

Guy Gogniat · Dragomir Milojevic
Adam Morawiec · Ahmet Erdogan
Editors

Algorithm- Architecture Matching for Signal and Image Processing

Best papers from Design and Architectures for Signal
and Image Processing 2007 & 2008 & 2009

Lecture Notes in Electrical Engineering

Volume 73

For other titles published in this series, go to
www.springer.com/series/7818

Guy Gogniat • Dragomir Milojevic •
Adam Morawiec • Ahmet Erdogan
Editors

Algorithm- Architecture Matching for Signal and Image Processing

Best papers from Design and Architectures
for Signal and Image Processing 2007 & 2008
& 2009

Editors

Guy Gogniat
Lab-STICC-CNRS, UMR 3192, Centre de
Recherche
Université de Bretagne Sud – UEB
BP 92116
56321 Lorient Cedex
France
guy.gogniat@univ-ubs.fr

Dragomir Milojevic
Université libre de Bruxelles
CP 165-56, Av. FD Roosevelt 50
1050 Bruxelles
Belgium
dmilojev@ulb.ac.be

Adam Morawiec
ECSI
Av. de Vignate 2
38610 Gières
France
adam.morawiec@ecsi.org

Ahmet Erdogan
School of Engineering
The University of Edinburgh
Mayfield Road
EH9 3JL Edinburgh
United Kingdom
ahmet.erdogan@ee.ed.ac.uk

ISSN 1876-1100
ISBN 978-90-481-9964-8
DOI 10.1007/978-90-481-9965-5
Springer Dordrecht Heidelberg London New York

e-ISSN 1876-1119
e-ISBN 978-90-481-9965-5

Library of Congress Control Number: 2010938790

© Springer Science+Business Media B.V. 2011

No part of this work may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Publisher, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work.

Cover design: VTEX, Vilnius

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Advances in signal and image processing together with increasing computing power are bringing mobile technology closer to applications in a variety of domains like automotive, health, telecommunication, multimedia, entertainment and many others. The development of these leading applications, involving a large diversity of algorithms (e.g. signal, image, video, 3D, communication, cryptography) is classically divided into three consecutive steps: a theoretical study of the algorithms, a study of the target architecture, and finally the implementation. Such a linear design flow is reaching its limits due to intense pressure on design cycle and strict performance constraints. The approach, called Algorithm-Architecture Matching, aims to leverage design flows with a simultaneous study of both algorithmic and architectural issues, taking into account multiple design constraints, as well as algorithm and architecture optimizations, that couldn't be achieved otherwise if considered separately. Introducing new design methodologies is mandatory when facing the new emerging applications as for example advanced mobile communication or graphics using sub-micron manufacturing technologies or 3D-Integrated Circuits. This diversity forms a driving force for the future evolutions of embedded system designs methodologies.

The main expectations from system designers' point of view are related to methods, tools and architectures supporting application complexity and design cycle reduction. Advanced optimizations are essential to meet design constraints and to enable a wide acceptance of these new technologies.

This book presents a collection of selected contributions addressing different aspects of Algorithm-Architecture Matching approach ranging from sensors to architectures design. The scope of this book reflects the diversity of potential algorithms, including signal, communication, image, video, 3D-Graphics implemented onto various architectures from FPGA to multiprocessor systems. Several synthesis and resource management techniques leveraging design optimizations are also described and applied to numerous algorithms.

The contributions of this book are split into three parts addressing major issues when designing embedded systems. The first part proposes key contributions in the domain of architectures for embedded applications and especially for image and

telecommunication processing. The second part focuses on data acquisition and design techniques for embedded systems. First, an optimized sensor for image acquisition is detailed. Then several multiplication and division operators are described. The end of this part proposes several contributions in the domain of partial and dynamic reconfiguration for signal and image processing. This technology leads to complex design issues which are addressed in this chapter. The third part targets embedded systems design. RTOS for embedded systems and scheduling techniques are first addressed. Finally CAD tools for signal and image processing are detailed. The coverage of this book is large and provides an in-depth analysis of existing techniques and methodologies to design embedded systems targeting image and signal processing.

Guy Gogniat
Dragomir Milojevic
Adam Morawiec
Ahmet Erdogan

Contents

Part 1 Architectures for Embedded Applications

Lossless Multi-Mode Interband Image Compression and Its Hardware Architecture	3
Xiaolin Chen, Nishan Canagarajah, and Jose L. Nunez-Yanez	
Efficient Memory Management for Uniform and Recursive Grid Traversal	27
Tomasz Toczek and Stéphane Mancini	

Mapping a Telecommunication Application on a Multiprocessor System-on-Chip

.	53
Daniela Genius, Etienne Faure, and Nicolas Pouillon	

Part 2 Data Acquisition and Embedded Systems

A Standard 3.5T CMOS Imager Including a Light Adaptive System for Integration Time Optimization	81
Gilles Sicard, Estelle Labonne, and Robin Rolland	

Approximate Multiplication and Division for Arithmetic Data Value Speculation in a RISC Processor

.	95
Daniel R. Kelly, Braden J. Phillips, and Said Al-Sarawi	

RANN: A Reconfigurable Artificial Neural Network Model for Task Scheduling on Reconfigurable System-on-Chip

.	117
Daniel Chillet, Sébastien Pillement, and Olivier Sentieys	

A New Three-Level Strategy for Off-Line Placement of Hardware Tasks on Partially and Dynamically Reconfigurable Hardware	145
Ikbel Belaid, Fabrice Muller, and Maher Benjema	

End-to-End Bitstreams Repository Hierarchy for FPGA Partially Reconfigurable Systems	171
Jérémie Crenne, Pierre Bomel, Guy Gogniat, and Jean-Philippe Diguet	

Part 3 Embedded Systems Design

SystemC Multiprocessor RTOS Model for Services Distribution on MPSoC Platforms	197
Benoît Miramond, Emmanuel Huck, Thomas Lefebvre, and François Verdier	
A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention	217
Pengcheng Mu, Jean-François Nezan, and Mickaël Raulet	
Multiprocessor Scheduling of Dataflow Programs within the Reconfigurable Video Coding Framework	237
Jani Boutellier, Christophe Lucarz, Victor Martin Gomez, Marco Mattavelli, and Olli Silvén	
A High Level Synthesis Flow Using Model Driven Engineering	253
Sébastien Le Beux, Laurent Moss, Philippe Marquet, and Jean-Luc Dekeyser	
Generation of Hardware/Software Systems Based on CAL Dataflow Description	275
Richard Thavot, Romuald Mosqueron, Julien Dubois, and Marco Mattavelli	
Index	293

Contributors

Said Al-Sarawi CHiPTec, Centre for High Performance Integrated Technologies and Systems, The University of Adelaide, Adelaide, Australia,
alsarawi@eleceng.adelaide.edu.au

Ikbel Belaid University of Nice-Sophia Antipolis/LEAT-CNRS, 250 rue Albert Einstein, 06560 Valbonne, France, Ikbel.Belaid@unice.fr

Maher Benjemaa National Engineering School of Sfax/Research Unit ReDCAD, Sfax, Tunisia, Maher.Benjemaa@enis.rnu.tn

Sebastien Le Beux Institut des Nanotechnologies de Lyon, Ecole Centrale de Lyon, 36, Avenue Guy de Collongue, 69134 Ecully Cedex, France,
Sebastien.Le-Beux@ec-lyon.fr

Pierre Bomel LAB-STICC, Université Européenne de Bretagne, Lorient, France,
pierre.bomel@univ-ubs.fr

Jani Boutellier Computer Science and Engineering Laboratory, University of Oulu, Oulu, Finland, jani.boutellier@ee.oulu.fi

Nishan Canagarajah University of Bristol, Bristol, UK,
Nishan.Canagarajah@bristol.ac.uk

Xiaolin Chen University of Bristol, Bristol, UK, Xiaolin.Chen@bristol.ac.uk

Daniel Chillet University of Rennes 1, IRISA/INRIA, BP 80518, 6 rue de Kerampont, F22305 Lannion, France, Daniel.Chillet@irisa.fr

Jérémie Crenne LAB-STICC, Université Européenne de Bretagne, Lorient, France,
jeremie.crenne@univ-ubs.fr

Jean-Luc Dekeyser LIFL and INRIA Lille Nord-Europe, Parc Scientifique de la Haute Borne, Park Plaza, Bât A, 40 avenue Halley, Villeneuve d'Ascq 59650, France, Jean-Luc.Dekeyser@lifl.fr

Jean-Philippe Diguet LAB-STICC, Université Européenne de Bretagne, Lorient, France, jean-philippe.diguet@univ-ubs.fr

Julien Dubois Laboratoire LE2I, Université de Bourgogne, 21000 Dijon, France,
julien.dubois@u-bourgogne.fr

Etienne Faure SoC Department, LIP6, 4 place Jussieu, 75252 Paris Cedex, France,
etienne.faure@lip6.fr

Daniela Genius SoC Department, LIP6, 4 place Jussieu, 75252 Paris Cedex, France,
daniela.genius@lip6.fr

Guy Gogniat LAB-STICC, Université Européenne de Bretagne, Lorient, France,
guy.gogniat@univ-ubs.fr

Victor Martin Gomez Computer Science and Engineering Laboratory, University of Oulu, Oulu, Finland, victor.martin@ee.oulu.fi

Emmanuel Huck ETIS Laboratory, UMR CNRS 8051, Université de Cergy-Pontoise/ENSEA, 6, avenue du Ponceau, 95014 Cergy-Pontoise, France,
huck@ensea.fr

Daniel R. Kelly CHiPTec, Centre for High Performance Integrated Technologies and Systems, The University of Adelaide, Adelaide, Australia,
dankelly@eleceng.adelaide.edu.au

Estelle Labonne TIMA Laboratory (CNRS, Grenoble INP, UJF), Grenoble, France

Thomas Lefebvre ETIS Laboratory, UMR CNRS 8051, Université de Cergy-Pontoise/ENSEA, 6, avenue du Ponceau, 95014 Cergy-Pontoise, France,
lefebvre@ensea.fr

Christophe Lucarz Microelectronic Systems Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, christophe.lucarz@epfl.ch

Stéphane Mancini GIPSA-lab, INPG-CNRS, 961 rue de la Houille Blanche Domaine Universitaire-B.P. 46, 38402, Saint Martin d'Hères, France,
stephane.mancini@gipsa-lab.inpg.fr

Philippe Marquet LIFL and INRIA Lille Nord-Europe, Parc Scientifique de la Haute Borne, Park Plaza, Bât A, 40 avenue Halley, Villeneuve d'Ascq 59650, France, Philippe.Marquet@lifl.fr

Marco Mattavelli SCI-STI-MM, Ecole Polytechnique Fédérale de Lausanne, CH 1015 Lausanne, Switzerland, marco.mattavelli@epfl.ch

Marco Mattavelli Microelectronic Systems Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, marco.mattavelli@epfl.ch

Benoît Miramond ETIS Laboratory, UMR CNRS 8051, Université de Cergy-Pontoise/ENSEA, 6, avenue du Ponceau, 95014 Cergy-Pontoise, France,
miramond@ensea.fr

Romuald Mosqueron SCI-STI-MM, Ecole Polytechnique Fédérale de Lausanne, CH 1015 Lausanne, Switzerland, romuald.mosqueron@epfl.ch

Laurent Moss Ecole Polytechnique de Montréal, Campus de l'Université de Montréal, 2500, chemin de Polytechnique, 2900 boulevard Edouard-Montpetit, Montréal, Quebec H3T 1J4, Canada, Laurent.Moss@polymtl.ca

Pengcheng Mu Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, P.R. China, pengchengmu@gmail.com

Fabrice Muller University of Nice-Sophia Antipolis/LEAT-CNRS, 250 rue Albert Einstein, 06560 Valbonne, France, Fabrice.Muller@unice.fr

Jean-François Nezan IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes, 20, avenue des Buttes de Coësmes, 35043 Rennes Cedex, France, jnezan@insa-rennes.fr

Jose L. Nunez-Yanez University of Bristol, Bristol, UK,
J.L.Nunez-Yanez@bristol.ac.uk

Braden J. Phillips CHiPTec, Centre for High Performance Integrated Technologies and Systems, The University of Adelaide, Adelaide, Australia,
phillips@eleceng.adelaide.edu.au

Sébastien Pillement University of Rennes 1, IRISA/INRIA, BP 80518, 6 rue de Kerampont, F22305 Lannion, France

Nicolas Pouillon SoC Department, LIP6, 4 place Jussieu, 75252 Paris Cedex, France, nicolas.pouillon@lip6.fr

Mickaël Raulet IETR/Image and Remote Sensing Group, CNRS UMR 6164/INSA Rennes, 20, avenue des Buttes de Coësmes, 35043 Rennes Cedex, France, mraulet@insa-rennes.fr

Robin Rolland CIME Nanotech, Grenoble, France

Olivier Sentieys University of Rennes 1, IRISA/INRIA, BP 80518, 6 rue de Kerampont, F22305 Lannion, France

Gilles Sicard TIMA Laboratory (CNRS, Grenoble INP, UJF), Grenoble, France

Olli Silvén Computer Science and Engineering Laboratory, University of Oulu, Oulu, Finland, olli.silven@ee.oulu.fi

Richard Thavot SCI-STI-MM, Ecole Polytechnique Fédérale de Lausanne, CH 1015 Lausanne, Switzerland, richard.thavot@epfl.ch

Tomasz Toczek GIPSA-lab, INPG-CNRS, 961 rue de la Houille Blanche Domaine Universitaire-B.P. 46, 38402, Saint Martin d'Hères, France,
tomasz.toczek@gipsa-lab.inpg.fr

François Verdier ETIS Laboratory, UMR CNRS 8051, Université de Cergy-Pontoise/ENSEA, 6, avenue du Ponceau, 95014 Cergy-Pontoise, France,
verdier@ensea.fr

Part 1

Architectures for Embedded Applications

Lossless Multi-Mode Interband Image Compression and Its Hardware Architecture

Xiaolin Chen, Nishan Canagarajah,
and Jose L. Nunez-Yanez

Abstract This paper presents a novel Lossless Multi-Mode Interband image Compression (LMMIC) scheme and its hardware architecture. Our approach detects the local features of the image and uses different modes to encode regions with different features adaptively. Run-mode is used in homogeneous regions, while ternary-mode and regular-mode are used on edges and other regions, respectively. In regular mode, we propose a simple band shifting technique as interband prediction and a gradient-based switching strategy to select between intraband or interband prediction. We also enable intraband and interband adaptation in the run-mode and ternary-mode. The advantage of LMMIC is to adaptively “segment” the image and use suitable methods to encode different regions. The simplicity of our scheme enables the hardware amenability. Experimental results show that LMMIC achieves superior compression ratios, with the benefits of enabling encoding any number of bands and easy access to any band. We also describe the hardware architecture for this scheme.

1 Introduction

The rapid advance of multimedia technology generates huge amount of image data, most of which are multispectral images. We define “multispectral images” here as images containing more than one spectral band. This includes a wide range of images from colour images to hyperspectral images. For instance, colour images, often stored as JPEG, BMP, or TIF format, have at least three bands, e.g. red, green and blue (RGB). In most printing systems, a four-band colour space CMYK (cyan, magenta, yellow and black) is commonly used. Moreover, some high fidelity image capture systems collect the spectral reflectance measured at different wavelengths

X. Chen (✉)
University of Bristol, Bristol, UK
e-mail: Xiaolin.Chen@bristol.ac.uk

in order to accurately capture the colour of a physical surface. For example, the VASARI imaging system [16, 23] developed at the National Gallery in London uses a seven-channel multispectral camera to capture paintings. In remote sensing, the LANDSAT 7 [15] satellite images have seven spectral bands, and the AVIRIS (Airborne Visible/Infrared Imaging Spectrometer) [7] hyperspectral images contain 224 contiguous bands. These images form the base of the widely used web mapping services, e.g. the Google Earth. In medical imaging, multispectral images also prevail. Examples include magnetic resonance imaging (MRI) which can simultaneously measure multiple characteristics of an object [6], and medical images formed by different medical imaging modalities such as MRI, CT and X-ray [8]. These images are normally compressed for transmission and storage. As many applications, e.g. remote sensing imaging, medical imaging, pre-press imaging and archiving [30], demand perfect reconstruction of images, lossless compression on multispectral images attracts increasing interests. Also for applications that need to transmit image data instantly after acquisition, real-time compression is desirable. As software compression often suffers from huge CPU resource occupation and memory consumption, we aim to design an efficient hardware amenable lossless image compression scheme, with the capability of real-time processing.

Unlike gray-scale image, multispectral image has not only spatial but also spectral (or called interband) redundancy among different spectral bands. Moreover, the existence of multiple spectral bands suggests two problems worth of concern: (1) to encode any number of bands, which is not offered by schemes with certain restricted structure; (2) to enable random access to whichever band such that any bands can be retrieved without processing the whole multispectral image. For example, each band of the LANDSAT image contains different information of the earth – water body or vegetation moisture contents etc., and different combinations of bands give illustrations on different issues such as mineral, soil and so on [15]. Therefore, it is desirable to have random band access. Many compression schemes in literature have achieved good compression ratios, but few consider these merits.

In literature, both transform and prediction techniques are used in interband image compression. Popular transform based interband coding techniques include vector quantization [12, 14, 21], discrete cosine transform [4], discrete wavelet transform [25], and vector-lifting schemes [2]. These techniques are efficient in reducing spectral redundancy, but their high computational complexity and sometimes jointly encoding several bands (e.g. 16 bands in [25]) are obstacles for hardware implementation and real-time processing. On the other hand, predictive coding does not only perform well in removing spatial redundancy but also spectral redundancy. Wu extended Intraband CALIC [29] to Interband CALIC [30], which is claimed to offer one of the best interband compression results in literature but requires complex interband correlation coefficients calculation and context formation. SICLIC [1] is a simple and efficient coder based on LOCO-I [28], but its 3-band joint-run mode, while offering good bit rates, constrains it from encoding any number of bands.

To relieve these problems, we propose a Lossless Multi-Mode Interband image Compression (LMMIC) scheme. The proposal of this scheme is inspired by the

concept of segmentation. Segmentation, in a general sense, is to partition an image into multiple segments in order to change the representation of an image into something meaningful or easy to analyse. However, traditional segmentation, e.g. statistical model-based methods [9] and graph-based methods [5], is too complex to implement in real-time compression. The novelty of our scheme is to apply the concept of segmentation to group pixels with similar features and use different methods to encode them. We called this method multi-mode strategy, where a new ternary-mode is designed to detect and encode the edges and smooth areas, and a run-length coder [28] is used to encode the homogeneous regions, while the rest of the image, say the texture regions, is coded by a regular-mode which consists of intraband and interband prediction. We propose a simple band shifting technique for interband prediction and adopt the Gradient-Adjusted Prediction (GAP) from CALIC [29] for intraband prediction. A new gradient-based switch is also designed to select the better predictor in regular-mode and to allow intraband and interband adaptation in run-mode and ternary-mode. The proposed scheme does not only offer excellent compression ratios, but also the distinctive feature of the flexibility of encoding any number of bands. As LMMIC only involves a limited number of addition and shifting, it is hardware amenable. Note that the proposed scheme in this paper is for general purpose (e.g. space, medical, archiving) images with more than one spectral bands but not specifically geared for hyperspectral images. Since some of the techniques for hyperspectral images make specific use of the structure of these images, for example, by including a band ordering process [26] or by clustering a number of bands [25], we do not include those highly specialized and not necessarily hardware amenable methods in our comparison study. However, we acknowledge that refining our techniques for specific use with hyperspectral images is an interesting topic for further research, and its findings will be reported elsewhere.

This paper is organized as follows. In Sect. 2 we present an overview of LMMIC. Then we explain the core techniques – multi-mode strategy in Sect. 3 and band shifting and gradient-based switching in Sect. 4. Context modelling is briefly described in Sect. 5. The performance comparison with other state-of-the-art schemes is presented in Sect. 6. We propose the hardware architecture to support LMMIC in Sect. 7 and conclude our work in Sect. 8.

2 An Overview of LMMIC

An image contains many features, such as smooth regions, edges, texture etc. The complexity of an image is an obstacle for compression, thus segmentation (also referred to as region-based methodology) is a viable approach to help with distinguishing these features. The lossless image compression method TMW [11], which achieves the best gray-scale image compression ratio so far, uses segmentation to analyse the image in the first pass. Shen [24] took advantage of the region-growing algorithm for segmentation of lossless compression of medical images. Ratakonda [20] used multiscale segmentation to encode general images. However, they are all

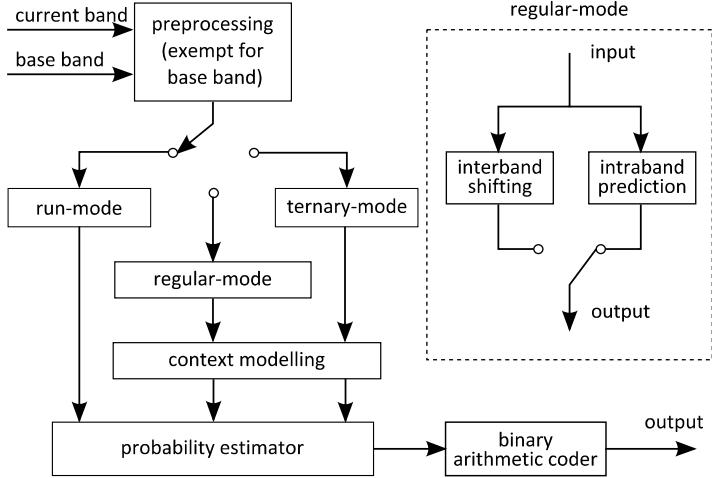


Fig. 1 Schematic of the proposed image compression system

complex two-pass schemes so cannot meet well the real-time processing requirement. Due to the complexity of segmentation, we skip the conventional segmentation methods, but instead propose to apply its concept, by using a simple switch to detect the image features adaptively, and choose suitable modes to encode these features. This switching technique resembles the function of “segmenting” the image into different areas. This is the idea that our scheme is based on.

Figure 1 shows the schematic of LMMIC. It consists of preprocessing, prediction, context modelling and arithmetic coding. At the beginning, a base band is chosen. It is encoded independently using intraband compression method. Then a preprocessing step is performed on all bands except the base band to calculate the band difference between the current band and the base band. The output of the preprocessing includes a difference band, an original band and a base band. They are then fed into the multi-mode predictor. The prediction step includes run-mode, ternary-mode and regular mode. In each mode, there is a choice between intraband and interband operations. For regular-mode, we proposed a band shifting technique for interband prediction, while the intraband prediction is based on the Gradient-Adjusted Predictor (GAP) [29]. A new gradient-based switching is designed to select the better predictor. We also enable adaptation on intraband and interband operations for run-mode and ternary-mode. This multi-mode strategy applies to all bands in an image. The context modelling is constructed in a similar but simpler way as in [29] to further exploit higher order redundancy. The probability estimator and arithmetic coding are described together with the hardware architecture in Sect. 7. LMMIC offers not only very good compression ratios, but also the distinctive feature of encoding any number of bands, since the structure of our scheme does not enforce any restriction on the number of bands to be coded. The simple multi-mode strategy and switching method make it hardware amenable.

3 Multi-Mode Strategy

As stated before, the multi-mode strategy is based on the concept of segmentation. As important as segmenting regions correctly in segmentation, it is crucial to carefully decide under which circumstances the system should be working under which mode, since it is exactly these entry conditions “segmenting” the image. In this section, we present how each mode works and the conditions for entering each mode. Prior to that, the preprocessing step is briefly described below.

3.1 Preprocessing

In Fig. 1, to start the process, one band is chosen as base band. For instance the band G in RGB images, or the first band received in other multispectral images. The base band is coded with intraband coding only. Then a preprocessing step simply subtracts the base band from the current band to get the band difference.

$$\text{Band}_{\text{diff}} = \text{Band}_{\text{curr}} - \text{Band}_{\text{base}}. \quad (1)$$

This seems to be a simple act, but leads to a lot of benefits. For example in RGB images, the bands that can be used during prediction without violating the reversibility of the algorithm are

$$G, R, B, R - G, B - G. \quad (2)$$

Instead of having three original bands, now we have five “bands” (some are difference bands) that can be used in prediction. For images with any number of bands, there always exist an original band and a difference band for encoding each band. This enables adaptation between intraband and interband prediction. Also, in this way, each band is only coupled with the base band and no multi-band joint encoding is performed. Since the base band is coded independently, it can be retrieved any time without processing other bands. Once the base band is retrieved, the current band can be retrieved. This allows the flexibility of compressing any number of bands and enables easy access to any bands. For any bands except the base band, multi-mode strategy is applied on both the original band and the difference band.

3.2 Run-Mode

Run-length coding is simple and efficient in grouping identical symbols [28]. It encodes the occurrence, i.e. the number of times that the symbol occurs consecutively, also called *run-length*. We use run-length coding to encode the homogeneous regions of the image. Figure 2 shows the neighboring pixels of the current pixel X according to their geographical positions. When $W = N = NW = NE$, the current pixel is assumed to be in a homogeneous region and is tried to be encoded in run-mode. If X is identical to W , the run-length increases by one; otherwise “run” stops and the current run-length is encoded. The latter case means that encoding symbol in run-mode is unsuccessful, so the symbol is encoded in regular-mode.

Fig. 2 Neighboring pixels of the current pixel

		NN	NNE
	NW	N	NE
WW	W	X	

3.3 Ternary-Mode

Ternary-mode is our new proposal and is inspired by the binary-mode in CALIC, which works on the binary area where there are only two different pixels in the neighborhood, e.g. black and white texts. However, unlike CALIC, our new ternary-mode targets on two types of areas – clear edge areas and smooth but not exactly homogeneous areas. Edge, which appears as the abrupt changes in pixel intensity, is very difficult to predict. Therefore, instead of predicting it, we propose to record the similarity between the edge pixels and their neighbouring pixels. In areas where a sharp edge occurs, as shown in Fig. 3a, pixels on the edge tend to be the same or similar but pixels at the two sides of the edge are usually different; also, in areas where a less sharp edge occurs, as in Fig. 3b, pixel values tend to be changing gradually from non-edge area to edge area. In both cases, we assume that within a small neighborhood of the current pixel, say the seven neighbouring pixels in Fig. 2, there are no more than three distinctive symbols and the ternary-mode is triggered. In operation, pixel W as the first unique pixel value, is represented as $s1$. Then the other six pixels in the neighbourhood are evaluated and the second and third distinctive pixels are represented as $s2$ and $s3$, respectively. By comparing the current pixel with $s1$, $s2$ and $s3$, we can assign a value to the current pixel by

$$T = \begin{cases} 0, & \text{if } X = s1; \\ 1, & \text{if } X = s2; \\ 2, & \text{if } X = s3; \\ \text{escape}, & \text{otherwise.} \end{cases} \quad (3)$$

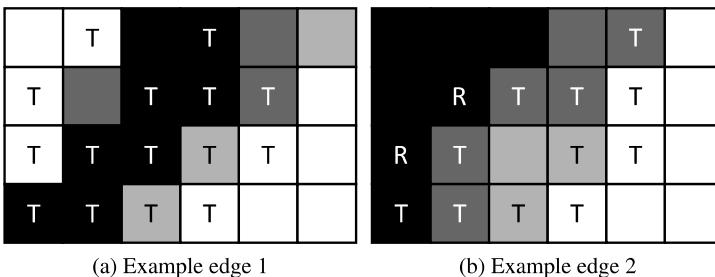


Fig. 3 Areas where ternary-mode is performed

In other word, the current pixel can be denoted by the order of its value appearance in the neighbourhood, given the condition that the checking in the neighbourhood is always conducted in the same order. “Escape” happens when the current pixel X is not equal to any of the pixel values in the neighbourhood and thus encoding in this mode fails. It is a way of switching among different modes. Figure 3 shows the areas that the ternary-mode is performed. T indicates the symbols encoded by ternary-mode, while R indicates run-mode, and the colour is the gray-level of the symbols. The figure tells that edge areas can be largely covered by this mode.

We choose to use three distinctive pixel values but not fewer or more for the following reasons:

1. In the cases shown in Fig. 3, there are usually more than two distinctive pixel values in the neighbourhood. Using only two pixel values cannot adequately describe the edge conditions.
2. In many cases, image edges are more complicated than the ones shown in Fig. 3. However, allowing more distinctive pixel values is very likely to result in more negative than positive effect, as explained below:
 - a. It makes entering the ternary-mode too easy, if four or more different pixel values are allowed in a 7 pixel neighbourhood. This would fail to characterize the specific areas that are suitable to be encoded in ternary-mode;
 - b. It would lead to a lot more “escapes”. Because more random areas are classified as applicable in ternary-mode, the current pixel is more likely to fail to find a match with any of the pixels in a more diverse neighbourhood;
 - c. Allowing more distinctive pixel values would increase the alphabet size, and hence the bits that are needed to encode pixels.

The alphabet size for encoding in this mode is only 4 instead of 256 in the original form, so lower entropy can be obtained. Ternary-mode also works as a “backup” of the run-mode in smooth but not exactly homogeneous regions.

3.4 Regular-Mode

The regular-mode is triggered, either when the entry conditions for run-mode and ternary-mode cannot be met, or when encoding in other modes fails. The regular-mode consists of intraband and inter-band prediction, which is selected according to the local features adaptively. Details of the interband prediction and switching strategy are discussed in next section.

4 Band Shifting and Gradient-Based Switching

We design a simple band shifting technique for interband prediction, and adapt the GAP from CALIC [29] for intra-band prediction. However, the performance of interband prediction depends on the interband correlation. In the case of strong inter-

band correlation, interband prediction is preferred, otherwise intra-band prediction is selected. A gradient-based switching method is proposed for the selection.

4.1 Band Shifting for Inter-Band Prediction

In the regions where bands are strongly correlated, pixel changes in one band often happen in another band. For instance, Fig. 4 plots the pixel values of one line in band G and band B of the image “peppers” respectively. It is clear that the dot plot of band G has a similar trend with the dash plot of band B. We also notice that although changing in a similar trend, the difference between two bands varies from areas to areas. Thus directly subtracting band B from band G tends to result in big errors. The ideal way would be to move the base band to a position that is as close to the current band as possible so that only a small difference between the current band and the shifted base band needs to be coded. There are a lot of possible ways to predict the value for band shifting. Since band shifting is only performed when the interband correlation is high, we assume that in this case the band difference is reasonably small and varies in a regular way. Therefore, we propose to use the simple Median Edge Detector [28] to predict the band shifting value. We denote the band difference at position W, N, NW as W_diff, N_diff, NW_diff , and calculate the band shifting by

```

if NW_diff >= max(N_diff, W_diff)
    shift_band = min(N_diff, W_diff);
else if NW_diff <= min(N_diff, W_diff)
    shift_band = max(N_diff, W_diff);
else
    shift_band = N_diff + W_diff - NW_diff;
end

```

The solid plot in Fig. 4 shows that this prediction method successfully generate a zero-mean band difference between the current band and the shifted reference band.

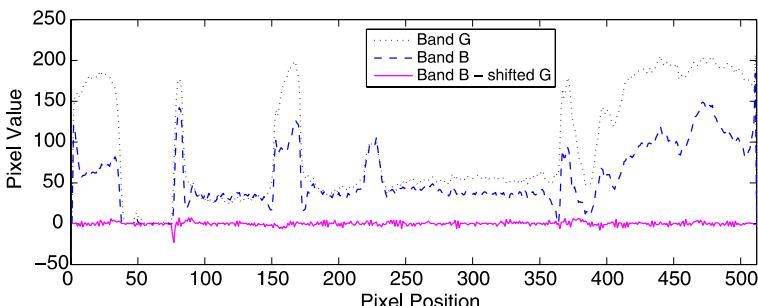


Fig. 4 Plots of one line in band G and band B and their difference after shifting

4.2 Gradient-Based Switching

For the regular-mode in the multi-mode strategy, we use the band shifting technique for interband prediction and adopt the GAP [29] for intraband prediction. Since the performance of the two predictors varies in different regions of an image depending on the spatial and spectral correlation, it is critical to decide which predictor to use in different areas. As we aim at designing a hardware amenable scheme, complex calculation of interband correlation coefficients is not desirable. We propose a simple switching method based on the local horizontal and vertical gradients, which are calculated by

$$dh = |W - WW| + |N - NW| + |N - NE|, \quad (4)$$

$$dv = |W - NW| + |N - NN| + |NE - NNE|, \quad (5)$$

where dv and dh are the vertical and horizontal gradients, respectively. When calculating the interband gradients, $W, N, NW, NE, NN, WW, NNE$ are substituted by the interband difference at the same positions. Interband gradients indicate how closely the two bands change in the same way. In addition to the gradients, the previous prediction error is taken into account to evaluate how well the predictor performs in the local area. Therefore, for both intraband and interband prediction, we calculate the switching coefficient S by

$$S = dv + dh + |e_w|, \quad (6)$$

where e_w means the prediction error at position W . The predictor that gives smaller S is selected to encode the current pixel. We counted the proportions of pixels that are treated by intraband and interband prediction in the regular-mode respectively, in band B and R on a set of RGB images in Table 1. We also calculated how often the predictor that gives smaller errors is selected, as `right_ratio`. The proportions of intraband and interband prediction do not sum up to 1 because the rest pixels are processed by the run-mode or ternary-mode. On average, more than 40% interband

Table 1 Proportions of pixels using intraband and interband prediction and the ratios of the better predictor is selected in the regular mode

Image	b_intra	b_inter	b_right_ratio	r_intra	r_inter	r_right_ratio
cmpnd1	6.41	17.66	70.13	4.62	18.64	73.33
cmpnd2	2.06	24.10	83.63	1.64	24.91	82.21
cats	1.78	47.78	87.12	1.10	47.56	90.29
water	5.33	44.19	76.53	3.04	45.49	82.15
lena	33.97	65.90	59.36	64.09	34.43	67.33
peppers	17.41	76.89	71.38	22.53	76.60	72.26
bike3	52.46	27.23	64.82	52.41	29.19	60.67
average	17.06	43.39	73.28	21.35	39.55	75.46

prediction is selected, meaning that there is a substantial amount of interband redundancy. The simple gradient-based switching technique has achieved over 70% correct choice in selecting a better predictor.

4.3 Adaptation in Run-Mode and Ternary-Mode

The above gradient-based switching is not only used in selecting the intraband and interband prediction in regular-mode, but is also modified to be used in enabling adaptation in the run-mode and ternary-mode. Since the run-mode encodes pixels directly and the ternary-mode only records the similarity among pixels, there is no prediction error generated by these two modes. Therefore, we eliminate the term of error from (6) to obtain an adaptation coefficient S' .

In the run-mode, when run-length is 0, either the pixels in the current band or in the band difference is selected according to which neighbourhood gives a smaller S' . The selected pixels are used in the run-mode and a flag is used to indicate this selection. When run-length is not 0, the previously used pixel values – whether from the current band or the band difference, are used to keep the continuity of the run.

In the ternary-mode, the whole neighbourhood of pixels used for ternary-mode are selected either from the current band or the band difference based on the value of S' . Since the coefficient S' can be calculated before receiving the current pixel, it is guaranteed that the current pixel used for comparing with its neighbouring pixels is from the same source. This adaptation in run-mode and ternary-mode improves the spatial and spectral decorrelation performance of our proposed scheme in our experiments.

We show in Table 2 the effect of using the original band, difference band or the two combined in the run-mode and ternary-mode. The “adaptation” on the fourth column means choosing the bands adaptively. The resulting bit rates vary for different images, but on average, using the adaptation technique slightly improves the compression ratios.

Table 2 Compression ratios comparison on LMMIC using different neighbourhoods in the run-mode and ternary-mode, in bits per pixel

Image	Current band	Difference band	Adaptation
cmpnd1	1.117	1.054	1.057
cmpnd2	1.037	0.972	0.969
cats	1.813	1.822	1.823
water	1.434	1.436	1.442
lena	4.233	4.230	4.233
peppers	3.339	3.363	3.356
bike3	4.274	4.353	4.289
average	2.464	2.461	2.453

5 Context Modelling

Context modeling is to group the prediction residue based on some local features, named *contexts*, in order to obtain a lower conditional entropy. In the proposed scheme, context is formed in a similar manner with CALIC [29] but is simplified to reduce the memory usage. We take 6 context symbols (W, N, NW, NE, NN, WW) to compare with the predicted value to obtain a texture pattern t , representing the local texture feature. Also, to indicate the activity of errors in a context, the *coding contexts* are generated with the local gradients dv, dh and a previous prediction error e of pixel W . The coding contexts are then quantized into 8 levels to form the coding context indexes. Combining the texture pattern and the coding contexts, a set of 512 compound contexts are formed with 6 bits texture pattern and 3 bits coding context indexes. In the case of interband context formation, original pixel values are replaced by the band difference at the same positions. These contexts are also used to generate an error feedback to the predictor, which will be discussed in Sect. 7.1. The 8 coding contexts are used to calculate the occurrence probability of pixels in the probability estimator presented in Sect. 7.2.

6 Performance Comparison

The performance of LMMIC is presented in this section. We firstly give two examples to show the areas where different modes apply in images. In example 1, Fig. 5b shows the regions where different modes are performed, comparing with the original image Fig. 5a. Run-mode works on the grey areas, which are smooth and homogeneous; ternary-mode works on the white regions, which often lie on the edge of the homogeneous regions, some smooth regions or clear edges; regular-mode works on the dark regions, which are mostly texture or noisy areas. In example 2, we only apply ternary-mode and regular-mode on image Fig. 6a to emphasize the function of ternary-mode. In Fig. 6b, white areas indicate where the ternary-mode applies, while the black areas indicate where the regular-mode applies. We can see that all the texts including those in the picture are covered by the ternary-mode, as well as some of the clear edges and homogeneous areas.

To have a quantitative measure of the proportion that the three modes apply on an image, we count the number of pixels being treated by each mode in Table 3. The “run” in the second column stands for the number of pixels being coded by the run-mode. The “ternary” in the third column means the number of pixels satisfying the entry condition of the ternary-mode, and the “successful ternary” in the fourth column means the number of pixels being successfully coded by the ternary-mode. The “successful ternary ratio” in the fifth column is the proportion of the “successful ternary” in the whole image. Table 3 shows that run-mode does not happen often except in relatively simple images like “bike3-g” and “CMPND1-G”. Ternary-mode occurs more often but still takes on a small proportion, as shown in the fifth column of “successful ternary ratio”. But it has a roughly 60% successful rate when comparing the fourth and the third column, which means 60% of the pixels entering the

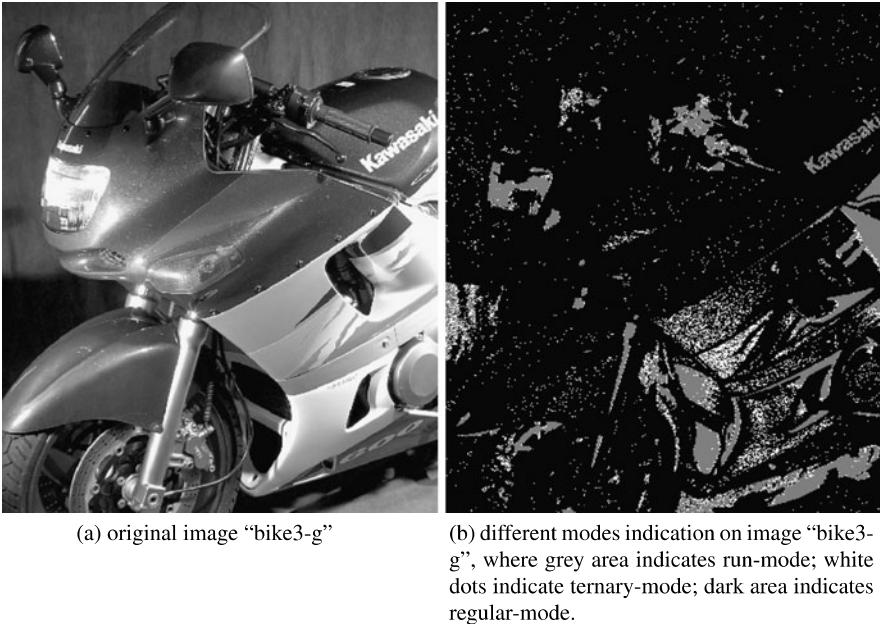


Fig. 5 Example 1: (a) Original image “bike3-g” and (b) image indicating different modes applied

Table 3 Mode counts of LMMIC

Images	Run	Ternary	Successful ternary	Successful ternary ratio (%)
lena	148	8117	4363	0.017
barb2	7492	10708	5585	0.013
hotel	2826	16823	11686	0.028
bike3-g	109012	146804	113654	0.158
CMPND1-G	276114	31227	27833	0.071

ternary-mode can be successfully coded by it. This makes it improve the overall bit rates compared to applying the regular-mode alone.

