The idea is to generate an

orthogonal matrix V such that the transformed matrix $AV = W$ has orthogonal columns. Having the matrix W , the Euclidean length of each non-zero column $W_{(:,i)}$ will be normalised to unity. The singular values and vectors are then computed as follows: $\sigma_i = \|W_{(:,i)}\|$, $U_{(:,i)} = \frac{W_{(:,i)}}{\sigma_i}$ and $W = U\Sigma$. The SVD of the matrix A is then given by: $AV = W \rightarrow AV = U\Sigma \rightarrow A = U\Sigma V^T$. Plane rotations represented by a matrix Q are incrementally applied to the matrix A to compute the matrix W . At the k -th step with the rotation matrix $Q^{(k)}$, we have: $A^{(k+1)} = Q^{(k)}A^{(k)}, 0 \leq k \leq k_r$. With a sweep defined to be a series of $\frac{n(n-1)}{2}$ pairwise column-orthogonalizations of the matrix $A^{(k)}$, the convergence of the matrix $A^{(k)}$ to W is guaranteed for $k_r = S \frac{n(n-1)}{2}$, where S is the number of sweeps, $A^0 = A$ and $W = A^{k_r}$. The multiplication of $A^{(k)}$ by $Q^{(k)}$ affects only the column-pair $(A_{(:,i)}^{(k)}, A_{(:,j)}^{(k)})$. The computation of $A^{(k+1)} = Q^{(k)}A^{(k)}$ is reduced to:

$$\begin{pmatrix} A_{(:,i)}^{(k+1)} \\ A_{(:,j)}^{(k+1)} \end{pmatrix} = \begin{pmatrix} \cos \Theta_{ij}^{(k)} & \sin \Theta_{ij}^{(k)} \\ -\sin \Theta_{ij}^{(k)} & \cos \Theta_{ij}^{(k)} \end{pmatrix} \begin{pmatrix} A_{(:,i)}^{(k)} \\ A_{(:,j)}^{(k)} \end{pmatrix} \quad (1)$$

The rotation angle $\Theta_{ij}^{(k)}$ is chosen such a way that the new column pairs are orthogonal. Using the formulas of Rustishauser[12], we set $\Theta_{ij}^{(k)} = 0$ if $\gamma_{ij}^{(k)} = 0$; Otherwise we compute $\xi^{(k)} = \frac{\alpha_j^{(k)} - \alpha_i^{(k)}}{2\gamma_{ij}^{(k)}}$ and $\tan \Theta_{ij}^{(k)} = \frac{\text{sign}(\xi^{(k)})}{|\xi^{(k)}| + \sqrt{1 + \xi^{(k)2}}}$ with $\alpha_i^{(k)} = A_{(:,i)}^{(k)} \bullet A_{(:,i)}^{(k)}$, $\alpha_j^{(k)} = A_{(:,j)}^{(k)} \bullet A_{(:,j)}^{(k)}$ and $\gamma_{ij}^{(k)} = A_{(:,i)}^{(k)} \bullet A_{(:,j)}^{(k)}$. This rotation angle always satisfies: $|\Theta_{ij}^{(k)}| \leq \frac{\pi}{4}$.

The main operations of this method are the generation of the rotation angle and the updating of the columns elements which rely on multiply accumulate (MAC) operations. Their dataflow nature as well as the large sizes of the matrices considered makes this computation a nice candidate for hardware implementation.

3 Implementation on a Reconfigurable System

Since the orthogonalization of column-pairs $(A_{(:,i)}^{(k)}, A_{(:,j)}^{(k)})$ are independent, they can be done in parallel. Brent and Luk [4] suggested the use of a set of $n/2$ processors connected together in a ring topology to orthogonalize a matrix of dimension n . Each processor has an exclusive access to a memory segment which stores the pairs of columns to be orthogonalized by that processor. After the orthogonalization of their column-pairs, the processors exchange data with their left and right neighbours. This process is repeated until all column pairs are orthogonalized, thus completing a sweep. The platform used is made upon a personal computer equipped with a RC-1000 FPGA board of the company Celoxica. The board contains up to four memory banks independently connected around the FPGA. Therefore we can implement up to four processing element (PE) in the FPGA. Because the size of the FPGA is limited, it will not be possible to implement 8 PEs (4 for the dot product and 4 for the column-pairs update). We

use the reconfiguration to implement the 8 PEs. The first reconfiguration loads the FPGA with the 4 PEs PE_11, PE_12, PE_13 and PE_14 (figure 1 a) which are used for the dot-products computation. The second configuration loads the 4 PEs PE_21, PE_22, PE_23 and PE_24 (figure 1 b)) which are used for the updating of the columns. The computation of the rotation angle is too complex for a hardware implementation. Since this computation is not often executed, it is left to the processor which collects the dot-product values from the FPGA and returns the rotation angle.

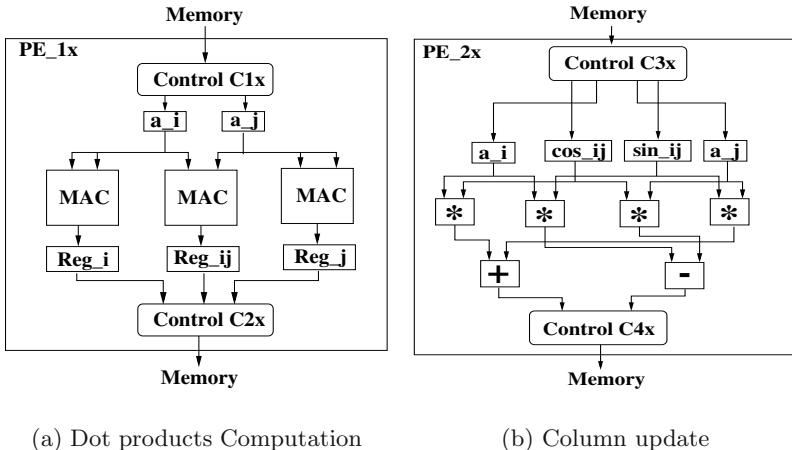


Fig. 1. Structure of the processing elements

4 Performance

The floating-point implementation of the PEs can be run with a clock frequency of maximal 20MHz. Therefore the total time needed by the 4 PEs PE_11, PE_12, PE_13 to compute all the dot products required in to sweep for 4 block matrices is given by: $(\text{number_of_rows} \times \text{number_of_columns} \times (\text{number_of_columns} - 1) \times (\text{cycle_period})) / 2$. For 4 block matrices with 100 columns and 50.000 rows each, using a 20MHz clock, the time need to complete the computation of the dot products required by a sweep is: $(50.000 \times 100 \times (10 - 1) \times (1/20.000.000)) / 2 = 12.375s$. The rotation process is much faster, since it does not need to sweep all the columns 2 by 2. So the computation time can be done as follow: $\text{number_of_columns} \times \text{number_of_rows} \times \text{cycle_period}$. The time needed to update a matrix of 50.000 x 400 with the same clock is 250ms. With the reconfiguration overhead of one second, our implementation will compute the complete SVD of a 400 by 50.000 matrices in 13.625s which is much higher than the time need on a Pentium processor with up to 450 MHZ.

5 Conclusion

In this paper we have dealt with an efficient implementation of the singular value decomposition of big matrices on a reconfigurable system made upon a PC and a FPGA-Board. With the structure of the RC 100-PP, it is possible to implement 4 PEs in parallel for each computation step. Because the complete function to compute the SVD could not fit on the FPGA, we made use of the hardware reconfiguration to implement the complete function needed for the SVD. The main bottleneck of the system is the floating point computation which occupy large space and slow down the clock. We are working on a solution based on the fixed-point operations to increase the design clock while decreasing the design area.

References

1. M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. Using linear algebra for information retrieval. *J. Soc. Indust. Appl. Math.*, 37(4):573–595, 1995.
2. M. Berry, T. Do, G. O'Brien, V. Krishna, and S. Varadhan. *SVDPACK(Version 1.0) User's Guide*, 1996.
3. C. Bobda and Nils Steenbock. Singular value decomposition on distributed reconfigurable systems. In *12th IEEE International Workshop On Rapid System Prototyping(RSP'01), Monterey California*. IEEE Computer Society, 2001.
4. Richard P. Brent and Franklin T. Luk. The solution of singular-value and eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, 1985.
5. S. Deerwester, S. Dumai, G. Furnas, T. Landauer, and R. Harshmann. Indexing by latent semantic analysis. *Journal of American Society for Information Science*, 41(6):391–407, 1990.
6. G. E. Forsythe and P. Henrici. The cyclic jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.
7. Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. North Oxford Academic Publishing, 1983.
8. Eldon R. Hansen. On cyclic jacobi methods. *J. Soc. Indust. Appl. Math.*, 11(2):448–459, 1963.
9. M. R. Hestenes. Inversion of matrices by biorthogonalization and related results. *J. Soc. Indust. Appl. Math.*, 6(1):51–90, 1958.
10. F. Luk J. Cavallaro. CORDIC arithmetic for an svd processor. *Journal of Parallel and Distributed Computing*, 5(3):271–290, 1998.
11. J. Cavallaro N. Hemkumar. Efficient complex matrix transformations with CORDIC. In *IEEE Symposium on Computer Arithmetic*, pages 122–129. IEEE, 1993.
12. H. Rustishauser. The jacobi method for real symmetric matrices. *Handbook for Automatic Computation*, Vol 2 (linear Algebra):202–211, 1971.
13. G. Salton. *The SMART Retrieval System*. Prentice Hall, Inc, 1971.
14. J. H. Wilkinson. *The algebraic eigenvalue problem*. Oxford University Press, 1965.

A New Reconfigurable-Oriented Method for Canonical Basis Multiplication over a Class of Finite Fields $\text{GF}(2^m)$

José Luis Imaña¹ and Juan Manuel Sánchez²

¹ Dpto. Arquitectura de Computadores y Automática, Universidad Complutense,
28040 Madrid, Spain

jluimana@dacya.ucm.es

² Dpto. Informática, Escuela Politécnica, Universidad de Extremadura,
10071 Cáceres, Spain
sanperez@unex.es

Abstract. A new method for multiplication in the canonical basis over $\text{GF}(2^m)$ generated by an all-one-polynomial (AOP) is introduced. The theoretical complexities of the bit-parallel canonical multiplier constructed using our approach are equal to the smallest ones found in the literature for similar methods, but the multiplier implementation over reconfigurable hardware using our method reduces the area requirements.

1 Introduction

Galois or finite fields have several applications in communication systems, such as satellite links, computer networks, or compact disks [1]. They use arithmetic operations in the Galois field for cryptography, error correction or algebraic codes. Finite fields with q elements are represented as $\text{GF}(q)$, and the fields with fundamental interest for technical applications are the *extension fields* of $\text{GF}(2)$, denoted as $\text{GF}(2^m)$. The representation of the field elements has crucial role in the efficiency of the architectures for arithmetic operations. Considering a basis representation of the elements, the *addition* is relatively inexpensive, whereas the *multiplication* is the most important operation. There are different basis representations for elements of $\text{GF}(2^m)$, and the most popular are the *canonical* [2], *normal* and *dual* bases. The complexity of the multiplier depends on the basis and the defining irreducible polynomial selected for the field.

The field $\text{GF}(2^m)$ can be considered as a vector space of dimension m over $\text{GF}(2)$, so it can be represented using any basis of m linearly independent elements. Therefore, elements of $\text{GF}(2^m)$ are represented by m -bit vectors. *Addition* is realized by a bit-wise XOR operation, whereas the *multiplication* is determined by the basis. *Canonical* basis Ω is the set $\Omega = \{1, \omega, \dots, \omega^{m-1}\}$, where ω is a root in $\text{GF}(2^m)$ of an irreducible polynomial of degree m over $\text{GF}(2)$. Using this basis, the elements of $\text{GF}(2^m)$ are polynomials of degree at most $m - 1$ over $\text{GF}(2)$, and arithmetic is carried out modulo the irreducible polynomial. We denote as $\underline{\alpha}_\Omega = (a_{\Omega_0}, \dots, a_{\Omega_{m-1}})^t$ the coordinate vector of α with respect to Ω .

In this contribution, a new method for multiplication in the canonical basis for the field $\text{GF}(2^m)$ generated by an irreducible *all-one-polynomial* (AOP) is introduced. Our approach, named the *transpositional* method, uses the permutations given by the value m , and the aim of the method is to *group* the coordinates of the operands in order to the reconfigurable synthesis tool could find a good mapping on the reconfigurable device selected for the implementation of the multiplier. The theoretical space and time complexities of the bit-parallel multipliers constructed using our approach are equal to the smallest ones found in the literature for similar approaches based on generating AOPs [3][4][2][5][6], but the practical implementation over reconfigurable hardware (FPGAs and CPLDs) using our method reduces the area requirements of the multipliers.

2 The Transpositional Method

The *transpositional* method [7] for multiplication in the canonical basis over $\text{GF}(2^m)$ is based on the computation of 1-cycles and 2-cycles which determine *product* and *sum-of-product* terms, respectively, of the operands coordinates.

Let $\alpha, \delta, \chi \in \text{GF}(2^m)$ and $\underline{\alpha}_\Omega, \underline{\delta}_\Omega, \underline{\chi}_\Omega$ be their coordinate vectors, respectively, with respect to Ω . The multiplication $\delta = \alpha \cdot \chi$ in Ω involves the presence of *inner products* [7], such as $\underline{\alpha}_\Omega^t \cdot \underline{\chi}_\Omega^r = a_{\Omega_0} c_{\Omega_{m-1}} + a_{\Omega_1} c_{\Omega_{m-2}} + \dots + a_{\Omega_{m-1}} c_{\Omega_0}$, where the c_{Ω_i} s are the coordinates of χ with respect to Ω and where $\underline{\chi}_\Omega^r = (c_{\Omega_{m-1}}, \dots, c_{\Omega_0})$. These *sum-of-product* expressions defined over $\text{GF}(2)$ can be represented using the notation given in group theory for the *permutations*, in which the upper row contains the subscripts of the coordinates of α which are multiplied by the coordinates of χ with subscripts given in the lower row [7]. It can be proved that the *permutations* so defined involve the presence of *1-cycles* (k) and *2-cycles* (i,j), where (k) represents a *product* term $x_k = (a_k c_k)$ and where (i,j) represents a *sum of products* $x_{ij} = (a_i c_j + a_j c_i)$. The 2-cycles (i,j) are called in group theory as *transpositions*. We can define functions \mathbf{EC}_i^m , \mathbf{OC}_i^m and \mathbf{MC}^m which give the cycles for the permutation corresponding to the values i and m . Defining the functions \mathbf{E}_i , \mathbf{O}_i and \mathbf{M} as the addition of the terms x_{ij} 's and x_k 's represented by the cycles (i,j)'s and (k)'s given by \mathbf{EC}_i^m , \mathbf{OC}_i^m and \mathbf{MC}^m , respectively, the following expressions [7] for the coordinates of the product δ can be given

$$d_{\Omega_i} = \mathbf{E}_0 + \begin{cases} \mathbf{O}_{i+1} & i \text{ even} \\ \mathbf{E}_{i+1} & i \text{ odd, } i \neq m-1 \\ \mathbf{M} & i = m-1 \end{cases} \quad (1)$$

where d_{Ω_i} s, with $i = 0, 1, \dots, m-1$, are the coordinates of δ with respect to Ω . We have named the multiplication method given by equation 1 as *transpositional* because it is based on the computation of 1-cycles and 2-cycles (*transpositions*).

In Table 1 a comparison for the theoretical complexities obtained with our approach and with other methods is given, for canonical multipliers with generating AOPs. It can be observed that the *theoretical* complexities obtained using our method are equal to the lowest ones obtained using other similar approaches.

Table 1. Theoretical complexities of bit-parallel canonical basis multipliers

	#XOR	#AND	Delay
Itoh-Tsu.[2]	$m^2 + 2m$	$m^2 + 2m + 1$	$T_{AND} + \lceil \log_2 m + \log_2(m+2) \rceil T_{XOR}$
Hasan [4]	$m^2 + m - 2$	m^2	$T_{AND} + (m + \lceil \log_2(m-1) \rceil) T_{XOR}$
Koç-Sun.[5]	$m^2 - 1$	m^2	$T_{AND} + (2 + \lceil \log_2(m-1) \rceil) T_{XOR}$
Halbut.[3]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil \log_2(m-1) \rceil) T_{XOR}$
Zhang[6]	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil \log_2(m-1) \rceil) T_{XOR}$
Transposit.	$m^2 - 1$	m^2	$T_{AND} + (1 + \lceil \log_2(m-1) \rceil) T_{XOR}$

3 Implementations over FPGAs and CPLDs

The theoretical complexity given in Section 2 is not an exact predictor of the area consumption if reconfigurable hardware is used [8]. We have used *Xilinx Foundation F2.1i* for the implementation of canonical multipliers over FPGAs and CPLDs using our *transpositional* method and using the method given in [3].

For FPGAs, 4013XLPQ160 devices from XC4000XL family have been used. In Table 2, the experimental results obtained for the multipliers implemented using the approach given in [3] and using our method are showed. The total CLB count using transpositional method is 6.6% lower than using the other approach, whereas the total maximum combinational path delay of our method is 3.6% slower, but this is because we have performed optimization for area. The reduction of the CLB count is due to the devices used are LUT-based, and gates with smaller number of inputs can be easily included in a LUT, which increases the possibility of obtaining a better mapping solution [9]. *Transpositional* method groups the coordinates of the operands by means of the 2-cycles and the 1-cycles, therefore helping to the mapping tool to reduce the CLB count.

Table 2. Experimental results for FPGA implementations

	Halbutogu.& Koç		Transpositional	
	CLBs	Max.Path(ns)	CLBs	Max.Path(ns)
$GF(2^4)$	6	14.7	5	14.8
$GF(2^{10})$	40	25.0	37	25.8
$GF(2^{12})$	54	27.3	53	28.6
$GF(2^{18})$	116	33.0	106	36.8
$GF(2^{28})$	281	45.3	259	43.0
$GF(2^{36})$	454	51.7	428	55.1
Total	951	197.0	888	204.1

For CPLDs, XC95288XV devices from XC9500XV family have been used. In Table 3, the experimental results obtained for the multipliers implemented using the method given in [3] and using the *transpositional* method are showed. The total MC count using our method is 12.8% lower than using the other approach, and for the total delay, the transpositional method is 29.7% faster.

Table 3. Experimental results for CPLD implementations

	Halbutogu.& Koç		Transpositional	
	MCs	T_{PD} (ns)	MCs	T_{PD} (ns)
$GF(2^4)$	10	9.8	9	9.1
$GF(2^{10})$	55	17.3	57	15.9
$GF(2^{12})$	89	24.1	79	20.7
$GF(2^{18})$	213	38.4	175	17.3
Total	367	89.6	320	63.0

Experimental results given in Tables 2 and 3 seem demonstrate, therefore, that the *grouping* technique provided by the *transpositional* approach is a good method when *any* reconfigurable platform is used for the implementation.

4 Conclusions

A new *transpositional* method for canonical basis multiplication over finite fields $GF(2^m)$ generated by irreducible AOPs has been presented. The theoretical complexities of the bit-parallel multipliers constructed using our method are equal to the lowest ones found in the literature, but the FPGA and CPLD implementations of multipliers using our transpositional approach lead to a lower count of CLBs and MCs, respectively, than using other similar approaches.

References

1. Menezes, A.J. (ed.): Applications of Finite Fields. Kluwer Academic (1993)
2. Itoh, T., Tsujii, S.: Structure of Parallel Multipliers for a Class of Finite Fields $GF(2^m)$. Information and Computation, Vol.83 (1989) 21–40
3. Halbutogullari, A., Koç, Ç.K.: Mastrovito Multiplier for General Irreducible Polynomials. IEEE Trans. Computers, Vol.49, No.5 (2000) 503–518
4. Hasan, M.A., Wang, M.Z., Bhargava, V.K.: Modular Construction of Low Complexity Parallel Multipliers for a Class of Finite Fields $GF(2^m)$. IEEE Trans. Computers, Vol.41, No.8 (1992) 962–971
5. Koç, Ç.K., Sunar, B.: Low-Complexity Bit-Parallel Canonical and Normal Basis Multipliers for a class of Finite Fields. IEEE Trans. Computers, Vol.47, No.3 (1998) 353–356
6. Zhang, T., Parhi, K.K.: Systematic Design of Original and Modified Mastrovito Multipliers for General Irreducible polynomials. IEEE Trans. Computers, Vol.50, No.7 (2001) 734–749
7. Imaña, J.L., Sánchez, J.M., Fernández, M.: Método de multiplicación canónica sobre campos $GF(2^m)$ generados por AOPs orientado a hardware reconfigurable. II Jornadas sobre Computación Reconfigurable y Aplicaciones, JCRA (2002) 215–220
8. Imaña, J.L.: Bit-Parallel Arithmetic Implementations over Finite Fields $GF(2^m)$ with Reconfigurable Hardware. Acta Applicandae Mathematicae, Vol.73, No.3. Kluwer Academic Publishers (2002) 337–356
9. Kao, C.C., Lai, Y.T.: A Routability Driven Technology Mapping Algorithm for LUT Based FPGA Designs. IEICE Trans. Fundamentals, Vol.E84-A, No.11 (2001) 2690–2696

A Study on the Design of Floating-Point Functions in FPGAs

Fernando E. Ortiz¹, John R. Humphrey¹, James P. Durbano², and Dennis W. Prather¹

¹ University of Delaware, 140 Evans Hall, Newark, DE 19716

{ortiz, humphrey, dprather}@ee.udel.edu

² EM Photonics, Inc. 102 East Main St., Newark, DE 19711

durbano@emphotonics.com

Abstract. Floating-Point Operations represent a common task in a variety of applications, but such operations often result in a bottleneck, due to the large number of machine cycles required to compute them. Even though the FPGA community has developed advanced algorithms to improve the speed of FLOPs, floating-point transcendental functions are still underdeveloped. In this paper, we discuss some of the tradeoffs faced when implementing floating-point functions in FPGAs. These techniques, including lookup tables, and CORDIC algorithms, have been used in the past for the implementation of fixed-point analytic functions. This paper seeks to apply those methods to floating-point functions. The implementation results from different versions of a floating-point sine function are summarized in terms of speed, area, and accuracy to understand the effect of different architectural alternatives.

1 Introduction

Nearly every modern microprocessor-based system is capable of processing non-integer values using floating-point methods to represent real numbers. In addition, floating-point operations (FLOPs) represent a common task in modern applications, ranging from 3D graphics, to simulations. To increase performance, floating-point operations have been highly researched and optimized for use in FPGAs. In comparison, more advanced operations, such as trigonometric functions, have not received as much attention. Although trigonometric functions have been realized in FPGAs, these implementations typically have been based on fixed-point, rather than floating-point, formats. As we will show, the advantages of the techniques used in fixed-point do not apply directly to floating-point designs, and therefore novel schemes are required. In this paper, we examine the various tradeoffs associated with developing floating-point trigonometric functions in FPGAs, using one of two general schemes: adapt the well known fixed-point methods to accept and produce floating-point numbers, or use techniques that use floating-point arithmetic to approximate arbitrary functions.

2 Design Alternatives

In this section, we discuss several alternatives involved in implementing trigonometric functions in reconfigurable hardware. Even though there exists a considerable amount of published work in this area [1-5], it is all based on fixed-point formats. First, we will show that the advantages of the techniques used in fixed-point arithmetic do not apply directly to floating-point designs (and then propose modifications to these methods). Then, techniques for implementing other well known functional approximation techniques in hardware will be presented, using only floating-point arithmetic throughout the computations.

2.1 Adapting the Fixed-Point Approximation Methods to Floating-Point Arithmetic

This section will study the feasibility of modifying the currently available (fixed-point) function approximation techniques that will enable them to accept and produce floating-point numbers. Every fixed-point implementation of a function can be trivially turned into floating-point one by the use of conversion modules between fixed-point and floating point-numbers at the input and output ports, as shown in figure 2. However, this alternative should be avoided because of the high extra latency that results from the addition of these modules, typically on the order of five clock cycles (each) for fully pipelined, high-speed versions [6].

LUT. The LUT method can be extended for use in floating-point systems with two steps. First, the addition of an FP2Int function that maps the floating-point input into an integer that can be used for the table lookup. The second modification applies to the values stored in the table; these should be stored in floating-point notation, eliminating the need for an output conversion module, but increasing the memory usage, due to redundant exponents.

Bipartite LUT. This method is unsuitable for floating-point conversion because it is based on a second order Taylor interpolation around a point derived from the decomposition of the fixed-point input argument. This technique cannot be applied for floating-point numbers, since the exponent bits cannot be interpreted in the same way as mantissa bits and the sign bit.

CORDIC. The CORDIC algorithm can be executed using floating-point numbers, but doing so would result in large inefficiencies. The simplicity of these operations remains no longer true for floating-point systems, particularly in the case of floating-point addition, where multiple variable-length shifters are required. The only comparative advantage of using CORDIC methods to implement floating-point functions in hardware is that the additional latency introduced by the floating-point conversions is small compared with the intrinsic latency of high-precision CORDIC units.

2.2 Floating-Point Approximation Methods

As was shown, the use of the fixed-point solutions for floating-point problems results in high latencies and large resource usage. In this section, we propose the use of interpolation techniques with small LUTs embedded in the hardware and computations being carried out in floating-point arithmetic.

Linear Interpolation. This method uses a small set of known values of the function and extrapolates to the complete interval of interest using a straight line approximation

In order to implement linear interpolation in hardware, three steps are required. First, the input is converted to an integer (n) that will serve as an index for the lookup. Then, the actual table lookup takes place and the values of $f(x)$, $f'(x)$ and $n\Delta x$ are output. Finally, the interpolation takes place using two floating-point adders and two floating-point multipliers (see Figure 1). Note that the $f'(x_0)$ and $n\Delta x$ can be calculated instead of stored in the table, and doing so reduces the memory usage significantly, but also adds drastically to the latency.

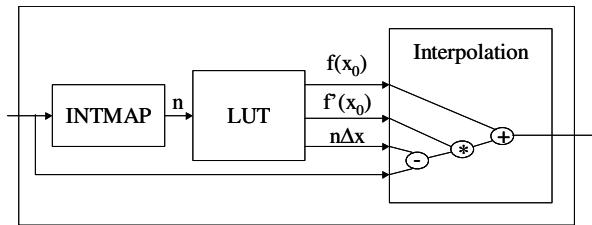


Fig. 1. Hardware Implementation of a Floating-Point Linear Interpolator

3 Example: Floating Point Sine

The previous sections presented several alternatives available for implementing floating-point functions in hardware. In this section, we will present the design of a floating-point sine function, used in a computationally intensive electromagnetic simulation algorithm, where the high speed and low latency are of primary importance. Low error is considered in the cost function, with a smaller weight factor.

Periodic functions require additional argument reduction, to force the input to be in a smaller interval where the function is defined. This interval is $[0, 2\pi]$ in the case of the sine function. The alternatives to consider were a full LUT, the CORDIC algorithm, and linear interpolation. Because of the symmetry of the function, only one quarter of a wavelength needs to be stored in the LUT. The remainder of the values can be determined with the appropriate sign and phase changes. This technique reduces the size of the table by 75% without any loss in precision. To better understand the impact of these factors in hardware, several of these choices were implemented in a Xilinx Virtex-II 6000-4 FPGA. The results are provided in Table 1.

Table 1. Implementation Results of Floating-Point Sine. This table compares the results of several implementation alternatives of a sine function. Sine Output = 32-bit, IEEE 754 floating-point number

Comp. Method	Area(Slices+BRAM)	Maximum Error(%)	Latency	Speed (MHz)
CORDIC-8bit	428	2.45	20	137.6
CORDIC-16 bit	889	1.00E-02	28	121
LUT-8bits-no Interp	222	2.45	7	226.2
LUT-16bits-no Interp	183+32	1.00E-02	7	177.8
LUT-8 bits-Interp	1431	7.53E-05	18	104.5

4 Conclusion

Several design alternatives exist when implementing floating-point functions in FPGA-based systems. In this paper, we have discussed these approaches and presented the relative tradeoffs involved with each. We showed that the traditional solutions, such as CORDIC and bipartite tables, do not offer the best solution when floating-point arithmetic is intended. We proposed two solutions to this problem: (1) Use floating-point conversion modules at the inputs and outputs of these algorithms and (2) use simple floating-point interpolation algorithms. We then presented implementation results from several versions of a sine function to quantify our analysis. Although CORDIC implementations have been historically preferred because they provide very high accuracy with reasonable hardware requirements, current FPGAs include embedded memory blocks that afford accurate LUT implementations with significantly higher speeds and reduced latencies. CORDIC algorithms should be used when the memory blocks are unavailable, if the required precision makes the size of the LUT impractical, or reduced-area requirements outweigh the long latency. Ultimately, the specific requirements of the end application dictate the architecture that provides the best complement in terms of area, speed, and precision.

References

- [1] J. Volder, "The CORDIC Trigonometric Computing Engine," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330-334, 1959.
- [2] J. Duprat and J. M. Muller, "The CORDIC algorithm: new results for fast VLSI implementation," *IEEE Transactions on Computers*, vol. 42, pp. 168-178, 1993.
- [3] D. S. Phatak, "Double Step Branching CORDIC : A New Algorithm for Fast Sine and Cosine Generation," *IEEE Transactions on Computers*, vol. 47, pp. 587-602, 1998.
- [4] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," presented at ACM International Symposium on Field-Programmable Gate Arrays, Monterey, California, 1998.
- [5] T. Vladimirova and H. Tiggeler, "FPGA Implementation of Sine and Cosine Generators Using the CORDIC Algorithm," presented at Military and Aerospace Applications of Programmable Devices and Technologies, Laurel, Maryland, 1999.
- [6] P. Belanovic, "Library of Parameterized Hardware Modules for Floating-Point Arithmetic with An Example Application," in *Electrical and Computer Engineering*. Boston, Massachusetts: Northeastern University, 2002, pp. 83.

Design and Implementation of RNS-Based Adaptive Filters

Javier Ramírez¹, Uwe Meyer-Bäse², Antonio García¹, and Antonio Lloris¹

¹ Department of Electronics and Computer Technology

University of Granada

javierrp@ugr.es {agarcia, lloris}@ditec.ugr.es

² Department of Electrical and Computer Engineering

FAMU-FSU College of Engineering

umb@eng.fsu.edu

Abstract. This paper presents the residue number system (RNS) implementation of reduced complexity and high performance adaptive FIR filters on Altera APEX20K field-programmable logic (FPL) devices. Index arithmetic over Galois fields along with a selection of a small wordwidth modulus set are keys for attaining low-complexity and high-throughput. The replacement of a classical modulo adder tree by a binary adder with extended precision followed by a single modulo reduction stage improved area requirements by 10% for a 32-tap FIR filter. A block LMS (BLMS) implementation was preferred for the update of the adaptive FIR filter coefficients. RNS-FPL merged filters demonstrated its superiority when compared to 2C (two's complement) filters, being about 65% faster and requiring fewer logic elements for most study cases.

1 Introduction

Residue number system (RNS) based [1, 2] digital filter designs can be effective for realizing high speed sum-of-products kernels. One of the main properties of the RNS is the inherent modularity which induces efficient implementations and excellent levels of performance. These characteristics hold for a variety of technologies, from cell-based integrated circuits (CBIC) [3] to field-programmable logic (FPL) [4].

FPGAs have intrinsically weak arithmetic capabilities when compared to ASICs. In addition, FPL deficiencies increase geometrically with precision, as a result of architectural limitations. FPL device families, such as Altera FLEX10K or Xilinx Virtex, are organized in channels (typically 8-bits wide) with local short delay propagation paths; also dedicated memory blocks to synthesize small RAM and ROM functions are included. Performance rapidly suffers when carry bits and/or data have to propagate across a channel boundary. We call this the *channel barrier* problem [4]. New trends in FPL device design tend to add dedicated hardware for multiplication to the new FPL device families, as has happened with Altera APEX II and Virtex II.

An alternative design paradigm is advocated in this paper with the development of efficient structures for RNS-based adaptive FIR filters. The RNS advantage is gained by reducing arithmetic to a set of concurrent operations that reside within small wordlength non-communicating channels. This attribute makes the RNS potentially attractive for implementing these systems with FPL technology.

2 RNS Background

In the RNS, numbers are represented in terms of a relatively prime basis set (moduli set) $P=\{m_1, \dots, m_L\}$. Any number $X \in \mathbf{Z}_M = \{0, \dots, M-1\}$, where $M = \prod m_i$, has a unique RNS representation $X \leftrightarrow \{X_1, \dots, X_L\}$, where $X_i = X \bmod(m_i)$. RNS arithmetic is defined modulo M by pair-wise modular operations:

$$\begin{aligned} Z = X \pm Y &\leftrightarrow \left\{ \langle X_1 \pm Y_1 \rangle_1, \dots, \langle X_L \pm Y_L \rangle_L \right\} \\ Z = X \times Y &\leftrightarrow \left\{ \langle X_1 \times Y_1 \rangle_1, \dots, \langle X_L \times Y_L \rangle_L \right\} \end{aligned} \quad (1)$$

where $\langle Q \rangle_j$ denotes $Q \bmod(m_j)$. It is the ability of the RNS to do arithmetic within independent small wordlength channels that makes it particularly attractive for FPL insertion, as has been referred in the literature [5-7].

On the other hand, index arithmetic [2] constitutes an efficient means for enhancing and reducing the complexity of RNS-based DSP applications. All the non-zero elements in a Galois field can be generated exponentiating a primitive element, denoted g_r . Thus, multiplication in $\text{GF}(m_r)$ can be implemented as:

$$|q_1 q_2|_{m_r} = g_l^{|i_1 + i_2|_{m_r - 1}} \quad l = 1, 2, \dots, L \quad (2)$$

where $q_1, q_2 \in \{1, \dots, m_r\}$ and $i_1, i_2 \in \{0, \dots, m_r - 1\}$ are the indexes of q_1 and q_2 , respectively. This multiplication scheme just requires LUTs for index computation, a modulo $m_r - 1$ adder for index addition and a LUT for the inverse index transformation.