The experimental result in terms of compression ratios is presented in Table 4. We choose a set of standard 3-band RGB images, a 4-band CMYK image “park” and a 7-band LANDSAT 7 image “coastal” from CCSDC (the Consultative Committee for Space Data Systems). The RGB images include continuous-tone images (“cats”, “water”, “lena” and “peppers”), compound images (“cmpnd1” and “cmpnd2”) and synthetic image (“bike3”). We compare the proposed compression scheme with JPEG2000 [27], intra-band CALIC [29], IB-CALIC [30] and SICLIC [1]. The results of IB-CALIC and SICLIC are extracted from [1]. Some results are absent due to the unavailability of the programs. JPEG2000 is the current standard for image compression. The results of IB-CALIC are claimed to be one of the best in

Dear Pam,

I was delighted to hear from you last week. Patti and I had a wonderful time during our week-long summer vacation. The weather was excellent, and the food was absolutely exquisite. I hope that we can repeat this next year and that you will join us too.

We came back with a lot of fantastic memories, which we would like to share with you through some snapshots that we took.



Our favorite is this picture of us aboard the "Top Hat", which I have pasted into this letter using some really neat advanced digital imaging technology on my home computer. We will ship the rest to you on a CD-ROM soon. Wishing you the best.

Love,
Susan

(a) original image "CMPND1-G"

(b) Ternary-mode and regular-mode indication on image "CMPND1-G", where white area indicates ternary-mode; dark area indicates regular-mode.

Fig. 6 Example 2: (a) Original image "CMPND1-G" and (b) image indicating ternary-mode and regular-mode areas

Table 4 Bit rates comparison on selected schemes, in bits per pixel per band

Image	JPEG2000	CALIC	IB-CALIC	SICLIC	LMMIC
cmpnd1	1.44	1.21	1.02	1.12	1.05
cmpnd2	1.30	1.22	0.92	0.97	0.97
cats	1.75	2.49	1.81	1.85	1.82
water	1.41	1.74	1.51	1.45	1.44
lena	4.53	4.40		4.46	4.23
peppers	3.41	4.62		3.25	3.35
bike3	5.17	4.21		4.41	4.29
average	2.72	2.84		2.50	2.45
park	5.72	5.39			5.30
coastal	2.89	2.68			2.62

literature in terms of general interband image compression, but not hyperspectral image compression, which is not the scope of our proposed method either. And SICLIC is a good trade-off between compression ratio and complexity. Table 4 shows that LMMIC outperforms JPEG2000 and intraband CALIC by 10% and 14%, respectively. It is superior than SICLIC on average, though slightly inferior than IB-CALIC which has higher computational complexity. Since the interband coding in LMMIC only couples two bands, it has the flexibility of compressing images with any number of bands and easy access to any bands.

7 Hardware Architecture

Hardware amenability is one of the priorities in the design of the proposed scheme. Therefore, as previously described, the whole procedure, including the prediction and mode switching, only involves a limited number of addition and shifting, and memory usage is strictly controlled. In this section we propose the suitable hardware architecture to support the proposed compression scheme, which can be mainly divided into two parts – the architectures of lossless image modelling and encoding. The modelling part includes prediction and context modelling, while the encoding part includes probability estimator and binary arithmetic coder. We will discuss them below respectively in details.

7.1 Lossless Image Modelling

Lossless image modelling here serves both gray-scale and multispectral images. The user can specify which category the input image belongs to. The data flow of the image compression scheme is shown in Fig. 7.

To optimize the speed and hence the throughput, the data flow of the scheme is designed as two pipelines running in parallel, as shown in Fig. 7. Line 1, indicated by the flow on the left in blue, operates for the current symbol. It takes in the input symbol and selects the suitable mode to encode it. The mean of errors and context index calculated from Line 2 are fed into the multi-mode prediction and probability estimator. The output from the multi-mode prediction, which is either the “runs” of the symbols, or the symbol order from the ternary mode, or the prediction error from the regular mode, are used to drive the probability estimator and arithmetic coder. Line 2, indicated by the flow on the right in red, works for the next symbol. It takes the input symbol to update the contexts, and calculate the prediction value and context index under the selected mode for the next symbol. The advantage of dividing the procedure into two parallel pipelines is, while not violating the sequential constraint, to halve the execution time and hence obtain higher throughput. A summary of the operations in each pipelines is as follows.

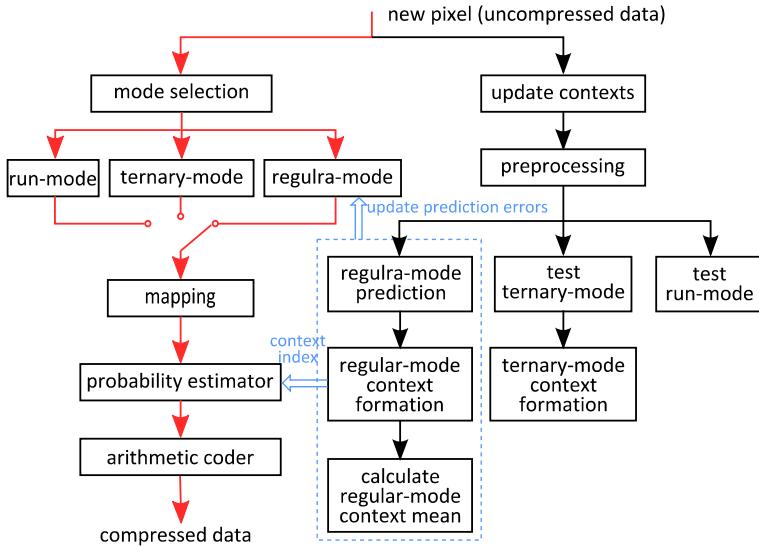


Fig. 7 Data flow of the prediction and context modeling module

Line 1:

1. Select a suitable mode and calculate prediction error $\epsilon = X - (\hat{X} + \text{error_mean})$;
2. Update the sum of errors sum and the number of pixels $count$ in each context;
3. Map prediction error ϵ to $\tilde{\epsilon}$;
4. Update coding context index Q ;
5. Encode the prediction error with probability estimator and arithmetic coder.

Line 2:

1. Update contexts with new symbol;
2. Calculate primary prediction value \hat{X} for regular mode, and evaluate the condition of entering run-mode and ternary-mode;
3. Calculate the texture pattern;
4. Calculate the error energy and the context index;
5. Calculate the mean of the errors error_mean .

During the above process of image modelling, there are two issues worth of special care, while the rest of the process is relatively straightforward numerical computation.

- There is usually heavy memory usage in image compression algorithms, either for storing prediction context or coding context, which are defined in Sect. 5. To minimize the memory usage for prediction context, we only store three lines of image pixel values in memory. We use three pointers to indicate the line orders. New input symbols are always stored in line A. Every time when a line is filled up, the three pointers to each line rotate in such an order that the oldest line is discarded and the newly formed line is saved.

- As introduced in [29], an error feedback technique is used in the prediction step to adjust the prediction bias in each context. We adapt this approach in our scheme, but we calculate the prediction bias based on a different context formation and provide special treatment for hardware amenability. The mean of errors $\bar{\epsilon}(C)$ in each context C is assumed to be the most probable prediction offset error, and hence is a good observation of the bias of the predictor. We improve the prediction by adding this term. It is calculated by

$$\bar{\epsilon}(C) = \text{sum}/\text{count} \quad (7)$$

where *sum* and *count* are the sum and occurrence count of errors in the context C , respectively. The calculation of $\bar{\epsilon}(C)$ requires arbitrary division, which resources demanding in hardware implementation. To solve this problem, instead of having an infinite range, we store the *count* with only 5 bits ($2^5 - 1 = 31$). When the *count* reaches its maximal value 31, it is halved by right-shifting one bit; meanwhile *sum* is also halved so as to maintain the mean $\bar{\epsilon}(C)$. For images with 8 bits per pixel, the mapped prediction error ranges from 0 to $2^8 - 1$. Therefore we only need 13 bits ($2^5 \times 2^8 = 2^{13}$) plus one sign bit to store the sum of errors. Experimental results prove that this rescaling technique slightly improves the compression ratio by “aging” the observed data. However, division is always a difficult problem in hardware, especially when the dividend can be as large as 13 bits. To make this division practical, we bound the dividend *sum* by 10 bits for two reasons: firstly, experiments on our image test set show that the chance of the *sum* being larger than 1023 is less than 0.001%; secondly, extraordinary large errors tend not to reflect the true behavior of the context because prediction errors should be moderately small given an adequate predictor. Therefore, we can safely clip the *sum* value to 1023. We use the most significant bits of the divisor *count* in the division, with the dividend being rescaled accordingly to maintain the same result. Consequently, we only need a lookup table of 1 KByte ($2 \times (2^{10}/2) = 1024$) to complete fast division. Although the result of division is only an approximation, it does not affect the compression performance in our experiments.

7.2 Probability Estimator and Arithmetic Coding

To encode the prediction errors generated from the image modelling stage, arithmetic coding is chosen in our implementation due to its best performance among other coding methods in terms of compression ratios. In actual implementation, it is advantageous to separate the arithmetic coding procedure as probability estimation and coding process [22]. Probability estimation is to calculate the probabilities of the data source adaptively and the coding process uses these probabilities to recursively calculate the proportion within the interval $[0, 1]$, which is used to encode the input symbols. In this section, we briefly introduce the arithmetic coder we choose in our proposal and present the architecture of the probability estimator.

7.2.1 Arithmetic Coding

Since probability estimator serves for the arithmetic coder, we can decide the implementation of the arithmetic coder first. A general arithmetic coder handles data source with multi-symbol alphabets. While providing good performance, it is complex and not easily amenable to hardware implementation. Alternatively, *binary arithmetic coding* has become a popular implementation of the arithmetic coding. It was proposed by Langdon and Rissanen in 1981 [10] and later adopted by the bi-level lossless image compression standard JBIG. Binary arithmetic coding works with binary source alphabet (0 and 1) and thereby the cumulative distribution vector of the alphabet, denoted by $pcum$, is simply $pcum = [0 \ p_0 \ 1]$, with p_0 being the probability of symbol 0. The cumulative distribution of each symbol is required to be updated adaptively for each coded symbol in arithmetic coding. As Moffat [13] and Said [22] pointed out, this task can be much simpler in binary arithmetic coding due to the simple data structure. Binary coding also avoids implementation of some resource expensive components such as multiplication, and thus obtains efficiency gains. Due to the amenability and fast speed of binary arithmetic coder for software and hardware implementation, we adopt it in our proposed system. For more details about the binary arithmetic coder we used, the readers are referred to [13, 22]. An implementation of this binary arithmetic coder is published in previous work of our group [18].

7.2.2 Probability Estimation

Overview As the prediction errors generated by the prediction step has a multi-symbol alphabet, e.g. $2^8 = 256$ symbols for images with 8 bits per pixel, they cannot be processed directly by a binary arithmetic coder. We construct a probability estimator to adaptively calculate the probabilities of symbol occurrence in each context and decompose these probabilities into binary symbols (0 or 1). It enables the application of a simple and efficient binary arithmetic coder and hence results in full pipelining and high throughput. This probability estimator is based on the one presented in [17]. In [17], the probability estimator is optimized for the statistical lossless general data compression algorithm which includes variable order contexts, while in our implementation we modified it for the fixed-order contexts of image compression. In particular, we studied the effect of using different amount of bits to store the symbol occurrence count and the initialization of the probability estimator. We will explain in detail soon below.

In the context modelling described here, we divide the prediction errors into different groups, which we called coding context. We assume that the prediction errors are independent identically distributed within each context. Therefore, probability estimation is performed for each coding context.

Working Mechanism of the Context Trees The main part of the probability estimator is a tree structure stored in a SRAM. Generally, each coding context is represented by an $n + 1$ level (n is the bits per pixel) balanced binary tree with 2^n

leaves associated with each symbol in the alphabet. For instance, a 9-level tree with $2^8 = 256$ leaves should be used for each context of the prediction errors ranging from 0 to 255. In each tree node, a register is used to store the symbol occurrence count. Initially, all the symbols in the alphabet are assigned to a certain probability, which is equal probability in most of the cases, and the whole range of the probability sums up to 1. Thus each tree leave has an initial value to represent its probability, and other tree nodes have the value of the sum of the value of their two sub-trees. When a new symbol is received, the value of the corresponding tree nodes increases to reflect the change of the probability distribution of symbol occurrence.

Here we demonstrate the working mechanism of the tree for probability estimation in Fig. 8. It shows a simplified binary tree with 3 level and 4 leaves, which represents a four symbol alphabet. The number in each tree node denotes the symbol occurrence. The value with underline on top of each tree leave is the symbol value represented by the leave. Firstly, all the trees need to be initialized before being used. In Fig. 8a, all the tree nodes are initialized to 0, except the tree root and the

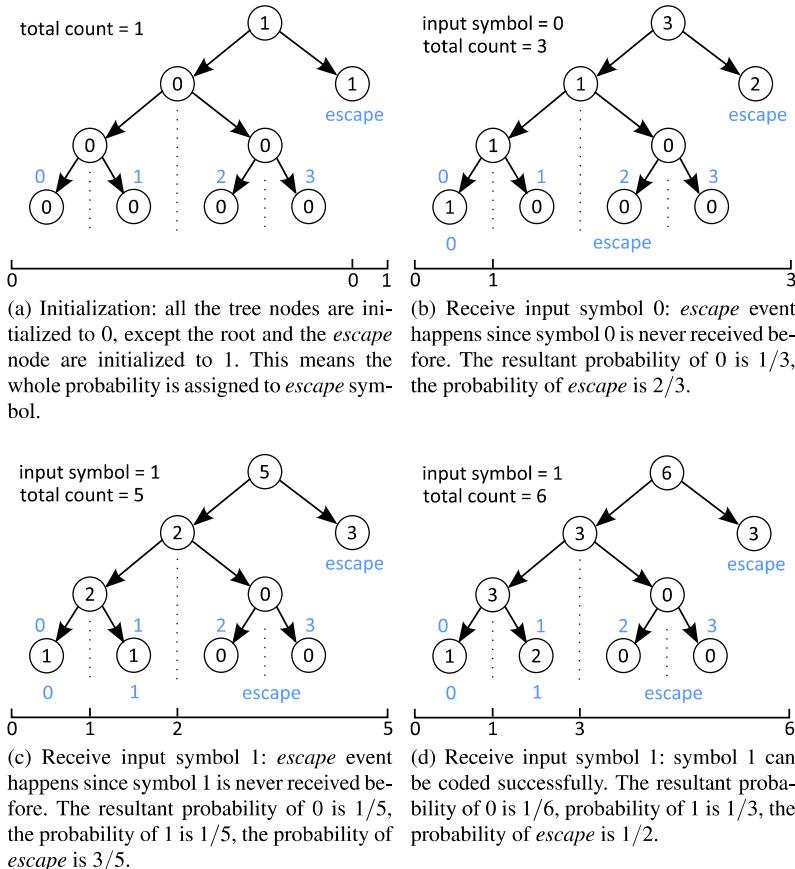


Fig. 8 Simplified tree structure of the probability estimator

escape node. *Escape* here means coding is not successful. It happens when a valid probability of a symbol cannot be found. For example, when an input symbol has not occur before and hence its tree node value (symbol occurrence) is 0, the *escape* event happens and the new symbol cannot be coded directly. We will show how it affects the tree in the rest of Fig. 8 and more discussion about *escape* would follow soon. In the initialization, the *escape* node is assigned to 1 to avoid coding failure. The value of the tree root is the sum of the symbol occurrence and the *escape* events, and hence is 1. We stored the tree root value as *total count*. It is clear that at this stage, the whole range of probability is assigned to *escape*. In Fig. 8b, a new symbol 0 is received. This symbol is not seen before so its tree node is 0. In this case, *escape* happens and the *escape count* increases by 1. Meanwhile, the tree leave for symbol 0 should also increase to reflect the symbol occurrence. Correspondingly, the tree nodes, which symbol 0 walks pass when going down from the second level tree node, should also increase accordingly. The coordinate below the tree reveals the current probability distribution, where symbol 0 has $1/3$ and *escape* takes on $2/3$. A similar situation happens in Fig. 8c, where a new symbol 1 is received. Both *escape* and 1 increase their occurrence. This would happen to all symbols occurring first time after initialization to 0 of the tree. But when the symbol already has a valid probability (non-zero occurrence), as in Fig. 8d, the input symbol can be coded directly. Figure 8d shows when a symbol 1 arrives for the second time. This time only the tree leave of symbol 1 increases, resulting its probability to $1/3$. When more symbols are received, fewer *escape* would happen and symbols are most likely to be directly coded. But there are exceptions which we will discuss in the next two paragraphs.

Context Tree Initialization In our proposed scheme, we use 8 trees for the regular-mode and one for the run-mode. Each tree has $2^8 = 256$ nodes. For ternary-mode, we use 64 trees, each of which has 4 nodes corresponding to the four options (s_1, s_2, s_3 and “*escape*”) in ternary-mode. These trees are dynamically updated during the coding process and thus are called *dynamic trees*. In addition, we use two trees, one of each kind, to represent the “*escape*” condition. Because these two trees do not change during the coding procedure, they are called *static trees*. Symbols coded by the static trees do not achieve any compression as the static trees do not change to reflect any probability distribution changes. Therefore, *escape* is undesirable. Then why do we still want to initialize the tree nodes to 0 even though there are more *escape* happening?

We had thought of a couple of initialization possibilities: (a) to initialize the tree nodes all to 1 just to reduce *escape*; (b) as the prediction errors tend to follow a Laplacian distribution [29], we can possibly initialize the tree nodes in a Laplacian way – let the small symbol be assigned higher probability and as the symbol becomes bigger the probability decreases as the Laplacian curve. Therefore, we carry out the following experiment. We apply the same lossless image compression scheme on the image “lena”. The only difference among these schemes is the initialization of the probability estimation. The comparison of compression ratios

Table 5 Performance comparison on different initialization of probability estimator

Initialization	Escape count	Compression ratio (bpp)
0	697	4.137
1	229	4.142
Laplacian	560	4.140

and *escape count* is shown in Table 5. Although the compression ratios do not vary dramatically under different initialization methods, it can help us with understanding the problem. When the trees are initialized as 0, despite the large amount of *escape* happening, it performs best because it ignores those symbols that never or rarely occur and thus reduces code space. When the trees are initialized as 1, it sets all occurrence count to 1, even for those that have never appeared. This might slightly distort the original histogram of prediction errors. When the trees are initialized as Laplacian distribution, it actually helps with building up the desirable error histogram in the first instance but this advantage might soon be overtaken by the possibly incorrect forced occurrence count for some symbols. From the above observation, we initialize the probability for all the symbols to 0.

Choice of Context Tree Node Size *Escape* takes place when the *occurrence count* of the input symbol is 0. This does not only happens after initialization, but also occurs when the value of the tree node is *rescaled*. Since the *occurrence count* is stored in a register, the size of the register decides its limit, which is the *maximum occurrence count*. When the maximum occurrence count of a symbol is reached, the *occurrence count* needs to be rescaled. Rescale is done by halving the occurrence count, which can be easily executed by right shifting. In order to maintain the probability distribution, all the tree nodes need to be rescaled. Consequently, the *occurrence count* of symbols that have only been received less than twice will be rescaled to 0, resulting in *escape* when those symbols occur later. Therefore, the size of the register, which is the number of bits, to store the symbol *occurrence count* needs to be carefully chosen. We carry out experiments on the image “lena” using the same image compression algorithm but with different number of bits for storing the *occurrence count*. The results of average compression bit rates are shown in Fig. 9. We can see that the number of bits does affect the compression performance considerably. When the *maximum occurrence count* is too small, more *escapes* are likely to happen; when the *maximum occurrence count* is too big, the “aging” effect of observed data is attenuated. Therefore, we choose 14 bits according to the result of Fig. 9.

Output of Probability Estimation In order to drive the binary arithmetic coder, the probability estimator output three values: *decision bit*, cumulative probability of 0 (denoted as *cum0*) and cumulative probability of 1 (denoted as *cum1*).

The cumulative probabilities of 0 and 1 here means the cumulative probabilities of going left or going right in the context tree. They can be calculated from the

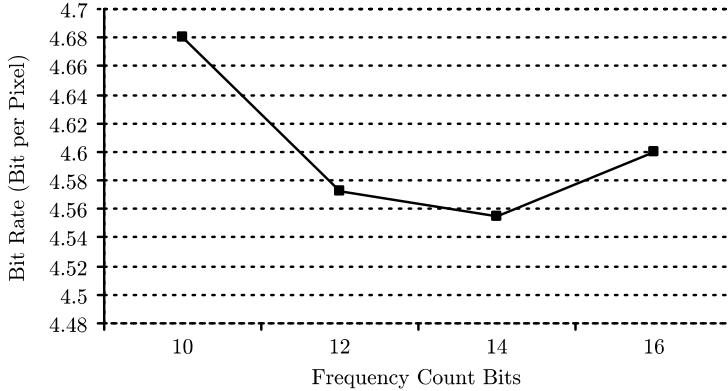


Fig. 9 Average bit rates under different probability precision, in bpp

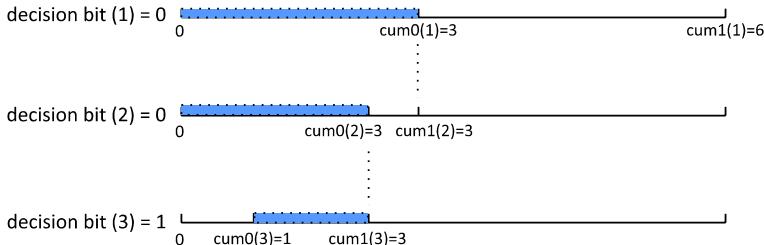


Fig. 10 Arithmetic coding using the output of probability estimator

tree node values. Let us recall Fig. 8d. When the symbol 1 arrives, the *total count* increases from 5 to 6. Since this symbol appears before, it goes to the left branch but not *escape*. Here it outputs one *decision bit* 0. The *cum0* is the value of the current tree node, which is 3, and *cum1* is the value of its parent tree node minus the one of the current tree node (*cum0*). Figure 10 shows how these procedure works and the effect of these output. The values are assigned based on the tree in Fig. 8d. The number in parentheses denotes the level of the tree. So at the first level, *cum0* = 3, *cum1* = 6. Because the *decision bit* is 0, the probability 1/2 is chosen. Since symbol 1 is 01 in binary, it goes to the left branch in the next level. In the selected interval from the previous level in Fig. 10, the whole range is assigned to 0 so we have *cum0* = *cum1* = 3. As *decision bit* is 0, the interval between 0 and *cum0* is chosen. In the last level, symbol 1 goes to the right branch of the tree. So the selected interval is between *cum0* and *cum1* and its probability is 2/3. The total probability of the symbol 1 can be calculated as $(1/2) \times (2/3) = 1/3$. In this way, the probability estimator only needs to output the *decision bit*, *cum0* and *cum1* at each clock cycle, and a total of 9 cycles are needed to encode one 8-bit symbol.

Architecture of Probability Estimator Figure 11 shows a simplified diagram of the probability estimator. It is modified from our previous work in [17]. As men-

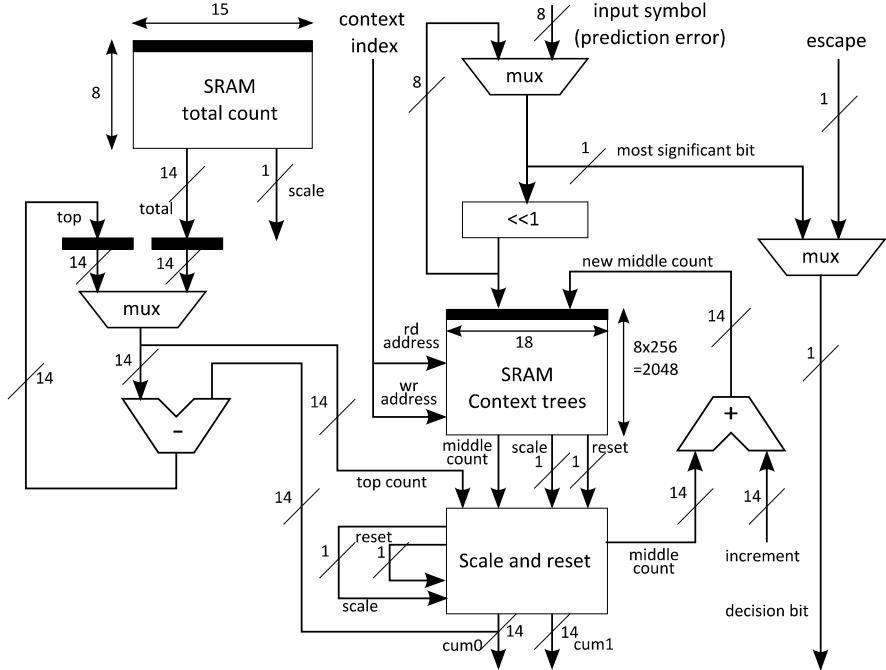


Fig. 11 Architecture of the probability estimator

tioned earlier, the probability estimator in [17] works for general data compression, while this diagram is modified to work for our proposed image compression. Two SRAM memory are used to store the context trees and the *total count* for each tree respectively. The size of the SRAM for the context trees is

$$N_{con} \times S_{tree} \times W_{node} \quad (8)$$

where N_{con} is the number of contexts, S_{tree} is the number of tree nodes in each tree, and W_{node} is the width of the register used to store the information of each tree node. This not only includes the 14 bits used to store the tree node value, but also include two scale bits and two reset bits. They are used to control whether the left or right sub-tree of the current tree node needs to be rescaled or reset. With the scale bits and reset bits, the operation of scale and reset can be done while the input symbol transverses down the tree, instead of being executed separately and wasting clock cycles. The input symbol, which is the prediction error, is shifted one bit each clock cycle. Its most significant bit is sent to the context tree for probability calculation. Meanwhile, the context index is read in to specify the corresponding context tree. The *increment* denotes the amount to be added to each tree node when a symbol walks pass. We set *increment* to 1. But it can also be set to bigger values if the user wants to accelerate the “aging” effect of rescaling. The *middle count* and *top count* are output as *cum0* and *cum1*, about which we have explained the role in calculating probability.

The probability estimator maps the probability data into a set of binary decisions and cumulative occurrence counts, which enable the use of efficient binary arithmetic coding. It needs 9 clock cycles to code a 8-bit symbol, without using any extra clock cycles for rescaling and resetting. Therefore, it is highly efficient and allows high throughput. Some preliminary implementation results of the regular-mode working for grayscale images are published in [3, 19]. This part of the architecture together with the binary arithmetic coder enables the system to process one bit per clock cycle, which translates into a throughput of around 123 Mbits/s on a Xilinx Virtex 4 FPGA. Full implementation of the proposed system will be part of our future works.

8 Conclusions

An original Lossless Multi-Mode Interband image Compression (LMMIC) scheme is proposed. The concept of segmentation is well ingrained in this scheme to deal with different regions in the image adaptively. The simple and efficient band shifting technique and the switching strategy successfully remove the interband redundancy. Experiments show that LMMIC achieves highly competitive compression ratios and provides the flexibility of compressing any number of bands as well as easy access to any bands, which are not offered by many other schemes. The complexity of the scheme is strictly controlled and hardware amenability is maintained. A corresponding hardware architecture is also proposed to support the functionality of the proposed algorithm.

Acknowledgements The authors would like to thank the support from EPSRC under grant EP/D011639/1.

References

1. Barequet R, Feder M (1999) SICLIC: a simple inter-color lossless image coder. In: Proc data compression conf, pp 501–510
2. Benazza-Benyahia A, Pesquet J-C, Hamdi M (2002) Vector-lifting schemes for lossless coding and progressive archival of multispectral images. IEEE Trans Geosci Remote Sens 40(9):2011–2024
3. Chen X, Canagarajah N, Nunez-Yanez JL, Vitulli R (2007) Hardware architecture for lossless image compression based on context-based modelling and arithmetic coding. In: Proc IEEE int system on chip conf, pp 251–254
4. Dragotti PL, Poggi G, Ragozini ARP (2000) Compression of multispectral images by three-dimensional SPIHT algorithm. IEEE Trans Geosci Remote Sens 38(1):416–428
5. Felzenszwalb PF, Huttenlocher DP (2004) Efficient graph-based image segmentation. Int J Comput Vis 59(2):167–181
6. Hu J-H, Wang Y, Cahill, PT (1997) Multispectral code excited linear prediction coding and its application in magnetic resonance images. IEEE Trans Image Process 6(11):1555–1566
7. Jet Propulsion Laboratory, California Institute of Technology. Airborne Visible/Infrared Imaging Spectrometer (AVIRIS). <http://aviris.jpl.nasa.gov/>

8. Kayyali MSE multispectral technology applications. http://www.articlealley.com/article_1604_11.html
9. Kim J, Fisher JW, Yezzi A, Cetin M, Willsky AS (2005) A nonparametric statistical method for image segmentation using information theory and curve evolution. *IEEE Trans Image Process* 14(10):1486–1502
10. Langdon GG, Rissanen JJ (1981) Compression of black–white images with arithmetic coding. *IEEE Trans Commun* 29(6):858–867
11. Meyer B, Tischer PE (1997) TMW – a new method for lossless image compression. In: Proc int picture coding symp
12. Mielikainen J, Toivanen P (2002) Improved vector quantization for lossless compression of AVIRIS images. In: Proc XI European signal processing conf
13. Moffat A, Neal R, Witten IH (1998) Arithmetic coding revisited. *ACM Trans Inf Sys* 16(3):256–294
14. Motta G, Rizzo F, Storer JA (2003) Compression of hyperspectral imagery. In: Proc data compression conf, pp 333–342
15. National Aeronautics Space Administration (NASA): the Landsat program. <http://landsat.gsfc.nasa.gov/>
16. National gallery: visual arts system for archiving and retrieval of images. <http://users.ecs.soton.ac.uk/km/projs/vasari/>
17. Nunez-Yanez JL, Chouliaras VA (2005) A configurable statistical lossless compression core based on variable order Markov modeling and arithmetic coding. *IEEE Trans Comput* 54(11):1345–1359
18. Nunez-Yanez JL, Chouliaras VA (2005) Design and implementation of a high-performance and silicon efficient arithmetic coding accelerator for the H.264 advanced video codec. In: Proc IEEE int conf on application-specific systems, architecture processors, pp 411–416
19. Nunez-Yanez JL, Chen X, Canagarajah N, Vitulli R (2007) Dynamic reconfigurable hardware for lossless compression of image, video and general data content. In: Proc XXII conf on design of circuits and integrated systems. Invited paper
20. Ratakonda K, Ahuja N (2002) Lossless image compression with multiscale segmentation. *IEEE Trans Image Process* 11(11):1228–1237
21. Ryan MJ, Arnold JF (1997) The lossless compression of AVIRIS images by vector quantization. *IEEE Trans Geosci Remote Sens* 35(3):546–550
22. Said A (2004) Introduction to arithmetic coding – theory and practice. Imaging Systems Laboratory, HP Laboratories Palo Alto
23. Saunders D, Cupitt J (2003) Image processing at the national gallery: the VASARI project. The National Gallery, Technical Bulletin 14(1):72–85. London, UK
24. Shen L, Rangayyan RM (1997) A segmentation based lossless image coding methods for high-resolution medical image compression. *IEEE Trans Med Imaging* 16(3):301–307
25. Tang X, Pearlman WA, Modestino JW (2003) Hyperspectral image compression using three-dimensional wavelet coding. Proc SPIE, vol. 5022. SPIE, Bellingham, pp 1037–1047
26. Tate SR (1997) Band ordering in lossless compression of multispectral images. *IEEE Trans Comput* 46(4):477–483
27. Taubman DS, Marcellin MW (1996) JPEG2000 image compression fundamentals, standards and practice. Kluwer, Norwell
28. Weinberger MJ, Seroussi G, Sapiro G (1996) LOCO-I: a low complexity, context-based, lossless image compression algorithm. In: Proc data compression conf, pp 140–149
29. Wu X, Memon N (1997) Context-based adaptive, lossless image coding. *IEEE Trans Commun* 45(4):437–444
30. Wu X, Memon N (2000) Context-based lossless interband compression – extending CALIC. *IEEE Trans Image Process* 9(6):994–1001

Efficient Memory Management for Uniform and Recursive Grid Traversal

Tomasz Toczek and Stéphane Mancini

Abstract This chapter presents the usefulness of predictive and adaptive caching methods for the traversal of both uniform and recursive 3D grid structures. Recursive data structures are used in several image processing kernels and their efficient management is one challenge to save silicon area and reduce the power consumption due to the data transport. The described architectures greatly reduce the needs in term of bandwidth by exploiting the spatial and temporal locality of memory accesses during ray shooting in uniform and recursive grids. To maximize the cache efficiency, the original kernel is transformed to a “phase locked” ray-packet based propagation algorithm. Our results show that well-suited caching strategies can indeed yield significant performance gains during the traversal of both uniform and hierarchical grids. This emphasizes the relevance of semi-general purpose multi-dimensional predictive caches.

1 Introduction

The management of high quantities of data is a challenge for many digital systems. This problem is getting more and more complex with the increase of the quantity of memory embedded in digital integrated systems such as System on Chip (SoC). As an example, the International Technology Road-map for Semiconductors Consortium plans that memory will occupy 90% of a circuit in the next years. Then, to alleviate the well known memory wall, it is mandatory to provide efficient memory hierarchies that are optimized together with a cache friendly optimization of applications.

T. Toczek (✉)

GIPSA-lab, INPG-CNRS, 961 rue de la Houille Blanche Domaine Universitaire-B.P. 46, 38402, Saint Martin d’Hères, France
e-mail: tomasz.toczek@gipsa-lab.inpg.fr

Several image processing algorithms are using multi-resolution images to perform tasks such as vision [4], video compression [3] and 3D rendering [1]. Multi-resolution images are used for vision algorithms to integrate some global information at the pixel level: the results at the low resolution are used to constrain the computations at the more detailed levels. To compress video, the motion estimation step also benefits from a multi-resolution pyramid of the input images: the coarse grain motion vectors at the low resolution are used to guide the computations at higher resolutions, preventing the algorithm to “fall” in some local minima. Mip-Map images are used in real-time 3D rendering to apply textures at a level of resolution that fits the raster scan sampling of the scene triangles. These multi-resolution textures allow to speed-up the rendering and increase the quality of the rendered images by preventing subsampling (aliasing). The efficient management of multi-resolution and recursive data structure is one of the challenge to design embedded systems for image processing.

Multi-resolution 3D grids are widely used for applications such as realistic rendering ray-tracing, medical visualization, volume rendering and tomographic reconstruction. These applications extensively use the *ray shooting* kernel to simulate the propagation of light in a 3D volume. Ray shooting based algorithms are widely considered as both computationally and bandwidth hungry. They have however the advantage of producing high quality results while being conceptually simple, thereby being a good example of what modern architectures may be used for. The principle is to compute ray paths through diversely structured scenes.

In this chapter we will focus on the traversal of both uniform and recursive grids, which can be seen as either space indexing means or direct volume representations. On the one hand, space indexing means are used to store a primitive-based scene (typically a set of textured triangles) in an easily traversable structure, commonly known as an acceleration structure (AS). The AS traversal returns a list of primitives or even a more complex sub-scene to which some further computations are performed. Examples of this approach include SAH-based *kd-trees*, bounding box hierarchies, and so on [7]. Direct volume representations, on the other hand, involve partitioning the space into small cubic fragments called voxels, and storing a meaningful value for each voxel, such as color, density, and so on. This is especially useful in medical visualization and imaging. With no loss of generality, we choose the later direct volume representation as our performance case study.

This chapter demonstrates the usefulness of the *nD-AP Cache* architecture (*n*-Dimensional Adaptive and Predictive Cache) [12, 15] and the associated methodology for some ray-shooting based applications. The results demonstrate the versatility of the *nD-AP Cache*, which performs well for the uniform as well as the recursive grid traversal. To tackle the later aspect, we used a set of small communicating *nD-AP Caches* to cache part of the scene tree.

To fit the prediction mechanisms, the original ray (line) propagation kernel needs to be transformed to exhibit more temporal and spatial locality. That for a “phase-locked” ray-packet based propagation algorithm is proposed. The idea is to enable a virtual loop to synchronize the propagation of a set of rays. Hence, this transformation improves the very low data-reuse ratio of the original iterative algorithm for which a grid cell is traversed only once.

The proposed memory hierarchies and “phase-locked” propagation algorithm, both for uniform and recursive grids, are validated by performance measures performed both on an emulation platform and through CABA (Cycle Accurate, Bit Accurate) simulations. The experiments lead to some improvements of the cache features so as to adapt the n D-AP Cache to the ray shooting kernel and any processing with similar patterns of memory references.

In Sect. 2, we briefly describe some of the hardware designed so far for memory management for ray shooting. In Sects. 3 through 6, we present our architecture, which includes the above mentioned cache and traversal pipelines. Finally, we discuss its performances in Sect. 7 and conclude in Sect. 8.

2 State of the Art

2.1 Dataset Traversal

The iterative parametric methods for scene traversal are the most popular ones. They are conceptually easy and their implementation is often efficient. Most of them are variants of the DDA (Digital Differential Analyzer) algorithm adapted to ray tracing [2], which parametrizes the ray and performs all the computations in the parameter space thereafter. Figure 1 illustrates the variables used by the DDA on a 2D example.

The algorithm iterates on the cells of a uniform grid, storing the parameters of the intersection between the ray and the yet-uncrossed cell faces’ planes along each axis (called t_x and t_y in Fig. 1). In 3D, it comes the next cell is the neighbors of the current one, sharing the face corresponding to the smallest of the (t_x, t_y, t_z) parameters (that is, t_y in our example). This smallest parameter t_l (where $l \in \{x, y, z\}$) is updated by being added Δt_l , the parameter difference between the intersection points of the ray and the two faces of a cell orthogonal to a given axis l . In our example, it comes the next cell is the one just above the dark gray one, and the new face intersection parameters are $(t'_x, t'_y, t'_z) \leftarrow (t_x, t_y + \Delta t_y, t_z)$. The Δt_x and

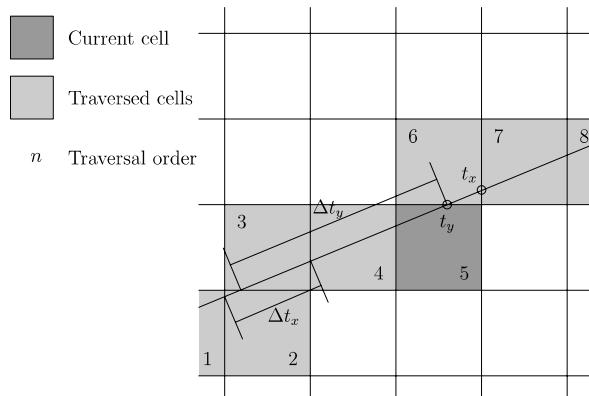


Fig. 1 Geometrical meaning of the variables used by the DDA algorithm

Δt_y parameters are computed once for all for a given ray at the initialization on the DDA algorithm. The resulting traversed nodes are shown in light gray in Fig. 1.

It should be noted that the DDA algorithm can be adapted to use projective geometry [14], which permits a higher traversal accuracy, and is a good substitute for floating-point arithmetics. It is the method we used in the architectures described hereafter.

When performing ray casting, the contribution¹ of each traversed cell is taken into account for the resulting pixel value computation. This is called compositing, and can be implemented in a variety of ways; for our tests, we used voxel-based volume rendering, considering each voxel as having uniform density, and integrating this density along the ray as a compositing rule. We could have used virtually any front-to-back compositing method instead. Also, the compositing unit can be turned into a primitive intersection test unit when the grid is used as a space indexing structure instead of a direct volume representation.

Parametric methods are quite versatile, and very similar techniques can be used for recursive grid traversal [9], the most well known special case being the traversal of octrees [20].

Amongst the non-parametric traversal and compositing methods, a very visually satisfying one consists in sampling the volume along the rays at regular intervals [8, 19]. It is especially efficient in the case of regular grids, where an element can be accessed in constant time. Aside from the oversampling/undersampling issues that may arise, this kind of approach tends to perform poorly quality-wise when used for the projection step of iterative tomographic reconstruction.

2.2 Memory Management

Most of the ray shooting dedicated hardware design with an emphasis on efficient memory management was done in the field of volume rendering. This can be explained by the fact that volume rendering naturally involves very large data sets, up to 1024^3 grids with recent medical appliances for instance. This is why memory bandwidth is likely to become the performance bottleneck of such systems. Therefore, most memory management strategies put a strong emphasis on data reuse and access locality through diverse means.

The cache efficiency of software systems can be optimized thanks to multi-threaded software [6, 11]: each thread deals with a small set of rays and the speed-up comes from the fact that some grid data used by each thread are in the cache memory. This implies rays are shot by coherent packets. Multi-threading based techniques is limited by task context switch, cache trashing and by the available computing power.