3 Design of RNS-Based Adaptive FIR Filter

An adaptive filter processes a digital input $x(n)$, obtaining an output sequence $y(n)$ through adjustable parameters whose values affect how $y(n)$ is computed. The output is compared to a second signal $d(n)$, called the *desired response* signal, thus getting the *error signal* $e(n) = d(n) - y(n)$. This is used to adapt the parameters of the filter, so the output matches the desired response signal, i.e., the magnitude of $e(n)$ must decrease with time. The LMS algorithm is simple to implement [8, 9] and powerful enough to evaluate the practical benefits of adaptation. It only requires the error signal, the input signal vector and a step size μ for adjusting the coefficients:

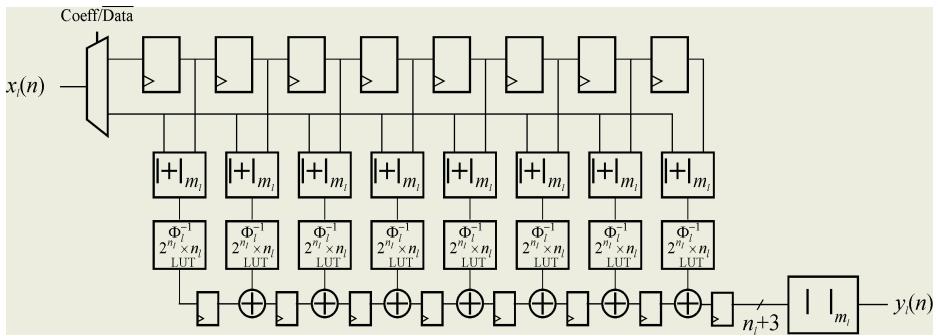


Fig. 1. Index-based Adaptive FIR filter design.

$$W(n+1) = W(n) + \mu e(n)x(n) \quad (3)$$

LMS computation requires multiplications and additions to be implemented, nearly in the same number as that of the FIR filter structure with fixed coefficient values, which is one of the reasons for its popularity. Also block (BLMS) implementations are possible, thus reducing further its computational costs.

An RNS-based BLMS adaptive FIR filter has been developed. The internal modular processing engine is intended to work externally in a 2C format in order to communicate with the LMS block that performs the FIR coefficient update. Efficient block decomposition 2C-to-index [3, 4] and ϵ -CRT-based [10] or CRT-based RNS-to-2C conversions are used within the modular processing engine, while Fig. 1 shows the structure of one of the \$L\$ internal index-based channels for an 8-tap FIR filter example. The channel accepts index inputs, so coefficient multiplication is implemented with modulo \$m_i\$ adders, while the filter product summation is efficiently implemented with an enhanced modulo \$m_i\$ adder chain consisting of conventional adders (with precision extension) and a modulo \$m_i\$ reduction stage.

Design examples were implemented using Altera APEX20K for both 2C arithmetic and RNS. Table 1 compares 2C FIR filters ranging from 8 to 32 taps with the filters proposed in this paper. Input signal and coefficients are 16-bit wide and, for the 2C design, the \$16 \times 16\$-bit multipliers are designed with five pipeline stages. The table shows the number of taps (\$N\$), the system dynamic range (\$W\$), the number of LEs, the maximum frequency and the modulus set used for the RNS study cases.

4 Conclusions

This paper has shown the benefits provided by the RNS for the implementation of adaptive FIR filters using FPL devices, with a BLMS structure for the update of the adaptive FIR filter coefficients that reduces the computational load while retaining good convergence properties. The proposed RNS filters are about 65% faster than 2C designs and require fewer logic elements in most cases. Concretely, complexity is reduced up to 13% when ϵ -CRT converters are used.

Table 1. Resource reduction and speed-up achieved by an adaptive FIR filter built in RNS-FPL technology when compared to the equivalent 2C system.

N	W	2C Adaptive FIR Filter Implementation		RNS-based Adaptive FIR filter implementation CRT/e-CRT				
		LEs	F (MHz)	Les	Resource reduction	F (MHz)	Speed-up	Modulus set
8	34	4255	91	5525/ 4276	-33% -1%	130/ 149	43% 64%	11,13,17,19, 23,29,31,32
16	35	8121	84	9070/ 7348	-12% 10%	128/ 137	52% 63%	11,13,17,19, 23,29,31,64
24	36	12102	80	13258/ 11009	-9% 10%	122/ 133	53% 66%	7,11,13,17, 19,23,29,31,32
32	36	16104	79	16637/ 14140	-3% 13%	119/ 131	51% 66%	7,11,13,17, 19,23,29,31,32

Acknowledgements

Part of this work was supported by Ministerio de Ciencia y Tecnología (Spain) under project TIC2002-02227. CAD tools were provided by Altera Corp., San Jose CA.

References

1. Szabo, N. S., Tanaka, R. I.: Residue Arithmetic and its Applications to Computer Technology. McGraw-Hill, New York, 1967.
2. Soderstrand, M., Jenkins, W., Jullien, G. A., Taylor, F. J.: Residue Number System Arithmetic: Modern Applications in Digital Signal Processing. IEEE Press, 1986.
3. Ramírez, J., Meyer-Bäse, U., Taylor, F., García, A., Lloris, A.: "Design and Implementation of High-Performance RNS Wavelet Processors Using Custom IC Technologies", Journal of VLSI Signal Processing, vol. 34, pp. 227-237, 2003.
4. Ramírez, J, García, A., Meyer-Bäse, U., Lloris, A.: "Fast RNS-based FPL Communications Receiver Design and Implementation", Lecture Notes in Computer Science, vol. 2438, pp. 472-481, 2002.
5. Hamann, V., Sprachmann, M.: "Fast Residual Arithmetic with FPGAs," Workshop on Design Methodologies for Microelectronics, 1995.
6. Safiri, H., Ahamadi, H., Jullien, G., Dimitrov, V.: "Design of FPGA Implementation of Systolic FIR Filters Using Fermat Number ALU," Asilomar Conference on Signals, Systems and Computers, Pacific Grove, 1997.
7. Meyer-Bäse, U., García, A., Taylor, F.: "Implementation of a Communications Channelizer Using FPGAs and RNS Arithmetic," Journal of VLSI Signal Processing, vol. 28, no. 1/2, pp. 115-128, 2001.
8. Jenkins, W. K., Schnaufer, B. A.: "Fault tolerant adaptive filters based on the block LMS algorithm", 1993 IEEE International Symposium on Circuits and Systems, vol.1, pp. 862-865, 1993.
9. Liu, C. M., Jen, C. W.: "A parallel adaptive algorithm for moving target detection and its VLSI array realization", IEEE Transactions on Signal Processing, vol. 40, no. 11 , pp. 2841-2848, 1992.
10. Griffin, M., Sousa, M., Taylor, F.: "Efficient Scaling in the Residue Number System", 1989 International Conference on Acoustics, Speech and Signal Processing, pp. 1075-1078, 1989.

Domain-Specific Reconfigurable Array for Distributed Arithmetic

Sami Khawam¹, Tughrul Arslan^{1,2}, and Fred Westall³

¹ School of Electronic and Engineering, The University of Edinburgh, KB,
Mayfield Road, Edinburgh EH9 3JL, UK,
S.Khawam@ee.ed.ac.uk

² Institute for System Level Integration, Livingston, EH54 7EG, UK
³ EPSON Scotland Design Centre, Livingston, EH54 7EG, UK

Abstract. Distributed Arithmetic techniques are widely used to implement Sum-of-Products computations such as calculations found in multimedia applications like FIR filtering and Discrete Cosine Transform. This paper presents a flexible, low-power and high throughput array for implementing distributed arithmetic computations. Flexibility is achieved by using an array of elements arranged in an interconnect mesh similar to those employed in conventional FPGA architectures. We provide results which demonstrate a significant reduction in power consumption in addition to improvements in timing and area over standard FPGA architectures.

Keywords: Embedded reconfigurable array, programmable, distributed arithmetic, FPGA, domain specific, low-power.

1 Introduction

The arrival of portable devices processing audio and video data endorses the need for solutions providing high-speed and low-power consumption for implementing the compute-intensive multimedia calculations. Hardwired implementations of such algorithms are not suitable, as a margin of flexibility is required due to the constantly changing algorithms and DSPs provide low-throughput and high power-consumption. In the past years reconfigurable hardware has emerged as a low-cost and flexible solution for high-throughput custom hardware at the cost of increased power consumption and area.

As reported earlier in [1], a reconfigurable array specific to one type of calculation provides a good compromise between flexibility, power-consumption, area and performance when compared to DSPs, FPGAs and hardwired solutions. This paper presents a reconfigurable architecture specific to computations that can be implemented in Distributed Algorithms [2]; this includes computations such as DCT and FIR filtering used in video and audio systems.

Previous domain-specific and coarse-grain reconfigurable architectures are more processor based; e.g. [3] provides simple programmable processors interconnected together for the execution of complex algorithms. In [4] the datapath of a processor can be reconfigured during run-time to adapt to calculations. The architecture proposed in this paper is based on a heterogeneous array with a mesh of interconnects

that is able to provide more parallel computations and a higher throughput at a lower frequency. Previous programmable and configurable architectures for DCT such as the one presented in [5] provide limited flexibility in the wide range of possible implementations that could be suitable. By using FPGA-style interconnects and elements we can provide greater flexibility at a lower-level.

The paper is organized as follows: In section 0 the algorithms to be supported are overviewed. Section 0 describes the reconfigurable system and the proposed array, and in section 0 the performance of the proposed array is assessed.

2 Target Algorithms

Distributed Arithmetic (DA) [2] is a technique used to compute the inner product of two vectors, where one of the vectors is a constant. Multiplications by fixed coefficients are replaced by ROM tables and shift-accumulate operations. The basic DA scheme has shift-registers to convert the N parallel input coefficients into bit-serial data. The bits at the output of the N shift-registers are combined to form the N bits wide address for the ROM table. N different ROM tables are provided. The output coefficients are found by shift-accumulation of the output of the ROM tables (see Fig. 3 below).

Discrete Cosine Transform (DCT) [7] is an algorithm used in many compression standards like MPEG and is suitable to be implemented using DA. A number of DA implementations of DCT exists, each having different features and compromises [6]. The array presented in this paper is flexible enough to support a number of DCT implementations with features such as: CORDIC based DCT [8], digit-serial implementations [9], memory reduction using the odd-even decomposition [9], DCT size and precision change.

3 Reconfigurable System

The authors recently introduced a System-on-Chip (SoC) architecture compromising domain-specific reconfigurable arrays in [1]. A number of configurable arrays can be embedded in the system, each specific to a computation, such as Motion Estimation or DCT. The arrays are configured dynamically by the processor or the DSP. The input data to the arrays is fed either by the DSP or the processor and the output is read back from the array. The elements of the array are described below.

3.1 Clusters for Distributed Arithmetic

A general DA implementation of a Sum-of-Product requires the following elements:

- Shift registers to convert bit-parallel input coefficient to bit-serial or digit-serial data.
- Memory elements to store the content of the ROM replacing the multiplication.
- Shift-accumulators for calculating the result.

Additionally, adders and subtracters are needed to implement techniques such as the odd-even DCT decomposition. It was thus chosen to use two types of elements in the array: A memory element and an element for add/sub/shift and accumulation, as detailed below. A cluster is formed by combining four such elements together using configurable switches. The clusters are arranged in an array as shown in Fig. 1. More add-shift clusters are used than memory clusters, due to the needs of the application. The columns are arranged uniformly to simplify manual routing and placement.

3.1.1 Memory Element

In this initial architecture the memory element used is a dual-port 1-Kbit RAM. Four such elements are packed into a *memory cluster*. The cluster contains logic, similar to the one found in [10], allowing to configure the memory in a number of geometries as shown in Table 1. The size of the memory was chosen according to the most used memory sizes in DCT calculations.

Table 1. Possible geometries of a memory cluster

Word Size	Bits per word			
	4-bit	8-bit	12-bit	16-bit
256	x	x	x	x
512	x	x		
768	x			
1024	x			

3.1.2 Add and Shift Element

To support most DA calculations, the add-and-shift element can be configured to act as the following:

- Adder/Subtractor
- Loadable shift-register useful for parallel-to-serial conversion. Right and left shifts supported.
- Accumulator. The adder in the accumulator can be dynamically configured to subtract.
- Shift-accumulator to be used at the ROM table output in DA calculations.

The elements have a programmable register at the output that can be enabled to support pipelined implementations. Every element is 4-bits wide; four elements are grouped into a cluster with interconnects provided to support cascading in order to allow wider bit ranges (up to 16-bit).

3.2 Mesh Interconnects

As described earlier in [1], symmetrical mesh interconnects are used to provide the connections between the clusters. Two types of tracks are provided: Six 8-bit wide tracks for data and six 1-bit tracks for control lines. Interconnects are composed of connection-boxes (C-Boxes) that connect the pins of a cluster to the tracks and switch-boxes (S-Boxes) that connect together the intersections of tracks. As in [1], the C-Boxes have a flexibility of $F_c=6$ and the S-Boxes of $F_s=3$, as defined in [11].

The configurable switches are implemented using tri-state buffers, which greatly increases the area, timing and power consumption of the array [12], when compared to using pass-transistors. Using tri-state buffers makes the architectures designed synthesizable, portable to any process and compliant with the design-flow used for the rest of the SoC.

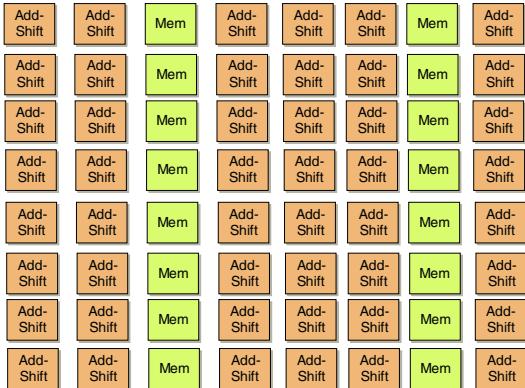


Fig. 1. Clusters arrangement in the array. More add-shift clusters are used due to their need.

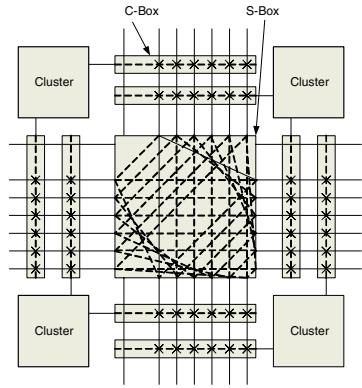


Fig. 2. Connection Boxes and Switch Boxes connect the clusters together [11].

4 Array Performance

The benchmark circuit used is a simple 8-point 1-D DCT using bit-serial DA without memory compression, as shown in Fig. 3. This circuit is manually mapped to the array as follows:

- Each 12-bits shift register is mapped to three add-and-shift elements.
- Each 2-Kbit memory is mapped to two 1-Kbit memories preprogrammed and used as ROMs.
- Each 16-bit shift accumulator is mapped to four add-shift modules from one cluster.

By implementing this benchmark circuit, *our array* was compared to a standard hardwired ASIC specially optimized for DCT and a commercial Xilinx Virtex-E FPGA. The measured power consumption, area and maximum frequency is shown in Table 2. All of these systems use .18 μ m CMOS technology and run at 1.8V and 10MHz.

The area of the Virtex-E and our implementations does not include the area used by the configuration memory, but includes the area used by interconnects and reconfigurable switches. The Xilinx area estimation is based on the assumption that a *slice* and its belonging C- and S-boxes have an area of 3303 μm^2 ; 71 slices are needed to implement one row. It can be seen that our implementation is 14% smaller than the Virtex-E area, however, this stays significantly larger than the hardwired implementation. This is partly caused by the fact that in the ASIC implementation no

RAMs are used for storing the coefficient but hardwired logic is used to implement the coefficient ROM tables, which greatly decreases the area and limits the flexibility.

The power consumption values measured for our array and for the hardwired implementation are obtained using post-routing simulation with typical switching activity. In the case of the Virtex-E FPGA, the power consumption is obtained with typical estimations provided by Xilinx. In both array cases, the power values include the power consumed by the configuration circuit. Our array consumes 38% less power than the Xilinx implementation since it has less interconnects and operates using larger clusters. Our reconfigurable array consumes 277% more power than ASIC due to the added switches, and also partly to the use of RAM over hardwired-LUTs.

With respect to timing, our array has a maximum frequency around 54% higher than that of Virtex-E., This is still 63% less than the maximum frequency achievable with hardwired ASIC due to the delays added in reconfigurable switches and to the higher-loads and longer routing.

Table 2. Performance comparison between hardwired ASIC, our array and a commercial Xilinx FPGA on one row of the array.

	.18µm ASIC	<i>Our array</i>	Xilinx's Virtex-E
Area (μm^2)	17 483	202 366	234 510
Power cons. (mW)	0.52	1.965	3.2
Max Freq. (MHz)	210	77	50

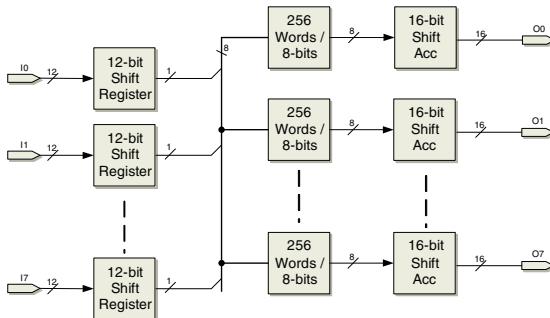


Fig. 3. 1-D DCT implemented using DA. For 8-points DCT 8 rows or elements are required.

When comparing the DCT implementation on our array to that on Virtex-E, it should be noted that the routing was done manually in the case of our array, while it was done with automatic software on Virtex-E, meaning that the performance of our array could have been more optimized if software was used for generating the routing and placement.

5 Conclusion

In this paper, we have introduced an embedded reconfigurable array targeting Distributed Arithmetic (DA) operations. The architecture is based on programmable clusters of add-shift and memory elements arranged in an array. Different levels of reconfigurable interconnects are provided to allow flexible mapping of diverse DA algorithms, such as a wide range of DCT computations, to the array.

The performance measured shows that DA implementations on the proposed architecture provide considerable improvements over standard low-level FPGAs ones: Power consumption is reduced by 38%, occupied area is decreased by 14% and the maximum operating frequency is increased by 54%.

When comparing the array with standard ASIC implementations, it becomes clear that this architecture provides a compromise between hardwired ASIC and generic FPGA solutions in terms of flexibility, area, timing and power consumption when used in portable multimedia devices.

References

1. S. Khawam, T. Arslan, F. Westall, *Embedded reconfigurable array targeting motion estimation applications*, 2003 IEEE International Symposium on Circuits and Systems (ISCAS 2003)
2. White, S.A, *Applications of distributed arithmetic to digital signal processing: a tutorial review*, ASSP Magazine, IEEE, Volume: 6 Issue: 3 , Jul 1989, Page(s): 4 -19
3. A. Abnous, J. M. Rabaey. *Ultra-low-power domain-specific multimedia processors*. IEEE VLSI Signal Processing. 1996
4. D. C. Cronquist, P. Franklin C. Fisher M. Figueroa and C. Ebeling. *Architecture Design of Reconfigurable Pipelined Datapaths*, 20th Anniversary Confe. on Advanced Research in VLSI, 1999
5. Burleson, W.; Jain, P.; Venkatraman, S., *Dynamically parameterized architectures for power-aware video coding: motion estimation and DCT*, 2nd International Workshop on DVC 2001, Pages: 4- 12
6. Sungwook Yu; Swartzlander, E.E., Jr., *DCT implementation with distributed arithmetic*, IEEE Transactions on Computers , Vol. 50 Issue 9 , Sept. 2001
7. N. Ahmed, T. Natarajan, K.R. Rao, *Discrete Cosine Transform*, IEEE Trans. On Computers. Vol. C-23, No. 1, pp.90-93, December 1984
8. Yi Yang; Chunyan Wang; Omair Ahmad, M.; Swamy, M.N.S., *An on-line CORDIC based 2-D IDCT implementation using distributed arithmetic*, Sixth ISSPA 2001 , Volume: 1
9. Kyounsoo Kim; Jong-Seog Koh, *An area efficient DCT architecture for MPEG-2 video encoder* ,Consumer Electronics, IEEE Transactions on , vol. 45 Issue: 1 , Feb. 1999
10. Wilton, S.J.E.; *Embedded memory in FPGAs: recent research results*, Communications, Computers and Signal Processing, 1999 IEEE Pacific Rim Conference on , 1999 , Page(s): 292 -296
11. Rose J., Brown S., *Flexibility of interconnection structures for field-programmable gate arrays*, Solid-State Circuits, IEEE Journal of , Vol.26, Iss.3, 1990, Pages: 277-282
12. V. George, H. Zhang J. Rabaye. *The design of low energy FPGA*, Proceedings. 1999 International Symposium on Low Power Electronics and Design, pp. 188-193. 1999

Design and Implementation of Priority Queuing Mechanism on FPGA Using Concurrent Periodic EFSMs and Parametric Model Checking

Tomoya Kitani¹, Yoshifumi Takamoto¹, Isao Naka², Keiichi Yasumoto³,
Akio Nakata¹, and Teruo Higashino¹

¹ Graduate School of Information Science and Technology, Osaka University
`{t-kitani,takamoto,nakata,higashino}@ist.osaka-u.ac.jp`

² Dept. of Tourism, Osaka Seikei University `naka@osaka-seikei.ac.jp`

³ Graduate School of Information Science, Nara Institute of Science and Technology
`yasumoto@is.aist-nara.ac.jp`

Abstract. In this paper, we propose a design and implementation method for priority queuing mechanisms on FPGAs. First, we describe behavior of WFQ (weighted fair queuing) with several parameters in a model called *concurrent periodic EFSMs*. Then, we derive a parameter condition for the concurrent EFSMs to execute their transitions without deadlocks in the specified time period repeatedly under the specified temporal constraints, using parametric model checking technique. From the derived parameter condition, we can decide adequate parameter values satisfying the condition, considering total costs of components. Based on the proposed method, high-reliable and high-performance WFQ circuits for gigabit networks can be synthesized on FPGAs.

1 Introduction

Due to recent progress of IP telephony and video/audio streaming systems, it has been very important to provide QoS (Quality of Service) in wide area networks.

In typical situations, it is required for high-end routers for backbones to process up to 10 Gbps traffic, while SOHO routers at most 100 Mbps traffic at cheaper cost. In order to reduce development costs in hardware implementation, it is desirable to use the same architecture for both high-end and SOHO routers, and to synthesize circuits with the specified performance only by adjusting parameters such as CPU/memory speed, circuit size, etc.

In this paper, we propose a flexible and reliable hardware design and implementation method using concurrent periodic EFSMs [3] and parametric model checking [4].

2 High-Reliable Design and Implementation Method

In the proposed method, we design and implement hardware circuits as follows: (1) describe behavior of a target system with several parameters in concurrent periodic EFSMs; (2) derive parameter conditions for deadlock freeness in the system using a parametric model checking technique; (3) derive a scheduler that allows all EFSMs to

execute only schedulable paths (satisfying time constraints, synchronization conditions, and so on) by assigning appropriate values to parameters; (4) derive VHDL-description which correspond the each EFSM and the scheduler module; and (5) implement the VHDL-description on FPGA with a commercial tool.

Concurrent Periodic EFSMs: EFSM is an extended FSM which has registers to deal with variables. Each transition rule is defined as $s_{cur} \xrightarrow{a[guard]} s_{next}$. Here, *guard* is a transition condition. If the value of the transition condition *guard* is true at state s_{cur} and event a is executed, then the EFSM moves to state s_{next} . By using time variables in transition conditions, we can give a constraint for execution time of each event.

We assume that every path (event sequence) from the initial state has the special dummy transition ψ as the last event of the path where the transition condition of ψ is specified so that the path can be executed in the specified time interval T . In our model, the multi-way synchronization mechanism [2] can be specified among EFSMs so that any subset of EFSMs can synchronize with each other by exchanging data when some conditions hold among the subset. By using multi-way synchronization, we can easily describe the real-time hardware system consisting of multiple parallel modules which frequently interact with each other by exchanging messages.

Parametric Model Checking: In [4], we have proposed a parametric model checking method for a periodic EFSM. In our method, temporal properties are written in *RPCTL* (Real-time and Parametric extension of Computation Tree Logic). Since the model is restricted to be periodic, our method can derive parameter conditions efficiently by analyzing at most three periods' behavior of a given periodic EFSM (see [4] for details).

Here, we use the parametric model checking method in [4] for obtaining the parameter condition. However, other model checkers such as Ref. [1] can be also used when the specifications are restricted in a class which the model checker can treat.

Hardware Synthesis of Concurrent Periodic EFSMs: In [3], we have proposed a tool to generate RT-level VHDL descriptions from given system specifications in concurrent periodic EFSMs. In the derived VHDL description, EFSMs are implemented as sequential circuits working with the same clock, and the multi-way synchronization among EFSMs is implemented as AND gates with priority encoders. Moreover, the scheduler which controls EFSMs to execute only schedulable paths (i.e., path satisfying time constraints) is implemented (see [3] for details).

3 Application and Evaluation

Specifying Priority Queuing Mechanism

In WFQ mechanism, each packet has its own priority called *class*, and different queues are used for the classes. We can assign priorities among classes so that total output amounts from queues are proportional to the fixed rates given to the corresponding classes.

As shown in Fig. 1, we compose the WFQ mechanism of four parts that are P_i , $Q(i)$, Po and Sch . Here, each $Q(i)$, $1 \leq i \leq CMAX$ is responsible for storing and extracting

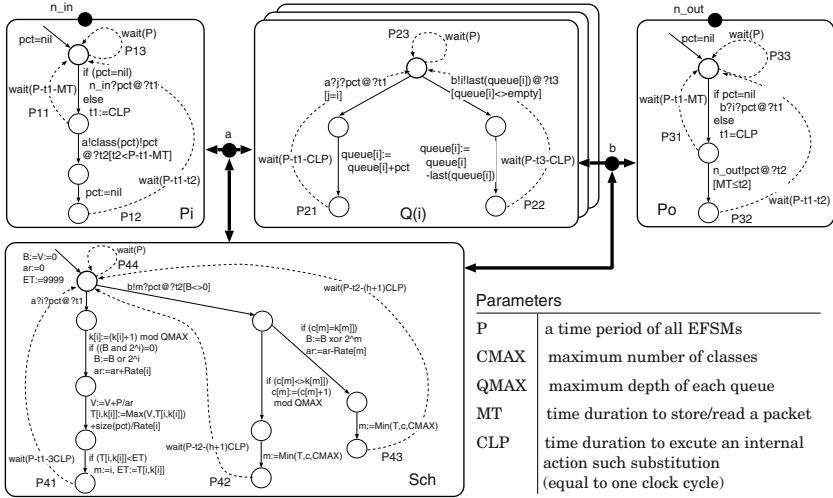


Fig. 1. Modules for WFQ in model concurrent periodic EFSM

packets with class i to/from the i -th queue. Sch is the WFQ algorithm described based on [5]. As a total, CMAX+3 periodic EFSMs are executed in parallel. In Fig. 1, n_in and n_out are gates which correspond to an input link from the network to the router and an output link from the router to the network, respectively. And, a and b are internal gates which are used as interaction points between P_i and one of $Q(1), \dots, Q(CMAX)$, between one of $Q(1), \dots, Q(CMAX)$ and P_o , respectively.

3.1 Experimental Results and Evaluation

Using our tools proposed in [3], we have synthesized the RT-level VHDL description from the specification of WFQ in Fig. 1. Before synthesizing circuits, we must assign appropriate values to constant parameters explained in the previous section. The parameter condition derived by the technique in Sect. 2 was

$$\begin{aligned} 2n(h+1) \cdot CLP &\leq Period \quad \text{and} \quad (2n-1)(h+1) \cdot CLP + MT \leq Period \\ \text{and} \quad 6n \cdot CLP &\leq Period \quad \text{and} \quad (2n-1) \cdot 3CLP + MT \leq Period \end{aligned}$$

Here, $h = \lfloor \log_2 CMAX \rfloor$. We also assume that n is the number of parallel execution of P_i , P_o and $2n \leq CMAX$ (When $n = 2$, two EFSMs are executed in parallel for P_i and P_o , respectively, that is, two packets can be processed in a period).

Parameter Decision for High-end Routers: In this case, it may be considered that performance is more important than component costs. So, we choose components to minimize $Period$, that is, make values of MT and CLP as small as possible. When we select Stratix series of Altera whose maximum clock frequency is 420MHz. Thus, the lower bound of CLP is $1/420\text{MHz}=2.38\text{nsec}$. Stratix series support PC1600 DDR SDRAM memory. If we suppose that packet size is 1500B (bytes) and data bus size is 64 bit, we obtain that $MT = 1500 \times 8/64/200\text{MHz}=938\text{nsec}$.

On the other hand, the circuit size of WFQ can be represented by $C + \alpha \cdot CMAX + \beta \cdot CMAX \cdot QMAX + \gamma \cdot (n - 1)$. Here, C , α , β and γ denote the common circuit size, the additional circuit size by adding one class, the size when increasing the queue depth by one, the size when increasing n by one, respectively. In our preliminary experiments, we know that $C=2718\text{LE}$ (logic elements), $\alpha=57\text{LE}$, $\beta=0.1\text{LE}$, and $\gamma=266\text{LE}$.

When we require that $CMAX=64$, $QMAX=256$, and $n=4$, the required circuit size will be 9180 LE. If we select Stratix series, therefore, we will find that EP1S10 (10570LE, 70USD) is enough with regard to the circuit size. The required memory size is $64 \times 256 \times 1500\text{B}=197\text{Mbit}$. Then using two 128Mbit chips of PC1600 DDR SDRAM (about 2USD per chip) is sufficient. Since $n = 4$, we need at least 4 DRAM chips. Consequently, the total cost is calculated as about 78USD in this case.

Actually, this WFQ circuit implemented on Stratix FPGA device can work at 108.9 MHz. From parameter condition $(2n - 1) \cdot (h + 1) \cdot CLP + MT \leq Period$, we obtain that $Period = 1.39\mu\text{sec}$. Since four packets can be processed every period for the best case, the performance will be about 34.6Gbps.

When we design and implement hardware circuits with different components to achieve various performance, we have to describe the corresponding specification for each circuit. In our method, based on the same specification, we can derive parameter conditions for satisfying a specified property such as deadlock freeness and synthesize the hardware circuits for satisfying different requirements by deciding parameter values.

4 Conclusion

In this paper, we have proposed a design and implementation method for a WFQ algorithm using concurrent periodic EFSMs and a parametric model checking. In our method, we can derive parameter conditions for deadlock free property with several parameters such as maximum input/output link speed of a router, the number of classes, memory speed, and so on. From experimental results, we believe that our method can be used for design and implementation of QoS routers for various environments by only adjusting parameter values considering cost and performance of components.

References

1. T. A. Henzinger, P-H. Ho and H. Wong-Toi : “HYTECH: A Model Checker for Hybrid Systems”, International Journal on Software Tools for Technology Transfer, vol. 1, no. 1-1, pp.110–122 (1997).
2. ISO : “Information Processing System, Open Systems Interconnection LOTOS”, ISO 8807 (1989).
3. H. Katagiri, M. Kirimura, K. Yasumoto, T. Higashino and K. Taniguchi, K. : Hardware Implementation of Concurrent Periodic EFSMs, Proc. of Joint Intl. Conf. on 13th Formal Description Techniques and 20th Protocol Specification, Testing, and Verification (FORTE/PSTV2000), pp. 285 - 300 (2000).
4. A. Nakata, and T. Higashino: “Deriving Parameter Conditions for Periodic Timed Automata Satisfying Real-Time Temporal Logic Formulas,” in Proc. of 21st IFIP Int'l Conf. on Formal Techniques for Networked and Distributed Systems (FORTE2001), pp. 151-166, 2001.
5. A. Parekh and B. Gallager : “A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks—The Single Node Case—”, IEEE/ACM Trans. on Networking, Vol. 1, No. 3, pp. 344–357 (1993).

Custom Tag Computation Circuit for a 10Gbps SCFQ Scheduler

Brendan McAllister, Sakir Sezer, and Ciaran Toal

School of Electrical and Electronic Engineering
Queen's University Belfast
Ashby Building, Stranmillis Road
Belfast, BT9 5AH, N. Ireland, U.K.

Abstract. This paper details the architecture and implementation of a tag computation circuit for a Self-Clocked Fair Queuing (SCFQ) Scheduler. The core objectives of the presented project is the implementation of a custom accelerator circuit that is optimized to process tag values for terabit router nodes operating at 10 Gbps per link. The system is implemented using FPGA technology and provides extended programmability to adapt the tag computation to a range of custom scheduling schemes.

1 Introduction

One of the major shortcomings of the current Internet is the best effort services. On the other hand, the traffic diversity is increasing with new emerging services producing variable and burst traffic patterns requiring extremely low network delay. It has become a necessity to differentiate Internet and service based traffic on their type and priority. The QoS issue with regards to the Internet has been one of the main research topics of the industrial and academic research community over the last six years. Numerous protocols, traffic handling schemes and techniques have been proposed and implemented

The research presented in this paper is based on IP QoS research and investigates hardware architectures for accelerator circuits and network processing elements for terabit core routers. It examines parallel processing architectures for programmable scheduling and presents an architecture and implementation of a SCFQ scheduler using a FPGA.