¹Which may be emitted or re-emitted light (rendering), density (PET reconstruction), attenuation (X-ray based reconstruction),

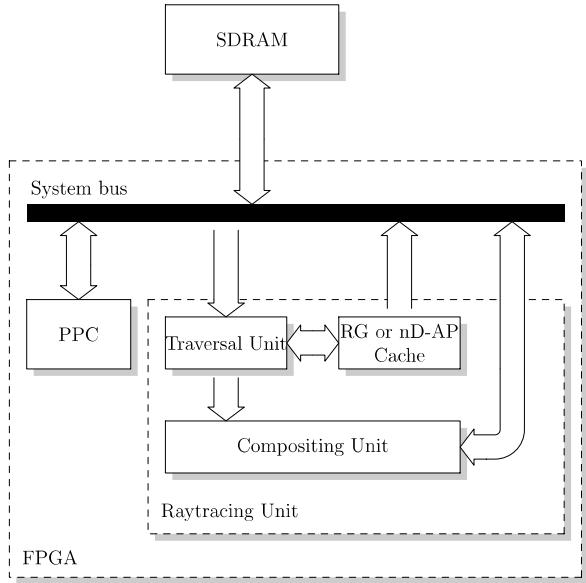
The *Cube* series is an example of regular-grid-sampling-based volume rendering hardware designed to spare this very bandwidth. While *Cube-3* [18] is a regular ray parallel ray tracing architecture, *Cube-4* [19] is based on object-order ray-tracing. The *Cube-4* hardware is designed so as each voxel is fetched exactly once per frame from the central memory. Therefore, it is optimally efficient in terms of memory bandwidth usage, if we admit that every voxel accounts for each frame. However, *Cube-4* has a number of severe limitations, one of them being a scene size limited to 256^3 equally sized voxels. Moreover, the very principle of object-order ray tracing upon which *Cube-4* was built makes perspective rendering implementation impractical; that is why VolumePro, a commercial implementation of *Cube-4*, only supports isometric rendering. Several other problems were underlined by [17].

VoxelCache [8] is a cache specifically crafted for sampling based ray casting, as well. It is small enough to be implemented on reconfigurable hardware. It uses only a single external memory bus, but has an internal 8 memory bank organization. This makes possible to fetch a tri-linearly interpolated sample every cycle. VoxelCache also has a prefetching mechanism, requiring that beams of 4×4 coherent rays are being shot. Despite the fact it was designed for uniform grids, VoxelCache was successfully used for full-octree-based volume rendering as well [22]. This however required the use of off-chip SRAM to keep the performances high, and the caching strategy was shown inefficient for large scenes due to cache trashing. On the contrary, we tried in our approach to use as much as possible inexpensive components found on most prototyping boards, while focusing on arbitrary large sparse octree-like acceleration structures, much more flexible than full octrees.

Since hardware systems benefit of uniform memory accesses, a solution is to allow only parallel rays, possible rendering a given volume slice by slice to produce an illusion of perspective. Such a strategy underlies volume rendering on commodity PC textured rasterization hardware [10, 21]. Some parallel rays are sampled at the same interval to provide “plane parallelism”. Perspective volume rendering is then simulated by a perspective transformation of the resulting image. This solution is very popular for visualization because it is fast but it suffers of too low accuracy for tomographic image reconstruction.

More recently, programmable graphics hardware has been used for volume rendering. It is quite suitable for brute force ray casting through uniform grids [16]: the built-in texture cache can handle 3D scenes efficiently, and the Single Instruction, Multiple Thread (SIMT) programming model is suited for casting rays in beams, ensuring reference locality. There are also other ways to perform volume visualization on GPGPUs, for instance through the well known marching cubes algorithm [13], which converts a volumetric scene into polygons before displaying it. Nowadays, GPUs can afford not only to display the generated polygons, but also to build them from the volumetric data in the first place [16], which used to be done on the CPU. The drawbacks of GPUs include a high power consumption, small per-core on-chip memory quantities making them unsuitable for recursive algorithms with high stack-space requirements, and a loss of efficiency in case of diverging code paths between threads, bad load distribution, and so on. Implementations of ray tracing algorithms are rather prone to such pitfalls.

Fig. 2 An overview of a complete rendering system



3 System Architecture

The architecture we propose is composed of two main parts: an adaptive and predictive cache for either uniform or recursive grids and a traversal unit, capable of determining the sequence of grid cells traversed by each ray of a coherent beam.

The generated sequences are meant to be communicated to a compositing unit. Since we chose visualization as an application, once a ray of the beam ends, its accumulated value may be written to a frame buffer. Figure 2 presents an overview of a whole rendering architecture as implemented on a prototyping board.

The traversal unit and the cache were designed and synthesized with the Xilinx Virtex 4 technology as a target. Depending of the number of units and the size of the FPGA, the results were obtained either by actual on-board runs, or CABA simulations.

4 The *n*D-AP Cache

The 3D-AP Cache is an instance of the *n*D-AP Cache which aims at caching multi-dimensional data as originally described in [15]. Also it performs pre-fetching by estimating the future zones of data the processing unit is supposed to fetch. As shown Fig. 3(a), the *n*D-AP Cache provides a virtual interface to the computing unit that issues multi-dimensional indexes in the data structure. The *n*D-AP Cache performs both the mappings between indexes and the external memory addresses and the internal memory addresses.

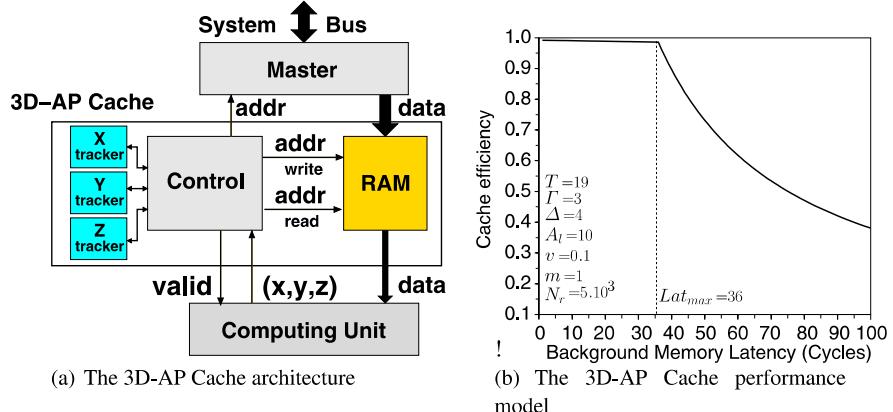


Fig. 3 The 3D-AP Cache aims at performing prefetching in multi-dimensional arrays

The pre-fetching mechanism relies on a tracking of indexes on each dimension. The trackers try to estimate the zones of indexes the computing unit may fetch from an analysis of the past fetches and a prediction model. The n D-AP Cache architecture is modular and several kinds of trackers are available. In the following, we consider a first order SC tracker which prediction model makes the hypothesis that the indexes of the fetch sequence evolve as a compound of a fast displacement around a low speed displacement. The trackers compute the estimated means of the indexes on each axis and request the control unit to grab a new zone of data when the estimated means cross a guard zone. This mechanism ensures that the new zone will be uploaded before the references reach the cached zone boundaries as demonstrated in [12] (see Fig. 3(b)). This prefetching is enabled when the 3D-AP Cache is tuned in such a way that the cut-off frequencies of the estimators and the different thresholds (T , Γ , Δ) fit the average speed of the indexes (v), their local amplitude (A_l) and the background memory latency (see [12] for more details). Above a threshold latency the cache efficiency drops but the cache parameters may be set to fit a given latency.

5 Uniform Grids

5.1 Uniform Grid Traversal

The traversal pipeline for regular grids is shown in Fig. 4. The RCPG-U unit (Ray Casting in Projective Geometry-Uniform grid) is made of a traversal unit connected to a 3D-AP Cache. The Line integral unit implements the compositing processing to compute sinograms in tomography applications. The traversal unit gets parameters from the PowerPC processor and manages the phase-locked propagation of a beam of rays. As can be seen, the memory references to the volume do not depend on

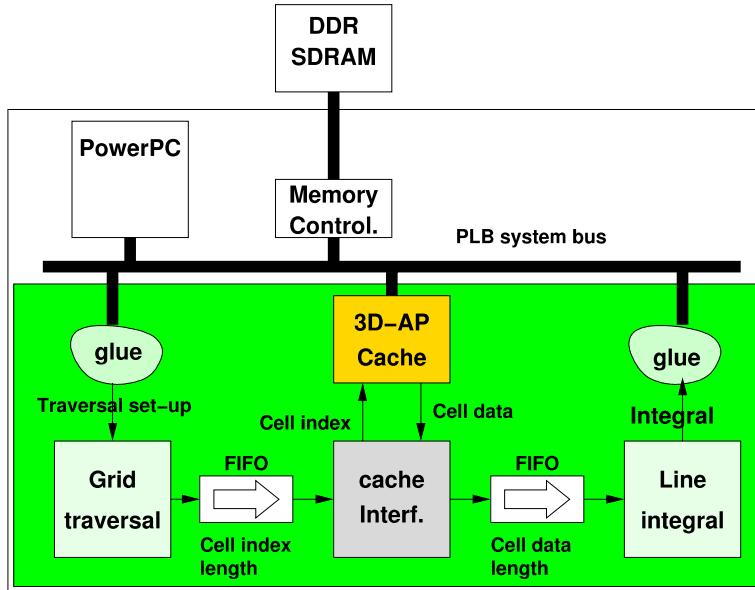


Fig. 4 RCPG-U pipeline and cache interface

the fetched data, and could theoretically be determined ahead of time. We chose nonetheless not to use this specificity, not to increase the hardware complexity of the traversal unit. This allows us to test the behavior of the cache while providing it with no clues regarding the future accesses.

In order to obtain good performances even if caching very small parts of the volume, we rely on “phase-locked” traversal. The phase-locked propagation enables a virtual loop over the traversed cells. Indeed, as shown in Fig. 5, the RCPG-U

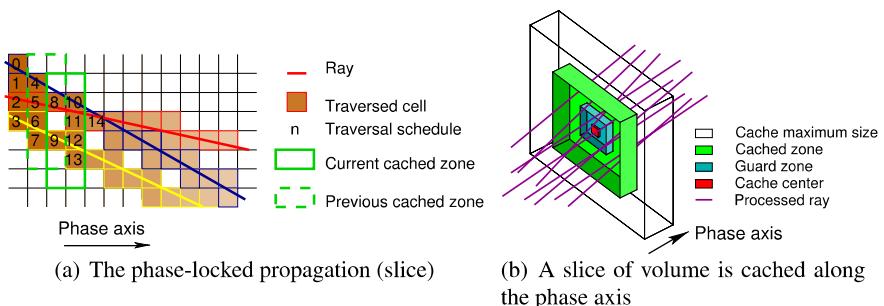


Fig. 5 The phase-locked propagation of a beam of rays improves the spatial and temporal locality of grid traversal

algorithm is synchronized over a set of rays to propagate them together along a main direction, orthogonal to a phase plane. This direction is collinear to the major axis which is the closest to the overall direction of the beam of rays, and is pre-computed at the same time the initial states of the rays of the beams are generated (typically, by a hard or soft processor). For each ray, a step of the RCPG-U algorithm returns the next cell to cross. This is iterated while the resulting cell remains in the phase plane. When all the rays of the set are out of the phase plane, then the phase is updated to the next one. The process loops until all the rays exit the volume. The phase acts as a virtual loop, the innermost one being the ray index. Since it is known that the *phase* moves in only one direction and that a lot of consecutive accesses will request cells sharing the same phase, we can afford to cache only a very narrow slice of the scene along the phase axis. Along the two other axes, the cached zone needs to be just broad enough to contain all the rays of the beam.

5.2 Uniform Grid Caching

The experiments on Ray Casting have raised the need of more efficient tracking mechanisms and new cache behaviors. To manage different classes of fetch sequences, multi-mode trackers implement together several tracking mechanisms that may be dynamically selected. The mapping of fetch indices to the internal cache addresses can also be dynamically chosen to fit different sizes of cache. The trackers are joined together to allow more complex cache zone displacements: a displacement request from a tracker makes the other trackers to center on their estimated center. Furthermore, the user module can now select the priority of misses over cache updates.

The dynamic selection of the priority of misses is efficient especially when the misses are faster than the uploading of new zones into the cache. At a first sight, updates of the cached zone have a higher priority than misses because it prevents the user module to always request misses if the trackers are too slow. But fetches can evolve differently on each dimension and misses can have different priorities depending on the cache geometry. At the time the user module requests a high priority miss, the cache update is interrupted, the miss is served and the update continues over.

The phase locked propagation of rays benefits from these improvements. The multi-mode tracking is necessary because the virtual loop on the propagation direction can be efficiently tracked by a linear tracker while the other dimensions are tracked by statistical trackers. The misses have high priority along directions perpendicular to the phase direction and have low priority along the later. Indeed, the phase increases (or decreases) uniformly faster than the other directions in the average case. The worst case is when the rays have a 45-degree angle with all or some of the main axes.

6 Recursive Grids

6.1 Caching the RG Data Structure

We generalize the above described pipeline and cache to the traversal of a generalization of octrees [5] we call recursive grids (for the lack of an established name). As their name somewhat implies, recursive grids are sparse hierarchical structures, and therefore induce a dependency between the memory access sequence and the data fetched from memory. Predictive prefetch mechanisms become therefore unavoidable in order to reach acceptable performances.

After a description of recursive grids, we will show how 3D-AP Caches can handle them, and which access patterns should be used to maximize caching efficiency.

6.1.1 Recursive Grids

A $2^n \times 2^n \times 2^n$ recursive grid is a tree with the following characteristics:

- each node is a cube
- each node has either a datum, or 2^{3n} equally-sized children

As a consequence, every node of the recursive grid has the same size as any other node on the same depth. We call the leaf nodes voxels. Figure 6(a) shows a $4 \times 4 \times 4$ recursive grid, with two non-voxel children at the root node. It is clear that, under our formulation, octrees are $2 \times 2 \times 2$ recursive grids.

$2^n \times 2^n \times 2^n$ recursive grids, especially for “moderately large” values of n such as 2 or 3, hold several interesting properties few other hierarchical acceleration structures do. While still having the benefits of sparse hierarchical structures, they also offer more regularity than most of them, allowing more efficient caching. It can be noticed that recursive grids are a subset of adaptive grids [9], and hence inherit of most of their advantages when it comes to their traversal.

For our tests, we chose to encode a $4 \times 4 \times 4$ recursive grid by an array of at most 32768 nodes, the node 0 being the root. The exact coding of a node is itself

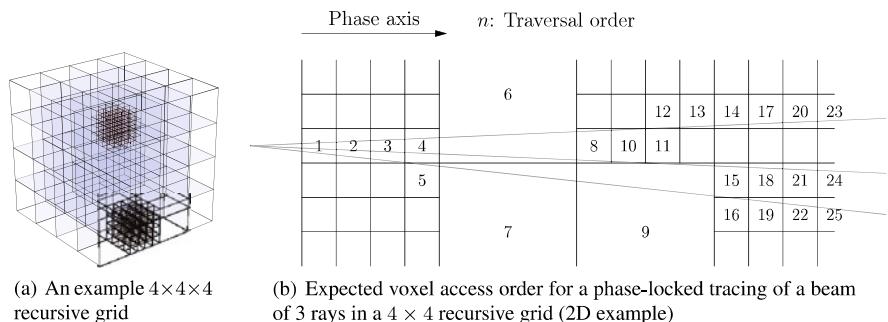


Fig. 6 Recursive grids and their phase-locked traversal

that of an array of 64 words of 16 bits, each coding for a child (15 bit datum and a 1 bit flag, determining if the child is subdivided or not). The datum a child contains is either a density, or index of the child node.

6.1.2 RG Cache

The Recursive Grid (RG) Cache, described in Fig. 7, aims at caching a part of the recursive grid by exploiting the spatial coherency of references. Furthermore, it provides a virtual interface to the processing unit. This means that the processing unit issues a 3D coordinate (x, y, z) together with a resolution level and the cache provides the corresponding datum, preventing the processing unit to manage the tree structure. The RG cache uses the n D-AP Cache as a basic block.

The proposed strategy is to cache each level of resolution with an n D-AP Cache and to perform prefetching in the tree. Indeed, when a reference occurs, it is likely that the next reference is at a close coordinate, either at the same, upper or lower resolution. Each n D-AP Cache is in charge of tracking references at a resolution and the neighboring ones.

The tree manager (TM) unit returns parts of the scene requested by the n D-AP Caches and maintains a coherent state of the caches at different resolutions. Each time a cache requests a part of the scene, the TM reads the corresponding data at the upper level to determine if the requested block is a leaf or a child node. In the later case, the TM fetches the data at the obtained address to fill the n D-AP Cache. As a consequence, the n D-AP Cache is slightly modified to allow the TM to read an n D-AP Cache concurrently with reads at the processing unit interface. Also, the cached zone at level n has to be inside the cached zone a level $n - 1$ to maintain cache coherency. Without this constraint, when the cache n would request a part out of the zone in the $n - 1$ cache, it would be too slow to get the data by traversing the tree from the root node.

Each of the cache is optimized to manage data at its level of resolution. The size of embedded cache memories fits the level of resolution to save space. For example, because the level 1 cache contains only $4 \times 4 \times 4 = 64$ data, it doesn't need trackers and has a simplified control management.

6.1.3 Improving Reference Locality

Just like we did for uniform grid traversal, we use a phase-locked propagation principle in order to keep the accesses coherent and the cached zone minimal.

More specifically, once the phase axis chosen, we propagate rays in a way that ensures the cell accesses during traversal are ordered by their coordinate on the phase axis. An example of such an access sequence is given in Fig. 6(b). Section 6.2.2 gives more details as how this can be implemented. Just like previously, the cached zone should be narrow along the phase axis.

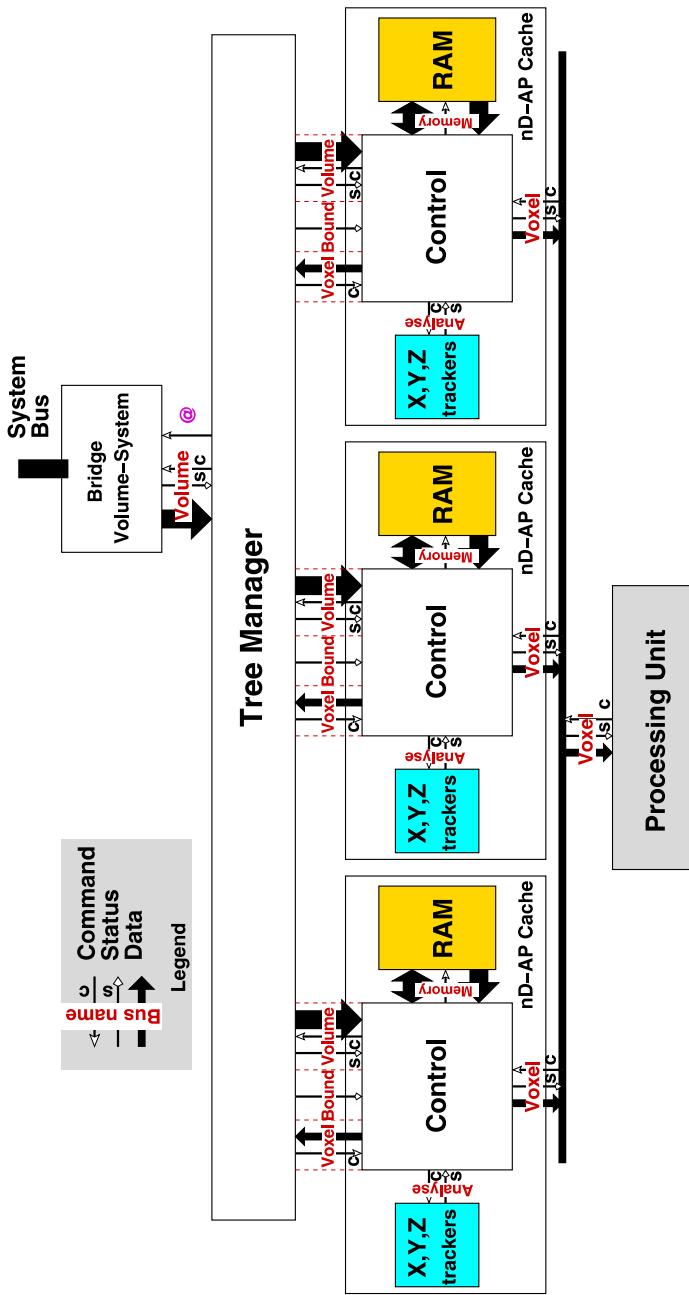
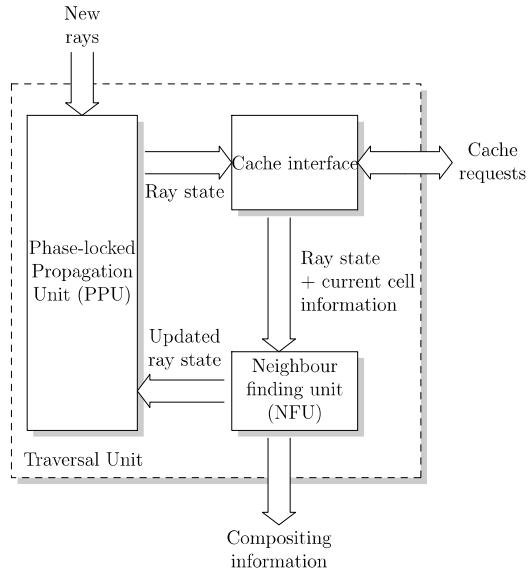


Fig. 7 Recursive Grid (RG) Cache

Fig. 8 Architecture of the recursive traversal unit



6.2 Recursive Grid Traversal

We implemented a hardwired traversal unit for efficient ray shooting through recursive grids. It works on beams of up to 256 rays. Figure 8 shows its structure. Its three main parts are:

- the phase-locked beam propagation unit, which determines the order in which grid nodes are fetched so as to minimize the probability of cache misses; do to so, it holds the states of all the rays of the current beam in such a way it is able to ensure they all propagate with the same speed along the phase axis
- the RG Cache interface, which performs cache requests while pipelining ray parameters linked to the request
- the neighbor finding unit, which determines the next node a ray should access and its updated parameters, based on the response from the cache and the former ray parameters

The cache interface simply contains two ray state holding FIFOs, which minimize the impact of small pipeline bubbles. Therefore, in the rest of this section, we will focus on the two other components.

6.2.1 Neighbour Finding Unit

We use a slightly modified version of the DDA algorithm for neighbor finding, in a fashion very similar to that presented in [2]. As we need to be able to vertically traverse the tree as well, we adapted the principles exposed for octrees in [20] to $2^n \times 2^n \times 2^n$ recursive grids, by considering each of our nodes as an n -level perfect

```

1 ray_state. $\vec{r}$   $\leftarrow$  unitary direction vector of our ray
2 ray_state.t  $\leftarrow$  current cell entry point parameter
3 ray_state.( $t_x, t_y, t_z$ )  $\leftarrow$  border intersection parameters
4 ray_state.( $\Delta t_x, \Delta t_y, \Delta t_z$ )  $\leftarrow$  parameter increments
5 ray_state.( $p_x, p_y, p_z$ )  $\leftarrow$  current cell absolute position
6 ray_state.depth  $\leftarrow$  depth of the current cell
7 max_depth  $\leftarrow$  maximum depth of the tree

```

Listing 1 Variables characterizing a ray state

octree. On a side note, this approach works for adapting pretty much any algorithm working on octrees to recursive grids, and without increasing its asymptotic cost.

To sum up everything, let us consider that a ray propagation state is fully characterized by the variables of Listing 1. The “parameters increments” are the differences between the parameters of the intersection points between the ray and two opposite faces of our cell, for each of the three possible such pairs of faces. The absolute position of the current cell is given in units corresponding to the size of cells located at a given maximum depth (the constant max_depth). If this depth is 6, for instance, the maximum detail level of a $4 \times 4 \times 4$ recursive grid will be the same as that of a 4096^3 uniform grid.

Let’s suppose without loss of generality that our ray propagates in the positive direction along each axis.² Our neighbor finding algorithm is then summarized by the pseudo code of Listing 2. Basically, our traversal unit advances to the next cell if the current cell is a leaf (this takes one cycle), or dives further if it is an internal node (n cycles for 2^{3n} grids). Figure 9(a) presents the unit’s organization. As the diving and advancing modes are mutually exclusive, there is hardware reuse between them which does not appear on this figure for the sake of clarity.

One should pay attention that the input and output data of the neighbor finding unit may grow moderately large depending on its exact coding. What call *ray propagation state* a structure composed of the variables seen in Listing 1, at the exception of the direction vector of the ray (which does not matter for the traversal assuming all the other variables are known). Assuming max_depth = 5, $n = 2$, and 32 bits per parameter, we need 260 bits per ray state. For each ray, those parameters need to be initially computed from the ray geometry before they are fed to the propagation unit. It involves obtaining the intersection points parameters between the ray and each of the faces of the root node cube.

6.2.2 Phase-Locked Ray Beam Propagation

The phase-locked propagation unit (PPU), as shown in Fig. 9(b), manages the indexes of rays to enter the propagation pipeline. The indexes of active rays are stored

²Indeed, if it is not the case along one or more axes, we can bring ourselves back to the case where it is by taking as absolute cell position the one’s complement of the actual cell position along those axes. Of course, the “correct” position must still be used for the memory accesses. This strategy is suggested in [20], where the reader may find extensive detail of such an approach.

```

1 if (current cell is a voxel) then
2     send ray and cell data to the compositing unit
3
4 // We need to find the next cell (on the same depth or above);
5 // determining which direction the next cell is:
6 k | tk = min(tx, ty, tz)
7
8 // When exiting a cell, undividing as far as necessary:
9 while pk[(h-1)..(h-n)]='1..1' with h=n*(max_depth-depth)
10    if depth > 0
11        depth ← depth - 1
12    else
13        traversal is over for current ray
14        for m in 0 to (n - 1)
15            h ← n * (max_depth - depth) - (m + 1)
16            foreach l ∈ {x, y, z}
17                if pl[h] = '1' then
18                    tl ← tl - Δtl
19                    Δtl ← 2 * Δtl
20
21 // Finally, advancing to the next cell:
22 pk ← pk + 2 ** (n * (max_depth - depth - 1))
23 tk ← tk + Δtk
24 else
25 // We need to dive further
26 for m in 0 to (n - 1)
27     h ← 2 * (max_depth - depth) - (m + 1)
28     foreach l ∈ {x, y, z}
29         Δtl ← Δtl / 2
30         tl ← tl - Δtl
31         if t > tl then
32             tl ← tl + Δtl
33             pl[h] ← '1' // hth bit of pl to 1
34         else
35             pl[h] ← '0' // hth bit of pl to 0
36     depth ← depth + 1;

```

Listing 2 Neighbour finding algorithm

in the “Index RAM” in a way to manage efficiently the synchronization of propagation along the beam phase. Hereinafter a “ray” stands for its index in the “State RAM”. The “Index RAM” is divided in several ranges to manage the phase synchronization over the different resolutions. The diving of rays being one of the most tricky behavior to deal with.

The beam phase is memorized in the PPU. It is updated when all the ray phases³ are further than the beam phase. To that end the memory is divided into in-phase rays and out-of-phase rays.

An in-phase ray is sent to the “Insertion Unit” to update its next state. On its way back, the ray is either still in-phase, if the propagation occurred on an other axis, or out-of-phase in other cases. When all the rays are out-of-phase, the PPU increments the beam phase and swaps the out-of-phase rays into the in-phase. This is actually done when all the rays sent to the propagation pipeline are back. That for, the number of processed rays is counted. Rays exiting the scene are counted back but not inserted in the “Index RAM”.

³A ray phase is the ray coordinate along the phase axis.

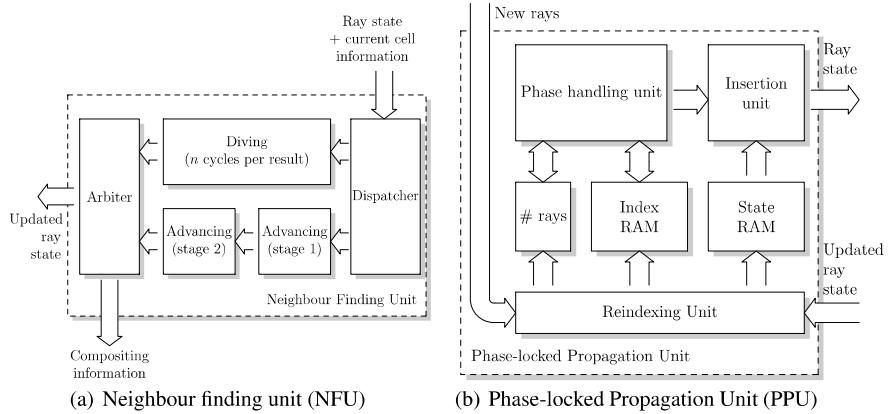
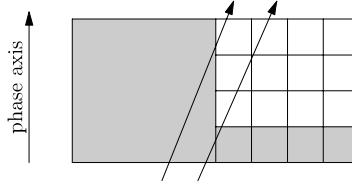


Fig. 9 Architecture of the two main elements of the recursive grid traversal pipeline

Fig. 10 In some circumstances, diving may out-phase some of the rays of a beam, while keeping others in-phase (gray voxels are on the beam phase)



Also, the phase synchronization have to be performed on all the levels of resolution. So, the “Index RAM” in-phase and out-of-phase parts are again divided in sub-parts for each level of resolution. Higher depth rays are first sent to the propagation pipeline and the lowest one are then sent, until there are no more in-phase rays at any resolution.

At last, rays diving into a higher resolution have to be sorted according to the beam phase. Indeed, the diving may cause a ray phase to be higher than the current one because the entry point in the higher resolution node may be anywhere on the child border (see Fig. 10). To speed-up the phase sorting, rays are stored in the “Index RAM” according to their relative phase in a node. This relative phase is simply the n bits starting at the $n(\max_depth - d)$ bit of the ray phase, where d stands for the current depth of the ray.

Hereof, we conclude that the “Index RAM” is a memory of $(\max_depth + 1).2^n \cdot \max_rays$ words of $\log_2(\max_rays)$ bits. A more tolerant phase synchronization allowing a 2^{dn} phase deviation would need a $2 \cdot \max_rays$ “Index RAM” but it would increase the cache’s memory.

7 Results

This section provides some results about the cache efficiency for a set of applications of the phase-locked ray tracing in uniform (RCPG-U) and recursive grids

Table 1 Area of the RCPG-U unit and 3D-AP Cache for a Xilinx Virtex IV technology. Percentages are relative to the V4FX60 device capacity. A group-level pipeline of 4 RCPG-U units shares a single 3D-AP Cache

	Logic cells	DFF	DSP48	BRAM (KB)
RCPG-U pipeline	1739 (3%)	996 (2%)	6 (4%)	9 (0.7%)
3D-AP Cache	1299 (2.5%)	365 (0.5%)	0	8 (0.7%)

(RCPG-R). We show that in both cases the cache performance highly depends on the geometry of the rays belonging to a beam. The distance between rays directly impacts the data-reuse ratio because a data is little reused when the rays do not cross the same voxels. At the opposite, the cache mechanism of the RCPG-U and RCPG-R systems are more efficient when a grid is visualized with a larger zoom because a voxel is crossed by the rays corresponding to neighboring pixels of the rendered image. The 3D-AP Cache shows to be more efficient when the RCPG-U unit is used to compute a sinogram. This later application computes the volume integral along each Line of Response (LOR) that connects a pair of detectors of a tomographic camera. The LORs of a sinogram form a 4D array. To increase the data-reuse ratio, the computation of the sinogram is split in a set of 4D sub-blocs. The performance of the RCPG-R system is the most difficult to measure because it also depends on the structure of the recursive grid. The RG Cache behavior depends both on the geometry of a beam of rays but also on the traversed levels. It happens that a few levels are traversed where the grid data is uniform. Then, the RG Cache efficiency may fall because the set-up time to initialize the traversal is higher but the total time to traverse the recursive grid is much smaller than a uniform grid traversal.

The RCPG-U system was implemented in a Virtex II Pro prototyping board and some details of its area occupancy and 3D-AP Cache performance are provided. The provided measures may be extrapolated to any FPGA or ASIC technology though. As a proof of concept, the RCPG-R is implemented in VHDL-RTL and the RG Cache performance is measured by simulation.

7.1 Hardware Complexity

7.1.1 Uniform Grid Traversal

The RCPG-U is designed in VHDL-RTL and implemented in prototyping board with a Virtex II Pro FPGA. The board runs at 30 MHz but this is not an issue as logic synthesis of the VHDL code for the Virtex 4 technology reports a clock frequency up to 200 MHz. A memory simulator allows to measure the 3D-AP Cache performance for different background memory configurations.

Table 1 provides the complexities of the RCPG-U pipeline and of the 3D-AP Cache. These results are obtained in fixed point arithmetic, with the bit widths set

to reach the accuracy needed by a tomographic reconstruction application. The 3D-AP Cache seems of the same complexity as the RCPG-U unit but a group-level pipeline up to 4 RCPG-U units shares a single 3D-AP Cache. Indeed, the RCPG-U unit has a pace of 3 to 4 clock cycles between each fetch. In this later configuration, the 3D-AP Cache occupies the third of the whole system complexity and the highest throughput is reached when the 3D-AP Cache releases a datum each clock cycle.

An interesting point is that only the BRAM size depends on the maximum size of the cached zone. The area of the control unit of the 3D-AP Cache is almost independent on the cache memory.

7.1.2 Hierarchical Grid Traversal

The RCPG-R unit is implemented in VHDL-RTL and has been validated has a proof of concept. The main objectives were to show in one hand that the phase-locked synchronization is efficient and on the other hand that the most complex part of the architecture are integrable. Table 2 gives the complexity of some parts of the RCPG-R. Most of the area is used by the index RAM and the ray state RAMs. Their sizes are directly linked with the quantity of rays in a beam and the levels of the recursive grid.

7.2 Cache Efficiency

Several criteria are used to measure a cache efficiency. In the following, we focus on the time performance and the cache efficiency is measured as the ratio between the number of fetches divided by the number of clock cycles to get all the data. Other measures such as the system bus occupancy, the re-use ratio and others are available but will not be discussed to gain in clarity. The measures performed by simulation or with the prototyping board are equivalent, but the prototype is faster.

Table 2 Area of some parts of the RCPG-R recursive grid traversal unit for a Xilinx Virtex IV technology

	Logic cells	DFF	DSP48	BRAM (KB)
Hierarchical Neighbour Finding	5639 (11%)	1474 (2.9%)	6 (4%)	0
Hierarchical Phase-locked Propagation	1035 (2.1%)	727 (1.4%)	0	14.6 (2.7%)
Tree Manager	2246 (4%)	286 (0.5%)	0	0

In order to study the effectiveness of pre-fetching, the efficiency is measured for a set of memory latencies. This later is the time to get a burst of data from the background memory. The latency includes the arbitration time of the system bus, the latency of the memory controller and the latency of the external memory device.

As a worst case hypothesis, we suppose that the latency is paid for each burst request (no pipeline). An update of the 3D-AP Cache is split in a set of bursts corresponding to lines in the x axis. Better results are achievable when the latency is paid only once for a 3D-AP Cache update.

The measures show that the pre-fetch mechanism is efficient for a large set of latencies depending on the application (visualization, sinogram computing), the viewpoint and the quantity of rays in a beam.

7.2.1 Cache Efficiency of the Uniform Grid Traversal

Visualization For the visualization application, ray casting is used to visualize a 3D grid on a 2D focus plane: each pixel of the resulting image is obtained by composing the voxels traversed by the ray issued from the pixel and passing through a viewpoint.

From Fig. 11 we clearly see that the pre-fetching mechanism allows to overcome the memory bottleneck: the efficiency is high before a latency threshold and

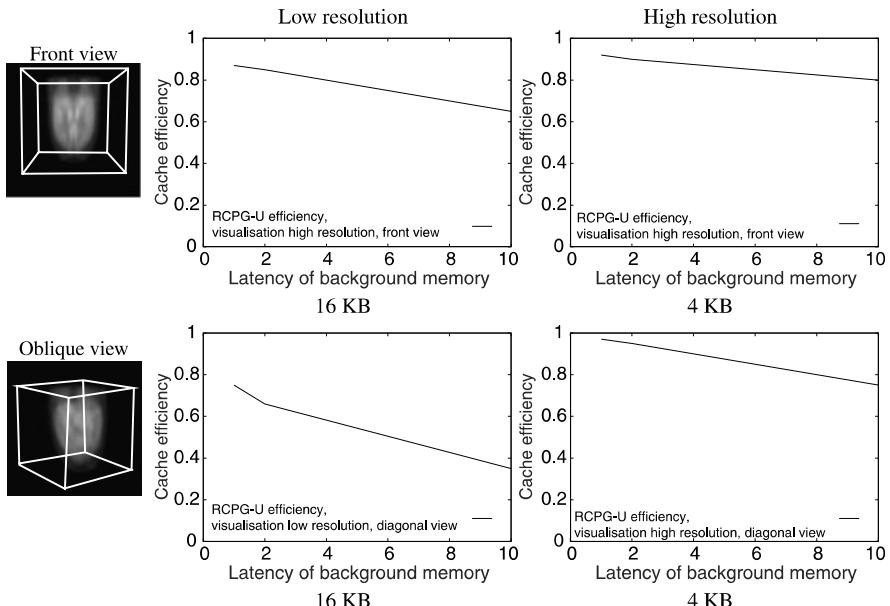
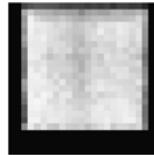


Fig. 11 RCPG-U visualization efficiency: the average 3D-AP Cache efficiency of the group level pipeline for the visualization application; The numbers provide the cache size

Fig. 12 RCPG-U visualization efficiency: the 3D-AP Cache efficiency for each beam of rays. The cache efficiency exceeds 90% for some tiles (white is 100%)



drops above this threshold. Figure 11 provides the average cache efficiency along the memory latency for two viewpoints and two resolutions. The computed images are divided in tiles of 12×12 pixels, to form beams of 144 rays: at low resolution there are 22×22 tiles and 44×44 tiles at the high resolution. The figure also provides the size of the cache memory. The background memory is 64 bit wide and a word contains a $2 \times 2 \times 2$ part of the volume.

The efficiency corresponding to the computation of each beam is provided in Fig. 12. Each block of that image corresponds to a beam and gives the efficiency at which are performed the fetches of all the voxels traversed by all the rays in the beam. A lower efficiency on the borders is due to the fact that these beams contain fewer rays and a small part of the volume is traversed when the rays hit the volume's edge.

The RCPG-U pipelines can be parallelized at two levels: within a group parallel level, some traversal units share a 3D-AP Cache, and at the cluster level, a set of group level pipelines is connected to a main 3D-AP Cache. In the later configuration, there is a cache hierarchy, with leaf 3D-AP Caches grabbing data from the main cache. Measures on a cluster-level parallel architecture show that two groups of pipelines sharing a higher level cache enables a 1.5 speed-up at the low resolution and 1.8 at the high resolution. The speed-up also depends on the view-point, the geometry of rays and the size of the tiles.

From these measure we conclude that the 3D-AP Cache allows to exploit the data-reuse and performs pre-fetching efficiently.

Sinogram Computing Because of a higher data-reuse it is more efficient to compute a sinogram than to visualize a volume, as shown in Fig. 13. The measures are

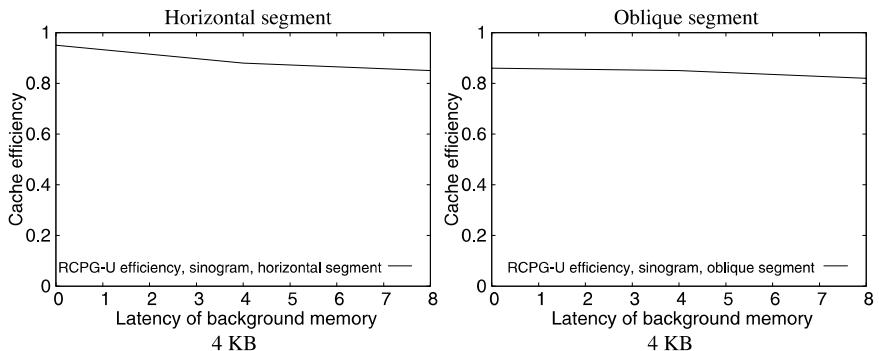
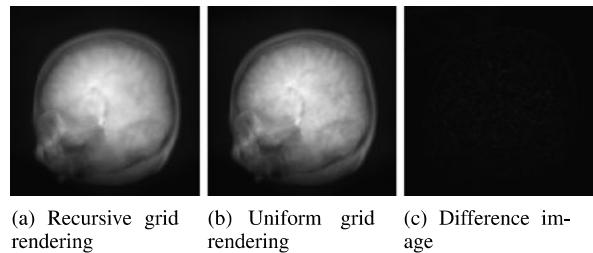


Fig. 13 RCPG-U sinogram efficiency: due to a high data-reuse ratio, the cache efficiency to compute a sinogram is higher

Fig. 14 Rendering of a recursive grid at 128×128 resolution; The PSNR is 46 db but uniformly distributed



performed to compute a sinogram simulating an “ECAT EXACT HR+” PET camera: it is a cylinder of 32 rings and each ring has 576 detectors. The RCPG-U unit is used to compute the integrals of the lines blending two detectors. The measures are performed in two scenarios: the start and end detectors belong to a single ring (horizontal segment) and in different rings (oblique segment). The rays to compute a sinogram are split in beams of 256 rays.

The pipeline has an efficiency higher than 80% with a typical background memory latency (3 to 4 clock cycles). The simulations show that a cluster of two group-level pipelines still has an efficiency about 85% and provides a speed-up of 1.9 compared to a single group-level pipeline. These very good results are due to the fact that the rays of a 4D tile are crossing together in a “tube” and their density is higher than in the case of visualization.

7.3 Cache Efficiency of the Recursive Grid Traversal

The RCPG-R architecture is evaluated by visualizing a reconstructed 256^3 MRI data from the PET-SORTEO database.⁴ Since the data is provided as a uniform grid, a $4 \times 4 \times 4$ recursive grid is built by merging adjacent voxels which are about the same density (i.e., within 37% of the dynamic range). Since each node has 64 children, this criterion does not lead to severe losses in quality.

Figure 14 shows a render of this recursive grid performed by the RCPG-R unit. The resolution of the rendered image is 128×128 . The quality is enough for a visualization application and the number of traversed cell is 6 times less.

Similarly to the regular grid traversal the RG-Cache efficiency increases with the image resolution because the data-reuse ratio is higher then. But, as some rays do not hit the highest depth in the recursive grid, the data reuse ratio of the intermediate depths is higher than the uniform grid ones, even for a low resolution image.

The plots in Fig. 15 show the efficiency to render a 128×128 image for different memory latencies. The efficiency is above 80% for a 4 clock cycle latency for most of the tiles and drops when the latency increases. The latency now corresponds to the clock cycles to wait before getting all the data of a $4 \times 4 \times 4$ sub-grid. Then the

⁴<http://sorteo.cermep.fr/>.

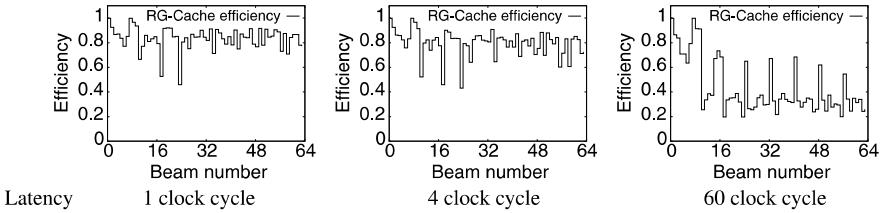


Fig. 15 RCPG-R visualization efficiency: the efficiency is plot for each tile of the rendered image

128 bytes (64 data) of the sub-grid arrive in 16 clock cycles thanks to a 64 bit width background memory.

These results show that the RG-Cache efficiently pre-fetches parts of the tree on time for a typical memory. Furthermore, the virtual interface to the cache prevents the computing unit from computing the effective addresses of the tree data. Hence, considering a latency of four, the RG-Cache allows to fetch a datum each 1.25 clock cycle in average. Eventually, the recursive grid traversal needed six times less memory references than the uniform one, which results in as much speed-up.

7.4 Discussion

Comparison with other solutions is not straightforward because all the architectures we can find in the literature implement a grid sampling algorithm, whereas the proposed algorithm implements an exact grid traversal ray casting. Furthermore, most of the systems implement early-ray termination, which is useful for visualization because unseen voxels do not need to be rendered, but our algorithm performs a complete grid traversal. Early ray termination could be added to our system but it is likely that stopping some rays would lead to too-much instability for a low global gain.

Reference [8] reports simulation results of a cache designed for a volume rendering hardware architecture. For each sample, the pipeline fetches 8 data from the VoxelCache which is a full associative cache that contains blocs of voxels. The VoxelCache holds 512 lines each of 64 voxels to make a 32 KByte memory. The simulation results of [8] show a 90% pipeline utilization but the size of the rendered volume is limited to 128^3 voxels because the VoxelCache is trashing when it cannot hold all the voxels along a line.

Our solution reaches an equivalent throughput with 4 times less memory and without associative memory: it is space-saving and has a better usage of FPGA resources. Also, a high level of parallelism is allowed, depending on the geometry and the density of rays. At last, the 3D-AP Cache and the “phase locked” RCPG allow any size of grid and the system is scalable.

To our knowledge there is no equivalent to the recursive grid traversal strategy presented in this chapter. The closest hardware architectures presented in the literature addresses full octrees [22]. The later are different from sparse octrees in the

way that nodes are always data and never pointers to a higher resolution. They are much like multi-resolution volumes and their traversal is done at a given resolution. Hence, the traversal of full octrees looks like a regular grid traversal, the resolution staying constant along a ray.

7.5 *Improvements*

Getting higher performances could be done by some improvement of the presented concepts and some new strategies may overcome some bottlenecks.

First, many of the architecture's parameters could be set to fit a particular use of the architecture or to take into account some integration constraints. As an example, the ability to pipeline the burst requests on a system bus would improve the performances dramatically. The placement of the volume data in the memory is also of high importance to reduce the relative latency by enabling longer bursts. As an example, a memory data word may contain a sub-volume and larger parts of volume may be stored at some contiguous addresses. The level of cluster-parallelism could be increased by tuning the bandwidth between the higher level cache and lower level caches. This should allow a better overlap of the computation with the update of the lower level caches.

The recursive grid traversal could be improved by a better synchronization of the different cache depths. Due to the recursive grid structure, the $n - 1$ -depth cache stores the pointers needed to update the n -depth cache and the two levels have to agree on a common zone to keep the cached tree coherent. At the moment, the $n - 1$ -depth cache constrains the n -depth cache and a back-pressure mechanism would allow a better pre-fetching mechanisms along several levels of the grid.

The automatic setting of the cache parameters is also a major challenge. Some of the cache parameters should be set dynamically to get higher performances, especially to manage special cases such as small set of rays at the border of the image. Currently the cache parameters are set manually, for a given beam of rays and a memory latency, and are expected to fit all the beams. Dynamically setting the cache parameters should allow a better cache efficiency for the different ray casting applications. Also, a dynamic setting of the parameters would allow a higher efficiency for different memory latencies.

8 Conclusion

In this chapter, we have presented a typical example of Algorithm Architecture Matching focused on the optimization of the memory hierarchy. It illustrates clearly that a co-design of the algorithm, the IP and the memory hierarchy leads to better results than a basic addition of standard IPs.

In a first step, the original grid traversal algorithm is transformed to exhibit more spatial locality than its theoretical expression. Indeed, a “virtual” loop is enabled

by the phase-locked propagation of blocs of rays. Then the memory references are coherent along this phase due to the initial geometrical coherence of the rays belonging to a beam. Splitting the computation of the result in blocs (or tiles) leads to coherent memory references. The phase-locked propagation is efficient both for the regular and recursive grid traversal.

In a second step, this transformation is shown to fit the 3D-AP Cache reference model. The 3D-AP Cache shows to be efficient at prefetching the volume data. The measures performed on an emulation board and by simulation have shown that the cache efficiency highly depends on the target application and the density of rays. For example, in the case of the visualization application, the efficiency drops when the volume is sub-sampled. This is not an issue as a volume of lower resolution could be used to render sub-sampled images.

The measures are performed in a worst-case situation. Indeed, on one side the background memory is supposed not to pipeline burst requests and, on the other side, the processing unit is able to perform a reference each clock cycle. An actual implementation may relax these constraints by allowing pipelining thanks to modern multi-bank memories. Also, the frequency of the references may be lower when the grid traversal would be implemented by software. Even better efficiencies would be reached in a less constrained implementation.

More generally speaking, the transformation applied to the original algorithm leads to a compromise between the efficiency of the memory hierarchy and the IPs area. Indeed, the phase-locked propagation needs some internal memories to store the intermediate results of the propagation. The phase-locked recursive grid traversal algorithm is the most memory hungry because it needs also to store the stacks of traversal of each rays. The proposed architecture is highly configurable to meet different target optimization criteria and fit integration constraints.

Further improvements and trade-offs are available to reach a better efficiency of the recursive grid traversal. Some new strategies to pre-fetch parts of the recursive grids are needed. The difficulty is to manage coherently the different levels of resolution.

We believe that the management of recursive data structure is of a great importance for many applications. For example, multi-resolution images are used in computer vision, video compression and 3D rendering. As the data traffic is one major of source of power consumption, optimizing the management of such data structures would enable complex image processing algorithm now restrained to desktop appliances to be implemented in embedded systems.

References

1. Akenine-Möller T, Haines E, Hoffman N (2008) Real-time rendering, 3rd edn. AK Peters, Natick
2. Amanatides J, Woo A (1987) A fast voxel traversal algorithm for ray tracing. In: Eurographics '87. Elsevier, North-Holland, Amsterdam, pp. 3–10
3. Ang S-S, Constantinides GA, Luk W, Cheung PYK (2008) Custom parallel caching schemes for hardware-accelerated image compression. *J Real-Time Image Process* 3(4):289–302

4. Felzenszwalb PF, Huttenlocher DP (2006) Efficient belief propagation for early vision. International Journal of Computer Vision 70(1)
5. Glassner AS (October 1984) Space subdivision for fast ray tracing. IEEE Comput Graph Appl 4(10):15–22
6. Grimm S, Bruckner S, Kanitsar A, Meister EG (October 2004) A refined data addressing and processing scheme to accelerate volume raycasting. Comput Graph 28(5):719–729
7. Havran V (November 2000) Heuristic ray shooting algorithms. Ph.D. thesis. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague
8. Kanus U, Wetekam G, Hirche J (July 2003) VoxelCache: a cache-based memory architecture for volume graphics. In: Eurographics/SIGGRAPH workshop on graphics hardware, pp. 76–83
9. Klimaszewski KS, Sederberg TW (January–February 1997) Faster ray tracing using adaptive grids. IEEE Comput Graph Appl 17(1):42–51
10. Krüger J, Westermann R (2003) Acceleration techniques for GPU-based volume rendering. In: Proceedings IEEE visualization 2003
11. Köse C, Chalmers A (July 1997) Profiling for efficient parallel volume visualisation. Parallel Comput 23(7)
12. Larabi Z, Mathieu Y, Mancini S (June 2009) Efficient data access management for FPGA-based image processing socs. In: Proceedings of the 2009 IEEE/IFIP international symposium on rapid system prototyping, pp. 159–165
13. Lorensen WE, Cline HE (1987) Marching cubes: a high resolution 3d surface construction algorithm. SIGGRAPH Comput Graph 21(4):163–169
14. Mancini S, Desvignes M (2006) Ray casting on a SoPC platform: algorithm and memory tradeoff. In: IEEE conference on computer information technology, Seoul, Korea. IEEE, Los Alamitos
15. Mancini S, Eveno N (November 2004) An IIR based 2D adaptive and predictive cache for image processing. In: DCIS 2004, p. 85
16. nVidia. Cuda sdk. <http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html>
17. Osborne R, Pfister H, Lauer H, McKenzie N, Gibson S, Hiatt W, Ohkami T (1997) EM-Cube: an architecture for low-cost real-time volume rendering. In: 1997 SIGGRAPH/eurographics workshop on graphics hardware. ACM, New York
18. Pfister H, Kaufman A, Chiueh T-c (1994) Cube-3: A real-time architecture for high-resolution volume visualization. In: Kaufman A, Krueger W (eds) 1994 symposium on volume visualization, pp. 75–82
19. Pfister H, Kaufman AE (1996) Cube-4 – a scalable architecture for real-time volume rendering. In: VVS, p. 47
20. Revelles J, Ureña C, Lastra M (2000) An efficient parametric algorithm for octree traversal
21. Strengert M et al. (2004) Large volume visualization of compressed time-dependent datasets on GPU clusters. Parallel Comput 31(2)
22. Wetekam G, Staneker D, Kanus U, Wand M (2005) A hardware architecture for multi-resolution volume rendering. In: HWWS '05: proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware. ACM, New York, pp. 45–51

Mapping a Telecommunication Application on a Multiprocessor System-on-Chip

Daniela Genius, Etienne Faure,
and Nicolas Pouillon

Abstract The particular form of the task graph of many telecommunication applications permits a high level of coarse grained parallelism. We consider a classification application on a telecommunication oriented multiprocessor system-on-chip (MP-SoC) platform. The hardware architecture hosting this type of application contains many programmable processors and dedicated hardware coprocessors, sharing the same address space. Inter-task communications are implemented via Multi-Writer Multi-Reader (MWMR) channels placed in shared-memory. To meet the strict requirements of this type of application, several performance bottlenecks have to be overcome. We show how our tool DSX (Design Space Explorer) helps to analyze these bottlenecks and outline the perspectives for further improvement.

Keywords Hardware/software codesign · Taskgraph · Kahn Process Network

1 Introduction

Telecommunication applications can be considered a special case of streaming applications. They usually process packet streams, where the same operations are performed on each packet, but the actual computing depends on the packet contents. For [1], this variable processing time, depending on the packet type, is the main characteristic of network applications. Throughput requirements are variable: backbone equipments, such as routers, require high throughput and little computation, while traffic analyzers require less throughput but intensive computation.

The classification application proved quite a challenge when initially mapping it to the MPSoC platform, which meant writing the netlists as well as the scripts

D. Genius (✉)
SoC Department, LIP6, 4 place Jussieu, 75252 Paris Cedex, France
e-mail: daniela.genius@lip6.fr

to vary architectural parameters by hand [2]. The Design Space Explorer (DSX) tool promised to facilitate this process; we thus rewrote our application to serve as a non-trivial example in a very early phase of the development of DSX [3]. DSX already contained some of the specific features we required, others were added at our request. However, performances of the mapped application were slightly inferior to those of the initial version.

DSX has undergone a major evolution since; in particular, cache and lock mechanisms were improved. DSX offers fine-grained mapping facilities, which were not yet fully exploited. It furthermore facilitates the analysis of performance bottlenecks. An attempt to improve the performance of our application was thus very promising.

The paper is organized as follows: Sect. 2 presents related work, Sect. 3 explains the particular form of our task graph. Sections 4 and 5 detail the hardware and application software, respectively. Section 6 shows how our architecture and application can be rewritten in DSX and how the mapping can be described. Furthermore, it details how DSX answered to the specific requirements of our application. Section 7 presents an incremental approach to eliminating the performance bottlenecks, which is facilitated by DSX. In Sect. 8, experimental results for the full application are shown. In Sect. 9 we draw conclusions and give an outline of future work in form of a non-exhaustive list of parameters to explore more comprehensively.

2 Related Work

We focus on telecommunication applications written in the form of a set of coarse grain parallel threads communicating with each other. Inter-task communications can be done through message passing like in STePNP [4], modeled in the form of data flow graphs like StreamIt [5] and Ptolemy [6], originally targeted to DSP, or can use the shared memory capabilities of the multiprocessor hardware architecture.

Kahn Process Networks (KPN) [7] propose a semantics of inter-task communication through infinite FIFO channels with non-blocking writes and blocking reads. Such infinite channels are impossible to implement, thus KPN formalism has been adapted for example by YAPI [8]. To deal with the select problem YAPI introduces the select function, which makes the model non-deterministic. Implementations of YAPI are COSY [9] and SPADE [10]. Digital System Design Environment (Disydent) [11], used in the initial mapping as described in [2], is also based upon KPN and uses point to point FIFOs.

While the KPN formalism is well suited to video and multimedia applications which can be modeled by a task graph where each communication channel has only one producer and one consumer, it is not convenient for telecommunication applications where several tasks access the same communication buffer in order to consume or produce packet descriptors.

MWMR (Multi-Writer/Multi-Reader) channels are software FIFOs that can be accessed by several reader and writer tasks. The communication protocol is described in more details in [2]. The generic MWMR communication channel supports

both hardware or software producers or consumers, making it possible to decide quite late whether a task should be implemented in software or hardware.

In the domain of codesign for signal and image processing domain, the two extremes are *platform configuration* (tuning the platform architecture parameters and exploring its configuration space) and *system-level synthesis*. Our work is closer to the former, with the exception that we designed I/O coprocessors which are *specific* to our domain [12].

Like for SESAME [13], we define a platform that is usable for several applications rather than designing a strictly application-specific platform. For this reason, we adopted SoCLib [14], a generic shared-memory multiprocessor-on-chip open platform (see Sect. 4). The core of the platform is a library of SystemC simulation models for virtual components (IP cores), with a guaranteed path to silicon.

The analytical system-level design approach for network processors presented in [1] is not based on simulation, considered too time-consuming in the context of design space exploration.

Recently, the work on ESPAM [15] examined the mapping of streaming media applications to shared memory MPSoC architectures. A variety of bounded KPN channels are implemented in form of specific hardware, whereas MWMR channels are software channels mapped to on-chip memory.

In the remainder of this paper, we will present the mapping of a telecommunication application, initially described in [2] in the form of POSIX threads which communicate via MWMR channels, to a shared memory MPSoC platform. We will show the specific way in which DSX was adapted to meet our requirements and how it helped to quickly identify and eliminate the performance bottlenecks.

3 Application Specification

In order to extract the coarse-grained parallelism from a sequential application, two basic approaches exist. The first one relies on the coarse-grained segmentation of the sequential application. The algorithm is split into functional tasks that execute sequentially. This is called *pipeline parallelism*. The other consists in duplicating the whole sequential application into many clones; all tasks are doing the same job, but each one on a different dataset. This is known as *task-farm parallelism*. The task-farm model is convenient for telecommunication applications processing successive and independent packets like in a Gigabit Ethernet stream.

Task-farm and pipeline parallelism can be combined to yield any hybrid of graph, as shown on the left hand side of Fig. 1. All communications between tasks use point-to-point channels, that can be implemented as software FIFOs, in order to handle the asynchronous behavior of the tasks. Communication channels are represented by arrows between tasks. The FIFOs implementing the communication channels are implicit.

In many cases, the data produced by a task is not destined to one particular task, but rather to a *class* of tasks. Assume that tasks T00, T01 and T02 are three instances of the same computation, and that T10, T11 and T12 are three instances of another

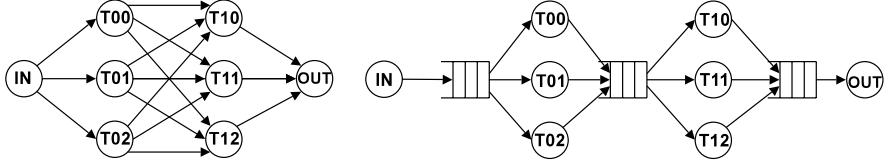


Fig. 1 Example of hybrid parallelism (*left*) with explicit MWMR channels (*right*)

computation. In this case, the first three tasks can send their output to any of the three others. It is evident that we should try to replace the nine separate communication channels by one single, multi-access communication channel. Figure 1 shows on its right one single FIFO, shared by three producers and three consumers.

Read and write operations can be blocking or non-blocking. The latter returns the number of items that have been transferred. A non-blocking write to a full channel and a non-blocking read from an empty channel thus return zero.

The new task and communication graph (TCG) is now a bipartite graph describing the intrinsic coarse grain parallelism of the application, without specifying implementation. As both programmable processors and hardware coprocessors can access a given software MWMR channel, each task can be implemented either as a software task (running on a programmable processor), or as a dedicated hardware coprocessor. KPN channels can be implemented as a special case of the MWMR communication formalism: in order to implement the KPN semantics, the task graph must have only one producer and one consumer per channel, and all the accesses to the FIFOs must be enclosed into a loop [16].

The communication protocol is based upon a shared memory multiprocessor architecture. All MWMR channels are mapped in shared memory. Their access is protected by a single lock per channel located in general-purpose memory and accessed through atomic operations. Each channel may have several readers and writers, but ignores their number. As illustrated by the following write request, the MWMR protocol requires five steps on the network-on-chip:

1. Get the lock protecting the MWMR channel (linked READ access).
2. Write the lock (conditional WRITE access).
3. Test the status of the MWMR (READ access).
4. Transfer a data burst (READ/WRITE access).
5. Update the status of the MWMR and release the lock (WRITE access).

All transfers to or from a MWMR channel must be a multiple of the 32 bits system bus width. Figure 2 shows an example hardware architecture with two processors and two memory banks. The MWMR channel implements a communication channel between a software task running on CPU0 and a hardware task implemented by coprocessor 1. One of the memory banks contains the locks ensuring the synchronization, implemented as spinlocks. On the TTY terminal, the progress of the application can be observed in the form of text messages.

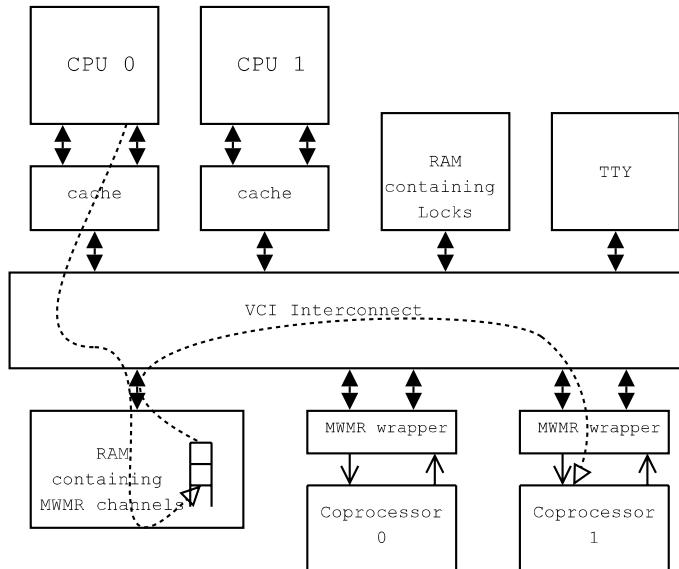


Fig. 2 Use of a MWMR channel located in on-chip memory, with one producer running on CPU0 and one consumer implemented in coprocessor 1

4 The Target Hardware Architecture

The target hardware architecture is a multiprocessor system on chip based on SoC-Lib [14] components and running the MutekH [17] kernel. It contains a variable number of 32 bits processors (currently MIPS32),¹ a variable number of embedded memory banks and other components such as a terminal emulator, an interrupt controller, and several I/O coprocessors. All these components are communicating through a VCI/OCB compliant micro-network [19]. There are two types of components: initiators and targets. Initiators send command packets, routed to the appropriate target by the interconnect, targets send response packets.

All initiators and targets share the same address-space. In such a hardware platform using large numbers of processors, coprocessors and memory banks, the central interconnect has to provide a high throughput between initiators and targets, which a conventional bus is unable to offer as it can only serve one communication at time. We replace the bus by a Network-on-Chip (NoC), which prevents us from using a snoop mechanism to ensure data coherency. We use a generic flat (one-level, non-clustered) interconnect. In order to obtain more realistic measurements, parameters are a minimal latency and the number of possible requests allowed to queue up for access to the same target component.

¹Meanwhile, instruction set simulators for ARM7 and PowerPC 405 have been added [18], others are in preparation.

For performance reasons, MWMR channels are however located in *cacheable* memory. Their coherency is guaranteed by a software mechanism where each cache line containing MWMR channel data (status, contents, etc.) is invalidated before execution of the five step access protocol, in order to ensure that data is fresh from memory, and flushed after the access, in order to ensure that memory is updated. The flush is only necessary in case of a write back cache mechanism, it is unnecessary for the write-through caching policy of our MIPS-based platform.

4.1 The Telecommunication Platform

Until now, we have presented a *generic* hardware architecture. A telecommunication specific platform can be obtained by replacing the two coprocessors in Fig. 2 by two application specific coprocessors called *InputEngine* and *OutputEngine*.

As usual in that domain, in order to take into account the limited size of on-chip memory banks, packets are cut into chunks of equal size, chained by pointers, which can be handled more efficiently [20]. We call such a structure a *slot* (Fig. 3). A packet is accordingly represented by a double-word *descriptor*, containing only a pointer to the beginning of the packet and the mandatory information to retrieve it. Necessary data are the address of the next slot, the total size of the packet, and an offset for potential additional headers. Our slots are 128 bytes long. A descriptor sits in front of each slot. This leaves 120 bytes for the payload. The first slot of a packet is pointed by a standalone descriptor that can be easily transferred. This use of descriptors allows us to avoid the copying of packets in memory most of the time. Consequently, and in contrast to image processing applications like MJPEG, where chunks of images are transferred, the MWMR channels contain only descriptors and packets payload is transferred directly in memory.

To take this duality into account, I/O coprocessors must have two interfaces: a MWMR interface to send and retrieve descriptors as well as a VCI interface to send and receive slots. The latter is directly connected to the on-chip interconnect, while the former is implemented in the form of a *wrapper*. SoCLib components are required to have a VCI interface, while coprocessors use FIFO interfaces. To implement MWMR communication protocol, a dedicated hardware wrapper is thus required, featuring a target interface for configuration and an initiator interface for fetching descriptors from the software channels located in on-chip memory. DSX handles this controller transparently, as shown later.

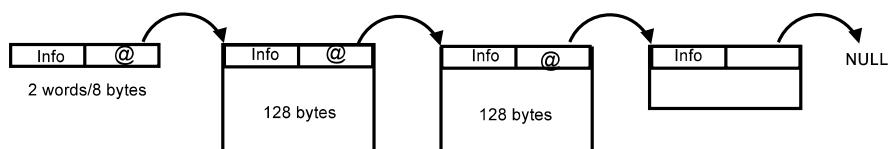


Fig. 3 Packets as chained slots

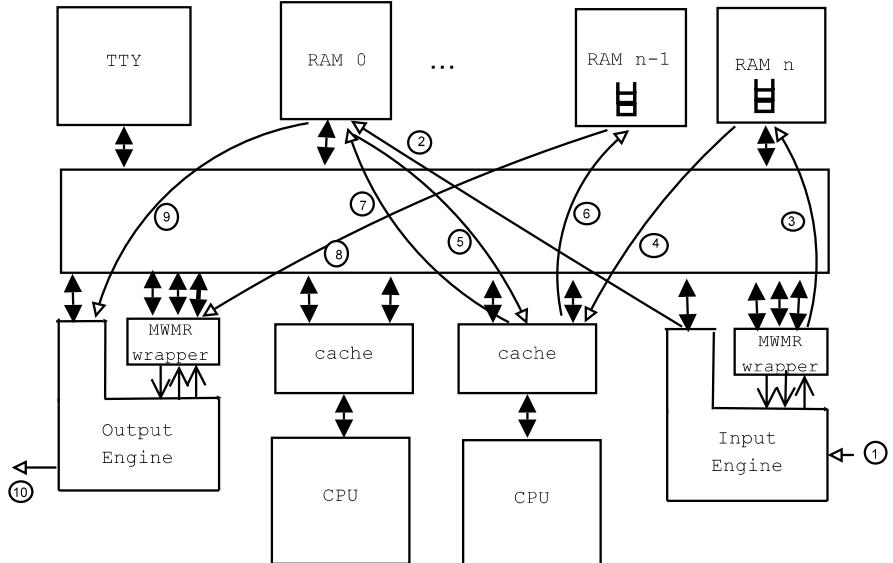


Fig. 4 Telecommunication platform with packet trajectory

A fundamental assumption of our architecture is that for a large majority of networking applications it is sufficient to consider only the beginning of a packet. We privilege this so-called first slot in so far as we store it in on-chip memory. The rest of a packet, also in the form of slots, is stored in off-chip memory. Nevertheless this policy can be modified by a minor change in the *InputEngine*: only one bit has to be manipulated in order to determine whether a slot is to be stored on-chip or off-chip.

Figure 4 shows on its lower right an *InputEngine* with one VCI interface and three FIFO channels, one for outgoing descriptors and one each for incoming on-chip and off-chip addresses. On its lower left, find an *OutputEngine* with one VCI interface and three FIFO channels, one for incoming descriptors and two for outgoing on-chip and off-chip addresses. The MWMR channels are stored on any of the memory banks, in our example on the last two. A typical packet trajectory, with an access only to the first slot, takes ten steps, all of which imply memory transactions:

1. The InputEngine MWMR controller reads a packet from a file or Ethernet link.
2. The InputEngine MWMR controller writes slots to a memory bank.
3. The InputEngine MWMR controller writes a descriptor to MWMR channel.
4. The processor reads descriptor from a MWMR channel.
5. The processor reads a slot from a memory bank.
6. The processor writes descriptor to a MWMR channel.
7. The processor writes a slot to a memory bank.
8. The Output Engine MWMR controller reads a descriptor from a MWMR channel.

9. The Output Engine MWMR controller reads a slot from a memory bank.
10. The Output Engine MWMR controller writes a packet to a file or Ethernet link.

Note that the same task never uses a given channel both ways.

5 The Classification Application

Classification is an important and resource-consuming part of many telecommunication applications [20] which takes place just after a packet arrives. Packet headers are analyzed, afterwards the packet is sent to one of several priority queues, and from there scheduled to the unique output queue. We consequently require a second level of tasks and obtain a hybrid tasks graph as seen in Sect. 3.

Our choice of application was also motivated by the particular challenge of mapping it onto a MPSoC. High demands on throughput require a strongly parallel TCG. MWMR channels are used in several different ways, for storing addresses, accepting descriptors on their way to and from coprocessors, and as priority queues. Limited on-chip memory requires fast feedback of liberated addresses once a packet has left the system. Most of all, memory accesses have to be very fast, as parallel as possible, while avoiding contention.

5.1 The Application Task Graph

Figure 5 shows the task graph of the classification application. In the following, we briefly describe the five types of tasks. Besides input and output tasks, there are two levels of software tasks. A hardware implementation additionally requires a bootstrap task that organizes coprocessor startup and address generation, a so-called *bootstrap* task.

Input Task: the input task reads a packet from a stream, determines its size, performs some basic checks on maximal packet size, checksum, etc., then computes the number of slots required to store it in memory, and finally copies the slots to memory. The first slot is copied to on-chip memory, subsequent slots, if they exist, are copied to off-chip memory. Only the eight-byte-long descriptor is sent to the outgoing MWMR channel. This write operation is non-blocking: if the channel is full, the packet is dropped and its addresses are recycled. Slot addresses, from which descriptors are constructed, are obtained from either one of three sources:

1. *Bootstrap task*: in the beginning, it creates slots from memory banks.
2. *Output task*: when a packet leaves the system through the OutputEngine.
3. *Classification task*: when errors are detected and packets can be discarded.

Note that packet addresses are always recycled.

Classification Task: the classification task reads one or more descriptors and suspends itself if this is not possible. If it succeeds, it reads the first slot from on-chip memory. The packet has to be dropped if one of several checks fails. Each slot begins with a descriptor containing the address of the next slot and one bit indicating whether or not the slot is destined for on-chip memory. Deallocation proceeds along these addresses from one slot to the next until the last slot is reached.

Scheduling Task: the scheduling task uses a simple algorithm which ponders an incoming descriptor by the priority of the current queue. The priority queues from which it reads are tested for eligibility in a round robin manner, necessitating a non-blocking read operation in order not to suspend on an empty queue. The descriptor of an eligible packet is then written to one unique output queue.

Output Task: the output task constantly reads the output queue of descriptors and blocks if this queue is empty. The address contained in the descriptor gives the memory location of the first slot. The address of the descriptor contained in this slot gives the address of the second slot, and so forth. During its reconstitution, a buffer holds fragments of the packet. Finally, the packet is written to an output stream. Each time a slot is written, its address is sent to either of the two channels containing on-chip and off-chip addresses and can again be used by the InputEngine.

Bootstrap Task: this task is responsible for the startup of the application. Originally part of the application main program, the bootstrap task has undergone significant changes, which are detailed in Sect. 6.5. It allocates on-chip and off-chip addresses at 128 bytes intervals (slot size) to the first and following slots, respectively, for as many packets as are simultaneously present in the system. Figure 5 shows that it writes into two channels: addresses of on-chip slots and addresses of off-chip slots.

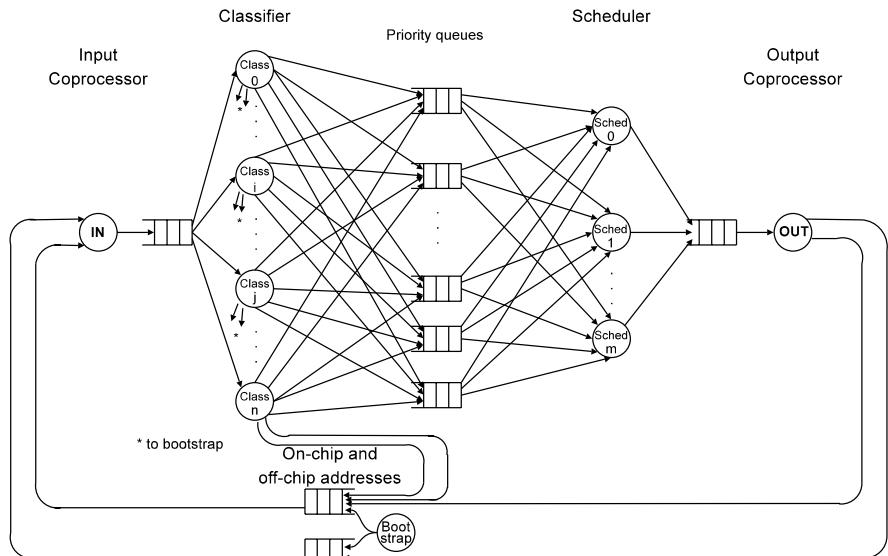


Fig. 5 Task graph of the classification application

off-chip slots. The bootstrap task runs only once, then suspends itself, liberating its processor.

Classification, scheduling and bootstrap are software tasks. Input and Output tasks exist as software and hardware versions, following the guidelines imposed by DSX: the hardware coprocessor is actually used in the final platform; the software versions will serve for a first validation under POSIX [21].

6 DSX Design Space Explorer

DSX [22] implements the task and communication graph, where the communications are of Multi-Writer Multi-Reader type. It comes with a multitasking multiprocessor kernel [17]. DSX extends Disydent by a comfortable user API enabling the user to describe architecture, application and mapping in *one* common language, Python [23]. It is suitable for both quick and in-depth design space exploration. The semantics of the application is preserved even if the mapping onto the hardware architecture changes.

DSX describes the task graph in a completely static manner. As the number of tasks is fixed, so is the channel sizes and their association to task ports. The management of software tasks, implemented as POSIX threads and created in the `main()` function of the embedded code, is automatic; the configuration of the MWMR wrappers is also done automatically. DSX can use SoCLib models of hardware components. The instantiation of the SoCLib components and the rather error-prone task of hardware netlisting is automatized and has completely disappeared from the designer's immediate view.

All build support files are generated automatically. DSX dimensions the memory by making a second compilation pass after mapping is completed. It finally generates binaries for both a purely software `pthread` version and for a SoCLib platform with SystemC models.

6.1 DSX Architecture Description

Input and output coprocessors are hardware tasks, instantiated with their name, the name of their controller and the name of the input file or Ethernet stream. They are located in memory segments which addresses are marked in the line `ctrl.addSegment`, along with their size and cacheability. Lines of the form `x//y` are used to generate the netlist, where `icn` stands for interconnection network. The role of initiators and targets connected to this network was explained in Sect. 4.

```
ie = dsx.TaskModel.getByName('input_eng').getImpl(soclib.HwTask)
ctrl, coproc = ie.instantiate(arch,
    'input_eng', 'input_eng_ctrl',
```

```

defines = { 'f_input': "in0.txt"})
ctrl.addSegment('ie0_ctrl',0x70200000, 0x100, False)
ctrl.vci_initiator // icn.to_initiator.new()
ctrl.vci_target // icn.to_target.new()
coproc.vci // icn.to_initiator.new()

oe = dsx.TaskModel.getByName('output_eng').getImpl(soclib.HwTask)
ctrl,coproc = oe.instanciate(arch,
    'output_eng', 'output_eng_ctrl',
    defines = { 'f_output': "out0.txt"})
ctrl.addSegment('oe0_ctrl',0x71200000, 0x100, False)
ctrl.vci_initiator // icn.to_initiator.new()
ctrl.vci_target // icn.to_target.new()
coproc.vci // icn.to_initiator.new()

```

The above lines describe only the two coprocessors, which are specific to our platform; the remaining hardware architecture is built from SoCLib components. We thus refer to the complete documentation and user's guides [14, 22].

6.2 DSX Application Description

Tasks are modeled in a Kahn-like fashion with input and output channels.² DSX provides both blocking and non-blocking primitives to access MWMR channels. Below find an example of a blocking write primitive, sending a four byte item located at address base to a channel named mwmr.

```
srl_mwmr_write(mwmr, base, 4);
```

The blocking read and write primitives return when the requested transaction is complete, i.e. the requested number of words was successfully read or written, whereas the corresponding non-blocking functions will always return an integer that indicates the number of words that have been actually transferred, even if the request is not satisfied. If the returned number is less than required, it is up to the software task to decide.

Tasks are modeled independently from each other and their implementations are interchangeable: software tasks, hardware coprocessors, and others.

6.3 DSX I/O Coprocessor Description

In DSX, we give a TaskModel which can then be instantiated several times in the TCG. The *InputEngine* is implemented (keyword `impls`) is a software and a hardware version (`SwTask` and `MwmrCoproc`). The implementation is cycle accurate

²Means of inter-task communication currently provided are MWMR channels, shared memory, mutex and synchronization barriers.

bit accurate (`caba:vci_input_eng`). The code of the software tasks has to be supplied by the programmer in a file `input_eng.c`. There are two ports `onchip` and `offchip` for receiving four byte on-chip and off-chip addresses, as well as an outgoing port for eight byte descriptors `desc` as well as `config` and `status` ports for configuration and interrogation about its status, respectively. The file name `f_input` is given as a parameter.

```
TaskModel( 'input_eng',
    ports = {
        'onchip' : MwmrInput(4),
        'offchip' : MwmrInput(4),
        'desc' : MwmrOutput(8),
    },
    impls = [
        SwTask( func = 'inputengine',
            stack_size = 1024,
            sources = [ 'input_eng.c' ],
            defines = [ 'f_input' ],
        ),
        MwmrCoproc( module = 'caba:vci_input_eng',
            to_coproc = [ 'onchip:onchip',
                'offchip:offchip' ],
            from_coproc = [ 'desc:desc' ],
            config = [ 'running' ],
            status = [ 'status' ],
        )));

```

In the TCG, an instance of the input task appears as follows, where the ports are connected to channels and the input file name is given:

```
Task( 'ie0', 'input_eng',
    portmap = {'onchip': channel_onchip,
               'offchip': channel_offchip,
               'desc': channel_desc },
    defines = {'f_input': "in0.txt"})
```

The OutputEngine is described and instantiated accordingly.

6.4 Classification and Scheduling Tasks

Classification tasks have one input port and fourteen output ports, one for each of the twelve priority queues, one for on-chip, and one for off-chip addresses of erroneous, thus discarded, packets. We only show the tasks model here, the task instantiation can be deduced easily.

```
TaskModel(
    'classif',
    ports = {
        'in_classif' : MwmrInput(8),
        'classif_ordo0' : MwmrOutput(8),
    ...
}
```

```

'clfif_ordinal': MwmrOutput(8),
'onchip': MwmrOutput(4),
'offchip': MwmrOutput(4),
},
impls = [ SwTask( 'clfif',
                  stack_size = 2048,
                  sources = [ 'classification.c' ] )
        ]
)

```

In the code of the task `classification.c`, once a slot has been loaded into local memory of the task's processor, the corresponding addresses will have to be invalidated. At our request, DSX now provides a primitive for selective invalidation to be used in the code of the tasks, rather than lines of assembler code: `srl_dcache_flush_zone` takes an address and a size in bytes as parameters.

6.5 Bootstrap Task

The perhaps most important adaptations triggered by the mapping of our classification applications and thus discussed in some detail below concern the bootstrap functionality. A generic task which manages the entire initialization process and the required primitives is now part of DSX. The work in [3] pointed out the need for a specific *bootstrap* task, featuring a mechanism which allows a task to run during a limited time only, whereas the usual tasks of DSX run continuously. Secondly, primitives were added permitting to access the strobe ports of the MWMR configuration wrapper in order to awaken the I/O coprocessors. In the previous version, for this purpose, we had to access internal information of the generated netlist.

```

TaskModel('bootstrap',
          ports = {'onchip' : MwmrOutput(4),
                   'offchip' : MwmrOutput(4),
                   'mem_onchip': MemspacePort(0),
                   'mem_offchip': MemspacePort(0),
                   },
          impls = [ SwTask(None,
                           bootstrap = 'bootstrap',
                           stack_size = 128,
                           sources = ['bootstrap.c'])
                    ]
)

```

The bootstrap task uses memory areas of explicit size that can be explicitly placed in memory (see Sect. 6.6), so-called *memspaces*. In order to distinguish between on-chip and off-chip memory regions to be used by the task, we define two *memspaces* of 384 slots and 2 K slots, respectively. Besides the two outgoing channels for created slots, memspaces also have ports of minimal size zero, i.e. the task dynamically creates slots until memory space exhaustion.