2 Weighted Fair Queuing

WFQ is probably the most well known fair queuing algorithm for fair scheduling of variable size packets. It allows an arbitrary number of end-to-end connections having a fair access to a link. WFQ is computationally complex and causes a significant implementation problem for high throughput rates. The computation complexity of

virtual time occurs, as whenever there is a change from busy to idle or reverse in a class, the algorithm requires further computation.

Using the same principles of WFQ, Self-Clocked Fair Queuing (SCFQ) is an approximation of the same calculation. The virtual time used in SCFQ is a measure of the progress of the system itself. Whenever the system changes state from busy to idle, virtual time resets to zero. In fact, SCFQ computation of virtual time is much simpler than that of WFQ and is therefore much more practical solution. It does not always achieve the delay and fairness properties of WFQ but these disadvantages are easily outweighed by the ease of implementation.

3 Tag Computation Architecture

Tag computation plays a vital role in the scheduling procedure of many fair queuing scheduling schemes. The method for computing finishing tags determines not only the fairness of the scheduling process but also the throughput rate i.e. how many finishing tags can be computed per second. Our research investigates architectures that are optimized for a specific technology, in this case FPGAs. The presented finishing tag computation circuit is highly parallel and pipelined. It is composed of a range of distributed on chip memory blocks.

Figure 1 shows the block level description of the finishing tag computation block. It is composed of four main blocks; R_k Evaluation, IP Acquire, Tag Value Calculation and Virtual Time. Two lookup tables, one for R_k lookup and another for tag data lookup, are implemented using Stratix embedded block RAMs

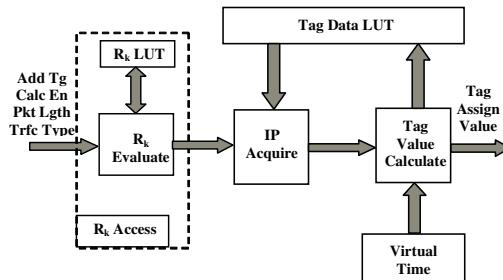


Fig. 1. Block Level Description

3.1 R_k Lookup and IP Acquire

The R_k lookup table is composed of 8 Stratix M4K RAM blocks of 4kbits per block accommodating up to 1024 lookup entries. Figure 2 shows the data flow of the R_k lookup. The lookup table can be addressed by the *TrafficType*, or by the *AddTg* which are determined by the packet classifier at the switch/router input port. This feature allows the tag computation block to be programmable to support per class or per flow queuing or a mixture of both. The lookup table translates the traffic type into an

equivalent R_k which will be used to calculate the individual finishing tag for each IP packet. The lookup table size determines the number of flows simultaneously supported by the SCFQ scheduler. The presented implementation supports up to 1024 flows. The QoS adjustment of each individual traffic type or flow is accomplished with the assigned R_k value.

The IP Acquire block is similar to the R_k lookup block and is composed of 8 Stratix M4K RAM blocks. It looks up the $PrvTgValue$ (previous finishing tag value) and the $PktNo$ (previous packet number) for finishing tag computation. Figure 3 shows the block diagram of the IP acquire block.

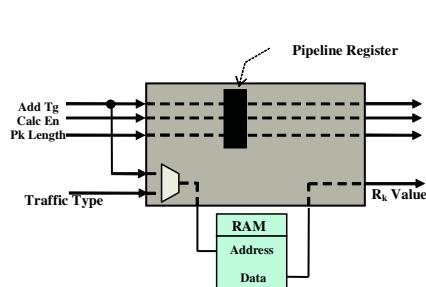


Fig. 2. R_k Lookup

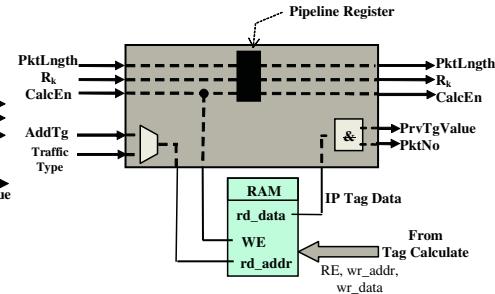


Fig. 3. IP Acquire

3.2 Virtual Time and Tag Calculator

The Virtual Time block is a counter circuit producing an integer count value to emulate the motion of time for the SCFQ scheduler. The finishing tag may run for infinite time whereas the CVT will only be a finite value. Tag computation is carried out by the tag calculator block. Figure 4 illustrates the data path of the tag calculator block.

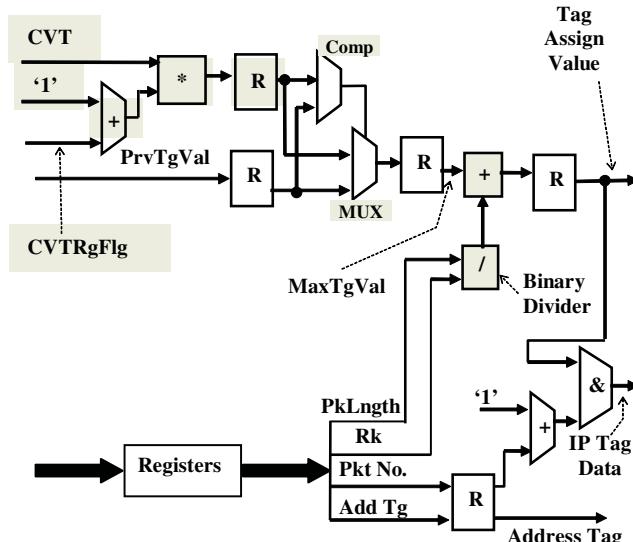


Fig.4. Tag Calculator Data Path

The tag computation circuit is a pipelined parallel map of the tag computation algorithm [1] and is able to compute one tag value per clock cycle.

4 Circuit Study and Conclusions

The finishing tag computation circuit is synthesized and targeted to the smallest Altera Stratix device [4] using Synplify Pro and Altera Quartus II tools. The speed and area performances are examined. The post-layout synthesis results are included in Table 1.

Table 1. Post-Layout Synthesis Results for Altera EPIS10F484C5

Tag Assign Circuit Post-layout Synthesis				
Device	Clock Speed	Logic Cells	Register Bits	RAM Blocks
EPIS10F484C5	17.65 MHz	1,662 (16%)	358 (4%)	16 M4K RAM (27%)

Although the latest high performance FPGA technology with embedded high speed memory has been used, the maximum speed of the circuit is significantly lower than expected. A circuit speed of 17.65 Mhz means 17.65 million tag computation per second can be achieved. The circuit is optimized to perform one finishing tag computation every clock cycle. Assuming a minimum IP packet length of 80 bytes, the tag computation circuit is able to service a link with a throughput rate of 10 Gbps, fulfilling the minimum requirement.

The presented architecture demonstrates that new generation FPGA architectures with a range of embedded peripherals and memories are an ideal platform for high throughput programmable network processing allowing the implementation of Tera-bit network nodes.

References

1. Mark W. Garrett, Bellcore. A Service Architecture for ATM: From Application to Scheduling, IEEE Network Magazine, Vol.10, No. 3, pp 6-14, May/June 1996.
2. A. K. Choudhury, E. L. Hahne. New implementation of multi-priority pushout for shared memory ATM switches, Computer Communications, vol. 19, pp. 245-256, March 1996.
3. K.C.Chang, Digital Systems Design with VHDL and Synthesis. An Integrated Approach, IEEE Computer Society Press and John Wiley & Sons, 1999, ISBN 0-7695-0023-4.
4. Stratix FPGA Family Data Sheet, Altera Corporation, San Jose, CA 95134, USA, Data Sheet Version 3.0, December 2002.

Exploiting Stateful Inspection of Network Security in Reconfigurable Hardware

Shaomeng Li, Jim Tørresen, and Oddvar Søråsen

Department of Informatics, University of Oslo, N-0316 Oslo, Norway
`{shaomenl,jimtoer,oddvar}@ifi.uio.no`

Abstract. One of the most important areas of a network intrusion detection system (NIDS), stateful inspection, is described in this paper. We present a novel reconfigurable hardware architecture implementing TCP stateful inspection used in NIDS. This is to achieve a more efficient and faster network intrusion detection system as todays' NIDSs show inefficiency and even fail to perform while encountering the faster Internet. The performance of the NIDS described is expected to obtain a throughput of 3.0 Gbps.

1 Introduction

“Stateful inspection” is applied in Network Intrusion Detection Systems (NIDS) and is a more advanced network security tool than firewalls. It is used for checking the handshakes in a communication session by using detailed knowledge of the rules of the communication protocol. This is to make sure that it is completed in an expected and timely fashion. By checking a connection (packet by packet) – not just one single packet, and knowing what has just happened and what should happen next, stateful inspection detects incorrect or suspicious activity and alerts flags to the system administrator [1].

1.1 TCP Connection Stateful Inspection

TCP (Transmission Control Protocol) [2] is an important Internet protocol. It provides a full duplex reliable stream connection between two end points in the TCP/IP network. The approach of using stateful inspection will be one of the best ways (maybe the only way) to monitor a TCP connection.

In NIDS Snort, a software based STREAM4 preprocessor with 3000 lines software code is designed to conduct the TCP stateful inspection performing two functions: Stateful inspection sessions (monitoring handshakes) and TCP stream reassembly (collecting together packets belonging to one TCP connection). Testing Snort on various networks has shown that the STREAM4 preprocessor leads to a bottleneck in Snort for some network traffic environments (details can be found in [3]). Thus, to improve the performance of Snort, we would like to explore how reconfigurable hardware might be used to replace the STREAM4 TCP stateful inspection.

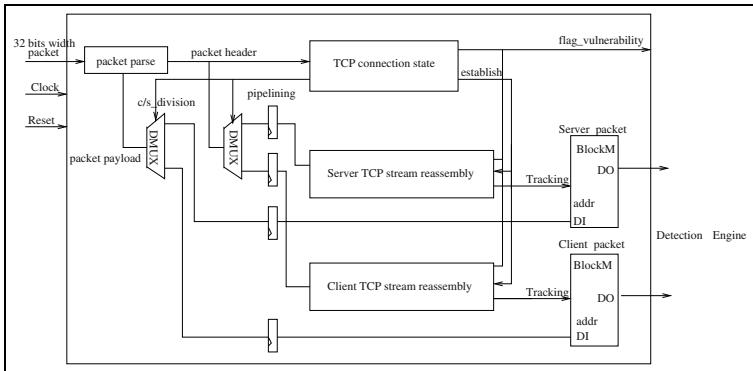


Fig. 1. Block diagram of reconfigurable hardware on TCP connection.

2 Exploiting New Implementation Methods for TCP Stateful Inspection

The new approach would be to process the stateful inspection in hardware rather than software as usual. Implementation in Field Programmable Gate Arrays (FPGAs) is appropriate to make such explorations. The new hardware architecture is proposed in Fig. 1. This unit will be an add-on unit for the computer running the Snort software.

Incoming packet data (32 bit width) is input to the reconfigurable hardware unit which processes the TCP three way handshake and the Server and Client TCP stream reassembly. The information of the packet header will be stored in some registers based on the libpcap library which is used in Snort to get a packet off from the wires.¹ The basic packet header information most frequently referenced are the sequence number, acknowledge number, window size and TCP flags such as the SYN and ACK bit.

The TCP connection state unit is implemented as a state machine to check the three way handshake of the TCP connection. After establishing the proper connection (by TCP three way handshake), the data over a TCP connection can be exchanged between the Client and the Server. The processing of data flowing to the Server side and the Client side can be performed separately and in parallel, even if the Server and the Client TCP stream reassembly units conduct the same function. This means that packets sent to the Server and the Client side are reconstructed individually in independent hardware units. By doing this, the processing of TCP stream reassembly units in a NIDS is accelerated, of course, at a cost of extra FPGA resources. Two 32 bit DMUXs (one for header and one for payload) are added to separate incoming packets into the Server and the Client packets. The reason for doing this is to feed incoming packets into the Server TCP stream reassembly unit and the Client TCP stream reassembly

¹ Registers for the packet header are not shown in Fig.1.

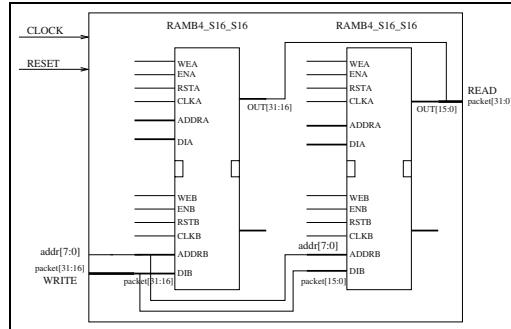


Fig. 2. The Server (or the Client) packet memory.

unit, respectively. The TCP stream reassembly units are running in parallel and determine which packets need to be stored in the “Client packet” or the “Server packet” memory. This avoids the need for large TCP stream reassembly buffers.

Two 32 bit comparators and one 32 bit adder are needed to implement one TCP stream reassembly unit. If the sequence number of an incoming packet is outside of the band size (band size is decided by the initial sequence number (ISN) and window number), the packet will be dropped. The payload of the packet is otherwise stored into the “Server packet” memory or “Client packet” memory respectively, to reconstruct the data for a succeeding detection engine. By pipelining the TCP stream reassembly, the “Server packet” memory and the “Client packet” memory unit, the total performance can be enhanced.

The size of the packet memories are 5x32 bit (16 bit data bus). 5x32 bit is required as the signature pattern can be matched at a maximum of 5x32 bits in the succeeding detection engine [4]. However, a dual port (write/read) memory is required for the Server/Client packet units with minimum size of 5x32 bits. Using a dual port RAM for the packet memory is important to be able to receive new data when matching (reading) is concurrently undertaken.

Virtex XCV1000-6 FPGA to be used in this work contains RAM blocks called SelectRAMs. Each has a capacity of full synchronous dual ported 4096-bit memory and is ideal to implement the Server/Client packet unit. One such block SelectRAM can be configured as a memory with different data widths and depths. However, since dual port RAM is required, the maximum data width is limited to 16 bits. The library primitives, the RAMB4-S16-S16 is dual ported where each port has a width of 16 bits and a depth of 256 bits which is available in the XCV1000-6. By considering the size of the RAMB4-S16-S16 and the packet which has 32 bit data width, two such block SelectRAMs are therefore needed to implement *one* 32 bit data bus packet memory – see Fig.2. Since there are two packet memories (the Client and the Server), a total of four block SelectRAMs are therefore needed to implement the “Server packet” and the “Client packet” memory units.

Processing the data flow on the Server side and Client side in parallel and eliminating the need for a large reassembly buffer are our main contributions to improve the process of TCP connection in a NIDS. This makes it different from the approach in [5]. Data path processing in parallel is the main feature used when implementing the Server and the Client TCP stream reassembly in FPGA. Thereby the performance of NIDS facilitated by the method could be enhanced.

3 Experiments

Our implementation of this study is analyzed by using the ISE FPGA tool from Xilinx [6]. Designs are to be mapped onto a Virtex XCV1000-6 FPGA.

All individual modules such as TCP connection state, TCP stream reassembly unit and DMUX are implemented in VHDL. The simulation of those functions were conducted by the Modelsim XE II v6.5a simulator [6].

Except for the packet parsing, the whole system has been placed and routed into a XCV1000-6 FPGA. The minimum clock period for data from input to output is 10.467 ns which corresponds to a throughput of 3.06 Gbps.

However in IP/TCP networks, the Server often needs to be able to handle multiple connections simultaneously. Hence, multiple TCP connections have to be considered in this study. The process which consumes most SLICEs in the FPGA is the module which does doing the TCP three way handshake. Although there are 12288 SLICEs in one XCV1000-6 FPGA, the possibility of having multiple TCP connections is limited to the capability of implementing units of the “Server packet” memory and the “Client packet” memory in one such FPGA. The reason for this is that the height of the CLB array in one FPGA decides the number of block SelectRAMs, consequently determining the size of the packet units. One XCV1000-6 FPGA with the amount of 32 block SelectRAMs can therefore implement only 8 TCP connections. Although the size of the SLICEs of such an FPGA should be checked to see if it is enough to implement remaining modules of 8 TCP connections simultaneously.

By using a Virtex XCV812E FPGA which has 280 block SelectRAMs as used in [5], 70 multiple TCP connections can be expected to be implemented in one FPGA.

4 Conclusions

Stateful inspection over a TCP connection is studied and implemented in FPGA based hardware to remove the bottleneck of TCP connection in a network traffic environment. A novel approach using reconfigurable hardware is introduced. Experiments show that the performance could be improved by this implementation to a throughput of 3.0 Gbps.

References

1. Michael Clarkin. “*Comparison of CyberwallPLUS Intrusion Prevention and Current IDS technology*”. NETWORK-1, Security Solutions, Inc., White Paper.