The bootstrap task only exists as a software task (SwTask). The programmer provides the code, using the appropriate primitives, in the file `bootstrap.c`.

For efficiency reasons, memspaces are mapped to cacheable memory (`cram0`). We will thus have to ensure cache coherency by software, selectively invalidating cache lines in the software tasks, as shown in Sect. 6.4.

```
Memspace('onchip', 49152)
Memspace('offchip', 262144)

mapper.map('onchip', buffer = 'cram1', desc = 'cram1')
mapper.map('offchip', buffer = 'cram2', desc = 'cram2')
```

The function `create_slots` performs the actual address generation; slots are 128 bytes long, addresses are consequently generated with that interval. The related software primitives, `SRL_MEMSPACE_ADDR` and `SRL_MEMSPACE_SIZE`, have been added to the API.

```
static void create_slots(srl_memspace_t memsp, srl_mwmr_t mwmr)
{
    uintptr_t base = SRL_MEMSPACE_ADDR(memsp);
    size_t size_left = SRL_MEMSPACE_SIZE(memsp);

    while (size_left) {
        srl_mwmr_write(mwmr, &base, 4);
        base += sizeof(papr_slot_t);
        size_left -= sizeof(papr_slot_t);
    }
}
```

The main function named `bootstrap` assigns names to ports, issues a strobe signal to the controller of the output engine (`oe0_ctrl`), generates the slots and finally issues a strobe signal to the controller of the input engine (`ie0_ctrl`), using the functions `srl_mwmr_config` and `BASE_ADDR_OF` added at our request.

```
FUNC(bootstrap)
{
    srl_mwmr_t addr_int = GET_ARG(onchip);
    srl_mwmr_t addr_ext = GET_ARG(offchip);

    srl_memspace_t memspace_offchip = GET_ARG(mem_onchip);
    srl_memspace_t memspace_onchip = GET_ARG(mem_offchip);

    srl_mwmr_config(BASE_ADDR_OF(oe0_ctrl), 0, 1);
    create_slots( memspace_onchip, addr_onchip);
    create_slots( memspace_offchip, addr_offchip );
    srl_mwmr_config(BASE_ADDR_OF(ie0_ctrl), 0, 1);
}
```

The bootstrap task runs only once, then suspends.

6.6 DSX Mapping Description

In the following code snippets, we show examples of mappings, leaving out nested loops only for simplicity of presentation.

To begin with, we state that a TCG will be mapped on a given hardware.

```
mapper = Mapper( hard, tcg )
```

MWMR Channels: The software objects of a channel are the channel itself (buffer), its description (sizes, status and buffer addresses) and its status (read pointer, write pointer, lock).

```
mapper.map('fifo_desc0',
           buffer = 'uram0',
           desc = 'cram0',
           status = 'cram0')
```

In the actual code, we use Python iterators to assign different channels to different memory banks, the assignment being made for example by whatever function yielding an integer value and respecting the boundaries, like the maximum number of available memory banks, for instance implementing a simple *round robin* strategy. Note that the memory bank where the channel is situated appears explicitly in the mapping, which is a significant improvement over SPADE where only entire tasks can be explicitly mapped.

Memspaces: Memspaces are located on one memory segment, either uncacheable or cacheable. The latter is faster but the coherency problem has to be solved in the code. They are connected to ports, and their declarations are completely static so that they can be allocated at compile time.

```
mapper.map('memspace_onchip',
           buffer = 'cram1',
           desc = 'cram1')

mapper.map('memspace_offchip',
           buffer = 'cram2',
           desc = 'cram2')
```

Coprocessors: The two coprocessors are mentioned by their instance names (`ie0` and `oe0`) their task model names and the name of their controller; the parameters have already been given in the hardware description (see Sect. 6.3).

```
mapper.map('ie0', coprocessor = 'input_eng',
           controller = 'input_eng_ctrl')
mapper.map('oe0', coprocessor = 'output_eng',
           controller = 'output_eng_ctrl')
```

Tasks: On the one hand the MIPS 32 does not support multiple contexts, context switching is thus expensive. On the other hand the MIPS internal architecture is relatively simple, we can thus afford to add a large number of processors and allocate only one task per processor. For each task, its software objects – code, stack and information about its status – are mapped to memory banks.

All classification and scheduling tasks can be mapped to their respective processors and memory banks in a loop using a Python iterator; each task can potentially have its own TTY for debug purposes.

```
for j in range(nclassif):
    mapper.map( 'classif%d' % j,
                desc = 'cram0',
                run = 'mips%d' % j,
                stack = 'cram0',
                tty = 'tty',
                tty_no = 0)
```

Shared and private memory segments are mapped to the processors.

```
for c in range(ncpu):
    mapper.map( 'mips%d' % c,
                shared = 'uram%d' % c,
                private = 'cram%d' % c)
```

We finally map the task and control graph.

```
m.map( tcg,
        private = 'cram0',
        shared = 'uram0',
        code = 'cram0',
        tty = 'tty',
        tty_no = 0)
```

In the following we will analyze the particularities of our application that may hamper performances, each in turn, before tackling the full-scale application in Sect. 8.

7 Eliminating the Bottlenecks

The theoretical maximum throughput on a 32 bit platform is 32 bits per cycle. The maximum achievable throughput however depends on the frequency the InputEngine can write to the interconnect. It is close to 8 bits per cycle (2.64 Gb/s for a 330 MHz platform), which means that the average interval between the sending of two 32 bits words is four cycles. This was determined experimentally by instrumenting the outgoing descriptor channel of the InputEngine.

Three main bottlenecks were already treated implicitly in [2] when tool support was scarce, and taken up shortly after in the early days of DSX [3]. In the following they are revisited, made explicit and analyzed in depth, taking advantage of the comprehensive support now provided by DSX:

1. Contention due to *accesses to the InputChannels*.
2. Contention due to *simultaneous accesses to memory banks* by several data objects.
3. Insufficient *burst length* for MWMR transfers.

The first is indicated by a high number of spins before obtaining access to a channel's lock, the second is indirectly indicated by long memory access latencies (the time passed through the interconnect for accesses to a memory bank), the third by

a large number of transactions per transferred words. All three can easily be determined by instrumenting the corresponding SoCLib components (MWMR wrapper for the first, memory banks for the second and third).

7.1 Accesses to the InputChannels

DSX permits to analyze the behavior of the MWMR channels. The MWMR controllers yield statistics. There is a simple scheduler (basically round robin testing if a channel is non-empty/non-full and going to the next if the test fails).

1. *Elects*: the task is elected by the scheduler.
2. *Spins*: accumulates the number of retries necessary to obtain the channel's lock.
3. *Bailouts*: insufficient space in channel, release the lock and return to scheduler.
4. *Xfers*: transfers.

Throughout this experiment we use a task graph with sixteen tasks, which is a power of two and yet not too large a number of initiators for a flat interconnect.

By intuition, more tasks reading from the same channel should empty it faster. On the other hand, due to the overhead of channel description, however small, more channels occupy slightly more memory even if their size is a fraction of the size of a unique channel. More importantly, there will also be more potential contention for access to memory banks.

We modify the InputEngine such that it sends descriptors onto several InputChannels instead of only one, the parameter being fixed at compile time, see Fig. 6. In order to exhibit the effects of multiple InputChannels without perturbation by other effects, we choose a TCG with only one intermediate level of software tasks. We still achieve 8 bit/cycle for two tasks which only copy descriptors but do not access packet memory. For one single task, the throughput is 3.24 bit/cycle (1.07 Gb/s). In the following, all throughput information is given in bits per cycle. The right side of the figure depicts four intermediate tasks reading from two InputChannels. By dividing the number of *spins* by the number of *elects*, we obtain the mean number of *spins* before a channel is elected. Figure 7 shows the experimental result for sixteen tasks. The result is obvious (one channel per task). However, increasing contention on the interconnect will mostly eliminate this benefit for the realistic application, see Sect. 8.

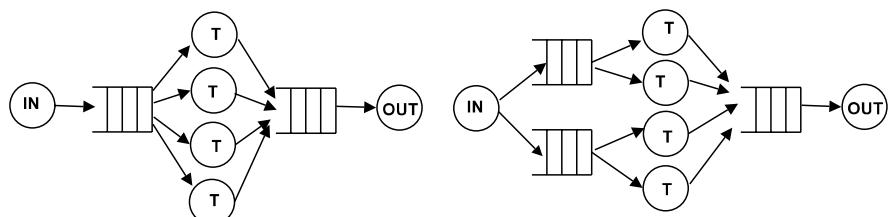


Fig. 6 One level of tasks with one (*left*) or several InputChannels (*right*)

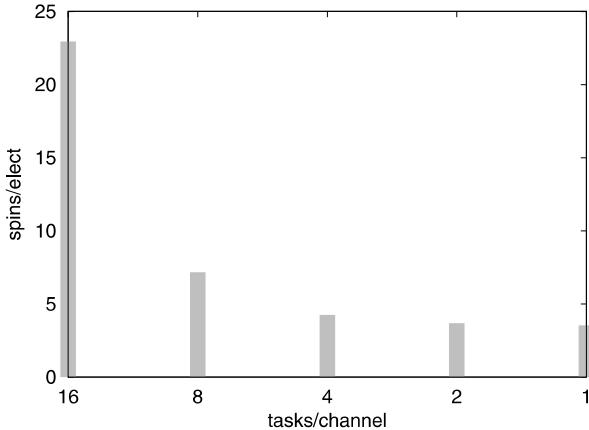


Fig. 7 Spins per *elect* depending on the number of tasks per InputChannel. For sixteen tasks, 22.9 spins are required when all tasks access the same channel, whereas the best result, 3.3 spins per *elect*, is obtained when every task has its own channel

7.2 Simultaneous Accesses to Memory Banks

As stated above in Sect. 6, the application contains essentially four kinds of software objects to be placed on the memory banks:

1. Application code.
2. Packets in the form of slots.
3. MWMR channels.
4. Stacks of the software tasks.

The mapping of application code is generally obvious, for the execution stacks the only challenge is to keep them as small as possible (less than 2 K are required for each classification, 1 K for each scheduling task, 128 bytes for the bootstrap task). The fact that MWMR channels are placed on general purpose memory banks makes them ubiquitous, but at the same time vulnerable to memory access contention. As soon as accesses to packet memory come into play, the potential for contention increases further. A further possible distinction is between InputChannels, channels containing on-chip and off-chip addresses, and priority queues.

We modify our intermediary tasks such that now they transfer of a packet, i.e. access slot memory, but without doing any further computation on the packet contents. Performance strongly differs depending on whether a unique memory bank holds all software objects or whether they are distributed over different banks. Two or more simultaneous accesses to the same memory bank may cause *contention* and increase latency as subsequent accesses may have to wait in a FIFO queue. Only the extreme cases are shown here: either all software objects are mapped onto the same or to different memory banks. Figure 8 shows that the latter mapping gains up to a factor of two. We also observe that while for eight tasks performance reaches the optimum, for sixteen tasks it degrades slightly. This is due to the contention of read and write accesses to the same channel.

Fig. 8 Throughput depending on the mapping of software objects to memory banks. We consider one up to sixteen tasks and the two extreme cases of mapping all objects to the same bank (same) and to different banks (diff)

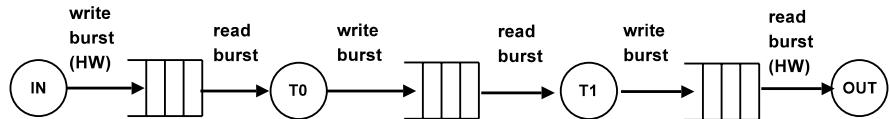
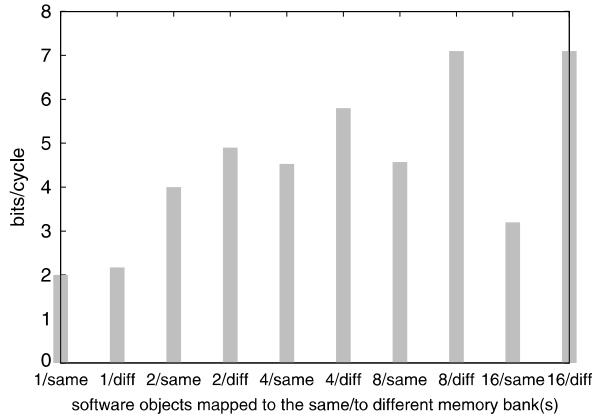


Fig. 9 Minimal two-level application in order to determine the effect of increased burst size

7.3 Burst Size

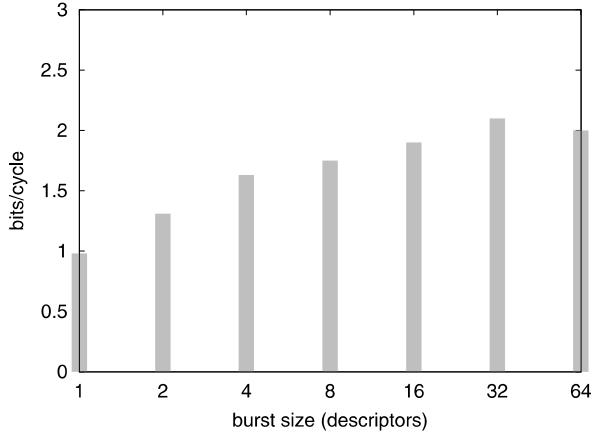
MWMR transfers are expensive as they require four additional memory accesses apart from the actual transfer of one item. As the destination of a packet is not known until the packet header is analyzed in the classification task, and subsequent descriptors are usually sent to different priority queues, previous versions of the applications transferred descriptors one by one between the two software task levels, which was inefficient. We rewrote the tasks such that descriptors destined for the same priority queue are now read and written as *bursts*.

The present exploration restricts to the priority queues.³ Again, in order to isolate the causes of the bottleneck, we only take two sequential tasks, one with an access to slot memory, the other only transferring descriptors (Fig. 9). The tasks are connected by one MWMR channel for which we vary the burst size.

We assume that there are sufficient memory banks and data objects are ideally mapped. We add a second level of software tasks and the MWMR channel(s) connecting these two levels. Figure 10 shows that throughput increases for larger burst lengths of 1 to 64 descriptors arriving at 2.1 bit/cycle. When bursts become too long, they take too much time to traverse the interconnect. For bursts larger than 64 descriptors, performances do not improve any more.

³The issue is the same between hardware and software tasks; the bursts size issuing from a hardware coprocessor can be configured at its creation.

Fig. 10 Throughput depending on the burst length; the burst length increases from 1 to 64 descriptors (2 to 128 words)



8 Performance Results

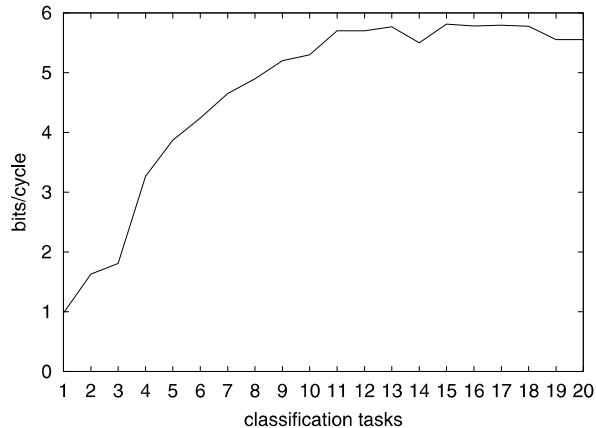
We are now prepared to take on the full classification application. Apart from SoCLib models, we use SoCLib compatible models of the I/O coprocessors. SystemCASS [24] from the Disydent environment is a cycle accurate bit accurate simulator about ten times faster than the SystemC event-based simulator. In all experiments, we focus on effects independent of instruction cache size. We do not replicate code, but chose the instruction caches sufficiently large to contain the entire executable (16 KiB). If the instruction cache is of a smaller size than the application executable, instruction cache misses occur each time a new part of the application executable has to be recharged. Exploration of the cache parameters is not part of this study, we accordingly chose the maximal size allowed by the SoCLib component for our data cache (16 KiB), and a direct mapped write through cache policy.

Small packets of 40 bytes payload and 14 bytes Ethernet header constitute the worst case for classification, because the number of headers to verify is largest with respect to the data throughput. Mean latency is measured in simulation cycles, from reading a packet from the input stream to writing it onto the output stream. We do not take into account the time for the execution of the bootstrap task, which varies with the number of slots generated before the strobe signal is given, neither the time for booting the operating system. The *steady state* is thus the situation after the bootstrap is terminated and buffers have filled up, such that measurements of throughput are stabilized. The throughput itself is measured at the output side. At configurable periodic intervals, the OutputEngine computes the average of the data transmission rate in bit/cycle.

We now replace the two generic task levels of the preceding section by the classification and scheduling task levels and determine the optimal number of processors for one InputChannel. The software classification and scheduling tasks are modified in order to regroup descriptors into bursts of configurable size. Throughout this section, the burst size is 32 descriptors (64 words).

Figure 11 shows throughput for increasing number of classification tasks, starting with one classification and scheduling task each. Even if beyond twelve tasks

Fig. 11 Throughput depending on the number of classification tasks. When the number of classification tasks increases beyond a certain limit, adding scheduling tasks further improves the throughput. These limits have been determined experimentally; they are located at eight and fifteen classification tasks for two and three scheduling tasks, respectively, indicating a ratio of around 1:8



(5.7 bit/sec) the throughput increases only marginally, the overall optimum of 5.83 bit/sec is obtained for seventeen classification and three scheduling tasks; adding more tasks only increases contention on the interconnect, outweighing the gain of increased computing power.

We furthermore study the influence of the number of InputChannels (Fig. 12). We obtain the best throughput for two channels, each connected to eight tasks. It degrades quickly if more channels are used.

Varying the burst size while keeping the number of tasks and InputChannels stable confirmed an optimal burst size of 32 descriptors (not shown here).

Finally, to give a first indication on the benefits of a judicious mapping of the software objects, Fig. 13 shows results for four possible mappings of data objects. All other parameters are unchanged (two InputChannels, seventeen classification and three scheduling tasks, etc.). It is no surprise that the presence of one single memory bank (*all-on-one*) provokes a lot of contention, whereas separating slots and channels (*slots+channels*) proves beneficial. Mapping the priority queues on separate banks (*priority-queues*) makes a further difference, whereas mapping all software

Fig. 12 Throughput depending on the number of channels for the twenty processor (seventeen classification and three scheduling tasks) version. Tasks are partitioned into groups of nearly equal size, accessing the same channel. All software objects are mapped onto different memory banks, the burst size is 32 descriptors

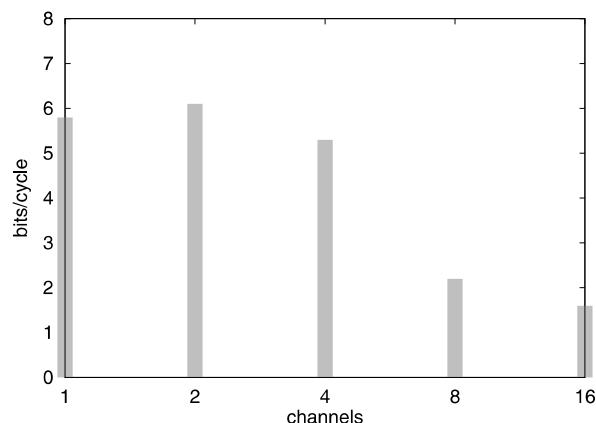
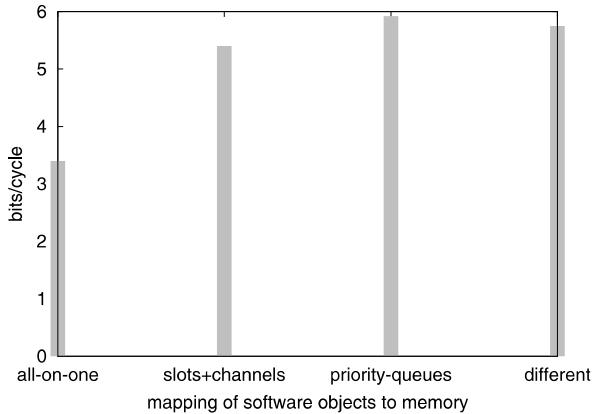


Fig. 13 Throughput depending on the mapping for four different mappings: all objects on one memory bank (all-on-one), MWMR channels and slots on different memory banks (slots+channels: for 54 byte packets, only on-chip slots are required), additionally mapping each priority queue on a different bank (priority-queues), and finally all objects on different memory banks (different)



objects on different banks (*different*) does not improve performance because with such a large number of memory banks, too many components are grouped around the interconnect and potential for contention increases. The best performance we obtain is 5.92 bit/cycle (1.95 Gb/s on a 330 MHz platform). Note that we do not completely explore the design space yet, which is vast, so that the results may improve further.

To summarize the differences to previous versions, the application now requires twenty instead of twenty-four tasks (three scheduling and seventeen respectively twenty-one classification tasks) in order to reach its maximum throughput, and a flat interconnect is sufficient. We employ two InputChannels instead of only one, and transfer bursts of thirty-two descriptors instead of a single descriptor between the two levels of software tasks. We now obtain a throughput of 5.92 bit/cycle instead of 1.67 bit/cycle obtained in the original application [2], a factor of about 3.5. As performances for the first DSX-guided mapping shown in [3] were slightly inferior to even those of the original application, this improvement is even more important.

9 Conclusion and Perspectives

With the help of DSX, a multi-threaded telecommunication application, modelled as a hybrid task graph between task-farm and pipelined, was mapped to a shared memory MPSoC platform. The classification application exhibited most of the features of DSX, on the other hand it posed interesting challenges: high parallelism, severe requirements on memory latency, etc.

During the last two years, the implementation of MWMR channels underwent several technical improvements, improving the throughput by a factor of 3.5 when exploited. The work described in [3] proved that DSX can handle task graphs that extend the KPN semantics to multiple readers and multiple writers; however it did not probe far into performance bottlenecks and mapping of software objects.

Meanwhile, MWMR channels have become well established as the communication channel of choice for DSX. They have proven their efficiency in the hardware/software codesign of other streaming applications, among those a stereoscopic pre-crash obstacle detection [25].

In the presence of a larger number of tasks, a *clustered* architecture is better adapted. The work described in [2] shows such a Non-Uniform Memory Access (NUMA) architecture, where memory access times differ depending on whether a processor accesses a memory bank local to the cluster, or on another cluster. It is obvious that NUMA adds even more parameters to our exploration, such as the number of clusters, of processors and memory banks per cluster. Data objects mapped injudiciously to a “wrong” cluster will incur even stronger penalties.

Variations of cache size and associativity will have to be studied, a rather classical issue of design space exploration. The impact of small instruction caches that cannot hold the entire executable, particularly important in the presence of clusters, will also have to be investigated in more detail.

As MWMR communication channels are placed in memory, we profit fully from DSX’s memory mapping capabilities. In practice there are not as many memory banks as data objects, thus trade-offs have to be accepted. Nevertheless, our experiments underline the importance of having extensive possibilities for fine-grained mapping of software objects.

Trying to summarize the parameters that can be varied for the classification application on the flat architecture yields the following non-exhaustive list:

1. Number of tasks for each level of the hybrid task graph.
2. Cache parameters.
3. Number of InputChannels.
4. Mapping of software objects upon memory banks.
5. Burst sizes for MWMR channels.

A large number of parameters, notably the cache size, associativity and word size, have remained untouched in the experiments shown here. In consequence, the design space for a full-scale exploration becomes extremely large. We are now prepared to undertake the exploration for a given performance requirement under realistic conditions such as limited memory size and power consumption.

References

1. Thiele L, Chakraborty S, Gries M, Künzli S (2002) Design space exploration of network processor architectures. In: 1st workshop on network processors at the 8th international symposium on high-performance computer architecture (HPCA8), Cambridge, MA, USA, pp 30–41
2. Faure E (2007) Communications matérielles-logicielles dans les systèmes sur puce orientés télécommunication (HW/SW communications in telecommunication oriented MPSoC). PhD thesis. Université Pierre et Marie Curie
3. Genius D, Faure E, Pouillon N (2007) Deploying a telecommunication application on multiprocessor systems-on-chip. In: Design and architectures for signal and image processing (DASIP)

4. Paulin P, Pilkington C, Bensoudane E (2002) StepNP: a system-level exploration platform for network processors. *IEEE Des Test Comput* 19(6):17–26
5. Drake M, Hoffman H, Rabbah R, Amarasinghe S (2006) MPEG-2 decoding in a stream programming language. In: International parallel and distributed processing symposium, Rhodes Island, Greece
6. Buck JT, Ha S, Lee EA, Messerschmitt DG (2002) Ptolemy: a framework for simulating and prototyping heterogeneous systems. In: Readings in hardware/software co-design. Kluwer Academic, Norwell, pp 527–543
7. Kahn G (1974) The semantics of a simple language for parallel programming. In: Rosenfeld JL (ed) *Information processing '74*. North-Holland, New York, pp 471–475.
8. de Kock EA, Smits WJM, van der Wolf P, Brunel J-Y, Kruijzer WM, Lieverse P, Vissers KA, Essink G (2000) YAPI: application modeling for signal processing systems. In: Proceedings of the 37th conference on design automation (DAC-00). ACM/IEEE, New York, pp 402–405
9. Brunel J-Y, Kruijzer WM, Kenter HJJN, Pétrot F, Pasquier L, de Kock EA, Smits WJM (2000) COSY communication IPs. In: Proceedings of the 37th conference on design automation. ACM/IEEE, New York, pp 406–409
10. van der Wolf P, Lieverse P, Goel M, La Hei D, Vissers KA (1999) A MPEG-2 decoder case study as a driver for a system level design methodology. In: CODES '99: proceedings of the 7th international workshop on hardware/software codesign. ACM Press, New York, pp 33–37
11. Augé I, Pétrot F, Donnet F, Gomez P (2005) Platform-based design from parallel C specifications. *IEEE Trans Comput-Aided Des Integr Circuits Syst* 24(12):1811–1826
12. Faure E, Greiner A, Genius D (2006) A generic hardware/software communication mechanism for Multi-Processor System on Chip, targeting telecommunication applications. In: ReCoSoC '06: proceedings of the 2006 conference on reconfigurable communication-centric SoCs. Univ. Montpellier II, Montpellier, pp 237–242
13. Erbas C, Cerav-Erbas S, Pimentel AD (2006) Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Trans Evol Comput* 10(3):358–374
14. SoCLib Consortium. Projet SoCLib: plate-forme de modélisation et de simulation de systèmes intégrés sur puce (The SoCLib project: an integrated system-on-chip modelling and simulation platform). <http://www.soclib.fr>
15. Nikolov H, Stefanov T, Deprettere E (2008) Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Trans Comput-Aided Des Integr Circuits Syst* 27(3):542–555
16. Parks TM (1995) Bounded scheduling of process networks. PhD thesis. University of California at Berkeley, CA, USA
17. Becoulet A (2010) Définition et réalisation d'un exo-noyau pour architectures multiprocesseurs hétérogènes à mémoire partagée (Definition and realization of an exo-kernel for heterogeneous shared-memory architectures). PhD thesis. Université Pierre et Marie Curie
18. Pouillon N, Becoulet A, Vieira de Mello A, Pêcheux F, Greiner A (2009) A generic instruction set simulator API for timed and untimed simulation and debug of MP2-socs. In: IEEE international workshop on rapid system prototyping. IEEE Comput. Soc., Los Alamitos, pp 116–122
19. VSI Alliance (2000) Virtual component interface standard (OCB 2.0)
20. Comer D (2003) Network system design using network processors. Prentice Hall, New York
21. Tanenbaum A (1995) Distributed operating systems. Prentice Hall, New York, pp 169–185
22. Pouillon N, Greiner A (2007) DSX: un outil d'exploration architecturale efficace pour systèmes multi-processeurs intégrés sur puce (DSX: an efficient design space exploration tool for MPSoC). In: Colloque GDR SoC. <https://www-asim.lip6.fr/trac/dsx>
23. Python Software Foundation: Python programming language. <http://www.python.org>

24. Buchmann R, Pétrot F, Greiner A (2004) Fast cycle accurate simulator to simulate event-driven behavior. In: Proceeding of the 2004 international conference on electrical, electronic and computer engineering (ICEEC'04), Cairo, Egypt. IEEE Comput. Soc., Los Alamitos, pp 35–39
25. Greiner A, Pétrot F, Carrier M, Benabdenbi M, Chotin-Avot R, Labayrade R (2006) Mapping an obstacles detection, stereo vision-based, software application on a multi-processor system-on-chip. In: IEEE intelligent vehicles symposium, Tokyo, Japan. IEEE Comput. Soc., Los Alamitos, pp 370–376

Part 2

Data Acquisition and Embedded Systems

A Standard 3.5T CMOS Imager Including a Light Adaptive System for Integration Time Optimization

Gilles Sicard, Estelle Labonne, and Robin Rolland

Abstract This paper presents a light adaptive system which allows an automatic management of the integration time value of a standard 3 transistors (3T) CMOS imager. A low resolution network of high dynamic range pixels is included in this standard CMOS sensor. This low resolution network is regularly distributed on the entire photosensitive array, and computes the average light power information. This value allows the control system to choice the optimized integration time value which provides the best image quality. This imager has been designed in a $0.35\text{ }\mu\text{m}$, 3.3 V CMOS technology. The basic photosensitive block layout contains four 3T standard pixels and one non-linear 2T pixel. Due to this distribution, we obtain a 3.5T per pixel. This sensor has been tested and TV video sequences show the efficiency of this very simple control system.

Keywords CMOS image sensor · Light adaptive system · Optimized integration time value · Low-cost camera

1 Introduction

The CMOS image sensors currently present on the market have average performances such as: an input dynamic range (DR) and a SNR about 60–70 dB, a correct sensitivity (limited by the integration time and the small size of the photodiode) and a correction of the fixed pattern noise (FPN) carried out in specific sample and hold system [1]. In comparison with CCD sensors, CMOS Active Pixel Sensors (APS) propose lower performances in term of dynamic range, sensitivity and noise (including dark current, temporal noise and fixed pattern noise). But CMOS technology offers advantages in term of production cost, power consumption and integration capabilities.

G. Sicard (✉)

TIMA Laboratory (CNRS, Grenoble INP, UJF), Grenoble, France

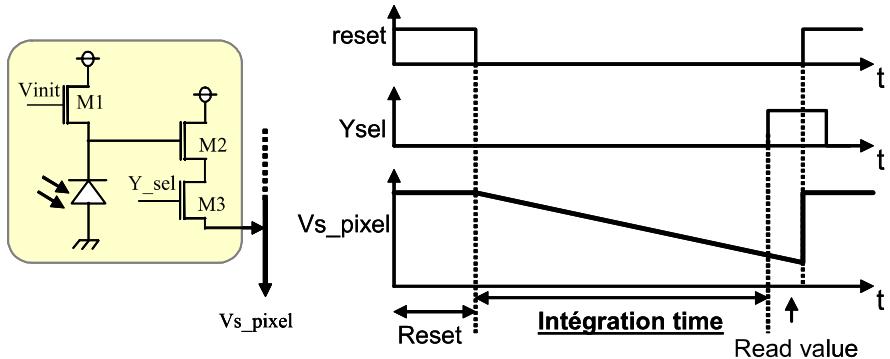


Fig. 1 Schematic and timing diagram of a standard 3T pixel

Researches are undertaken to improve CMOS imagers and to reduce their major drawbacks. Basically, the sensitivity improvement and the dark current minimization could be resolved with optimized CMOS technology. But, dynamic range, temporal noise and FPN problems concern the electronic design. To minimize the noise, several structures exist [1]. The architecture of a typical 3 MOS transistors active pixel sensor with an integration mode and its transient characteristic is presented in Fig. 1.

In order to extract the photo generated information, three successive steps are carried out. In a first time, the photodiode (and its parasitic capacitance) is initialized to a value close to V_{dd} . Next, the M1 transistor is opened; the photocurrent can discharge the capacitance linearly during a given time, called integration time. Once this integration time reach, the last step, the readout, can take place.

The value of the integration time impacts directly some performances of the imager, as the Input Dynamic Range (IDR), as illustrated in Fig. 2. In case of high value of photocurrent (dot line), the capacitance is fully discharged before the end of the integration time. This leads to a saturation effect, due to the too long integration time. In case of low value of photocurrent (gray line), the capacitance is not discharged: There is no detected information due to the too short integration time.

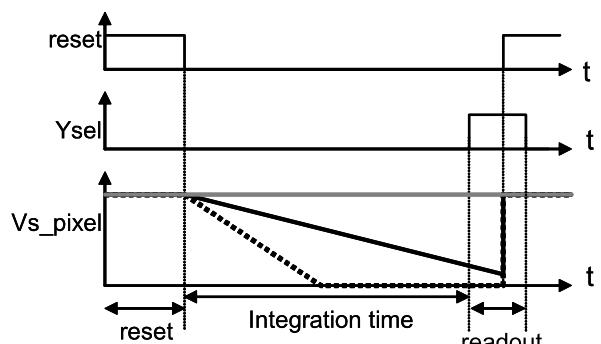


Fig. 2 Standard 3T pixel responses timing diagram with three different illumination levels

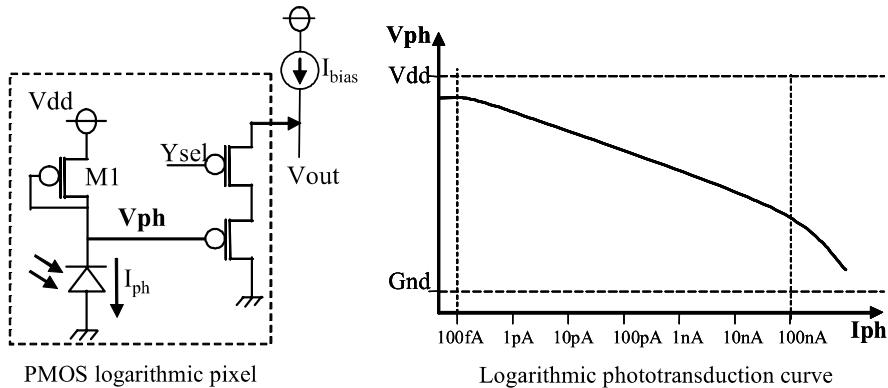


Fig. 3 Logarithmic pixel architecture and its phototransduction curve

It means this kind of CMOS sensors, with a 60–70 dB IDR, needs an integration time management. In a standard way, this management is done at the camera level.

To increase the input dynamic range over 100 dB (thus better than CCD sensors), several works investigate different methods or pixel structures: a fixed integration time with specific processing algorithms [2, 12–15], a variable integration time with specific processing algorithms or readout mode [3–5, 16–18] and continuous operating pixels using a pixel with a logarithmic response as shown in [6, 8].

Majors disadvantages of these high dynamic range (HDR) integration pixels are a higher pixel area compared to a standard 3T pixel, a very long readout phase (cumulative integration time) or complex external processing in order to compute the final HDR image. In another way, continuous operating pixels have the advantage of being very simple (pixel with 3 transistors, Fig. 3), providing an instantaneous high dynamic range, about 120 dB. But this very simple architecture presents a lower sensitivity, a huge Fixed Pattern Noise (FPN) and a non-linear response.

For consumers market, like webcams or mobile phones, all these improved propositions are not really profitable due to the extra costs.

In this work, we propose an intermediate solution using a standard 3T CMOS imager where the 60–70 dB dynamic range is automatically adapted to the light conditions. The sensor changes automatically its integration time in order to obtain the best image. In theory, this system would allow to obtain for a given scene a same image whatever the illumination of this scene, highlighted or darkness. To obtain this light adaptive system, an in-pixel system computes the sensor average illumination and modifies automatically the integration time value.

In the state of the art, [7, 8, 10] propose very interesting solutions: they obtain a specific phototransduction curve (based on logarithmic pixel) and they propose to shift this curve according the illumination condition. These works are bio-inspired systems (Silicon Retina). The major disadvantages of these methods are a large silicon area pixel and a non-linear response. Another solution, proposed by [9], explains how authors control the image variation (with the same scene) based on histogram information. But their pixels contain a high number of transistors.

Based on these results, we have work on the design of a light adaptive system which provides an automatic computation of the integration time value. The major constraints are to preserve the linear response of a standard pixel and to obtain a minimum silicon area overhead. Another constraint is to implement the simplest possible solution, in order to minimize the cost, the power consumption and to preserve the main electrical and electro optical characteristics of a standard CMOS imager.

The following section presents the principle of our low-cost light adaptive system. In Sect. 3, the sensor architecture is described. In Sect. 4, experimental results are reported and an overview of the sensor is dressed. Section 5 discusses the results. Finally, conclusions and perspectives are presented.

2 Automatic control of the integration time value

In order to detect the average light power variation, a dedicated photosensitive array has been designed. This array has a lower resolution than the standard one and it is spread across the photosensitive array. The aim of this matrix is to provide an output voltage ($V_{ph_average}$) in relation to the average light power (Fig. 4).

Through a dedicated feedback loop, this output voltage $V_{ph_average}$ controls the integration time value. As shown in Fig. 5, the analogue voltage $V_{ph_average}$, is amplified and converted in a digital word via a 3 bits Flash Analog to Digital Converter

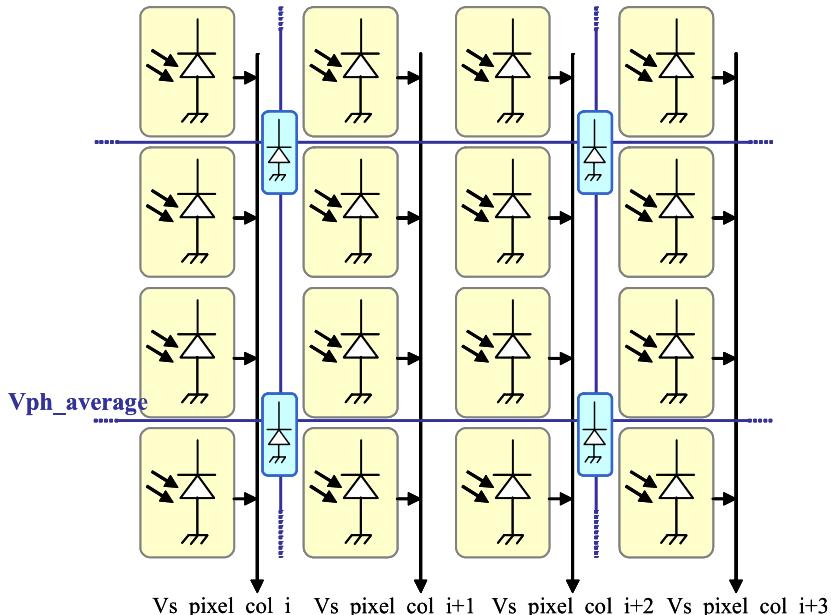


Fig. 4 Block diagram of our CMOS imager

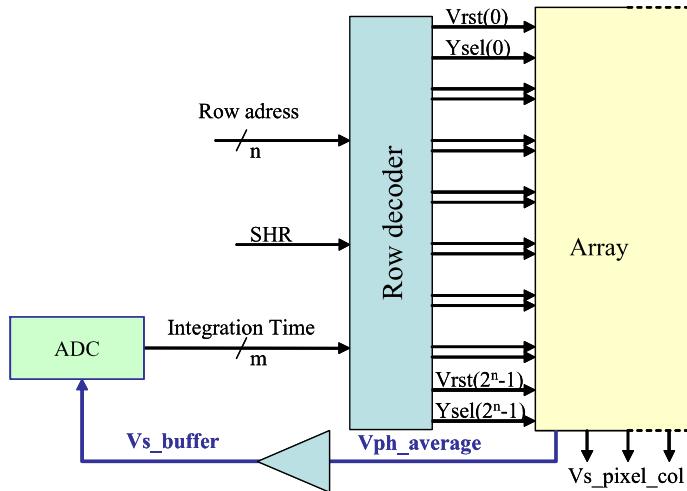


Fig. 5 Integration time control system

(ADC). Once digitalized, this information drives the pixel integration time through the reset control signal, managed by the row decoder.

In order to provide the average incident light value, we have chosen to implement an independent photosensitive array with a high dynamic range. The first feature, the independency, has been decided in order to keep a completely standard functional array (3T pixels), without any interaction with this dedicated array. It means that the readout scheme is exactly the same than a standard CMOS camera.

The second feature, the high dynamic range, has been decided in order to always obtain a valid output, whatever the light condition, without saturation effect.

The chosen pixel architecture is the logarithmic one, originally presented in [6], as it is simple, robust and allowing a high dynamic range. The implemented structure is derived from the one proposed in [8]. The logarithmic pixel we have designed includes only two NMOS transistors and a dedicated photodiode (Fig. 6).

All logarithmic pixels have a common node and this node voltage is logarithmically dependent of the average photocurrent value. This logarithmic law is due

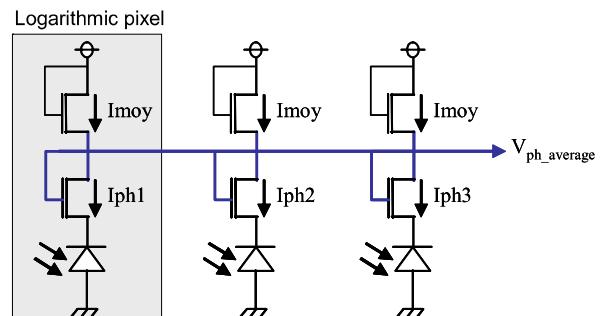
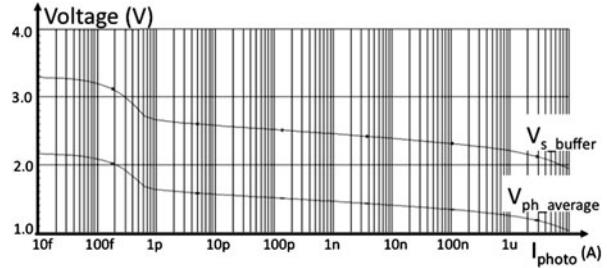


Fig. 6 Logarithmic pixels network

Fig. 7 Logarithmic pixel response obtained by simulation



to NMOS transistors working in subthreshold mode. The drain current of saturate transistors has an exponential relation with Gate, Bulk, Drain and Source transistor voltages:

$$I_d = I_{do} \cdot e^{\frac{(-V_{GB})}{nU_t}} \cdot \left(e^{\frac{V_{SB}}{U_t}} - e^{\frac{V_{DB}}{U_t}} \right) \quad (1)$$

n and I_{do} are process dependent parameters, U_t is the thermal voltage kT/q . If we assume that all NMOS transistors have the same size, the mean photocurrent value is computed as:

$$I_{moy} = \frac{\sum_{i=0}^m I_{ph(i)}}{m} \quad (2)$$

m is the number of logarithmic pixels. From these equations, the obtained output voltage follows a logarithmic law:

$$V_{ph_average} = V_{dd} - n \cdot U_t \cdot \ln\left(\frac{I_{moy}}{I_{do}}\right). \quad (3)$$

The simulated transfer function curve of this pixel is presented in Fig. 7. The output voltage $V_{ph_average}$ is sensitive to more than 7 decades of photocurrent, i.e. a 140 dB dynamic range.

3 Architecture of the Sensor

The proposed image sensor, called IMAGYNE2, is composed of two arrays (Fig. 8): a 128×128 standard integration 3T pixel array and a 64×64 logarithmic pixel array, which is regularly distributed with the standard array, and 128 column amplifiers. Two address decoders drive respectively the array rows and the column amplifiers.

The basic layout is shown in Fig. 9. This block contains four standard integration pixels and one logarithmic pixel. By abutment of this block, we obtain 128×128 standard pixels including a 64×64 sub-matrix which provides the average value of luminosity. The area of this basic block layout is $24 \times 24 \mu\text{m}^2$. The standard pixel includes three NMOS transistors and a $36 \mu\text{m}^2$ N⁺-P-well photodiode. In this layout, the logarithmic pixel photodiode area is $17 \mu\text{m}^2$. The fill factor is about 25%.

The integration pixel outputs are connected to a specific sample and Hold system called “column amplifiers”. These readout circuits are located at the bottom of

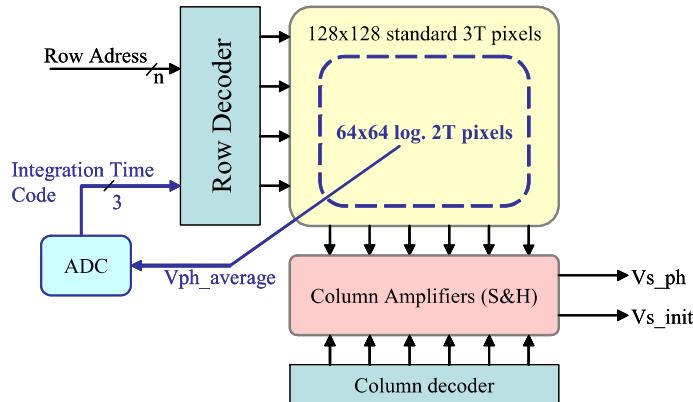


Fig. 8 Block diagram of our CMOS vision sensor

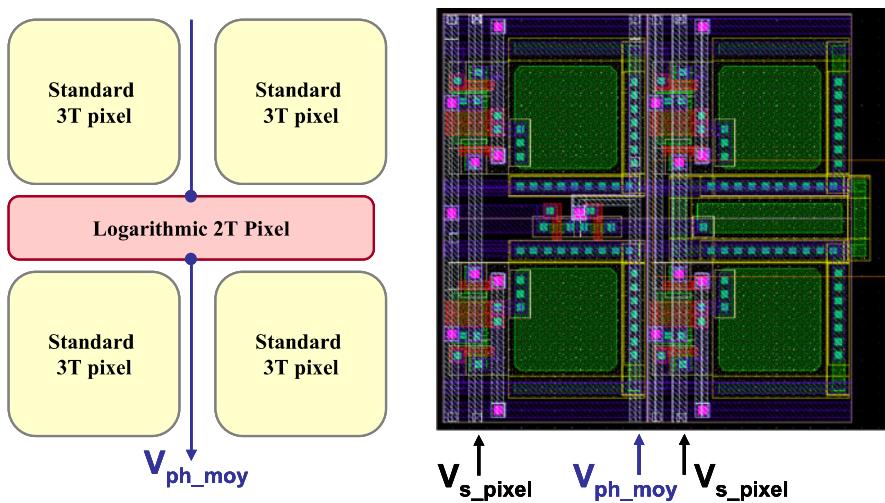


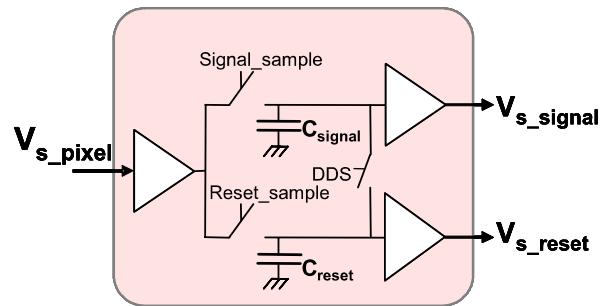
Fig. 9 Block Diagram and layout view of our light adaptive system: a logarithmic pixel added in a matrix of four standard 3T pixels

each column. These amplifiers allows to sample and hold the two pixel levels corresponding to the photocurrent output level and the reset output level (according to the classical readout of the standard integration imagers presented in [1], Fig. 10).

As the column amplifiers are a Fixed Pattern Noise (FPN) source, a special care has been carried out in their design. To reduce the offset variations, our column amplifiers present a structure described initially by [1]. This structure allows Correlated Data Sampling (CDS) and Double Delta Sampling (DDS) techniques in order to minimize the pixel to pixel and column to column FPN.

The logarithmic photosensitive array provides only one output corresponding to the analogue voltage $V_{ph_average}$. This voltage is amplified and converted into a 3 bits

Fig. 10 Block diagram of a standard column amplifier



data. The row decoder uses this data to compute the optimized integration time value and to drive the reset control signal of each line.

4 Overview and Measures of Our Circuit

This 128×128 pixel image sensor IMAGYNE2 has been designed in a standard, $0.35\text{ }\mu\text{m}$, four-metal layers, 3.3 V CMOS technology. This sensor has been designed in a multi-projects IMAGYNE test chip, integrating four different imagers. One is a standard 3T imager called REFERENCE. Figure 11 shows an overview of this chip. Sensor IMAGYNE1 implements a logarithmic 4T pixel with on-chip FPN reduction [11]. Sensor IMAGYNE3 integrates a logarithmic pixel with a light adaptive system which is spread between the 12T pixel, the column amplifier and the ADC [19].

Table 1 resumes the main characteristics of our CMOS imager.

Figure 12a–e illustrates the light adaptive capability of our sensor. This figure shows 2 films (TV video format) of the same scene with the same evolution of

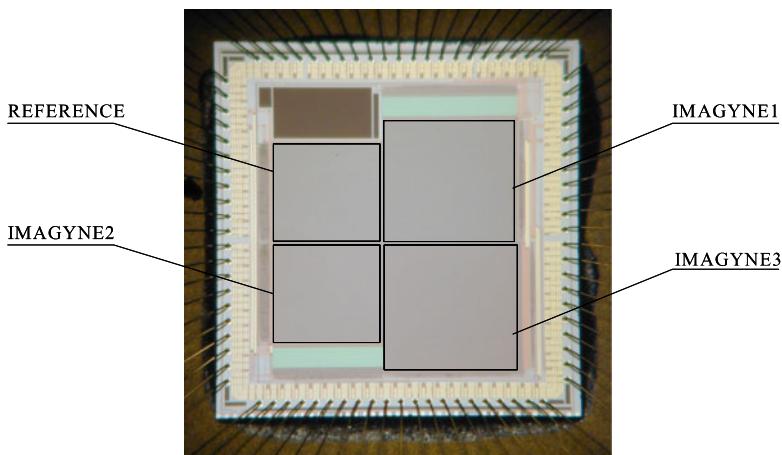


Fig. 11 Chip photograph

Table 1 Main characteristics of the proposed sensor

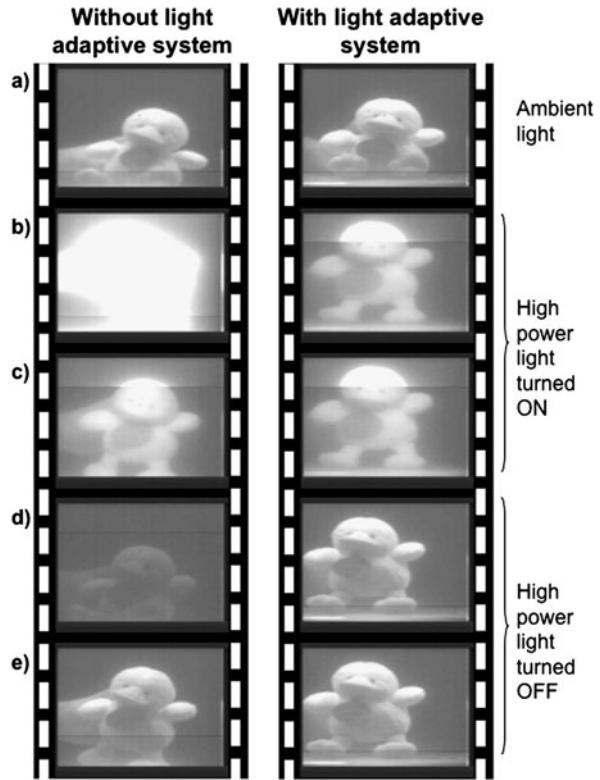
Prototype Chip summary	
Technology	0.35 μ m CMOS
Standard array resolution	128 \times 128 pixels
Log. network resolution	64 \times 64 pixels
Transistors per pixel	3.5 NMOS
Pixel pitch	12 μ m
Photodetectors	N ⁺ -P-well photodiode
Fill factor	25%
Acquisition mode	Rolling shutter
Power supply	3.3 V
ADC resolution	8 bits
Integration time	According average illumination
Dynamic range	as a standard 3T imager
FPN	as a standard 3T imager
Temporal noise	as a standard 3T imager

the light condition. On the left, a film with the REFERENCE array is shown. This standard REFERENCE imager consists in a 128 \times 128 3T pixel array without any feedback loop control. The integration time is controlled with an external command. The images obtained with the light adaptive system (IMAGYNE2 array) are shown on the right.

Under ambient light (Fig. 12a), the light adaptive system allows obtaining an image with a good trade-off of gray levels. An appropriate integration time value is choosing in order to obtain the same trade-off with the reference imager. When a high power light is switched on, the light adaptive system adapts instantaneously the integration time, allowing a good image, while the image obtained by the standard imager presents a majority of saturated pixels (Fig. 12b). The integration time is too long and a shorter value is chosen to obtain a better image (Fig. 12c). When the high power light is switched off, again, the light adaptive system adapts instantaneously the integration time, allowing a good image, while the standard imager provides a darker response due to the shorter integration time (Fig. 9d). A longer value is needed to obtain a good image (Fig. 12e). Whatever is the luminosity, the light adaptive system allows to adapt instantaneously the integration time and to obtain good images, while the same imager without this system presents darker or saturated images.

In both films, images obtained with a high light power (Fig. 12c) show two parts in the image with two different integration times. This problem occurs during the integration phase and is due to our implementation of the row decoder: It doesn't take into account the duration of the row blanking which is necessary to provide a TV video signal. Due to this, first rows of the image have a longer integration time than others due to the addition of the integration time with the blanking row time duration.

Fig. 12 TV Video sequences obtained with this sensor



5 Discussion

Table 2 presents the main characteristics of the state of the art and of the work presented in this paper (IMAGYNE2).

Compared to the existing works, the circuit designed using adaptive integration time is an alternative solution: a linear response, a small pixel, and no additional electronic or specific algorithm. Its way of work is totally standard. From user's point of view, the overall behavior (electrical and temporal) is similar to a standard CMOS imager. The difference is on the camera architecture: there is no electronic system dedicated to the computation of the integration time.

At the pixel level, the prototype presents an area overhead (about 50%) due to the logarithmic array. But it could be drastically reduced with a more aggressive layout (about 10 to 20%).

6 Conclusions and Perspectives

A light adaptive system has been implemented in a standard CMOS image sensor in order to control its integration time value. The average value of the global incident

Table 2 Comparative table

Authors	Techniques	IDR	Frame rates	Transistors per pixel	Pixel size	Technology	Fill factor
[12]	Fixed T_{int}	100 dB	30 fps	5	7.5 μ × 7.5 μ	0.35 μ	
[2]	Fixed T_{int}	130 dB	25 fps	25	25 μ × 25 μ	0.35 μ	11%
[13]	Fixed T_{int}	120 dB	1 kfps	43	19 μ × 19 μ	0.18 μ	50%
[14]	Fixed T_{int}	100 dB	1 Hz	19	30 μ × 30 μ	0.5 μ	
[15]	Fixed T_{int}	100 dB	30 fps	5	7.5 μ × 7.5 μ	0.18 μ	49%
[16]	Multiple T_{int}	90 dB	15 fps	3	10 μ × 10 μ	0.5 μ	41%
[17]	Multiple T_{int}	108 dB	100 kpix/s	4	20.4 μ × 20.4 μ	1.2 μ	1.5%
[4]	Multiple T_{int}	250 fps	5.5		10.5 μ × 10.5 μ	0.35 μ	29%
[18]	Multiple T_{int}	119 dB	30 fps		10 μ × 10 μ	0.25 μ	54.50%
[5]	Multiple T_{int}	120 dB	50 fps	3	26 μ × 26 μ	1 μ	65%
IMAGYNE2	Adapted T_{int}	Adapted 120 dB	30 fps	3.5	12 μ × 12 μ	0.35 μ	25%

light power is measured in the photosensitive array and allows choosing the optimal integration time in a quasi-continuous way. If the mean luminosity is changing, only one image will include two integration time values. But, with 25 images per second, it's invisible.

This light adaptive system is implemented through a feedback loop: a network of High Dynamic Range logarithmic pixels provides information on the sensor average illumination and this data allows computing the optimal integration time through a dedicated 3 bits flash ADC. The logarithmic pixels, all connected to a common node, are regularly distributed in the standard pixel array. One logarithmic pixel is inserted in the middle of four standard pixels. The output voltage of this network is logarithmically dependent of the average photocurrent value.

Some improvements are under studying concerning the logarithmic network. Due to its high output dynamic voltage, compared to the very low resolution of the ADC used (3 bits), the photodiode area of the logarithmic pixel can be reduced. Furthermore, we are investigating on the reduction of this sub matrix resolution. A new circuit, under designed, proposes two logarithmic networks: 32×32 and 16×16 pixels, regularly distributed in a standard 128×128 pixels array.

To conclude, this light adaptive system allows obtaining a very good and simple control of the integration time value. No anti-blooming system and no mechanical aperture control are needed, contributing to the entire camera cost reduction.

References

1. Mendis SK, Kemeny SE, Gee RC, Pain B, Staller CO, Kim Q, Fossum ER (February 1997) CMOS active pixel image sensors for highly integrated imaging systems. *IEEE J Solid-State Circuits* 32
2. Stoppa D, Simoni A, Gonzo L, Gottardi M, Dalla Betta GF (December 2002) Novel CMOS image sensor with a 132 dB dynamic range. *IEEE J Solid-State Circuits* 37
3. Burlucicello E, Etienne-Cummings R, Boahen KA (February 2003) A biomorphic digital image sensor. *IEEE J Solid-State Circuits* 36
4. Yang D, El Gamal A, Fowler B, Tian H (December 1999) A 640×512 CMOS image sensor with ultrawide dynamic range floating-point pixel-level ADC. *IEEE J Solid-State Circuits* 34
5. Schanz M, Nitta C, Bußmann A, Hosticka BJ, Wertheimer RK (July 2000) A high dynamic range CMOS image sensor for automotive applications. *IEEE J Solid-State Circuits* 35
6. Mead C, Ismael M (1989) Analog VLSI implementation of neural systems. Kluwer, Boston
7. Delbrück T, Mead CA (1995) Analog VLSI phototransduction by continuous-time, adaptive, logarithmic photoreceptor circuits. In: Koch C, Li H (eds), *Vision chips: implementing vision algorithms with analog VLSI circuits*. IEEE Comput. Soc., Los Alamitos, pp 139–161
8. Sicard G, Bouvier G, Lelah A, Fristot V (1998) A light adaptive 4000 pixels analog silicon retina for edge extraction and motion detection. In: *Workshop on machine vision and applications (MVA '98)*, Chiba, Japan, November
9. Ni Y, Devos F, Boujrad M, Guan JH (July 1997) Histogram-equalization-based adaptive image sensor for real-time vision. *IEEE J Solid-State Circuits* 32(7):1027–1036
10. Delbrück T, Oberhoff D (May 2004) Self-biasing low power adaptive photoreceptor. In: *IEEE international symposium on circuits and systems, ISCAS 2004*, pp 844–847
11. Labonne E, Sicard G, Renaudin M (2007) An on-pixel FPN reduction method for a high dynamic range CMOS imager. In: *33rd European solid-state circuits conference, ESSCIRC 2007*, Munich, Germany, September 11–13, pp 332–335

12. Akahane N, Sugawa S, Adachi S, Mori K, Ishiuchi T, Mizobuchi K (April 2006) A sensitivity and linearity improvement of a 100-dB dynamic range CMOS image sensor using a lateral overflow integration capacitor. *IEEE J Solid-State Circuits* 41(4)
13. Rhee J, Joo Y (February 2003) Wide dynamic range CMOS image sensor with pixel level ADC. *Electron Lett* 39(4):360–361
14. McIlrath LG (May 2001) A low-power low-noise ultrawide-dynamic-range CMOS imager with pixel-parallel A/D conversion. *IEEE J Solid-State Circuits* 36(5)
15. Acosta-Serafini PM, Masaki I, Sodini CG (September 2004) A 1/3 VGA linear wide dynamic range CMOS image sensor implementing a predictive multiple sampling algorithm with overlapping integration intervals. *IEEE J Solid-State Circuits* 39(9)
16. Schrey O, Huppertz J, Brockherde W, Hosticka B (July 2002) A high DR CMOS image sensor with on chip programmable region-of-interest readout. *IEEE J Solid-State Circuits* 37(7)
17. Yadid-Pecht O, Fossum ER (October 1997) Wide intrascene dynamic range CMOS APS using dual sampling. *IEEE Trans Electron Devices* 44(10)
18. Mase M, Kawahito S, Sasaki M, Wakamori Y, Furuta M (December 2005) A wide dynamic range CMOS image sensor with multiple exposure-time signal outputs and 12-bit column-parallel cyclic A/D converters. *IEEE J Solid-State Circuits* 40(12)
19. Labonne E, Sicard G, Renaudin M (2006) A 120 dB CMOS imager with a light adaptive system and digital outputs. In: 2nd conference on PhD research in microelectronics and electronics, PRIME 2006, Otranto, Italy, June 12–15, pp 269–272

Approximate Multiplication and Division for Arithmetic Data Value Speculation in a RISC Processor

Daniel R. Kelly, Braden J. Phillips,
and Said Al-Sarawi

Abstract Arithmetic data value speculation increases the throughput of a processor by using approximation to speculatively issue instructions dependent on an arithmetic result. This chapter describes new approximate multipliers and dividers. The performance advantage of these units is demonstrated in a practical context through simulation of a 32 bit RISC processor, modified to use these approximate units for arithmetic data value speculation. Instruction throughput improved by more than 15% for some media benchmarks.

Keywords Speculation · Approximation · Estimation · Arithmetic · Value speculation · Prediction · Probabilistic computing · Probability · Multiplication · Division · Mediabench · Simplescalar

Nomenclature

- n number of input bits to a counter
- m number of output bits from a counter
- z dividend
- d divisor
- q exact quotient after division $q = z/d$
- r remainder after division
- \tilde{d} approximate divisor
- q_i quotient after round i
- r_i remainder after round i
- t maximum number of division rounds
- f number of fractional bits maintained in q_i

D.R. Kelly (✉)

CHiPTec, Centre for High Performance Integrated Technologies and Systems, The University of Adelaide, Adelaide, Australia
e-mail: dankelly@eleceng.adelaide.edu.au

1 Introduction

Contemporary high-performance microprocessors make extensive use of speculation: they predict values that are not yet known and execute assuming this result is correct, while simultaneously checking if the prediction was correct. If the value was predicted incorrectly, the processor incurs a performance penalty while it backtracks and resumes with the correct value. Hence a complex trade-off exists between the cost of exact evaluation, the probability of correct predictions, the cost of correction, and the hardware overhead required to implement the predictions and corrections. Branch prediction is widely employed in modern microprocessors because of the high accuracy obtainable and the potential throughput benefit to typical programs. Many other forms of speculation have been proposed [17] but only a few have been adopted to any significant degree [5].

This chapter concerns the specific case of *arithmetic data value speculation* (ADVS) in which a hardware unit is used to quickly generate the likely result of an arithmetic operation such as an integer multiplication. This approximate result is carried forward into subsequent processing steps while an exact result is evaluated. Should the approximation prove correct, the processor can continue execution having benefited from the speculation. On the other hand, if the approximated result is incorrect, the processor will have to revoke its speculative steps and re-execute them using the exact result.

Like branch prediction and other widely used forms of speculation, ADVS requires support hardware in the pipeline to recover from mis-speculation [10, 19]. It also requires both an approximate unit and an exact unit for each arithmetic operation that is approximated.

1.1 Contributions

This chapter reviews two new approximate arithmetic units presented at the 2009 Conference on Design and Architectures for Signal and Image Processing (DASIP 2009): an approximate multiplier [11]; and an approximate divider [12]. Logic synthesis results are used to compare the speed of these units with baseline exact units. Operands extracted from media benchmarks running on a 32 bit RISC processor are used to characterize the probability of the approximate units generating correct results.

New results are also presented from execution-driven simulations of media benchmarks on a RISC processor that was modified for ADVS with the approximate integer multiplication and division. Operand caching is also introduced to further reduce the average latency of repeated operations, and mitigate the performance loss of repeated incorrect speculations. In some cases instruction throughput improved by over 15%. The improvement was less for some benchmarks, but none of the benchmarks simulated suffered a significant degradation in throughput.

1.2 Overview

Section 2 reviews the operation of approximate arithmetic units, and their uses. Section 3 describes the RISC processor we used, how we modified it to support ADVS, and the benchmarks we used to measure its performance. The synthesis methodology used to measure circuit performance is also described. The approximate multiplier and divider are presented in Sects. 4 and 5 respectively. Results of processor simulations including ADVS using the approximate multiplier and divider appear in Sect. 6.

2 Background

2.1 Approximate Arithmetic

The goal of arithmetic approximation is to reduce arithmetic calculation time for individual units at the expense of potential errors. Approximate arithmetic units are permitted to occasionally return erroneous results. By relaxing the constraint of always producing a correct result, they can be made faster, smaller, or more energy efficient than conventional, exact arithmetic units.

In ADVS we use approximation and speculation to improve instruction throughput in a processor. In this case errors are corrected before they leave the processor's pipeline, but there are many applications of approximate arithmetic in which the errors are not corrected, and are instead tolerated. Examples include radar processing [8], digital filters [25], CORDIC processing [26] and LDPC decoding [22].

An arithmetic unit can be made approximating by deliberately over-simplifying its hardware architecture so that correct results are no longer guaranteed for every combination of inputs. We call this *logical incompleteness*. The multiplier and divider presented in Sect. 4 and Sect. 5 are logically incomplete circuits. A benefit of logical incompleteness is that synchronous discipline is maintained. This means that circuit behavior is deterministic and conventional software tools and practices can be used to design, verify and optimize the logic circuits. Other logically incomplete adders are described in the literature [18, 19].

Alternatively, an arithmetic unit can be over-clocked, or its supply voltage over-scaled, so that signals are no longer guaranteed to evaluate within a clock cycle and errors may occur. We call this *temporal incompleteness*. Examples of temporally incomplete circuits include *Probabilistic CMOS* (or PCMOS) [1, 8, 14] and those built using the Razor latch [6, 7]. Benefits of this approach include tolerance to process variation and noise.

2.2 Arithmetic Data Value Speculation

Like branch prediction, ADVS is a speculative micro-architectural scheme that operates transparently to software. In the processor pipeline an arithmetic operation is

simultaneously issued to an approximate unit and an exact counterpart. The approximate unit delivers a result in fewer machine cycles than the exact unit, enabling the processor to speculatively execute until the exact result is known. In the common correct case the processor has saved cycles. In the uncommon incorrect case the processor must recover and replay issued dependent operations.

For ADVS it is necessary to design approximate units that execute at least 1 cycle faster than their corresponding exact units. It is also important they produce correct results with high probability. Each incorrect result is not just a missed opportunity to save cycles, it also incurs a performance penalty as the pipeline must recover from the mis-speculation.

While the probability of correct results is important, the magnitude of errors is inconsequential because the exact unit corrects the result. In ADVS all that matters is whether a result is correct or incorrect. In other applications, the magnitude of error might be important if the results are visible to the user.

The probability of correctness of an approximate unit depends on the distribution of program inputs. Hence it is important to benchmark with typical input cases instead of random numbers. For example, it has been empirically demonstrated that the distribution of carry propagations in adders is highly data dependent [16]. Loop counters are usually small and positive, and produce small carry chains in adders; on the other hand, memory addressing calculations can produce quite long carry lengths [16].

3 Simulation and Synthesis Tools

This section discusses the software tools and benchmarks we used to evaluate our approximate multiplier, divider and RISC processor. It also describes how the processor was modified to use ADVS.

3.1 SimpleScalar

SimpleScalar is a set of software tools to simulate a processor (also referred to as *SimpleScalar*), that implements the PISA architecture. The PISA architecture is a set of RISC instructions similar to the MIPS ISAs. The tool set includes a modified version of `gcc` to cross compile C and Fortran programs for the simulated *SimpleScalar* processor [4].

PISA was created by the *SimpleScalar* authors and is a close derivative of the MIPS architecture. The *SimpleScalar* pipeline is based on a 5-stage MIPS pipeline, with an additional stage in the back end. Up to 4 instructions can be issued to reservation stations, executed and retired per cycle. The processor allows out-of-order execution and branch speculation.

SimpleScalar includes execution driven simulators. The out-of-order simulator was modified for this project to include functional models of approximate arithmetic units and operand caches.

3.2 *MediaBench*

The performance of the estimating arithmetic units and the ADVS processor was evaluated using *MediaBench* benchmarks. This suite of programs and input data was assembled by Saint Louis University to be representative of communications and multimedia applications [15]. The suite also includes codecs for audio, video, PDF, coding and encryption. Each program in the suite is freely available. The *MediaBench* programs exhibit statistically different characteristics compared to *SPEC CINT2000* benchmarks in four metrics: retired instructions per clock (IPC), instruction cache hit rate, data cache read hit rate, and memory bus utilization [15].

The benchmarks were compiled using the version of `gcc` provided with *SimpleScalar*. A total of 14 tests from the set were successfully compiled. In these tests, integer arithmetic operations, excluding addition, accounted for approximately 1% of the total number of retired instructions, while floating point instructions accounted for approximately 5% of retired instructions.

3.3 *Operand Caches*

Operand caches store the operands and results of long latency operations to avoid the delay of recalculating repeated operations. Relatively small operand caches can increase throughput because typical programs repeat many operations. Simulations have shown that hit rates of over 50% can be achieved with simple direct mapped operand caches for multi-cycle arithmetic operations [23].

Operand caches provide the correct result for an arithmetic operation when that operation has been repeated recently enough that it is still available in the cache. Checking the cache and retrieving a result can be faster than recalculating a result for some long latency operations. If implemented in parallel with traditional hardware, operand caches do not incur a penalty for a miss.

Operand caches and ADVS complement each other when implemented in parallel. ADVS reduces the average latency for operations that have not occurred recently; operand caches reduce latency further when operations are repeated. Moreover, the operand caches spare the approximate arithmetic from repeatedly mis-speculating on repeated pathological cases, because the cached correct result is returned before the processor speculates on the approximate result.

Operand caches were used in this work, and were tuned to improve the overall cache hit rate by adjusting their associativity, replacement scheme and index hashing. Simulations were performed with separate caches for each integer and floating point operation. 64 entry direct mapped caches were hit over 50% of the time, and improved the average IPC by 5% for *MediaBench* benchmarks. For this work we used 4-way, FIFO replacement caches, indexed by the operand bits with the closest-to-equal probability of assertion. These increased the hit rates to over 60%, and improved IPC by approximately 6.5%.

3.4 Logic Synthesis

The delay of the approximate arithmetic circuits was measured using logic synthesis. The circuits were described in synthesisable VHDL and synthesized with *Synopsys Design Compiler* (version B-2008.09-SP2). Standard cells from the Artisan 1.8 V SAGE-X™ library for the TSMC 0.18 μm process were used [2]. In addition, a standard drive and load of a unit-sized *D flip-flop* (DFF) was used as a synthesis constraint.

Estimates of operand cache size, area and read latency were obtained using the Cache Access and Cycle Time Information (*cacti*) tools [24], using version 4.2 for 180 nm technology.

4 Approximate Multiplication

This section reviews a family of approximate multipliers designed for 32×32 bit signed and unsigned integer multiply instructions in a processor using ADVS. Further details, including the distribution of errors from these multipliers, can be found in [11].

The approximate multipliers are based on a conventional tree topology [21]. The multiplicand is multiplied bitwise by the multiplier to generate an array of partial products. A tree of counter circuits is used to sum the partial products until they form two operands to a carry-propagate adder (CPA). The output from the CPA is the product result. To make the multiplier approximating, exact counters are replaced by approximate counters. More details are provided below.

4.1 Counters

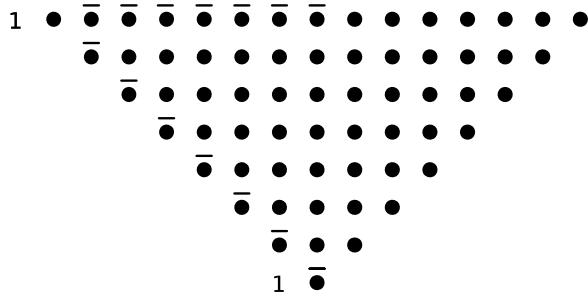
Counters are commonly used components in multipliers. An $(n; m)$ counter takes n input bits and computes their m bit sum. Common examples are the $(3; 2)$ counter (full-adder) and the $(2; 2)$ counter (half-adder).

The term *exact counter* is used if the sum of its n input bits can be correctly represented in m output bits; $m \geq \lceil \log_2(n) \rceil$. A counter is *approximate* if $m < \lceil \log_2(n) \rceil$. Approximate counters are faster than exact counters, but their output is not correct for all input combinations. The output sum is truncated to m output bits with the upper sum bits discarded because they are the least likely to be asserted.

4.2 Multiplier Topology

Partial products are generated from the input operands, and can be represented in dot notation as in Fig. 1 where each dot represents a logical AND of input bits from

Fig. 1 Partial products using dot notation for a signed 8×8 bit Baugh–Wooley multiplier



the multiplicand and multiplier. The dots are allocated to columns according to their numerical significance. Figure 1 also shows how the Baugh–Wooley method [3] has been used to handle 2’s complement signed operands. This method inverts some of the partial product bits and inserts extra partial product bits of fixed logical value of ‘1’.

Partial products bits of the same significance (in the same column) are grouped and form inputs to $(n; m)$ counters. This is repeated until there are one or zero remaining partial products for each level of significance. Each counter’s m output bits are distributed to the appropriate column and form the partial products for the next level.

The construction of an 8×8 bit tree multiplier that uses 61 $(3; 2)$ counters is shown in Fig. 2. The tree structure of the multiplier requires 4 levels, and a final CPA 11 bits wide that is shown as a dashed line in the figure. This multiplier is exact: its probability of correctness is 100%.

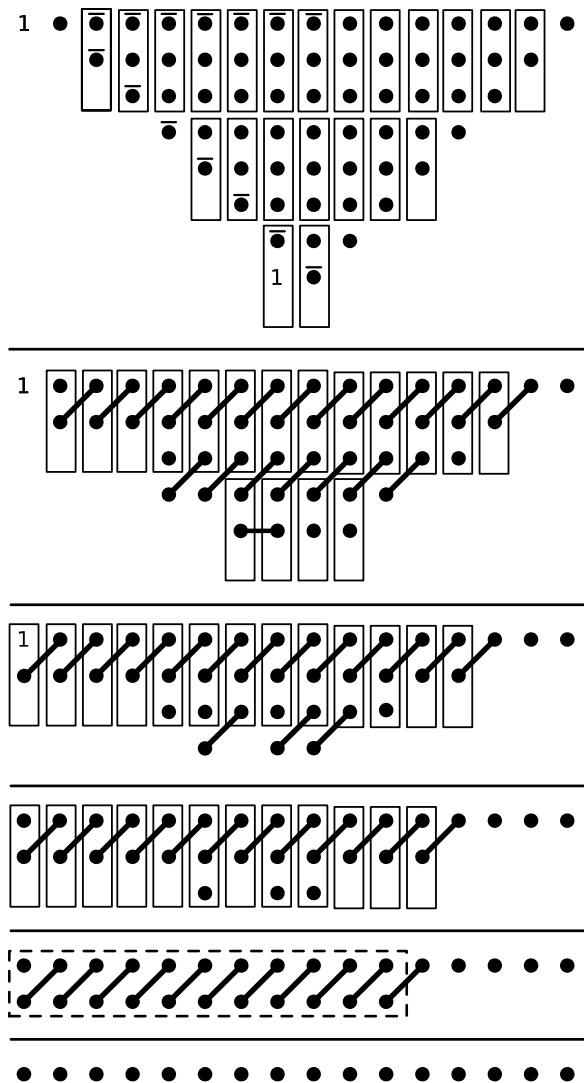
An approximate $(4; 2)$ 8×8 bit multiplier is shown in Fig. 3. The tree multiplier is constructed with approximate $(4; 2)$ counters instead of exact $(3; 2)$ counters. In this case only 31 $(4; 2)$ counters are needed in 2 levels, requiring a 13 bit CPA. The multiplier product is incorrect when one or more approximate counters discard an asserted sum bit. Simulation of all input combinations to the $(4; 2)$ 8×8 bit signed multiplier shows an average probability of correctness for uniformly distributed random inputs of 71.52%.

For the multipliers described here, one counter type is used throughout the tree. At each level of the tree, partial product bits are input to as few counters as possible. Where counters had open inputs or outputs, the synthesizer tool was left to optimize the logic.

4.3 Multiplier Results

Signed $(n; m)$ 32×32 bit multipliers were simulated operating on data collected from *MediaBench* programs. Each product was identified as correct or incorrect, and the probability of correctness of each multiplier was calculated as the proportion of correct multiplications. The probability of correctness of signed Baugh–Wooley tree

Fig. 2 Exact signed (3; 2)
8 × 8 bit tree multiplier



multipliers with homogeneous approximate counters is shown in Table 1. Further details of the multiplier topology can be found in [11].

A high probability of correctness was obtained because typical multiplication operands are small and positive. 33.6% of signed operands were zero, and only 7.6% were negative. Figure 4 shows the cumulative distribution of the absolute magnitude of the input operands.

The multipliers were synthesized from VHDL to determine their critical path delay. (Detailed area and power measurements can be found in [11].) Figure 5 shows the delay versus probability of correctness for signed ($n; m$) 32 × 32 bit multipliers. The shaded region of interest is expanded in Fig. 6. Multipliers in this region

Fig. 3 Approximate signed (4; 2) 8×8 bit tree multiplier

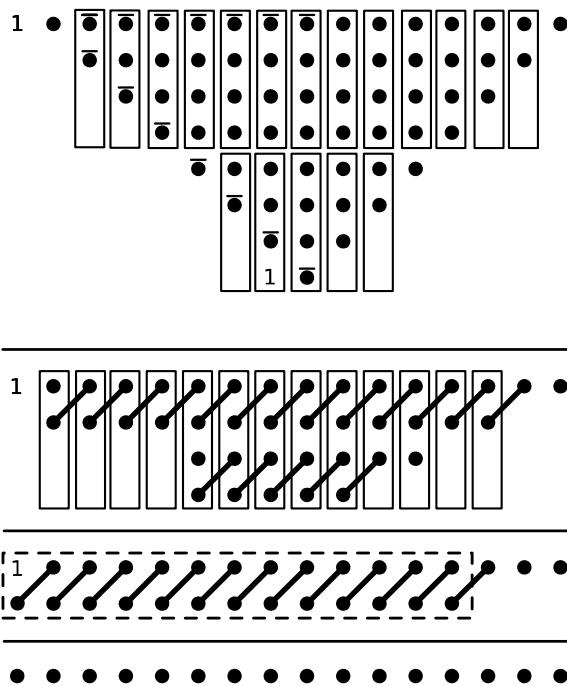


Table 1 (n ; m) Multipliers: Probability of correctness (%) for signed 32×32 bit multipliers constructed from counters with n input bits and m output bits, operating on benchmark data

		m				
		1	2	3	4	5
n	2	0.00				
	3	0.00	100.00			
	4	0.00	88.80	100.00		
	5	0.00	87.97	100.00		
	6	0.00	87.38	100.00		
	7	0.00	87.05	100.00		
	8	0.00	86.31	98.53	100.00	
	9	0.00	85.97	98.38	100.00	
	10	0.00	85.28	98.04	100.00	
	11	0.00	85.43	98.14	100.00	
	12	0.00	85.36	97.98	100.00	
	13	0.00	85.35	97.97	100.00	
	14	0.00	85.28	97.84	100.00	
	15	0.00	84.95	97.08	100.00	
	16	0.00	85.38	96.58	98.54	100.00

Fig. 4 Cumulative distribution function of the absolute magnitude of multiplication operands in benchmark programs

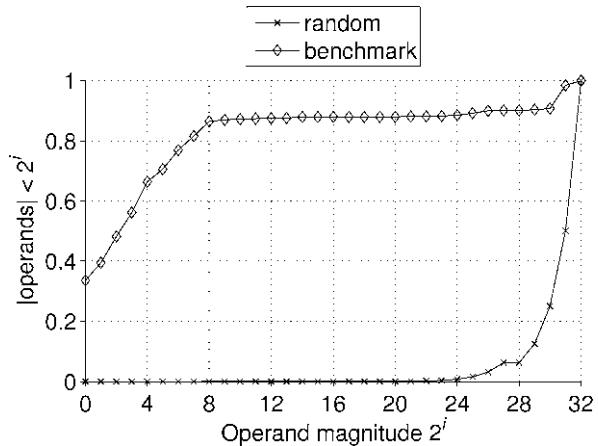
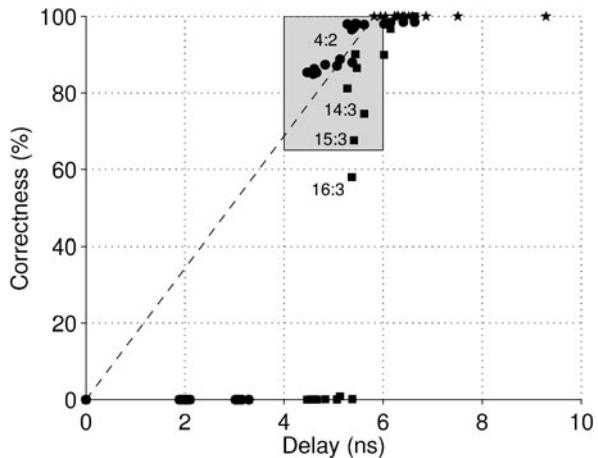


Fig. 5 Scatter plot of signed $(n; m)$ 32×32 bit multipliers, showing multiplier delay vs. correctness. Data points for random data are shown as squares; data points for *MediaBench* data are shown as circles



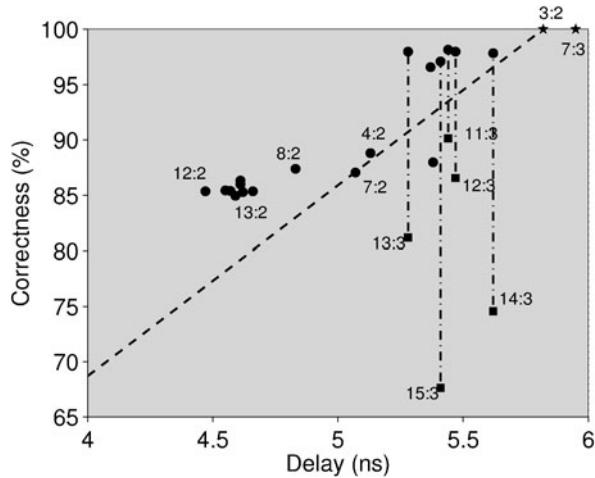
have a high probability of correctness. Some interesting labelled cases are discussed below. Note that the (3; 2) (full-adder) multiplier corresponds to a conventional, high-performance, exact design. The dashed line indicates linear trade-off with this baseline.