2. J. Postel. “*Request For comment 793, Transmission control Protocol*”. 1998.
3. Sergei et al. “*SNORTTRAN: An Optimizing Compiler for Snort Rules*”. Fidelis Security Systems, Inc., 2002.
4. Shaomeng Li et al. “*Exploiting Reconfigurable Hardware for Network Security*”. in Proc. of 11th Annual IEEE Symposium on FIELDS-Programmable Custom Computing Machines (FCCM’03), 2003.
5. Marc Necker et al. “*TCP-Stream Reassembly and State Tracking in Hardware*”. in Proc. of 10th Annual IEEE Symposium on FIELDS-Programmable Custom Computing Machines (FCCM’02), School od Electrical and computer Engineering, Georgia Institute of Technology, Atlanta, GA, 2002.
6. <http://www.xilinx.com>

Propose of a Hardware Implementation for Fingerprint Systems

Vanderlei Bonato¹, Rolf Fredi Molz¹, João Carlos Furtado¹,
Marcos Flôres Ferrão¹, and Fernando G. Moraes²

¹UNISC – Departamento de Informática
Av. Independência, 2293 Bairro Universitário. CEP: 96815-900
Santa Cruz do Sul/RS – Brazil
rolf@unisc.br, vbonato74@hotmail.com

²PUCRS – Faculdade de Informática
Av. Ipiranga, 6681 - Prédio 30.
CEP: 90619-900 - Porto Alegre - Brazil
moraes@inf.pucrs.br

Abstract. Fingerprint is graphical flow-like ridges presents on human fingers. Each fingerprint is unique, offering a clear and unambiguous method to identify an individual. The uniqueness of each fingerprint is determined by fine details embedded in its overall structure, named *minutiae*. Fingerprint classification system is CPU time intensive, usually implemented in software. This paper presents an alternative way to identify the *minutiae* from fingerprints, aiming real-time processing. In the first part is implemented the alternative algorithm in software (Delphi) and after this is presented an architecture to be implemented using a configurable devices (FPGA). The performance of this algorithm, in hardware and software, are analyzed, presenting the spent time within each system block.

1 Introduction

Fingerprint is graphical flow-like ridges presents on human fingers. They have been widely used in personal identification for several centuries. The uniqueness of each print is determined by fine details embedded in its overall structure that is known as minutiae. In figure 1 is showed some fingerprint details, where are presented minutiae, core and axis.

2 Proposed Algorithm

The algorithm proposed in this paper is considered the state-of-the-art and well different of the existing solutions found in automated fingerprint systems. The main goal is to implement a system in a hardware environment to obtain high performance. To obtain this performance the algorithm used is based on simple tasks.

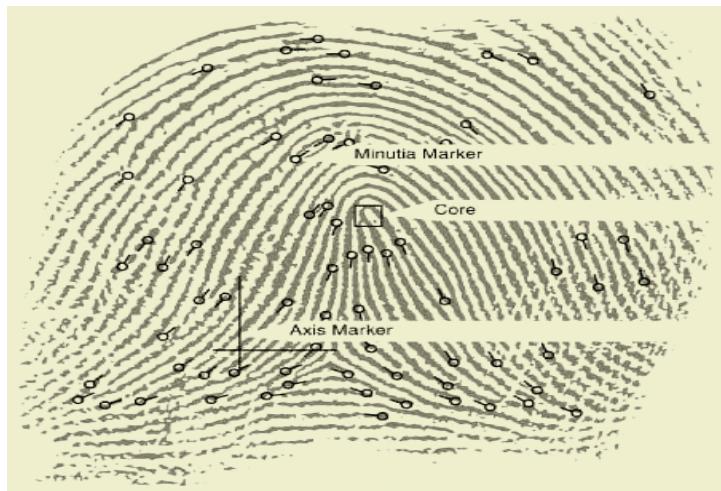


Fig. 1. Some fingerprint details

The algorithm is divided in the following processing steps: (i) input filter represented by a Gaussian filter; (ii) gradient and direction computation; (iii) ridge detection; (iv) minutiae detection. These steps are illustrated in Figure 2. In the Figure 3 we can see the results of the processing.

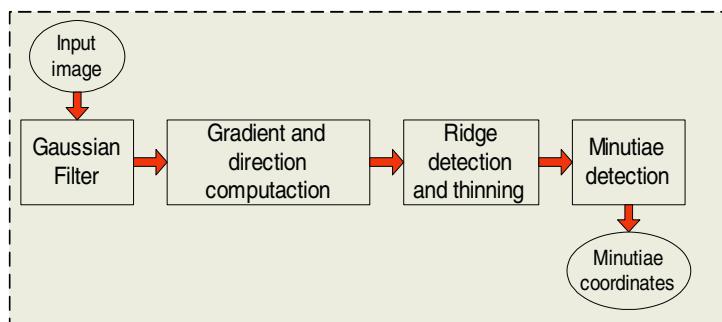


Fig 2. Blocks of the algorithm proposed

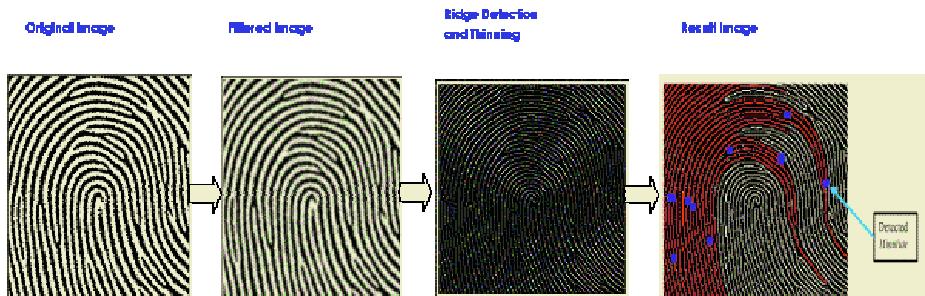
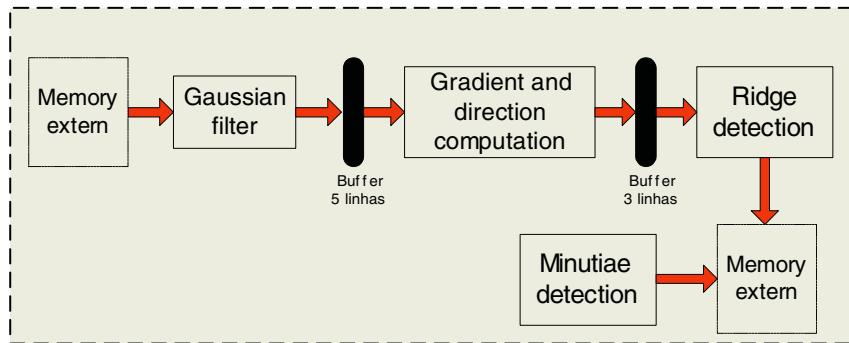


Fig. 3. Results after steps processed.

**Fig. 4.** Hardware partition.

3 Proposed Architecture

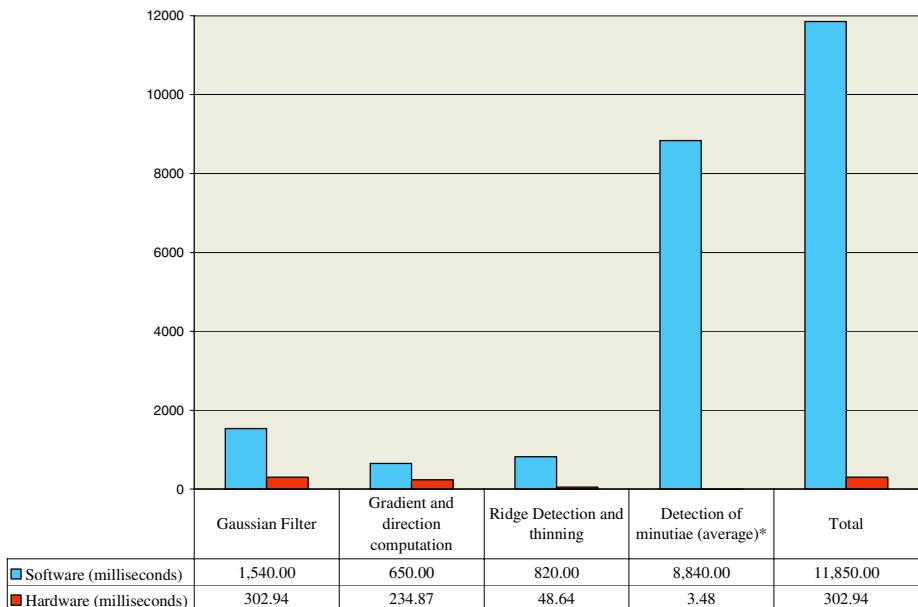
The block diagram presented in Figure 4 corresponds to the hardware partition of the system. The system is organized as a pipeline, to increase the final throughput. The tasks executed in hardware, by the FPGA device are the Gaussian filter, the gradient and direction computation, the ridge detection and thinning, and the minutiae detection.

4 Hardware x Software Results

In Table 1 are presents software and hardware performance for complete fingerprint image (256 x 256 pixels). Software processing was made by PentiumII 233 MHz and hardware processing was made using device FPGA EPF10K50EQC208-1 from FLEX10KE family. The performance of this device was 27.65Mhz, that means 35.9ns for each clock period.

Conclusion and Future Works

This paper present a techniques study used in systems of fingerprints processing. It was also presented a proposal that can be implemented in configurable devices (FPGA). One of the main advantages of this algorithm of minutiae location in relation to those presented in the section State of the Art, is that the functions are accomplished without use complex calculations, that is to say, that allows that this algorithm is implemented easily in hardware. For future work that can be made, is implementation of the classification method of the fingerprints. Like this being, at the end of the whole processing a complete system will be implemented in a device FPGA (System-on-to-chip - SOC).

Table 1. Performance between Software x Hardware

* Due scale range, is not possible to see the hardware time in the chart

Acknowledgement

The author Rolf Fredi Molz would like to gratefully acknowledge the support of FAPERGS project number 01/0168.5-Brazil

APPLES: A Full Gate-Timing FPGA-Based Hardware Simulator

Damian Dalton, Vivian Bessler, Jeffery Griffiths, Andrew McCarthy, Abhay Vadher,
Rory O’Kane, Rob Quigley, and Declan O’Connor

Neosera Systems Ltd, Nova Centre, Belfield, Dublin 4, Ireland
applesinfo@neosera.com
<http://www.neosera.com>

Abstract. Verification of large VLSI digital circuits is primarily accomplished through simulation. In general, there is a trade-off between speed of processing and accuracy. Software simulation tools can be very accurate but are very slow compared to logic accelerators and emulation systems. These latter systems, many FPGA based, while two to three orders of magnitude faster than software, deliver inferior timing analysis, in the latter case and cycle-based simulation it is merely equivalent to functional simulation. **APPLES (Associative Parallel Processor for Logic Event-driven Simulation)** is the first Full Gate-timing Logic Hardware Simulator, implemented in Xilinx Virtex-II technology. APPLES is a true simulator, delivering timing analysis with the accuracy of a software simulator, but has the distinction that processing is executed entirely in hardware devoid of any machine code. This has the potential to permit APPLES to be one to two orders of magnitude faster than equivalent software systems.

1 Introduction

In the testing and verification of digital circuits Logic Simulation plays a pivotal position, occupying an area where speed of computation is as an important consideration as the accuracy of the results. Parallel processing has been investigated extensively as a means to accelerate computational speed, but has had limited success. Compiled code and Event-driven simulation [1],[2],[3],[4],[5] are the two strategies from the sequential environment that have been employed in parallel logic simulation. Particularly important are synchronous event-driven MIMD. To obtain optimal performance, consideration must be given to Global synchronisation between processors. Unfortunately some of these tasks contribute significantly to the Communication overhead. Soule and Blank [6] and Mueller-Thuns et al [7] have studied these systems and under optimal conditions the best speedup figures were between 3 and 5 on an 8-processor iPSC-Hypercube. Fundamental communication and synchronisation issues in parallel logic simulation can be found in [8],[9],[10],[11],[12] and [13]. State of the art accelerators can be found at various commercial websites [14].

2 The APPLES Architecture

In APPLES (Associative Parallel Processor for Logic Event-driven Simulation), a succession of signal values that have appeared on a particular wire over a period of time are stored in a specific word in an Associative memory in a time ordered sequence.

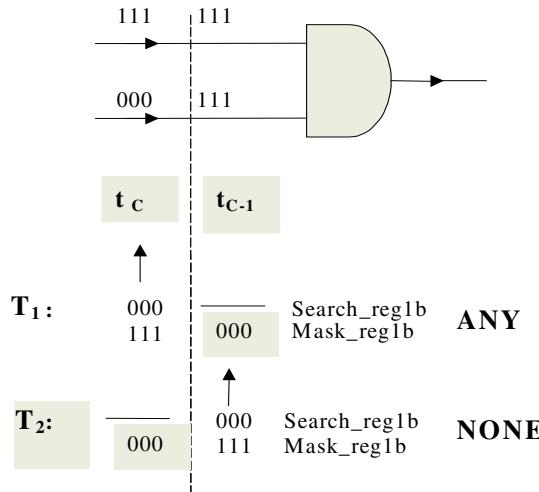


Fig. 1. The APPLES Algorithm for a Unit Delay AND Gate Transitioning to Logic 0

This associative memory structure permits gate evaluation to be performed through a number of parallel test patterns. Therefore, all gates of a particular type can be evaluated in the time it takes to apply the test patterns. For example, as shown in Fig. 1., if a unit-delay AND gate output is going to transition from logic 1 to logic 0, it is necessary to know the signal values at the current time t_c and the previous time interval t_{c-1} . Assume we are using 8-state logic, where 3 bits are required for each signal value. Two simple bit pattern tests will suffice. If ANY current input value is logic 0 (Test T1) and NONE of the previous input values are logic 0 (Test T2), then the output will change to logic 0. Different delay models are constituted by alternative test patterns.

Active gates are identified as gates that have passed all necessary tests. When all tests for the current time period are concluded the fan-out gate inputs affected by these active gates are updated. All signal values are time incremented by shifting the old and new input values into the least time position in the words of the associative array. A more detailed description of the APPLES processor can be found in [15],[16].

3 Benchmark Results

To evaluate the performance of the APPLES prototype, a cycle accurate Verilog version was designed and several ISCAS-85 benchmarks simulated. The gate count of these benchmarks ranged from 622 gates (C880) to 4392gates (C7552). Furthermore, it was assumed that all circuit data was accommodatable by the APPLES' arrays and no memory caching system was necessary. The number of cycles required to evaluate each active gate including all overhead tasks ranged from 3.6 to 5.2 cycles, where a cycle is defined as a register transfer to memory.

Initial implementation of the APPLES processor to accommodate the ISCAS benchmarks and designs up to 150K gates was realised on a Xilinx V1000BG560 (0.22 micron technology) FPGA with a 16Mbyte memory system. This design had a simple paging system, which simply brought sequentially from memory all pages to be processed at every time interval. This APPLES processor implementation running at 10MHz, simulated a circuit containing approximately 140K gates running at the same processing rate as Modelsim running on a 1.2 GHz Pentium processor. The number of clock cycles required for the Modelsim software simulator to compute each active gate was found to be in the range 1,500 to 2,100.

The V1000 FPGA implementation imposes a limited on the APPLES processor clock frequency at 21 Mhz. Transferring the same design to a Virtex-II XC2V1000 raises the clock ceiling to 56Mhz. Additional APPLES modifications such as the introduction of a cache memory system and transferring the design to the Virtex-II XC2V8000 will move this limit towards 100MHz.

4 Integrating the APPLES Processor into Existing CAD Tools

To be acceptable in existing design flows the APPLES system has been designed so that it integrates transparently. Physically, APPLES is on a board which connects to the PCI-bus of a PC. APPLES extracts its netlist information through the standard Verilog PLI interface.

During simulation runs, APPLES transmits information via the PLI interface to the host Verilog simulator. This structure enables any existing netlist Verilog files to be excuted on APPLES and the output to be displayed by the GUI of the Verilog simulator. Only one line of modification is required in a standard netlist file to execute it on APPLES. Essentially, the host Verilog simulator acts as a front-end.

Analysis of APPLES system performance with circuits having gate counts of 1 million gates or more indicates that there are many issues and factors that contribute to the overall processing rate. Important contributors to simulation speed are similar to conventional software simulators and emulators, composition of testbench, idiosyncrasies of the circuit being simulated and memory access rates. Nevertheless, initial benchmarking of large, million gate circuits, have mantained APPLES speed advantage. Furthermore, through the PLI interface, access is made to the Verilog testbench stimulating the circuit and other behavioural/RTL modules in the circuit.

5 Conclusion

The APPLES design indicates that the speed and capacity of current FPGA technology can be considered as a target structure not merely for prototypes but also for the eventual implementation technology of the processor it-self. Decomposing standard algorithms partially or fully into hardware may be sensible economically and in terms of performance, but greater benefits may be derived when radically different approaches are considered as exemplified by the APPLES processor.