The correctness of the approximate multipliers is higher for *MediaBench* data than uniformly random data due to the distribution of operands: benchmark operands are much smaller in magnitude and contain a high proportion of zeroes.

In general, the higher the degree of truncation of the counter, the lower probability of correctness of the multiplier. An exception is the (11; 3) multiplier compared to the (10; 3) multiplier. It is possible that in later levels in the tree, accumulated carries and sums in a particular column will be susceptible to a few pathological cases, where a smaller counter may be allocated more asserted bits.

Many factors affect the delay of the multiplier circuits. Delay is comprised of delay of the counters, the number of levels in the tree, interconnect delay and delay

Fig. 6 Zoomed region of the shaded scatter plot in Fig. 5



through the CPA. As the number of counter output bits m increases, the additional counters required dominate, and delay increases. As the number of counter input bits n increases, the delay of each counter increases, but fewer counters are required. Additionally, as n increases, the number of levels in the tree can decrease, reducing delay, but increasing the width of the CPA. Small changes to the width of the CPA have marginal effect on its delay when a fast tree adder is used.

Many of the multipliers in Fig. 6 exhibit a better than linear trade-off of probability of correctness for delay compared to the signed (3; 2) multiplier. Multipliers such as the (4; 2) and (7; 2) multipliers deliver a small reduction in delay for a small penalty to probability of correctness. Another cluster, including the (8; 2), (12; 2) and (13; 2) multipliers are slightly faster again, with further degraded probability of correctness. For multipliers constructed of counters with a high degree of truncation, such as the (13; 2) counter, the probability of correctness is highly dependent on the input data distribution.

5 Approximate Unsigned Division

Although division is an infrequent operation in general purpose computing, poor implementations can degrade the performance of many programs [20]. This problem is exacerbated in wide-issue processors. Hence integer division is suitable for approximation: it is a long latency arithmetic operation that is often on the critical path of execution.

This section reviews an approximate unsigned 32 bit integer divider from [12], that was developed from a new division algorithm, devised specifically for ADVS.

5.1 Division Algorithm

Given a divisor d and dividend z , a divider calculates the integer quotient q and integer remainder $0 \leq r < d$ to satisfy:

$$\frac{z}{d} = q + \frac{r}{d}. \quad (1)$$

To produce an approximate division algorithm, we began with the following recurrence using \tilde{d} , an approximation to the divisor d :

$$\begin{aligned} q_0 &= 0, \\ r_0 &= z, \\ q_{i+1} &= q_i + \frac{r_i}{\tilde{d}}, \end{aligned} \quad (2)$$

$$r_{i+1} = \frac{-r_i(d - \tilde{d})}{\tilde{d}}. \quad (3)$$

Note that each round requires two divisions by \tilde{d} . The algorithm was motivated by the idea of selecting values of \tilde{d} to simplify these division steps.

Provided $|d - \tilde{d}| < |d|$ and given sufficient precision and enough rounds, this recurrence will converge on the exact quotient and remainder. We make it approximating by limiting the precision of intermediate results, terminating before an exact result is guaranteed, and by only performing an approximate multiplication for $r_i \times (d - \tilde{d})$ in (3).

A fixed-point number representation with f fractional bits was used for the intermediate values q_i and r_i . The algorithm was terminated after t rounds.

To simplify the divider hardware, we selected \tilde{d} to be the smallest binary power greater than or equal to d . This allowed the divisions by \tilde{d} in (2) and (3) to be implemented as a right shift. Furthermore, the shift amount is constant, and hence was only calculated once in the initialization stage of the divider.

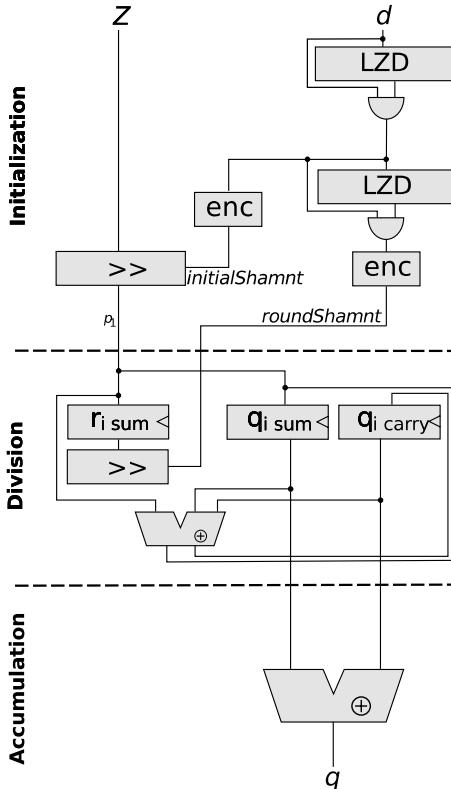
Experiments showed that approximating the multiplication $r_i \times (d - \tilde{d})$ in (2) had only a minor effect on the probability of producing a correct quotient. This is because the fractional bits of r_i and q_i are eventually discarded, and the magnitude of the partial remainder r_i decreases in each round. For the approximate divider only the most significant non-zero bit of $(d - \tilde{d})$ was used to approximate the product of $r_i \times (d - \tilde{d})$. This bit is constant throughout the division and hence was only detected once. In hardware, this approximation simplified the multiplier to a shift operation.

5.2 Divider Implementation

The approximate divider was implemented with three stages:

Initialisation The divisor d is inspected to determine the next greater binary power \tilde{d} . This is encoded as an initial shift amount (*initialShift*), and is

Fig. 7 An approximate divider. Control elements such as multiplexors, and timing elements and signals such as clocks are not shown



used to shift z from the initial value q_1 in the iteration. The next most significant bit is used to determine the approximate divisor \tilde{d} . As this value is a power of two, and is applied in (3) in each division round, it is stored as the round shift amount, $roundShamt$.

Division The recurrence in (2) and (3) is performed for t rounds. Repeated shifts and subtractions implement the iteration for the number of specified rounds. Intermediate results are kept in stored carry form with a carry save adder (CSA) for efficiency.

Accumulation The stored carry result for q is converted to non-redundant form using a carry propagate adder (CPA).

The hardware implementation is shown in Fig. 7. It uses a leading zeroes detector (LZD), encoders (enc), variable logical right shifters (\gg), latches, carry save adders (CSA), carry propagate adders (CPA). The approximation to $r_i \times (d - \tilde{d})$ is labelled p_1 .

The approximate divisor \tilde{d} could be chosen to be either power of two nearest to d . In this case \tilde{d} is the power of $2 \geq d$, hence the divider is called the greater binary power (GBP) divider. This implementation aggressively reduces the number of division iterations, width of data structures and data paths between each stage. Other design possibilities are presented in [12].

The division round is fast because the shift operation is simple, negating the need for the quotient digit look-up tables used in SRT division.

5.3 Divider Results

The approximate divider was simulated with operands extracted from the *MediaBench* suite. The probability of correctness was measured over a range of fractional bit lengths f , and division rounds t . The results are shown in Fig. 8. Despite the gross simplifications in the approximate divider, the probability of correctness obtained was over 99% with $f = 3$ fractional bits and $t = 7$ division rounds.

The input operands can be considered in three cases. Firstly, when the divisor is small or zero, the initial shift amount is small, so there are few fractional bits. There are also few division rounds required for the convergence.

Secondly, when the most significant bit (MSB) in the dividend is close to the MSB of the divisor, the quotient is small. The initial shift amount is large, and the number of division rounds required for a high probability of correctness depends on the magnitude of the operands. This is shown in Fig. 9, where the operands are biased uniform random inputs. The operands are biased such that $d < z$ because it was observed that this is rarely the case in benchmark data.

Thirdly, the case where the number of division rounds and fractional bits must be large to result in high correctness is when the dividend MSB is large, and the divisor MSB is approximately half the dividends. In this case the magnitude of the quotient is near the divisor and the algorithm must operate for many rounds to converge because the least significant bits of the quotient toggle in each division round due to carries rippling through the fractional bits.

The latency of the approximate divider was compared to that of a high-performance radix 4 SRT divider [21]. The SRT algorithm requires that both operands are normalized, so the operands are scaled before and after the operation. The SRT algorithm can terminate when all of the integer digits in the quotient

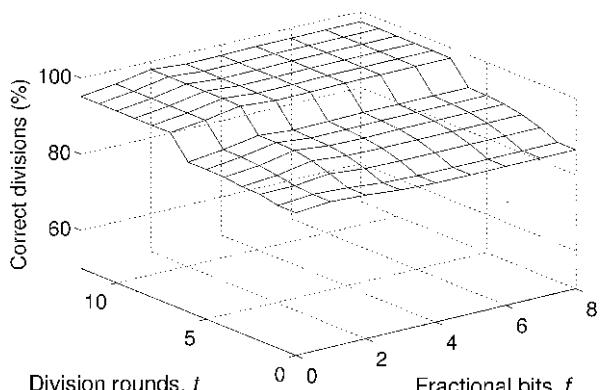
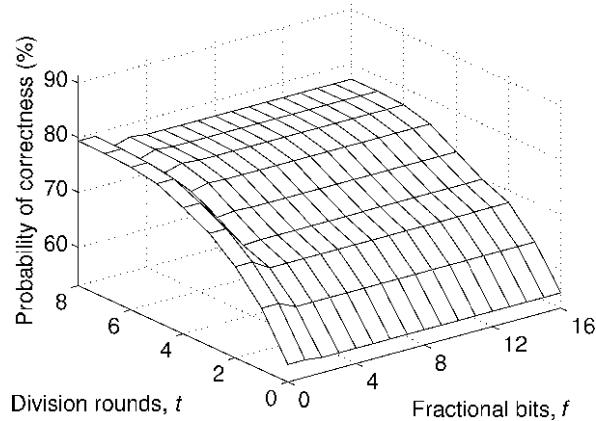


Fig. 8 Average probability of correctness of the approximate GBP divider operating on *MediaBench* data

Fig. 9 Average probability of correctness of approximate GBP divider operating on biased uniform random inputs



are calculated. Like the GBP divider, an SRT divider can be terminated early to obtain an approximate result. Figure 10 shows that the SRT divider required 6 division cycles for 90% probability of correctness on *MediaBench* data. 8 cycles are required for 99% probability of correctness.

Both dividers require initialization, division and post-processing cycles. For 99% probability correctness, the SRT divider required 8 division cycles, one pre-shift and one post-shift cycle: a total of 10 cycles. The approximate GBP divider required 7 division cycles, one accumulation cycle, and 2 initialization cycles because the initialization stage was broken into 2 stages to balance the pipeline delay. Hence both dividers require 10 cycles in total.

To determine the correctness vs. delay tradeoff of the approximate SRT divider and the GBP divider, they were synthesized from VHDL descriptions. Table 2 shows the results. For the same probability of correctness using *MediaBench* inputs and the same number of machine cycles required to perform the approximate division, the new divider is faster than an SRT divider due to the reduced clock period. In both

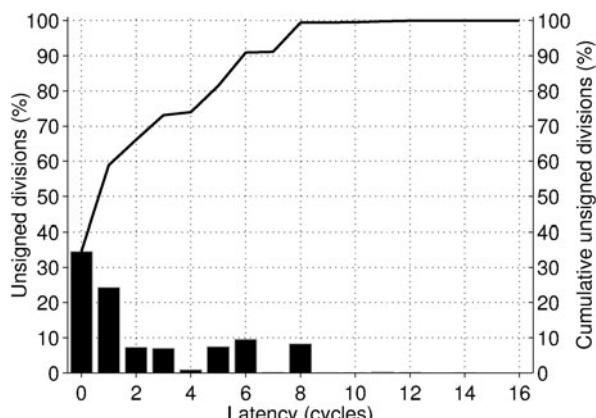


Fig. 10 Histogram of the average probability of correctness of the baseline radix-4 SRT divider when the number of division rounds are restricted. The distribution bars are measured on the left y-axis, and the cumulative line plot on the right y-axis

Table 2 Results from synthesis of 32/32 bit dividers. Each design was optimized for delay. The minimum achieved clock period is shown for typical operating conditions and wire-load

Design	Clock (ns)	Cycles	Latency	
			(ns)	(%)
Approximate radix-4 SRT divider ^a	4.45	10	44.50	100.0
New approximate divider	3.45	10	34.5	77.5

^aSRT latency is based on 8 division rounds so its probability of correctness for *MediaBench* inputs is the same as for the new approximate divider

cases, the minimum clock period was set by the initialization or accumulation stage. Power and area were also measured and are reported in [12].

6 Simulation of a RISC Processor with ADVS

This section presents the results of simulation of the *SimpleScalar* RISC processor with ADVS and operand caching, and contains the new contributions of this work. ADVS was used for signed and unsigned integer multiplication and unsigned division instructions. Operand caching was used for these instructions and also for signed divisions and floating point multiplications, divisions and square roots.

6.1 Operand Cache Simulation

Operand caches were simulated using CACTI 4.1 [9] for a 180 nm process to determine their size, access times and power consumption. A separate cache was maintained for each arithmetic operation. A total of 2^6 entries was selected as the size of each cache, with 4 way associativity. Up to 2^{12} entries was tested, but increasing the size of the caches above 2^6 entries increased performance only marginally. The caches were implemented aggressively to constrain the read time to within 5 ns; the cycle latency of the approximate dividers. Table 3 shows the total size in bytes required for each cache to store the input operands and result. Table 4 shows the area, power and delay estimates from CACTI.

The behavioral characteristics of the 180 nm caches were integrated into the *SimpleScalar* model and used to determine the cache hit rates and impact on throughput. The caches were assumed to have a 1 cycle read latency, and were read in parallel with the instruction execution. Whenever a result was found in an operand cache, the cached result was used in preference to the results from the approximate or exact arithmetic units.

Table 3 Total size in bytes of the operand caches, per line. The maximum size in bytes is shown in brackets

Cache	Operand (B)	Result (B)	Size (B)
Multiplication	2×4	2×4	16 (64)
Division	2×4	1×4	12 (48)
FP multiplication	2×8	1×8	24 (96)
FP division	2×8	1×8	24 (96)
FP SQRT	1×8	1×8	16 (64)

Table 4 180 nm result caches process using CACTI 4.1

Operation	Entry (B)	Ways	Lines	Total size (kB)	Access (ns)	Read Pwr. (W)	Area (mm ²)
Multiplication	12	4	64	3	1.402	0.213	0.726
Division	12	4	64	3	1.402	0.213	0.726
FP multiplication	24	4	64	6	1.393	0.532	1.026
FP division	24	4	64	6	1.393	0.532	1.026
FP SQRT	16	4	64	2	1.401	0.215	0.649

6.2 SimpleScalar Simulation

The 14 tests from the *MediaBench* suite were simulated with the modified version of *SimpleScalar*. The baseline simulations were first run without operand caching or ADVS. The exact arithmetic units were pipelined, and the number of stages was determined using timing information from VHDL synthesis. The default *SimpleScalar* cycle latencies were changed in the baseline model where they did not match the cycle latency determined from synthesis.

The *SimpleScalar* model was then modified to include operand caching and ADVS. The simulations were performed again, including functional models of the approximate arithmetic units with faster latencies extracted from synthesis. The approximate arithmetic units were configured to obtain peak correctness. Table 5 shows the cycle latencies used in simulation.

ADVS is beneficial above a threshold probability of correctness, otherwise if there are too many erroneous speculations the cost of repeatedly flushing the pipeline reduces performance. The minimum correctness of the approximating units was shown to be approximately 95% in a previous study [13]. The 95% correctness target is shown as a dashed line.

Caching generally increases the proportion of correct operations for each unit because arithmetic operations are repeated many times in typical programs. For repeated operations, a speculation flush is only performed once while the result remains cached, because the cache conclusively determines the result before the approximate result is returned.

Table 5 Default arithmetic latencies in *SimpleScalar*, and modified values derived from synthesis. These were used in the *SimpleScalar* simulations of the baseline and ADVS-enabled processors

Arithmetic unit	Default cycles	Exact		Approx.	
		period (ns)	cycles	period (ns)	cycles
Unsigned multiplier	3	3.41	3	3.77	2
Signed integer multiplier	3	3.46	3	3.77	2
Unsigned integer divider	20	4.45	18	3.45	10

Figure 11b shows the average probability of correctness of the approximate arithmetic units in the ADVS-enabled pipeline. Figure 11b shows the changed probability of correctness when operand caching is introduced. All issued instructions were included in the totals, even if they were issued speculatively.

Figure 11c shows the average hit-rates of all of the operand caches in each benchmark. All caches were sized to 4 K entries, with indexing on data value, 4 way associativity and FIFO replacement. Many of the caches show hit rates above 50% because of the highly repetitive operand values used in most programs.

Figure 11d shows the resulting average improvement in throughput measured in IPC. All of the benchmarks were compiled with different optimization levels of the gcc compiler, shown in the legend. The highest performing benchmarks such as *ghostscript* and *mesa* improved throughput due a high probability of correctness for speculated approximate arithmetic, and high hit rates in the operand caches. Maintaining separate caches for different arithmetic operations is beneficial because infrequent operations like floating point square root would otherwise be evicted if the caches were unified.

The average increase in IPC of the *MediaBench* benchmarks was and 3.96% with operand caching, but varied considerably between benchmarks. Both ADVS and operand caching favor programs with a high density of arithmetic operations, particularly long latency operations such as division. *ghostscript* and *mesa* both contain many division operations. However, despite high cache hit rates and arithmetic correctness, benchmarks such as *adpcmencode*, *adpcmdecode* and *epic* contained fewer than 0.2% long latency arithmetic operations, limiting the potential gain. The average proportion of arithmetic operations per benchmark is shown in Table 6. The maximum potential gain for ADVS and result caching is determined by the density of arithmetic operations in the program.

A pathological case for throughput gain was the *jpeg* benchmark. The program contains a relatively high proportion of integer multiplication and division operations, but they were poorly approximated and not very repetitive. Hence, caching was of limited benefit, and the flush penalties from ADVS resulted in an overall small drop in performance.

A high proportion of arithmetic operations is necessary but not sufficient to improve IPC with ADVS and operand caching. The distribution of operand values determines the probability of correctness of the approximate arithmetic units, and operand repetitiveness determines the ability to re-use data values with result caches.

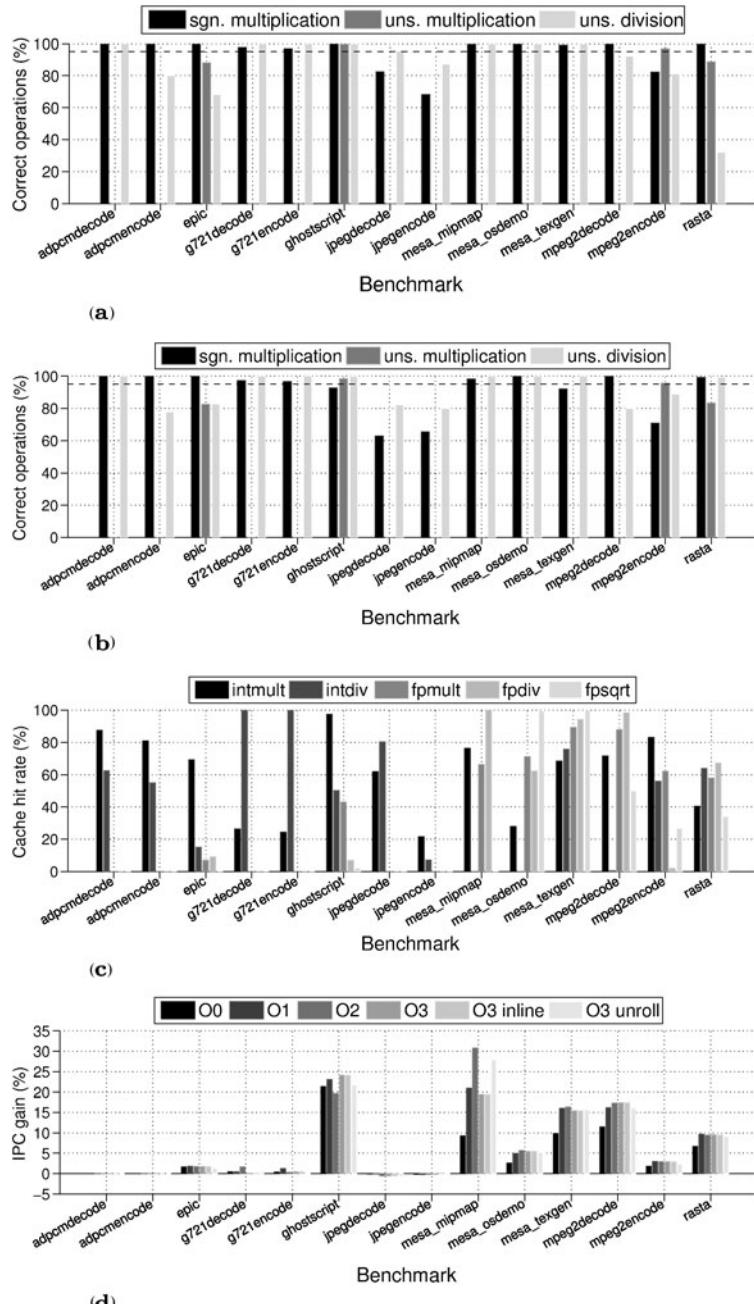


Fig. 11 (a) Correctness of arithmetic units in an ADVS-enabled pipeline. (b) Correctness of arithmetic units with operand caching enabled. (c) Hit rates of the operand caches. (d) Throughput increase with ADVS and operand caching enabled

Table 6 Average proportions of arithmetic operations in benchmark programs. All operations are shown as a percentage of the total number of retired instructions. In *SimpleScalar* floating point division and square-root share common hardware. Matching pairs of benchmarks such as encode/decode are grouped into a single result

Benchmark	Integer				Floating point					
	Signed		Unsigned		Single			Double		
	mult	div	mult	div	mult	div	sqrt	mult	div	sqrt
adpcm	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
epic	0.190	0.060	0.000	0.000	5.070	0.000	0.000	0.000	0.120	0.000
gsevenwoone	1.100	0.000	0.000	0.080	0.000	0.000	0.000	0.000	0.000	0.000
ghostscript	1.130	0.010	0.000	1.130	0.000	0.000	0.000	0.010	0.000	0.000
jpeg	0.820	0.050	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
mbmesa	1.690	0.080	0.000	0.000	11.200	0.450	0.000	4.370	0.560	0.040
mpegtwo	0.290	0.100	0.000	0.000	0.000	0.000	0.000	5.590	0.000	0.000
pegwit	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
rasta	0.150	0.030	0.320	0.110	1.510	0.060	0.000	1.400	0.130	0.020
Average	0.597	0.037	0.036	0.147	1.976	0.057	0.000	1.263	0.090	0.007

In general, programs that contain 0.5% or more of long latency arithmetic operations can benefit from arithmetic data value speculation and operand caching.

7 Conclusions

This chapter reviewed two approximate arithmetic units suitable for arithmetic data value speculation: a multiplier and a divider. For operands from media benchmarks, these produced correct results over 85% of the time. The approximate multiplier executed a cycle faster than a baseline exact multiplier and the approximate divider saved 8 cycles compared to an exact version.

New results were presented for a 32 bit RISC processor using arithmetic data value speculation for integer multiplication and division and using operand caching for all long-latency arithmetic instructions. These architectural enhancements improved the instruction throughput of the processor by over 15% for some media benchmarks. None of the benchmarks simulated showed a significant degradation in throughput.

Operand caching and arithmetic data value speculation work together to deliver this improved throughput. Without operand caching, arithmetic data value speculation can suffer from repeated mis-speculation of a few frequently used results.

ADVS and operand caching can be applied to yield significant throughput gains in certain programs. Good candidate programs have an arithmetic density of at least 0.5%. Programs that frequently repeat operations with the same operands are suitable for operand caching. Further inspection of typical operand values can provide

insight in to the likely correctness of the approximate arithmetic units. In the case of the units presented in this chapter, operands with a small magnitude are more likely to be correctly approximated.

Acknowledgement The authors would like to thank *eResearch SA* for their support and resources used to complete the simulations used for this chapter.

References

1. Akgul B, Chakrapani L, Korkmaz P, Palem K (2006) Probabilistic CMOS technology: a survey and future directions. In: 2006 IFIP international conference on very large scale integration, pp 1–6
2. Artisan Components, Inc., Sunnyvale, CA, USA (2002) TSMC 0.18 μm Process 1.8 Volt SAGE-XTMStandard Cell Library Databook
3. Baugh CR, Wooley BA (1973) A two's complement parallel array multiplication algorithm. IEEE Trans Comput C-22:1045–1047. Reprinted in Swartzlander EE, Computer arithmetic, vol 1. IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990
4. Burger D, Austin TM (1997) The SimpleScalar tool set version 2.0. SIGARCH Comput Archit News 25(3):13–25
5. Burger D, Goodman JR (2004) Billion-transistor architectures: there and back again. IEEE Comput Mag 37(3):22–28
6. Das S, Roberts D, Lee S, Pant S, Blaauw D, Austin T, Flautner K, Mudge T (2006) A self-tuning DVS processor using delay-error detection and correction. IEEE J Solid-State Circuits 41(4):792–804
7. Das S, Tokunaga C, Pant S, Ma WH, Kalaiselvan S, Lai K, Bull D, Blaauw D (2009) RazorII: in situ error detection and correction for PVT and SER tolerance. IEEE J Solid-State Circuits 44(1):32–48
8. George J, Marr B, Akgul BES, Palem KV (2006) Probabilistic arithmetic and energy efficient embedded signal processing. In: Hong S, Wolf W, Flautner K, Kim T (eds) Proceedings of the 2006 international conference on compilers, architecture, and synthesis for embedded systems, CASES 2006, Seoul, Korea, October 22–25, 2006. ACM, New York, pp 158–168
9. HP labs: Hp labs: cacti (2008) online: <http://www.hpl.hp.com/research/cacti/>
10. Kelly DR, Phillips BJ (2005) Arithmetic data value speculation. Lect Notes Comput Sci (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol 3740. Springer, Singapore, pp 353–366
11. Kelly DR, Phillips BJ, Al-Sarawi SF (2009) Approximate signed binary integer multipliers for arithmetic data value speculation. In: Proceedings of the conference on design and architectures for signal and image processing (DASIP). Sophia Antipolis, France. <http://www.ecsi-association.org/ecsi/dasip/dasip09>
12. Kelly DR, Phillips BJ, Al-Sarawi SF (2009) Approximate unsigned binary integer dividers for arithmetic data value speculation. In: Proceedings of the conference on design and architectures for signal and image processing (DASIP). Sophia Antipolis, France. <http://www.ecsi-association.org/ecsi/dasip/dasip09>
13. Kelly DR, Phillips BJ, Al-Sarawi SF (2009) Increasing throughput of a RISC system using arithmetic data value speculation. In: Conference record of the forty-third Asilomar conference on signals, systems, and computers, 2009. IEEE, Pacific Grove
14. Korkmaz P, Akgul B, Palem K (2008) Energy performance, and probability tradeoffs for energy-efficient probabilistic CMOS circuits. IEEE Trans Circuits Syst I, Regul Pap 55(8):2249–2262
15. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: Proceedings of the annual international symposium on microarchitecture. IEEE, Triangle Park, pp 330–335

16. Li A (2002) An empirical study of the longest carry length in real programs. Master's thesis. Department of Computer Science, Princeton University
17. Lipasti MH, Shen JP (1996) Exceeding the dataflow limit via value prediction. In: Proceedings of the 29th annual IEEE/ACM Int. symposium on microarchitecture. IEEE, Washington, pp 226–237
18. Liu T, Lu SL (2000) Performance improvement with circuit level speculation. In: Proceedings of the 33rd annual international symposium on microarchitecture. ACM, New York, pp 348–355
19. Lu SL (2004) Speeding up processing with approximation circuits. *IEEE Comput Mag* 37(3):67–73
20. Oberman SF, Flynn MJ (1997) Design issues in division and other floating-point operations. *IEEETC: IEEE Trans Comput* 46
21. Parhami B (2000) Computer arithmetic: algorithms and hardware designs. Oxford University Press, New York
22. Phillips BJ, Kelly DR, Ng BW (2006) Estimating adders for a low density parity check decoder. In: Proceedings of SPIE—the international society for optical engineering, Proc. SPIE vol 6313. SPIE, Bellingham
23. Richardson SE (1992) Caching function results: faster arithmetic by avoiding unnecessary computation. Tech. Rep., Mountain View, CA, USA
24. Tarjan D, Thoziyoor S, Jouppi NP (2006) Cacti 4.0. Tech. Rep. HPL-2006-86, HP Labs. Available online <http://www.hpl.hp.com/techreports/2006/HPL-2006-86.html>
25. Wang L, Shanbhag N (2000) Adaptive error-cancellation for low-power digital filtering. In: Conference record of the thirty-fourth Asilomar conference on signals, systems, and computers. IEEE, Pacific Grove
26. Yang L, Tong Z, Parhi K (2008) Analysis of voltage overscaled computer arithmetics in low power signal processing systems. In: Conference record of the forty-second Asilomar conference on signals, systems, and computers. IEEE, Pacific Grove

RANN: A Reconfigurable Artificial Neural Network Model for Task Scheduling on Reconfigurable System-on-Chip

Daniel Chillet, Sébastien Pillement,
and Olivier Sentieys

Abstract With increasing embedded application complexity and flexibility requirements, hardware designers more and more frequently introduce reconfigurable pieces of silicon inside System-on-Chip architectures. In this type of specific hardware blocks, several tasks can be allocated and executed simultaneously, and the number of running tasks supported by this reconfigurable hardware depends on its total number of elementary computing elements and on the number of elementary computing elements of each application task. For this context, the problem of the task scheduling and the task placement is NP-hard. To efficiently answer to this problem, specific resource constrained services must be developed to ensure the correct application execution on the reconfigurable resource. This paper presents the model of a Reconfigurable Artificial Neural Network (RANN) developed for task scheduling and placement on the reconfigurable resource of a System-on-Chip architecture. An adaptation of the Hopfield model is proposed with a regular reconfiguration of the Artificial Neural Network (ANN). Simulation results, which evaluate the RANN convergence, show significant improvement in comparison with the previous techniques. We also present how the RANN structure can be implemented and we discuss some optimizations in terms of hardware cost.

1 Introduction

Embedded systems are usually implemented on complex Systems-on-Chip (SoCs) which are built around heterogeneous processing units. Recent developments propose to implement general purpose processor (GPP) cores, specialized Intellectual Property (IP) blocks and Dynamically Reconfigurable Accelerators (DRA) in the

D. Chillet (✉)

University of Rennes 1, IRISA/INRIA, BP 80518, 6 rue de Kerampont, F22305 Lannion, France
e-mail: Daniel.Chillet@irisa.fr

same circuit, thus designing a so-called Reconfigurable System-on-Chip (RSoC). The increasing number of resources included in modern RSoCs requires the use of a software abstraction layer to efficiently manage such complex architectures. Classically, the system is organized around a GPP which runs an Operating System (OS) in charge of the management of the available computing resources [1, 2].

In this context, different characteristics of the application behavior and the hardware platform have an influence on the required OS services. Firstly, the system is composed of multiple heterogeneous execution resources, with potentially different execution delays for the application tasks. The latter characteristic defines a heterogeneous multiprocessor architecture which requires specific management [3]. Secondly, to ensure an efficient use of the architecture and an optimized execution of the application, the OS must manage the dynamicity provided by the DRA [4]. This feature can support the flexibility required by the application execution, by providing dynamic and partial reconfigurations of the architecture. Through this concept, it is possible to configure one part of the circuit without disturbing the other parts of the circuit. Recent FPGA circuits, such as Xilinx Virtex 5 family, support this possibility. Nevertheless, the overhead cost (in time and in energy consumption) of the bitstream loading during reconfiguration must be limited as much as possible. For this reason, the task model for the DRA resource is generally defined as a non-preemptive model. The conditions allowing more complex hardware task preemptions are defined in [5]. Thirdly, unlike classical processor core, the DRA resource is limited by its number of elementary computing elements (e.g. Configurable Logic Blocks, CLB, memory or arithmetic blocks). So task management must be able to schedule and place the execution of several simultaneous tasks, leading to a variable number of tasks running simultaneously.

In this paper, we propose a resource constraint scheduling and placement service for a reconfigurable resource included in an RSoC. This service is also called *spatio-temporal scheduling*. In our model, the scheduler manages non-preemptive tasks with precedence constraints between tasks. One optimization objective is the efficient usage rate of the DRA resource area (resource constraint). The tasks are supposed to be defined in columns, and a one-dimensional (1D) placement is considered. The rest of this paper is organized as follows. The problem definition is exposed in Sect. 2. Section 3 presents related works on scheduling techniques for multiprocessor systems, scheduling through Artificial Neural Network (ANN) models and some hardware implementations of ANN in FPGA circuits. In Sect. 4, the Reconfigurable Artificial Neural Network (RANN) implementing our proposed spatio-temporal scheduling is explained. Section 5 discusses the dynamic behavior of the RANN. Section 6 presents simulation results of the RANN, while Sect. 7 gives results of hardware implementation of the proposed scheduler. Finally, Sect. 9 concludes and discusses our future work.

2 Problem Definition

The main problem that we want to tackle is the schedule and placement of a set of tasks into a limited reconfigurable area embedded into an RSoC. More precisely, in

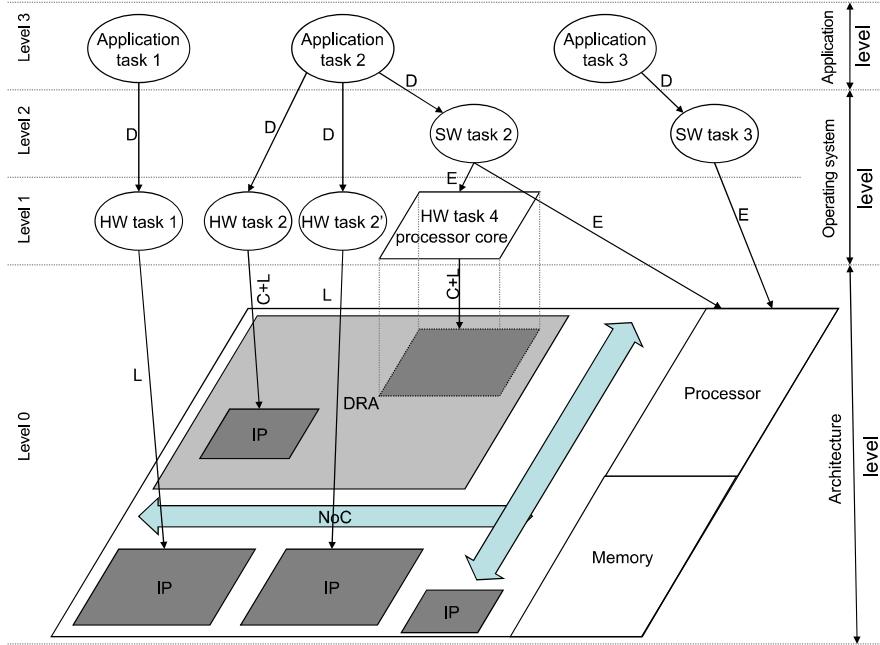


Fig. 1 Application mapping example on an RSoC. The application is defined as set of tasks, each is implemented by one or several OS tasks. These tasks are supported by the execution resources of the RSoC platform. In this example, application task 2 is defined by three following different OS tasks: HW task 2, HW task 2', and SW task 2. In this case, task 2 is called a *hybrid* task. On the other hand, tasks 1 and 3 are defined with only one OS task (i.e. for one execution target). Before their execution, some tasks need to be configured (C) and then launched (L)

addition to the classical temporal task allocation on execution resources, the spatial allocation must be ensured to define the position of each task within the reconfigurable area of the RSoC. For an RSoC, each task can be defined for several execution resources as shown in Fig. 1 where *D* arrows indicate task definitions for a particular software and/or hardware resources, *C* arrows represent a configuration step, and finally *L* arrows represent the task firing. The scheduler must decide at run-time on which resource the tasks should be instantiated. Due to the dynamicity of current applications, an on-line and dynamic scheduling algorithm is often advantageous.

Due to the number of constraints that must be satisfied, the spatio-temporal scheduling is known as an NP-hard problem. To manage this complexity, various heuristic methods were developed, like genetic algorithms [6], simulated annealing [7] and Artificial Neural Networks (ANNs) [8]. ANNs have demonstrated their efficiency for optimization problems that take several constraints into account. They converge in a reasonable time (i.e. a few cycles) if the number of neurons and connections between neurons can be limited as much as possible. Until now, ANNs have been used for task scheduling on homogeneous SoC architectures, i.e. SoC without reconfigurable hardware. These solutions cannot be applied directly to an RSoC which is intrinsically heterogeneous and could handle multiple task executions.

The main drawback of ANNs is the convergence time when the number of neurons increases. Some works have studied how ANNs can be implemented in hardware structure to reduce the convergence time [9]. The neuron state evolutions are naturally parallel and this leads to target parallel hardware structures. Furthermore, ANNs are frequently adapted through the modification of their inputs and weight connections, which leads to design adaptable structures. These two main characteristics can be very efficiently handled by FPGA circuits.

The main objective of our proposition is to support a variable number of tasks to schedule according to their area usage rate in the reconfigurable hardware. Our proposal also provides task placement within the reconfigurable area. For this purpose, we define a Reconfigurable Artificial Neural Network (RANN) which is updated at specific scheduling cycles. The task placement is done under resource constraint, which corresponds to a 1D task placement within the reconfigurable area (i.e. the tasks are placed in column into the reconfigurable resource). Our model takes task dependencies into account and supports a non-preemptive model of tasks. This characteristic ensures a minimal number of reconfigurations and then limits the energy consumption of the reconfigurable unit.

3 Related Works

After a review of task scheduling for heterogeneous multiprocessor platforms and task placement into reconfigurable area, this section discusses the use of ANN as a model for solving optimization problems. Since our objective concerns the management of a set of tasks into a reconfigurable resource, this section also reviews some works on hardware implementations of ANNs.

3.1 Temporal and Spatial Task Scheduling

Most of scheduling algorithms developed for real-time systems are limited by very specific and homogeneous constraints. Various algorithms have been proposed in the literature for periodic, sporadic or aperiodic tasks, allowing or not the task preemption, targeting monoprocessor or multiprocessor architectures, but rarely combining these properties together.

In the context of SoC architectures, specific task scheduling services have been proposed to tackle the heterogeneity characteristics of the execution resources [10, 11]. Implementing these services is often complex and is not always appropriate for real-time systems [12]. They are generally time consuming and do not consider the dynamic behavior of the applications. For example, the PFair algorithm [13] focuses on an optimal solution for periodic tasks on homogeneous multiprocessors. In [14], the authors have proposed an approximate solution to reduce global complexity and to design hardware implementations of the PFair approach. Nevertheless, this type of solutions is not adapted to manage the dynamicity of the DRA. Furthermore,

PFair introduces large number of task migrations which is not acceptable for the execution of the tasks onto a reconfigurable area of an RSoC.

Nowadays, most of SoCs embed a reconfigurable unit to deal with application flexibility and adaptability. The reconfigurable unit can support the execution of a high number of tasks but, in general, only few tasks are executed at the same time. The dynamic reconfiguration of this unit thus allows to sequentially execute tasks onto the same area, but a static scheduling is often insufficient to efficiently manage this execution resource. Due to the dependency between scheduling and placement functionalities, these two steps are often tightly coupled. In this case, the scheduling and placement OS services have been studied, and two major models are usually proposed: the one-dimensional (1D) or two-dimensional (2D) schemes to map tasks onto the reconfigurable unit [15, 16]. For the 1D scheme, the tasks are mapped on entire columns and the area occupied by the task is specified by the number of used columns. For the 2D scheme, the tasks are mapped on a rectangle of N logic cell columns by M logic cell rows. In this scheme, the task area is specified by the $N * M$ used logic cells. 2D techniques lead to the fragmentation of the reconfigurable area but the performances can be better than 1D techniques. Among the studies published about 1D and 2D task placements, we can cite the KAMER method (Keep All Maximum Empty Rectangles) which limits the area fragmentation by packing hardware tasks and keeping empty rectangles as large as possible.