References

1. MacMillan et al: An Industrial View of Electronic Design Automation: IEEE Trans CAD of ICs and Systems, Vol 19, No 12, Dec (2000) 1428-1449
2. Darringer et al: EDA in IBM: Past, Present and Future: IEEE Trans CAD of ICs and Systems, Vol 19, No 12, Dec (2000) 1476-1498
3. Breuer et al: Fundamental CAD Algorithms: IEEE Trans CAD of ICs and Systems, Vol 19, No 12, Dec (2000) 1449-1476
4. Dunn: IBM's Engineering Design System Support for VLSI Design and Verification, IEEE Design and Test of Computers, Feb (1984) 30-40
5. Agrawal et al: Logic Simulation and Parallel Processing, IEEE Proc Intl Conf on CAD (ICCAD) 1990.
6. Soule et al: Parallel Logic Simulation on General Purpose Machines, IEEE Proc Design Automation Conf, June (1988) 166-171
7. Mueller-Thuns et al: Benchmarking Parallel Processing Platforms: An Application Perspective, IEEE Trans on Parallel and Distributive Systems, Vol 4, No 8, Aug(1998)
8. Chandy, Misra: Asynchronous Distributed Simulation via Sequence of Parallel Computations. Comm ACM 24(ii), April (1981)
9. Bryant: Simulation of Packet Communications Architecture Computer Systems. Tech Rept MIT-LCS-TR-188. MIT Cambridge, USA (!977)
10. Briner: Parallel Mixed Level Simulation of Digital Circuits Virtual Time. PH.D Thesis, Dept of Elec Eng, Duke University, (1990)
11. Jefferson: Virtual Time. ACM Trans Programming Languages Systems, July (1985), 404-425
12. Soule, Gupta: Characterisation of Parallelism and Deadlocks in Distributed Digital Logic Simulation, Proc 26th Design Automation Conf, June (1989) 81-86
13. Ghosh, Lu: An Asynchronous Distributed Approach for the Simulation of Behavior-level Models on Parallel Processors, IEEE Trans on Parallel and Distributed Systems, Vol 6, No 6, June(1995)
14. www.cadence.com, www.memtor_graphics.com, www.aldec.com, www.aptix.com, www.axis.com , www.tharas.com, www.eve.com
15. Dalton: The Speedup Performance of an Associative Memory Based Logic Simulator, Proc 5th Intl Conf on Parallel Computation Technologies (PaCT-99), St Petersburg, Russia, LNCS Springer-Verlag, Sept(1999)
16. Dalton: Avoiding Conventional Overheads in Parallel Logic Simulation: A New Architecture, ACM/IEEE Proc Intl Conf on High Performance Computing, Calcutta, India, Dec(1999)

Designing, Scheduling, and Allocating Flexible Arithmetic Components

Vinu Vijay Kumar and John Lach

Department of Electrical and Computer Engineering
University of Virginia
351 McCormick Road, P.O. Box 400743
Charlottesville, VA 22904 USA
{vv6v,jlach}@virginia.edu

Abstract. This paper introduces new scheduling and allocation algorithms for designing with hybrid arithmetic component libraries composed of both operation-specific components and flexible components capable of executing multiple operations. The flexible components are implemented primarily in fixed logic with only small amounts of application-specific reconfigurability, which provides the flexibility needed without the negative area and performance penalties commonly associated with general-purpose reconfigurable arrays. Results obtained with hybrid library scheduling and allocation on a variety of digital signal processing (DSP) filters reveal that significant area savings are achieved.

1 Introduction

The optimal schedule and allocation of components during high-level synthesis are hardware dependent, requiring algorithms to be altered based on the target component library. This paper introduces algorithms for scheduling operations and allocating components based on a novel hybrid arithmetic library composed of both fixed-logic components and flexible components capable of performing multiple operations. Applications implemented with such a hybrid library can reap significant area benefits.

However, hardware flexibility must not be gained at the expense of performance and area, as is the case with general-purpose reconfigurable fabrics such as field programmable gate-arrays (FPGAs). This paper introduces a new technique for designing area- and delay-efficient flexible components. *Small-scale reconfigurability* minimizes area and delay penalties by inserting into fixed-logic only the amount of reconfigurable logic and interconnect required to achieve the desired component flexibility. Therefore, arithmetic components designed with this technique have the flexibility to perform multiple operations but are ASIC-like in their efficiency. This enables the area gains provided by the hybrid component library scheduling and allocation algorithms to be maintained.

2 Small-Scale Reconfigurability

A flexible component could simply be the multiplexed set of fully implemented individual components. To receive any area benefit, however, the flexible component must be smaller than the total size of all individually implemented components. Partitioning operations across time instead of space, each operation can be implemented in the same physical space, invoking the proper component configuration at the necessary time. Chiricescu describes an implementation of such a ‘morphable’ unit utilizing common sub functions to implement addition and multiplication [1].

This approach can be extended using reconfigurable logic to replace fixed-logic gates in the circuit. This would even enable circuits without common sub functions to be shared spatially. Taken to an extreme, the entire circuit would be implemented much like an embedded FPGA core, with the associated area and performance penalties of general-purpose reconfigurable fabric. The key to improving efficiency is to limit the added reconfigurable logic and interconnect. The highly optimized nature of arithmetic components requires custom design of flexible components, but logic synthesis techniques such as Boolean matching [2] can be used to find the minimum distance between the set of functions to be implemented. In addition, design decisions must be made as to how to add flexibility, such as multiplexing a set of fixed-logic gates or using a single lookup table (LUT) capable of implementing each gate.

We have designed a bit-sliced flexible component capable of executing 4-bit fixedpoint addition, multiplication, and comparison. The design of the adder/multiplier part is similar to that in [1]. The base arithmetic structures used are a carry lookahead adder and a parallel array multiplier, with sections of the multiplier partial product summation network utilized for addition when in adder mode. Using small-scale reconfigurability, flexibility is added to the logic cones of the output bit-lines to implement the comparator operation.

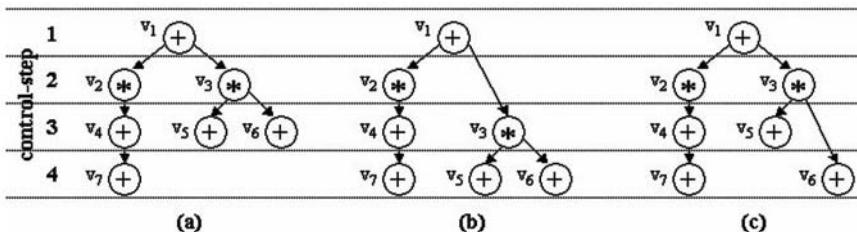


Fig. 1. Scheduling example: (a) Original DFG, (b) Conventional force directed list schedule, (c) Hybrid force directed list schedule

3 Hybrid Library Scheduling and Allocation

Given that optimality depends on the target component library, high-level scheduling and allocation algorithms have been modified with the introduction of flexible arithmetic components. The well-known force directed list scheduling algorithm is used, with a modified force calculation and node selection criteria. For a detailed description of the conventional algorithm, refer to [3, 4]. To demonstrate the modifications to the algorithm for hybrid scheduling, consider the example data flow graph (DFG) in Fig. 1a. The

conventional algorithm produces the schedule shown in Fig. 1b, with the intent to reduce operator concurrency. Hence, nodes v2 and v3 will not be scheduled in the same time step, as they are both multiply operations with high area costs. This schedule requires 3 adders and 1 multiplier. The hybrid scheduling algorithm produces the schedule shown in Fig. 1c. The force calculations are modified such that total number of operations per cycle is minimized, rather than individual operator concurrency. This schedule requires 2 adders and 2 multipliers if only fixed-logic components are used, leading to higher area cost than the earlier conventional schedule. However, if we allocate on this schedule using a hybrid library, only 2 flexible components are needed, which has a lower area cost than the previous solutions.

As with the conventional algorithm, the number and type of components is first initialized based on the nodes along the critical path. Operations are scheduled from the ready list of nodes, using modified force as the metric for scheduling efficiency. An additional parameter, slack, which is a measure of urgency of a node, is used to guide the selection of candidate nodes. If no resources are available to schedule a node with zero slack, a component is added and the scheduling process is iterated. When choosing between equal slack nodes, the selection is made such that individual operation concurrency is equalized with the previous step. This is to reduce the probability that a flexible component would need reconfiguration when used in that particular step. Scheduling is continued until either all of the nodes are scheduled or the time budget is exceeded, in which case the process iterates with the addition of a component.

Allocation is trivial for a case with only two operation types. The two types can be considered individual sets, with their intersection representing the nodes to be implemented by the flexible component. Hence, the formula from set theory for calculating the cardinality of the intersection of two sets given their individual cardinalities and that of their union is directly applicable here. For cases with more than two disparate operations, more complicated techniques (e.g. linear programming) are needed to derive the optimum allocation.

4 Results and Conclusion

The area efficiency of small-scale reconfigurability is revealed by the component implementations described in Section 2. Normalized to the area of a comparator X: Adder - 1.44X, Multiplier - 4.5X, Adder/Multiplier Limited Flexible Unit (LFU) - 4.81X, and Comparator/Adder/Multiplier Full Flexible Unit (FFU) - 5.31X. In comparison, an equivalent multiplier on an FPGA has an area of approximately 126X.

Normalized to the delay of the multiplier (the slowest fixed component) Y, the delays of the LFU and FFU are 1.27Y and 1.36Y, respectively, and the length of each control-step must be increased accordingly. These increases (which are significantly less than that of FPGAs) must be traded off against the area savings obtained. The reconfiguration time is not considered as only a small number of configuration bits need to be set and reconfiguration will not happen every control-step. In addition, the scheduling algorithm minimizes the flexible component reconfiguration frequency.

Using a library of these components, DSP examples from the high-level synthesis literature [5,6,7] have been scheduled and allocated with our modified algorithms. The examples were scheduled with the smallest number of control-steps, and the area results (normalized to a single comparator) are revealed in Table 1. FDLS AL is the component allocation via traditional force directed list scheduling and allocation using fixed

components. FDLS HAL is the allocation of hybrid library components on the exact same schedule. HFDLS HAL uses our hybrid scheduling and allocation algorithms. The last column shows the area savings obtained, which are maximized by the combination of the hybrid scheduling and allocation algorithms and the efficient flexible component implementations provided by small-scale reconfigurability.

Table 1. Area savings of hybrid scheduling and allocation with flexible components

CIRCUIT	FDLS AL		FDLS HAL		HFDLS HAL		FDLS AL - HFDLS HAL		
	#fixed	area	#fixed	#flex	area	#fixed	#flex	area	
ELLIP	5	13.3	3	1 LFU	12.2	1	2 LFU	10.6	20.27%
EDGE	3	10.4	1	1 LFU	9.31	1	1 LFU	9.31	10.82%
ARFILT	6	20.9	6	0	20.9	1	3 LFU	18.9	9.34%
FIRFILT	4	11.9	4	0	11.9	2	1 LFU	10.8	9.51%
DIFFEQ	5	12.9	2	1 FFU	11.3	2	1 FFU	11.3	12.66%

Acknowledgements

This work is supported in part by the National Science Foundation under grant No.CCR-0105626.

References

- Chiricescu, S., et al.: Morphable Multipliers. In: Glesner, M., et al. (eds.): FPL 2002, Lecture Notes in Computer Science, LNCS 2438. Springer-Verlag, Berlin Heidelberg (2002) 647–656
- Savoj, H., et al.: Boolean Matching in Logic Synthesis. Proceedings of the European Design Automation Conference (1992) 168-174
- Paulin, P.G. and Knight, J.P.: Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 8, no. 6 (1989) 661-679
- Verhaegh, W.F.J., et al.: Improved Force-Directed Scheduling in High-Throughput Digital Signal Processing. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 8 (1995) 945–960
- Haralick, R.M., and Shapiro, L.G.: Computer and Vision. Addison-Wesley, Reading, MA (1992) 6. Högstedt, K., and Orailoglu, A.: Integrating Binding Constraints in the Synthesis of Area-Efficient Self-Recovering Microarchitectures. Proceedings of the International Conference on Computer Design (1994) 331-334
- Karri, R., and Orailoglu, A.: High-Level Synthesis of Fault-Secure Microarchitectures. Proceedings of the Design Automation Conference (1993) 429-433

UNSHADES-1: An Advanced Tool for In-System Run-Time Hardware Debugging

M.A. Aguirre, J.N. Tombs, A. Torralba, and L.G. Franquelo

Electronic Engineering Dpt.
Escuela Superior de Ingenieros. University of Sevilla
c/Camino de los Descubrimientos s/n 41092 Sevilla (SPAIN)
{aguirre,jon,torralba,leopoldo}@gte.esi.us.es

Abstract. The aim of a Rapid Prototyping System for electronic circuit design is to obtain a physical model as similar as possible to the final system as the hosting technology can allow. Large digital integrated circuits are substituted by complex and advanced Field Programmable Gate Arrays (FPGA's) which emulate the whole circuit functionality. These devices can provide more information than the pure circuit emulation itself, they provide a special scheme to access the device configuration and execution time information of the design state registers. This paper describes the UNSHADES-1 system and is focused on the set of software tools that provide easy management and access to this execution time information.

1 Hardware Debugging

Software debugging normally provides a set of tools that can help the programmer to look into the running code and inspect the contents of the variables during the execution (on a step-by-step basis). All this information is linked to the high level source code (such as C, C++, BASIC, ...). Software debuggers provide a means of selecting breakpoints where the execution will be stopped and the code can be inspected. When a breakpoint is reached, then the software can be run step by step, run to the next breakpoint or run until the conclusion of the execution.

In our hardware debugging concept, different than that found in [1], but not opposed to, we try to reproduce this software debugging model, providing a closer link between the designer and the running code. Other approaches are Altera Signal Tap, or Xilinx Internal Logic Analyzer, that are intrusive methods for registering pre-selected internal signals. In our approach, the main objective consists in obtaining a hardware scenario that can interchange the information between the emulator system and the man-machine interface in a comprehensive way, but restricted to one snapshot. Once the hardware problem is solved, the development of a set of tools to provide control and manage hardware data is our task.

This paper presents the platform UNSHADES-1 that stands for University of Sevilla HArdware DEbugging System. UNSHADES consists of a hardware platform based on a Xilinx Virtex device and a set of software tools running on a personal computer. Other Tools for inner inspection that compete with UNSHADES-1 are Jbits, LabView, JHDL, Xilinx ILA, but none is able of forcing a value to a single register.

2 UNSHADES-1 Hardware. Highlights of the Emulation System

The UNSHADES-1 is described in [2] hardware consists of a board with two FPGAs: The emulation system (the VIRTEX FPGA) called S-FPGA and a smaller FPGA with a fixed configuration, called the C-FPGA. The C-FPGA performs the transfer tasks (protocol adaptation and others) with the host PC and certain control functions. One of the tasks of the C-FPGA is related to the control of some general purpose IO lines that can be used for to provide, if needed, extra control over the emulation system. These IO lines are useful for some particular tasks related to the debugging system.

3 UNSHADES-1 Software. The Run-Time Capture and Scheme System

From software point of view UNSHADES is fully integrated into the Xilinx standard design flow. Two files are needed for integration of UNSHADES into any Xilinx design flow: the bitstream file and the bit allocation file. The first is necessary because the initial configuration process is controlled by the UNSHADES software and the second provides the map that about the location of every register placed across the S-FPGA core and its design level name. Together these two file provide a link between the physical information and their names given during the high level design stages, in other words, it provides a method for closing the loop back. The execution time information can be displayed using comprehensive names. The primary task of the software consists of reading the low-level bitstream information and associating it to busses and registers. The UNSHADES software provides a graphic user interface (GUI) that presents a scheme of the registers and bits, associated with their last captured value.

3.1 External Snapshot

The simplest debugging tool is to take snapshots using an external line (figure 1, b). The PC sends a signal that requests that the Virtex launch the capture mechanism. An external IO line is used to initiate the capture and the S-FPGA continues to operate at all times at design speed. After capture, the UNSHADES software uploads the information to the PC and presents it in the GUI. A natural extension of this tool is to launch the capture macro every certain amount that is programmed. The designer will have information about the evolution of selected signals. Also the information can be recorded into a file to provide display in a waveform viewer. The aim of this tool is to

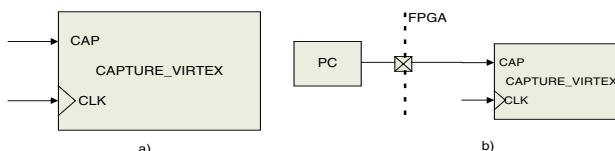


Fig. 1. Capture macro and external snapshot

observe the evolution of the system that has ‘slow variables’ like state variables in power system controllers or to capture the system state once a design under emulation has begun to operate incorrectly.

3.2 Single Clock Cycle Evolution

The system evolution can be frozen when a run-time event is satisfied. Run-time conditions have to be foreseen at design time. They can refer to register contents that have enough interest that it should be studied when it occurs. Usually they’re comparisons with bus values or bits. More complex conditions can be introduced that are combined with time conditions.

All Virtex flip-flops that belong to the design under debug can be controlled by means of their ‘clock enable’ input. Using this pin the system evolution can be totally or partially frozen without any risk of malfunctioning due to glitches or clock skew problems. If the ‘clock enable’ input is only asserted during a single clock cycle, the system will perform a ‘single step’ evolution. In the UNSHADES system an external IO line, called ‘debug clock’ is used to allow the software perform this single stepping of the S-FPGA. For this option, a small circuit that detects changes on this control line must be included in the S-FPGA design. We use 160 system gates for this circuit. Using a second line, called the ‘resume’ line, normal execution can be re-launched until the next run-time condition.

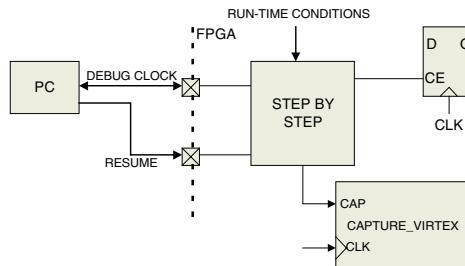


Fig. 2. Single step scheme

After each rising edge of the ‘debug clock’ line the flip-flop ‘clock enable’ input is asserted during a single clock cycle, and a ‘capture’ is launched. After this, the software uploads the information to be represented in the GUI. UNSHADES can launch a sequence of a configurable number of steps and record the information to be displayed in a classical waveform viewer.

3.3 Flip-Flop Contents Modification

UNSHADES software can change the contents of the S-FPGA registers during run-time. This task has never been reported before in the literature for the Virtex technologies. Previous work and vendor information affirm that changes cannot be per-

formed on selected bits. To make the change correctly the process can only be performed when the system is in a frozen state, this is because a sequence of steps is required and no single access is sufficient. The main difficulties for inducing a new value in a flip-flop is determined by the Configurable Logic Block (CLB) architecture which doesn't permit direct modification of the design Flip-Flops. An original read-modify-write-unmodify scheme has been developed for this purpose and is able to change the flip-flop state within a modified CLB and then return the CLB back to the original design configuration.

4 UNSHADES-1 Graphic User Interface

A good interface human machine is necessary for a comprehensive interaction between the emulation hardware and the information displayed. During synthesis stage all combinational parts are collapsed into look-up tables and cannot be rebuilt into their original schematics. Only registers can display their information.

5 Conclusions

A new technique for hardware debugging has been presented exploiting features of a commercial field programmable FPGA. This new debugging environment brings software debugger techniques into the field of hardware debugging. The FPGA devices chosen to emulate the complete digital design can provide observability, controllability and insertion of desired states into internal registers. UNSHADES tools have been presented as a set of software tools in spite of the fact that a custom hardware system has been designed to match with the software. The software has been developed in a hardware independent way and can be adapted to other commercial hardware platforms with few requirements.

References

- [1] Hutchings B., Nelson B. and Wirthlin M.J.. "Designing and Debugging Custom Computing Applications". IEEE Design & Test of Computers. Jan-March 2000. pp 20-28.
- [2] M.A. Aguirre, J. Tombs, A. Torralba and L.G. Franquelo. "Improving the Design Process of VLSI Circuits by Means of a Hardware Debugging System: UNSHADES-1 Framework". To be published in Proceedings of the 28th IEEE Industrial Electronics Conference. IECON'02. Sevilla November 2002.
- [3] Xilinx Data book. 2002.
- [4] Xilinx XC6200 series datasheet.
- [5] J. Faura, C. Horton, Bernd Krah, J. Cabestany, M.A. Aguirre and J.M. Insenser. "A New Field Programmable System On-a-Chip for Mixed Signal Integration" European Design & Test Conference 1997.
- [6] Xilinx application notes number xapp138 and xapp151.

Author Index

- Aa, T.V. 230
Agís, R. 1048
Aguirre, M.A. 1170
Akella, V. 520
Akil, M. 934
Almeida, C.B. 818
Almeida, M. 976
Almeida, P. 976
Almenar, V. 988
Almenar-Terre, V. 31
Amano, H. 161, 171, 766
Amaral, J.N. 648
Angarita, F. 988
Anjo, K. 161, 171
Antoš, D. 964
Arias, J. 1087
Arias-Estrada, M. 1008, 1053, 1103
Arostegui, J.-M.M. 681
Arslan, T. 1139
Astarloa, A. 497
Augusto, J.S. 818

Ballagh, J. 1099
Ballester, F. 533
Barat, F. 230
Barreiros, J. 141
Bartic, A. 595
Barton, R.J. 365
Bellows, P. 869
Benkrid, A. 553, 1012
Benkrid, K. 553, 1012
Benoit, P. 722
Bertels, K. 900
Berube, P. 648
Bessler, V. 1162
Beuchat, J.-L. 101
Bidarte, U. 497
Blodget, B. 565
Bobda, C. 272, 1123
Bober, M. 543
Boluda, J.A. 458
Bonato, V. 1158
Bossuet, L. 921
Brakensiek, J. 1111
Brebner, G. 385

Burgess, N. 808
Caffrey, M.P. 948
Çakır, M. 627
Callaghan, M. 1079
Cambon, G. 722
Canet, M.J. 988
Cañas, A. 1048
Cardells-Tormo, F. 31, 988
Carline, D. 1115
Carter, N.P. 1
Catthoor, F. 585
Cerdá-Boluda, J. 776, 1016
Charot, F. 282
Cheung, P.Y.K. 355, 396, 606, 796, 1071
Cheung, T. 1000
Chodowiec, P. 869
Choi, S. 507
Chu, M. 11
Clark, C.R. 956
Cochran, J. 996
Cocorullo, G. 661
Cohoon, J.P. 131
Colom-Palero, R. 1016
Colsell, S. 944
Compton, K. 121
Constantinides, G.A. 606
Corporaal, H. 230
Corsonello, P. 661
Costa, E. 141
Coulton, P. 1115
Court, T. Van 365
Crawford, J. 51
Cret, O. 1067
Crookes, D. 553, 1012
Cuadrado, C. 1087
Curran, P. 11

Dalton, D. 1162
Daly, A. 786
Danne, K. 272, 1123
Deconinck, G. 230
Delvai, M. 733
Demigny, D. 722
Denning, D. 980
Deprettere, E. 911

- Devlin, M. 980
Dharmapurikar, S. 859
Dias, T. 745
Díaz-Pérez, A. 303
Diniz, P.C. 313
Drutarovský, M. 1075
Dulay, N. 890
Dunham, M.E. 948
Durbano, J.P. 1131

Ebeling, C. 21
Edwards, R. 944
Eisenmann, U. 733
Ejlali, A. 849
El-Ghazawi, T. 204
Elmenreich, W. 733
Entrena, L. 220
Enzler, R. 151
Erdogan, O. 11
Estrada, R. 1036

Feldmann, R. 478
Ferrão, M.F. 1158
Ferrari, A.B. 468
Ferrera, S. 1
Fischer, V. 1075
Fisher, C. 21
Flidr, J. 869
Franquelo, L.G. 1170
Fučík, O. 964
Fujimoto, K. 437
Fujiwara, T. 437
Fukushima, T. 766
Furtado, J.C. 1158

Gadea-Girones, R. 776, 1016
Gaj, K. 204, 869
García, A. 1135
García, I. 1036
García Ortiz, A. 1111
Garzón, E.M. 1036
Gharai, L. 869
Glesner, M. 1111
Goda, B. 11
Gogniat, G. 921
Gomez, F.J. 194
Gómez Pulido, J.A. 1091
Gonzalez, I. 194
Graham, P.S. 948
Grandpierre, T. 934

Griffiths, J. 1162
Grimpe, E. 627
Guo, J.-R. 11

Haglund, P. 1040
Harkin, J. 1079
Harold, N. 980
Haubelt, C. 478
Hauck, S. 121
Heikaus, R. 11
Henz, M.J. 488
Herbordt, M.C. 365
Herrero, V. 776
Heřmánek, A. 1095
Higashino, T. 1145
Höller, R. 960
Hollstein, T. 1111
Hong, C.P. 670
Huang, R. 1000
Humphrey, J.R. 1131
Hwang, J. 1099

Iachino, M.A. 661
Ichikawa, S. 1024
Imaña, J.L. 1127
Irvine, J. 980

James-Roxby, P. 385, 565
Jayapala, M. 230
Jiang, J. 1057
Jimenez, J. 497
Jouraku, A. 161

Kadlec, J. 1095
Kalte, H. 272
Kamakoti, V. 1044
Kaouane, L. 934
Kapur, D. 996
Keller, E. 385, 565
Kennedy, I. 262
Kerins, T. 786
Khawam, S. 1139
Kienhuis, B. 911
Kim, C.H. 670
Kim, J.J. 670
Kitani, T. 1145
Kitaoka, T. 171
Kobori, T. 755
Koch, A. 1083
Kok, T. 1000

- Koorapaty, A. 426
 Kraft, R.P. 11
 Krasniewski, A. 828
 Krishnan, R. 240
 Ku, H. 968
 Kühl, M. 984
 Kumar, L.K. 1044
 Kumar, V.V. 1166
 Kummert, A. 1004
 Kuzmanov, G. 81
 Kwon, S. 670
 Kwong, K.H. 375
 Lach, J. 1166
 Lai, K. 638
 Laskowski, E. 71
 Lauwereins, R. 61, 230, 595
 Lázaro, J. 1087
 Lee, B. 808
 Lee, D.-U. 796
 Lee, G. 252
 Lee, K.H. 375
 Lee, P. 543
 Lee, T.K. 890
 Legat, J.-D. 181
 Leong, P.H.W. 375
 Li, S. 1153
 Liao, J. 334
 Líčko, M. 984
 Lim, D. 859
 Linarth, A. 1123
 Liu, H. 21
 Lloris, A. 1135
 Lockwood, J.W. 859, 968
 López, O. 1036
 Lopez-Buedo, S. 194
 Lorenz, M.G. 220
 Lu, S.-L. 638
 Ludewig, R. 1111
 Luk, W. 324, 396, 606, 796, 890, 1040,
 1057, 1071
 Lupu, E. 890
 Ma, H. 1099
 MacGregor, M. 648
 Maguire, L. 1079
 Man, H. De 61
 Mandado, E. 1107
 Manohar, R. 345
 Marescaux, T. 595
 Marín-Roig, J. 988
 Marnane, W. 786
 Martín, J.L. 1087
 Martina, M. 712, 1028
 Martinez, A. 691
 Martinez, J. 194
 Martínez de Alegría, I. 497
 Maruyama, T. 437, 448, 755
 Maya-Rueda, S. 1103
 Mazzeo, A. 292
 Mazzocca, N. 292
 McAllister, B. 1149
 McCanny, J.V. 111
 McCarthy, A. 1162
 McCulloch, S.T. 131
 McDermott, T. 51
 McDonald, J.F. 11
 McGinnity, T.M. 1079
 McLoone, M. 111
 McMillan, S. 565
 Mei, B. 61
 Mencer, O. 1040
 Mengibar, L. 220
 Meyer-Bäse, U. 1135
 Michalski, A. 204
 Mignolet, J.-Y. 595
 Milne, B. 1099
 Milne, G. 252
 Miremadi, S.G. 849
 Mitra, T. 334
 Miyajima, Y. 448
 Moffat, W. 595
 Mohanty, S. 41
 Molino, A. 712, 1028
 Molz, R.F. 1158
 Monien, B. 478
 Moraes, F.G. 1158
 Mora, F. 533
 Morillas, C.A. 691
 Moscola, J. 859
 Moure, M.J. 1107
 Mozos, D. 585
 Mupid, A.J. 1044
 Murgan, T. 1111
 Naka, I. 1145
 Nakata, A. 1145
 Nebel, W. 627
 Neely, C. 859
 Nestor, J.A. 992

- Netin, A. 1067
 Neto, H.C.C. 818
 Nicola, M. 1028
 Novotný, J. 964
- O'Connor, D. 1162
 Oelkrug, B. 1111
 O'Kane, R. 1162
 Oldeneel tot Oldenzeel, L. van 701
 Oliver, J. 520
 Ortega, J. 1048
 Ortigosa, E.M. 1048
 Ortigosa, P.M. 1036, 1048
 Ortiz, F.E. 1131
 Osana, Y. 766
- Paar, C. 91
 Panainte, E.M. 900
 Pantelimon, A. 839
 Pardo, F. 458
 Park, J. 313
 Parreira, A. 839
 Paschalakis, S. 543
 Payá, G. 533
 Peiró, M.M. 533
 Pelayo, F. 691
 Pérez, A. 988
 Perkins, C. 869
 Perri, S. 661
 Peters, C. 1079
 Petrov, M. 1111
 Philippe, J.-L. 921
 Pileggi, L. 426
 Pineda de Gyvez, J. 240
 Platzner, M. 151, 575
 Plessl, C. 151
 Pnevmatikatos, D. 880
 Pohl, Z. 1095
 Popovici, E. 786
 Prasanna, V.K. 41, 507
 Prather, D.W. 1131
 Pusztai, K. 1067
- Quaglio, F. 712
 Quigley, R. 1162
 Quintáns, C. 1107
 Quisquater, J.-J. 181, 701
- Ramani, A.S. 1044
 Ramírez, J. 1135
- Ramirez-Agundis, A. 1016
 Reorda, M.S. 616
 Resano, J. 585
 Robert, M. 722
 Rodríguez-Henríquez, F. 303
 Rodríguez Lozano, D. 1091
 Roma, N. 745
 Roman, D. 1067
 Romero, S. 691
 Ros, E. 1048
 Rouvroy, G. 181
 Royal, A. 355
 Rueckert, D. 1057
 Ryan, P. 51
- Saggese, G.P. 292
 Samyde, D. 701
 Sanchez, E. 681
 Sánchez, J.M. 1127
 Sánchez Pérez, J.M. 1091
 Sánchez-Reillo, R. 220
 Sansaloni, T. 988
 Santos, M.B. 839
 Saqib, N.A. 303
 Sassatelli, G. 722
 Sawitzki, S. 1119
 Schier, J. 984
 Schimmel, D.E. 956
 Schmidt, C. 1083
 Schmit, H. 406, 426
 Schuehler, D.V. 968
 Sebastià, A. 776
 Sedcole, N.P. 606
 Sezer, S. 1149
 Shayee, K.R.S. 313
 Shen, M. 21
 Shirazi, N. 1099
 Šimka, M. 1075
 Singh, V. 1099
 Siozios, K. 1032
 Skliarová, I. 468, 976, 1020
 Sklyarov, V. 976, 1020
 Sloman, M. 890
- Søråsen, O. 1153
 Sorel, Y. 934
 Soudris, D. 1032
 Sourdis, I. 880
 Sousa, J.T. de 839
 Sousa, L. 691, 745
 Souza, L. de 51

- Spallek, R.G. 1119
 Standaert, F.-X. 181, 701
 Stansfield, T. 416
 Stefanov, T. 911
 Stefanović, D. 996
 Steiger, C. 575
 Strollo, A.G.M. 292
 Stroomer, J. 1099
 Styles, H. 324
 Sukhsawas, S. 1012
 Sundararajan, P. 565
 Sutton, P. 1062
 Tai, B. 1040
 Takamoto, Y. 1145
 Takano, S. 952
 Tatas, K. 1032
 Teich, J. 478
 Teifel, J. 345
 Teixeira, J.P. 839
 Tempesti, G. 681
 Thanailakis, A. 1032
 Thoma, Y. 681
 Tichý, M. 984
 Toal, C. 1149
 Tørresen, J. 1153
 Tomás, P. 691
 Tombs, J.N. 1170
 Torralba, A. 1170
 Torres, L. 722
 Torres, V. 988
 Torres-Huitzil, C. 1008
 Tudruj, M. 71
 Turner, R.H. 972
 Vacca, F. 712, 1028
 Vadher, A. 1162
 Valdés, M.D. 1107
 Valls-Coquillat, J. 31, 988
 Vassiliadis, S. 81, 900
 Veendrick, H.J.M. 240
 Velten, J. 1004
 Verkest, D. 61, 585, 595
 Vernalde, S. 61, 585, 595
 Vicedo, F. 988
 Villasenor, J. 796
 Violante, M. 616
 Văcariu, L. 1067
 Wagner, C. 282
 Walder, H. 575
 Wang, S.Z.Q. 488
 Westall, F. 1139
 Wiangtong, T. 396, 1071
 Wollinger, T. 91
 Wong, K. 51
 Wong, W.-F. 334
 Woods, R.F. 972
 Xicotencatl, J.M. 1053
 Xing, G. 21
 Yahya, E. 282
 Yamamoto, H. 1024
 Yamamoto, S. 1024
 Yap, R.H.C. 488
 Yasumoto, K. 1145
 You, C. 11
 Yu, C.W. 375
 Yusuf, S. 890
 Zhou, K. 11
 Zhu, J. 1062
 Zinky, A. 648
 Zipf, P. 1111
 Zissulescu, C. 911
 Zuloaga, A. 497
 Zuver, C. 859
 Zyner, G. 51