Although static and dynamic schedulers are proposed, on-line solutions are often preferred, since they provide higher flexibility and dynamicity. In [17], an efficient placement and scheduling flow is proposed for hardware and software tasks. The authors show that the average overhead for task scheduling is equal to few dozens of ms. In [16], the authors show how a 2D on-line placement can be applied for hardware tasks which have long execution time. Furthermore, the authors consider the placement and scheduling problem as a multiprocessor scheduling. For example, in [18], the authors consider the mapping of real-time tasks onto 2D arrays of computers, which reveals that the time overhead of the placement and scheduling is too high to be used in the context of reconfigurable SoC. These works use specific algorithms and heuristics, which limit the number of tasks that can be handled by the scheduler.

3.2 ANNs Models for Task Scheduling

In [19], the authors have proposed the use of ANNs for on-line real-time scheduling by extending the results obtained in [20] for the Hopfield Artificial Neural Network model [21]. The theoretical basis on ANN for optimization problems is defined in [22] and in [23]. By using a Hopfield model, the authors of [19] ensure the existence of a Lyapunov function, called *energy function*. They show that the network

evolution converges towards a stable state for which the optimization constraints are respected. This energy function is defined as

$$E = -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N W_{ij} \cdot x_i \cdot x_j - \sum_{i=1}^N I_i \cdot x_i, \quad (1)$$

where N is the number of neurons, W_{ij} is the connection weight between neurons n_i and j , x_i is the state of neuron n_i , and I_i is the external input of neuron n_i . The state x_i of the neuron n_i is computed throughout the following expression

$$x_i = \begin{cases} 1 & \text{if } u_i > 0, \\ 0 & \text{if } u_i \leq 0, \end{cases} \quad (2)$$

with u_i the state of neuron n_i before threshold function. The neuron states are updated (potentially in parallel) according to the rule

$$u_i = \left(I_i + \sum_{j=1}^N x_j \cdot W_{i,j} \right). \quad (3)$$

Based on this model, a design rule that facilitates the ANN construction can be defined using equality or inequality constraints. The *k-outof-N* rule [20] allows the construction of an ANN of N neurons for which the evolution leads to a stable state with “*exactly k active neurons among N*”. This rule is an important result in ANN for several optimization problems. The corresponding energy function is defined as

$$E_{k\text{-outof-}N} = \left(k - \sum_{i=1}^N x_i \right)^2. \quad (4)$$

This function is minimal when the sum of active neurons is equal to k , and is positive in the other cases. Equation (4) can be rewritten in the form of (1) such as

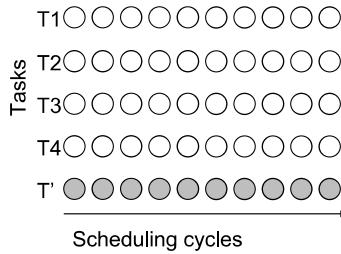
$$\begin{aligned} E_{k\text{-outof-}N} = & -\frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N W_{ij} \cdot x_i \cdot x_j - \sum_{i=1}^N I_i \cdot x_i \\ \text{with } & \begin{cases} W_{ij} = -2\overline{\delta_{i,j}} & \forall i, j \in (1, N), (1, N), \\ I_i = 2k - 1 & \forall i \in (1, N) \end{cases} \end{aligned} \quad (5)$$

and with $\overline{\delta_{i,j}}$ the inverse Kronecker function equals to zero if $i = j$, or to one in the other cases.

Cardeira and Mamperi demonstrated, in [19], the additive character of the Hopfield model and applied it to a monoprocessor architecture. In this case, the scheduling problem is modeled through ANN by the following representation (see Fig. 2):

- Neurons $n_{i,j}$ are arranged in an $N_t \times N_c$ matrix form, where the line i corresponds to the task T_i and the column j corresponds to the schedule time unit j . The number of time units N_c depends on the hyperperiod of tasks (i.e. the least common multiple of all the task periods) and N_t is the number of tasks.
- An active neuron $n_{i,j}$ indicates that, during the corresponding schedule time unit j , the task T_i is being executed.

Fig. 2 Classical structure used to model the scheduling problem with ANN for a monoprocessor architecture. T' is the fictive task added to manage the inactivity of the processor during schedule time steps. Each row represents a task while each column is a scheduling cycle. The connections between neurons ensure that the network converges toward a state which corresponds to a valid scheduling solution



- One specific line of neurons is added to model the possible inactivity of the processor during schedule time steps. These neurons are called *slack* or *hidden* neurons since they are not used to represent the solution. The total number of neurons to model this problem is $(N_t + 1) \times N_c$.

By using the *k-outof-N* rule on each line of neurons with $k = WCET_{T_i}$ (Worst Case Execution Time of task T_i), and on each column of neurons with $k = 1$, this structure can provide scheduling solution for monoprocessor architecture.

For homogeneous multiprocessor architectures with p execution resources, the problem can be modeled by the same structure with p fictive tasks and by applying a *p-outof-N* rule on each column.

In the case of heterogeneous multiprocessor architectures, several plans (or layers) of neurons are used to model the different execution resources [24] (see Fig. 3). The tasks are then defined by several WCETs, one for each layer ($WCET_{T_i, R_j}$) defined the WCET of task T_i on the resource layer R_j). New slack neurons are then necessary to manage the exclusive execution of each task on resources. The number of slack neurons to add for the resource layer R_j and the task T_i is equal to $WCET_{T_i, R_j}$. By applying a *k-outof-N* rule on each couple (T_i, R_j) with $k = WCET_{T_i, R_j}$, we can ensure that each set of neurons modeling the couple (T_i, R_j) can converge towards $WCET_{T_i, R_j}$ active neurons.

Figure 3 presents an example of an ANN with p resource layers. In this case, the total number of neurons increases by a factor p .

The main drawback of this structure concerns the high number of neurons to model the scheduling problem which depends on tasks and on numbers of scheduling cycles.

As a conclusion, we can note that none of the previous propositions has been able to efficiently manage the task scheduling on reconfigurable hardware. The main problem concerns the number of tasks that can be supported simultaneously on this resource. In our context, this number is not fixed and can evolve according to the area usage rate of each task.

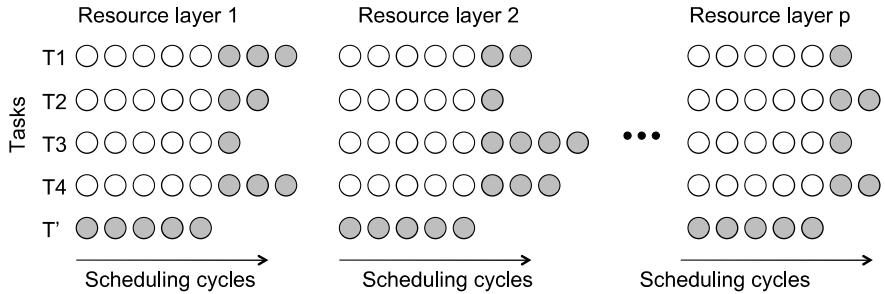


Fig. 3 Structure used to model the scheduling problem with ANN for a heterogeneous multiprocessor architecture with p execution resource layers. Grey circles represent the slack neurons. T' is the fictive task added to manage the inactivity of the processor during scheduling cycles

3.3 Implementation of ANNs

Software implementations of ANNs can be easily developed, but this type of implementations is not efficient because of the sequential evaluation of neurons. Indeed, the parallel evolution of neurons during the convergence step is not exploited, and this leads to very long simulation time. In order to alleviate this limitation, several works propose to study hardware implementation of ANNs. Due to the characteristics of the Hopfield model, FPGA circuits are good candidate for hardware implementations. In [25], the main advantages and drawbacks of FPGA circuits for ANN implementations are presented. The dynamic reconfiguration property permits to support dynamic evolution of ANN. The parallel evaluation of neurons thanks to the FPGA structure leads to a significant saving in convergence time. However, the main drawback concerns the processing element of a neuron which requires the computation of a sum of products, see (3). This computation needs a multiplier operator which remains expensive in hardware structure. For example, in [26], the authors have designed a basic structure including an RAM, a multiplier-accumulator, a subtractor, a multiplier, an accumulator and finally a Look Up Table (LUT) for the threshold function included in neural computation. For an ANN with only three neurons, the proposed structure is costly (approximatively 700 LUTs, 300 slices, 350 flip-flops). To reduce this cost, some authors propose to modify the values of inputs and weights by replacing them by power of two [27]. In this case, the multiplier operators are not necessary and can be replaced by simple shifters which are less expensive. But this modification is not feasible for all ANNs.

All the proposed implementations correspond to the resolution of one specific problem with the definition of a fixed ANN structure. However, the spatio-temporal scheduling problem on a reconfigurable resource requires a dynamic implementation able to schedule a variable number of tasks on the resource. This problem needs a particular ANN structure and classical solutions cannot be used. In the next section, we present our Reconfigurable ANN structure which supports the schedule of a variable number of tasks and the 1D task placement within the reconfigurable resource.

4 Scheduling for Reconfigurable Hardware using ANN

Spatio-temporal scheduling of heterogeneous tasks on a multiprocessor platform supporting dynamic adaptation leads us to the definition of the RANN model. This new ANN structure supports the management of an unfixed number of tasks. All tasks are considered as non-preemptive to limit the number of reconfigurations. Indeed, an important contribution of the energy consumption is due to these reconfigurations. Another key point is that none of the commercial FPGAs supports efficient preemption which includes bitstream readbacks.

4.1 Management of an Unfixed Number of Tasks Within the Reconfigurable Unit

Spatio-temporal scheduling of an unfixed number of independent tasks is the first challenge. The optimization criterion concerns the area usage rate of the reconfigurable area, defined as the ratio between the resources used and the total resources on the reconfigurable unit (RU). Let us consider a dynamically reconfigurable unit with a total area equal to TA [au] (with [au] the area unit) and a set of tasks $\mathcal{T} = \{T_i\}$ with T_i defined as

$$T_i = \{A_i, E_i\}, \quad \forall i = 1, \dots, N_t, \quad (6)$$

with A_i the area usage rate of the task T_i on the RU in [au], E_i the execution time of the task T_i in time unit [tu], and N_t the total number of tasks in \mathcal{T} . Note that the reconfiguration time of each task is included in its global execution time E_i . The problem is then to find all possible instantiated task combinations that ensure a maximum configuration, i.e. the maximum use of the reconfigurable area.

Definition 1 A Maximum Configuration (MC_i) is a configuration that cannot accept a supplementary task, due to the area usage of the reconfigurable unit.

So, the set of all the Maximum Configurations, MC , can be defined as the set of task subsets $\{T_i\}$

$$MC = \bigcup_i MC_i, \quad (7)$$

with MC_i defined by

$$MC_i = \left\{ T_j \middle| \left[\sum_{j|T_j \in \omega_i} A_j \leq TA \right] \wedge \left[\nexists T_k \mid k \notin \omega_i \wedge \sum_{j|T_j \in \omega_i} A_j + A_k \leq TA \right] \right\}$$

with ω_i a subset of \mathcal{T} , i.e. $\omega_i \subset \{T_1, T_2, \dots, T_{N_t}\}$.

Let us consider a simple example with a dynamically reconfigurable hardware having a total area of 50 [au] and an application composed of four tasks. The area and the execution time of each task on the hardware resources are as follows:

- task T_1 : $A_1 = 10$ [au]; $E_1 = 30$ [tu];
- task T_2 : $A_2 = 20$ [au]; $E_2 = 20$ [tu];
- task T_3 : $A_3 = 10$ [au]; $E_3 = 30$ [tu];
- task T_4 : $A_4 = 40$ [au]; $E_4 = 20$ [tu].

Table 1 shows all task combinations in the case that all tasks are independent. In this example, the configurations $Conf_7$, $Conf_9$, and $Conf_{12}$ are maximum and defined as

$$MC = \{\{T_1, T_2, T_3\}; \{T_1, T_4\}; \{T_3, T_4\}\}. \quad (8)$$

Some other configurations are incomplete ($Conf_0$ to $Conf_6$ and $Conf_8$), since it is possible to add a task in all these configurations. For example, in configuration $Conf_1$, it is possible to instantiate task T_2 and T_3 or T_4 in the remaining area. All remaining solutions are impossible configurations ($Conf_{10}$, $Conf_{11}$ and $Conf_{13}$ to $Conf_{15}$), since they require more area than available.

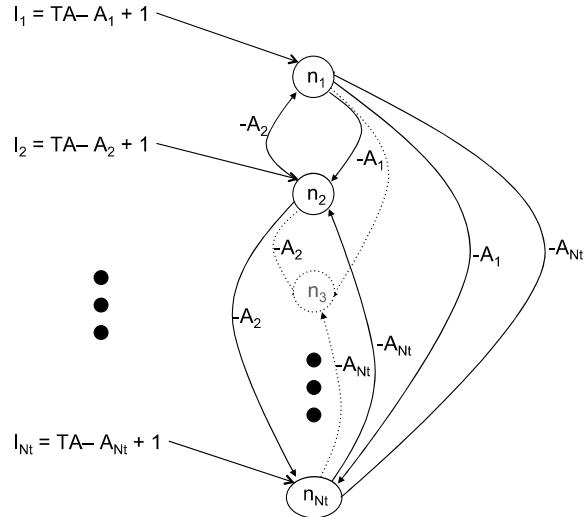
For the three maximum configurations, we can notice that the numbers of tasks are variable (the cardinalities of these configurations are equal to 2 or 3).

To manage this unfixed number of tasks, we propose a specific ANN structure shown in Fig. 4, where each neuron represents a task to schedule. This structure is defined for one scheduling cycle and is adapted for each new cycle. In this case, an

Table 1 List of all combinations of four tasks. MC is a maximum configuration (gray rows), NFC is a non-feasible configuration since area requirement is greater than the total available area (light gray rows), and finally IC is an incomplete configuration since at least one task can be added (see (4.1))

Configurations $Conf_i$	Tasks				Area req.	Conf type	Nb tasks
	T_1	T_2	T_3	T_4			
$Conf_0$					0	IC	0
$Conf_1$	X				10	IC	1
$Conf_2$		X			20	IC	1
$Conf_3$	X	X			30	IC	2
$Conf_4$			X		10	IC	1
$Conf_5$	X		X		20	IC	2
$Conf_6$		X	X		30	IC	2
$Conf_7$	X	X	X		40	MC	3
$Conf_8$				X	40	IC	1
$Conf_9$	X			X	50	MC	2
$Conf_{10}$		X		X	60	NFC	2
$Conf_{11}$	X	X		X	70	NFC	3
$Conf_{12}$			X	X	50	MC	2
$Conf_{13}$	X		X	X	60	NFC	3
$Conf_{14}$		X	X	X	70	NFC	3
$Conf_{15}$	X	X	X	X	80	NFC	4

Fig. 4 RANN principle ensures the schedule of an unfixed number of tasks at each cycle. This figure shows the input and weight values of the RANN to manage the available reconfigurable area for a given schedule tick. The RANN is defined for a generic schedule tick, and adapted for each new tick. Each neuron n_i receives a negative energy to each others neurons n_j (with $j \neq i$) through a valued edge which represents the necessary area of the task T_j . The neuron n_i represents the task T_i



active neuron n_i in the cycle s corresponds to a running task T_i at this cycle. The main idea of the RANN structure is to enable the neuron activation (task activation) if the available area is sufficient. Thus, the neuron input I_i of task T_i is defined as:

$$I_i = TA - A_i + 1 \quad (9)$$

which corresponds to the remaining area when the task T_i is instantiated on the reconfigurable unit. It also corresponds to the maximum area that can be used by other tasks to guarantee T_i instantiation. Furthermore, each neuron receives the area consumed in the reconfigurable unit by all other scheduled tasks, this is modeled by the connection weight with a value equal to $-A_i$. These connections are not all represented in Fig. 4. The complete connection matrix \mathcal{W} is defined by the elements $w_{i,j}$ which are equal to:

$$w_{i,j} = -A_i \quad (10)$$

defining that a connection exists from task i to task j with a weight equal to $-A_i$. The above definitions of I_i and $w_{i,j}$ (9) and (10)) are then used in (2) to evaluate the neurons' activity.

4.2 Management of Task Dependencies

The above proposition is insufficient to model a real task graph, since no dependency between tasks is considered. To support task dependencies, we propose to complete the previous structure by adding a controller and a logical function at each neural input. The goal of the controller is to manage the neuron inputs to make neuron activation dependent on the others.

The task dependencies are modeled as follows:

- Let \mathcal{D} be the task dependencies matrix ($N_t \times N_t$), with $d_{i,j}$ a binary variable equal to 1 if task T_i precedes task T_j , and equal to 0 if no dependency exists between these two tasks. Note that $d_{i,i}$ is equal to 0.
- Let \mathcal{F}_s be a binary vector of size N_t , which depends on the schedule cycle s and is composed of $f_{s,i}$ binary elements equal to 1 if task T_i has finished its execution before schedule time s , or equal 0 otherwise. For example, if task T_1 (with $E_1 = 30$ [tu]) starts its execution at schedule time 0, then the binary values of $f_{s,1}$ are: $f_{s,1} = 0 \forall s < 30$, and $f_{s,1} = 1 \forall s \geq 30$.
- Let \mathcal{X}_s be the task execution vector of size N_t , with $x_{s,i}$ an integer variable which represents the number of schedule cycles obtained by the task T_i until the schedule cycle s . At each cycle s , the values $x_{s,i}$ are incremented for all active neurons n_i at this cycle. This is formulated as

$$x_{s,i} = x_{s-1,i} + 1, \quad \forall i | n_i \text{ is an active neuron at cycle } s. \quad (11)$$

Then at cycle s , the binary variables $f_{s,i}$ are evaluated as

$$f_{s,i} = \begin{cases} 1 & \text{if } x_{s,i} \geq E_i, \\ 0 & \text{otherwise,} \end{cases} \quad \forall i = 1, \dots, N_t. \quad (12)$$

These evaluations (performed by the controller) check if the task represented by the neuron n_i has resumed its execution (i.e. if allocated time is greater than execution time).

From these previous variables, a control input $c_{s,i}$ of each neuron can be defined by the *maxterm* operation between the vector \mathcal{F}_s and the logical inverse row i of the matrix \mathcal{D} as

$$\begin{aligned} c_{s,i} &= (f_{s,1} \vee \overline{d_{1,i}}) \wedge (f_{s,2} \vee \overline{d_{2,i}}) \wedge \dots \wedge (f_{s,N_t} \vee \overline{d_{N_t,i}}) \\ &= \bigwedge_{j=1}^{N_t} (f_{s,j} \vee \overline{d_{j,i}}) \quad \forall i = 1, 2, \dots, N_t \end{aligned} \quad (13)$$

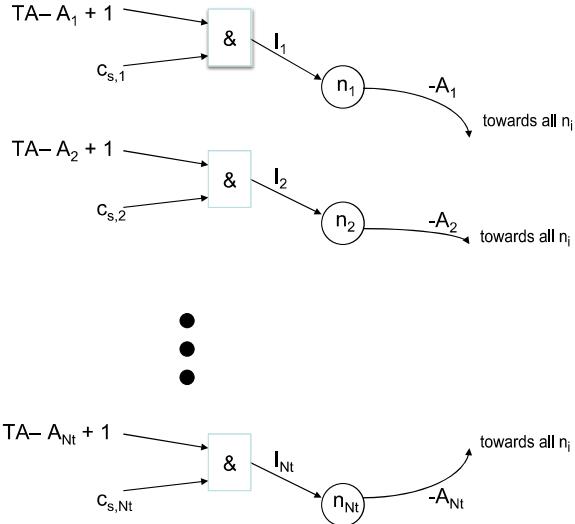
with \wedge the *logical-and* operator, \vee the *logical-or* operator and \bigwedge the *logical-and* of binary values. If a dependency exists between task T_i and T_j (task T_i precedes task T_j), this expression forces the execution of task T_i before the execution of task T_j . Then we can define the control input vector \mathcal{C}_s by

$$\begin{aligned} \mathcal{C}_s^* &= (f_{s,1} \quad f_{s,2} \quad \dots \quad f_{s,N_t}) \odot \begin{pmatrix} \overline{d_{1,1}} & \overline{d_{1,2}} & \dots & \overline{d_{1,N_t}} \\ \overline{d_{2,1}} & \overline{d_{2,2}} & \dots & \overline{d_{2,N_t}} \\ \dots & \dots & \dots & \dots \\ \overline{d_{N_t,1}} & \overline{d_{N_t,2}} & \dots & \overline{d_{N_t,N_t}} \end{pmatrix} \\ &= \mathcal{F}_s^* \odot \overline{\mathcal{D}} \end{aligned} \quad (14)$$

with \mathcal{F}_s^* the transpose vector of \mathcal{F}_s , $\overline{\mathcal{D}}$ the complementary matrix of \mathcal{D} and \odot the logical matrix *maxterm* operator.

The dependency control, defined by the $c_{s,i}$ variables, is implemented through a *logical-and* function placed on the neuron input, as shown in Fig. 5.

Fig. 5 Adding *logical-and* function in the RANN structure ensures the management of task dependencies. The input of each neuron n_i is combined with the control input variable $c_{s,i}$ to manage the task dependencies



4.3 Example of an RANN Structure

As proof-of-concept, we have implemented the four-task example defined in Sect. 4.1. If we consider two task dependencies from tasks T_1 and T_4 to task T_2 , the dependency matrix is given by

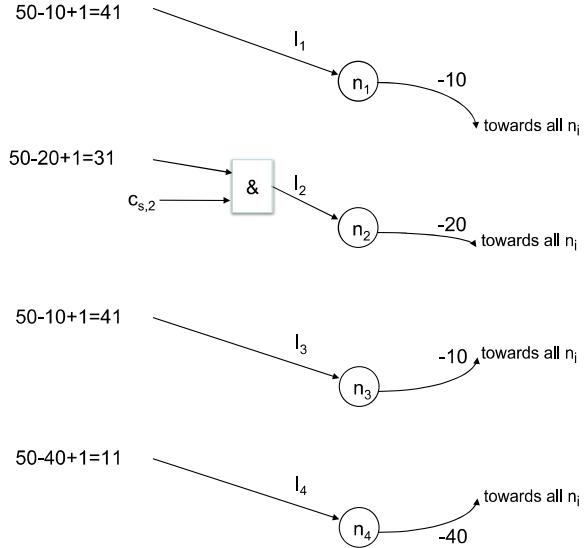
$$\mathcal{D} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{matrix} T_1 \\ T_2 \\ T_3 \\ T_4 \end{matrix}. \quad (15)$$

In this case, the control input vector is

$$\begin{aligned} \mathcal{C}_s &= \begin{pmatrix} c_{s,1} \\ c_{s,2} \\ c_{s,3} \\ c_{s,4} \end{pmatrix} = \begin{pmatrix} (f_{s,1} \vee 1) \wedge (f_{s,2} \vee 1) \wedge (f_{s,3} \vee 1) \wedge (f_{s,4} \vee 1) \\ (f_{s,1} \vee 0) \wedge (f_{s,2} \vee 1) \wedge (f_{s,3} \vee 1) \wedge (f_{s,4} \vee 0) \\ (f_{s,1} \vee 1) \wedge (f_{s,2} \vee 1) \wedge (f_{s,3} \vee 1) \wedge (f_{s,4} \vee 1) \\ (f_{s,1} \vee 1) \wedge (f_{s,2} \vee 1) \wedge (f_{s,3} \vee 1) \wedge (f_{s,4} \vee 1) \end{pmatrix} \\ &= \begin{pmatrix} 1 \\ f_{s,1} \wedge f_{s,4} \\ 1 \\ 1 \end{pmatrix}. \end{aligned} \quad (16)$$

Figure 6 presents the RANN structure for this example. Note that, when $c_{s,i} = 1$, the input function of task T_i is not necessary and the neuron input is simply equal to $TA - A_i + 1$.

Fig. 6 Management of the task dependencies in the RANN structure (dependencies from T_1 and T_4 to T_2). The neuron inputs are combined with the variable $c_{s,i}$ to control the neuron activation



5 Discussion

The RANN architecture is designed to schedule a variable number of tasks on a dynamically reconfigurable architecture and to ensure the management of task dependencies. While the tasks are defined by their area, the schedule is done under area constraint. If we replace the task area by the width of tasks onto the reconfigurable resource, the schedule ensures a 1D placement of tasks. In this case, each task is placed in column within the reconfigurable resource, and the convergence of the RANN always ensures that the task placement is possible.

At design time, the RANN is defined for a maximum number of simultaneous tasks (N_{\max} with $N_{\max} \geq N_t$) which can be instantiated in the reconfigurable hardware (this number is computed off-line). In our model, a neuron represents a task to be scheduled and because it is not realistic to modify the schedule at each cycle, we define a new time cycle, the Reconfigurable Schedule Tick.

Definition 2 *The Reconfigurable Schedule Tick (RST) corresponds to a time interval without any modification into the reconfigurable resource. It is defined as the greatest common divisor of all the task execution times.*

This RST is defined statically if all the tasks are known at compile time. Otherwise, if all the tasks are not known at compile time, the RST can be adapted on-line. This adaptation is managed by the operating system and has no impact on the neural network structure. However, this adaptation can modify the time allowed to the neural network for the convergence and the operating system tick must be larger than the neural network convergence. For this work, we consider that the tasks are all known at compile time.

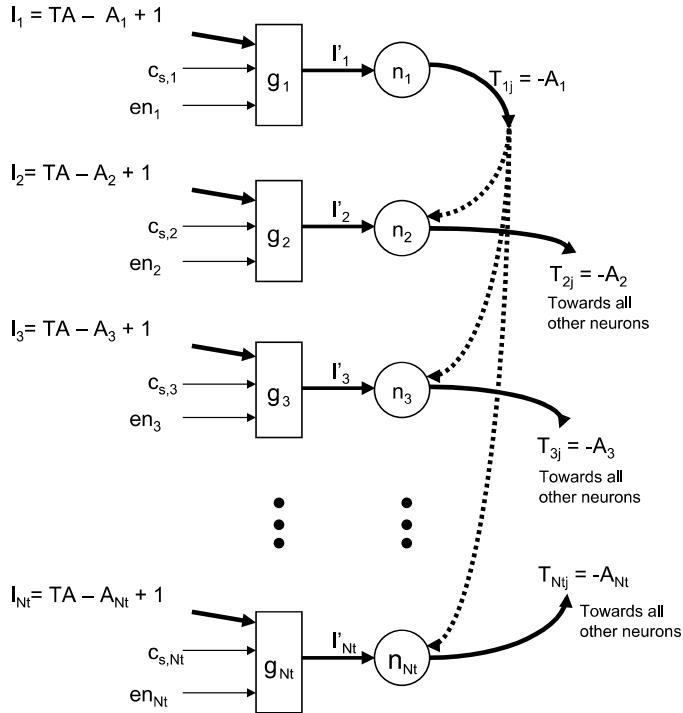


Fig. 7 RANN model of the scheduling problem for Dynamically Reconfigurable Architecture, with resource constraints (area constraint or width constraint)

For the proof-of-concept case (Sect. 4.1), the RST is equal to 10 [tu] and the RANN must converge between two consecutive RTSs.

Furthermore, to optimize the management of the number of scheduled tasks during a specific RST, the RANN allows to include or exclude neurons at each RST. Exclusion of a neuron (i.e. the corresponding task doesn't have to be scheduled during the RST) is done by forcing its input to zero. In this case, the neuron state automatically converges to the inactive state.

Figure 7 shows the complete architecture of an RANN. The number of neurons to evaluate in the RANN is managed through the \$en_i\$ input of each block \$g_i\$ in front of each neuron \$n_i\$. If this input is forced to zero, the corresponding neuron cannot become active, i.e. the neuron is then excluded from the RANN convergence.

As illustrated in Fig. 7, to ensure the convergence of the global structure, we have defined the inputs and connection weights as:

$$\begin{aligned}
 W_{ij} &= -A_i \cdot \overline{\delta_{i,j}} && \forall i, \forall j, \\
 I_i &= TA - A_i + 1 && \forall i, \\
 I'_i &= g_i(I_i, en_i, c_{s,i}) && \forall i, \forall s, \\
 n_i &= h_i(I'_i, W_{ij}) && \forall i, \forall j,
 \end{aligned} \tag{17}$$

with en_i the enable control input, and $c_{s,i}$ the control input to manage the dependencies of task T_i . As mentioned before, W_{ij} is the value of the connection from task T_i to task T_j as defined in the Hopfield model, TA is the total area available in the reconfigurable hardware, A_i is the area of task T_i and I_i the remaining available area if task T_i is scheduled. g_i is the function which computes I'_i according to the task dependencies and the possible scheduling configurations. This function also provides a zero value for I'_i when the task T_i has completed its execution. Finally, the function h_i computes the neuron state n_i following the Hopfield model (as (1)).

6 Convergence Case Study

The RANN was simulated using the proof-of-concept application. For this example, we assume that two task dependencies exist: one between task T_1 and task T_2 ($T_1 \rightarrow T_2$), and another between task T_4 and task T_2 ($T_4 \rightarrow T_2$), as defined in the previous section. In this case, the RANN is defined with four neuron, the connection and input values are defined following (17). One possible RANN evolution is illustrated in Fig. 8. The evolution of the RANN is presented in the following five RST steps.

- RST 1: Due to dependency constraints, task T_2 cannot be scheduled. So, the input of neuron n_2 is forced to zero (due to $c_{s,1}$ bit equal to zero), and n_2 cannot be switched to an active state. Then, we suppose that neuron n_1 corresponding to task T_1 is evaluated. For this neuron, the function u_1 is sufficient to switch to the active state ($u_1 = I_1 + \sum_{j=1}^{N_t} x_j \cdot W_{1,j} = I_1 = 41$). Then, the neuron n_4 is fired, and since the function u_4 is equal to $u_4 = I_4 - A_1 = 11 - 10 = 1$, it is switched to the active state. As this is a Maximum Configuration, no other neuron can be activated.
- RST 2: The evaluation of the new schedule tick is done between RST 1 and RST 2. The previous neuron states are conserved before starting this new step. So, since tasks T_1 and T_4 are not finished, no input and nor connection weight modifications are applied on the RANN, and the previous neuron states are conserved. The RANN stays in its previous state, even if n_2 and n_3 are evaluated. For n_2 , the dependencies from T_1 and T_4 prevent its activation. For n_3 , the function u_3 is equal to $u_3 = I_3 + \sum_{j=1}^{N_t} x_j \cdot W_{3,j} = 41 - 10 - 40 = -9$, so n_3 cannot become active.
- RST 3: At this tick, task T_4 has finished its execution. The binary value $f_{s,4}$ is set to 1 indicating that task T_4 is completed, hence preventing the re-schedule of the task. Nevertheless, due to the dependency between tasks T_1 and T_2 , the input neuron of task T_2 remains forced to zero. Then, we suppose that neuron n_3 corresponding to tasks T_3 is evaluated. For this neuron, the function u_3 is equal to $u_3 = 41 - 10 = 31$, so this neuron can be switched into active state.
- RST 4: At this tick, task T_1 has finished its execution. The binary value $f_{s,1}$ is set to 1 indicating that task T_1 is completed, this prevent the re-schedule of the task. In this case, all the dependencies of task T_2 are respected and task T_2

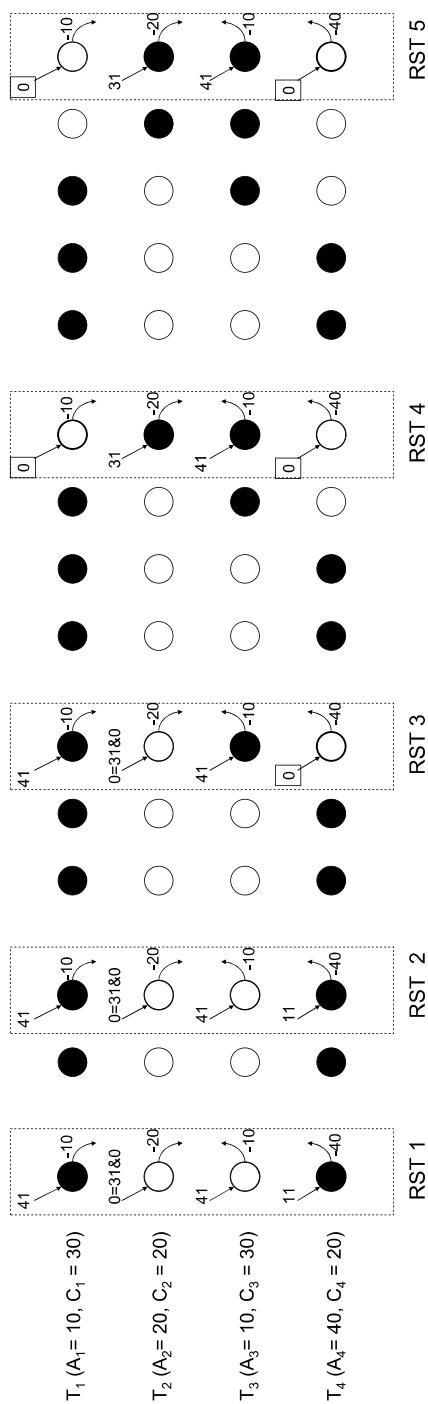


Fig. 8 Example of the RANN evolution for several RST. In this example, four tasks are considered. The inputs of neurons are managed to control the neuron activations

Table 2 Example of neuron evaluations for a scheduling example composed of four tasks with dependencies from tasks T_1 and T_4 to task T_2

Evaluated neurons	Reconfiguration schedule tick (RST)				
	1	2	3	4	5
1st neuron fired	n_1	–	n_2	n_2	–
Neuron evolution	$0 \rightarrow 1$		$0 \rightarrow 0$	$0 \rightarrow 1$	
2nd neuron fired	n_2	–	n_1	–	–
Neuron evolution	$0 \rightarrow 0$		$1 \rightarrow 1$		
3rd neuron fired	n_4	–	n_4	–	–
Neuron evolution	$0 \rightarrow 1$		$1 \rightarrow 0$		
4th neuron fired	–	–	n_3	–	–
Neuron evolution				$0 \rightarrow 1$	
Nb of neuron evaluations	3	0	4	1	0

can now be scheduled. The value $c_{s,2}$ is now equal to 1, so the input of neuron n_2 is then fixed at value $I_2 = TA - A_2 + 1$. So, if we suppose that n_2 is evaluated, its function u_2 is equal to $u_2 = 31 - 10 = 21$, so this neuron can be switched into active state.

RST 5: For the same reason than RST 2, the RANN stays stable at this tick, i.e. the previous neuron states are conserved. At the end of this tick, all the tasks have been scheduled and executed.

Table 2 shows the number of neuron evaluations at each RST for an example of RANN convergence. For each RST, the list of evaluated neurons and the state evolution of these neurons are given. The “–” character indicates that the RANN has converged and stays stable for the current simulation steps. For example, at RST 1, the first evaluated neuron is n_1 , which is switched to the active state. Next, n_2 is evaluated and maintained in the inactive state due to its dependencies from tasks T_1 and T_4 . Next, n_4 is evaluated and switched to an active state. Finally, no other neurons can change and the RANN is stable. The last line of the table shows the number of evaluated neurons at each RST. For this simple example, the total number of neuron evaluations for the complete scheduling is equal to 8. An equivalent scheduling problem, with the same number of tasks the same number of RST and modeled by classical ANNs (as described in Sect. 3.2) would need more than 100 neuron evaluations. In [19] a very similar example is presented: the scheduling problem with four tasks and height cycles requires about 200 neuron evaluations to converge. These results show that our proposal needs approximatively 10 times less evaluations to converge for a simple example. Furthermore, the number of neurons to model the problem with RANN is equal to N_t (in our example $N_t = 4$). Knowing that for the classical ANN model the number of neurons is greater than $(N_t + 1) \times N_c = 40$ (with N_t the number of tasks and N_c the number of cycles), the reduction factor is equal to $N_c = 8$.

Figure 9 shows the convergence time of the RANN by plotting the evolution of neurons evaluated to ensure convergence of the RANN, according to the number of tasks to schedule in the reconfigurable resource. If there is no dependency between tasks and if there is no area constraint, the number of neurons to evaluate is given by the line called *minimum constraint*. This is the worst case for the RANN since all neurons need to be evaluated. Generally, some dependencies exist between tasks, and the reconfigurable resource cannot handle all task executions simultaneously. In this case, several constraints can be exploited to limit the number of neurons that need to be evaluated. In the extreme case, where only one task can be instantiated at a time (due to area incompatibility or chain of dependencies), only one neuron needs to be evaluated (line *maximum constraint*). More realistic cases have intermediate number of constraints, with some dependencies between tasks and some incompatible area placements for tasks.

To evaluate the RANN convergence complexity, we can define two metrics as seen above: the *dependency constraint* which represents the temporal parallelism of the scheduling, and the *area constraint* which represents the spatial parallelism of the scheduling.

Definition 3 *The dependency constraint represents the link between the tasks of the application. If there are very few dependencies, the application is highly parallel.*

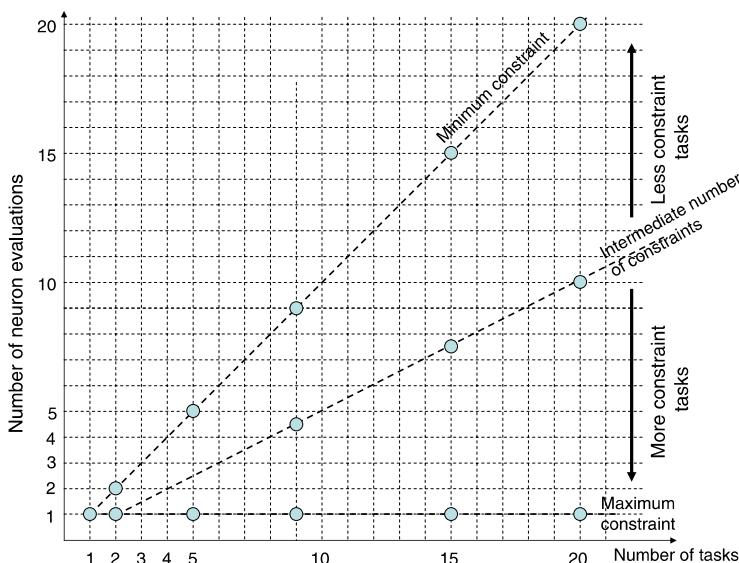


Fig. 9 Evolution of number of evaluated neurons according to the number of tasks to schedule. The line *Minimum constraint* corresponds to a set of independent tasks. The line *Maximum constraint* corresponds to a graph with a serial execution of tasks, i.e. tasks are closely dependent

lel. Therefore, we can evaluate the dependency constraint as the ratio between the number of task dependencies and the total number of tasks

$$\text{DependencyConstraint} = \left(\sum_{i=1}^{N_t} \sum_{j=1}^{N_t} d_{i,j} \right) / N_t. \quad (18)$$

Definition 4 The area constraint can be evaluated by the ratio between the minimal number of tasks which can be instantiated and the total number of application tasks

$$\text{AreaConstraint} = \frac{\min\{\text{Card}(MC_i)\}_{\forall MC_i \in MC}}{N_t}. \quad (19)$$

These two constraints need to be evaluated and handled separately to estimate the overall complexity of applications. In the simple example presented earlier, the system is more constrained by area resources than by task dependencies.

As shown before, the RANN architecture can schedule a dynamic number of tasks according to the environment status. In order to be effective, this spatio-temporal scheduler needs to be implemented in the RSoC. The next section presents a hardware implementation of the RANN structure.

7 Implementation Results of the RANN

As previously mentioned, the activation function of the state x_i of neuron n_i is computed by (2) and (3) that need a Multiplication-Accumulation operation (MAC) and a comparison with a threshold. Based on these expressions, Fig. 10 shows one possible internal structure of neurons. In this approach, the output value x_i of each active neuron n_i is the area used by the task placed within the reconfigurable resource.

The first drawback of this basic architecture is the data bit width of neuron outputs which must be able to encode the area usage rate of tasks. Due to this data size, the adder must be costly for a hardware implementation.

To limit the implementation complexity, the calculation method is modified. Rather than computing the neuron state by MAC operation, we define the logical function that produces the activation of each neuron. For each task, the necessary conditions to active the corresponding neuron are defined. For example, for the task T_1 defined in Sect. 4.1, its neuron n_1 can be switched in active state if the other neurons/tasks do not consume more than 40 [au]. This is possible for each following case:

- Tasks T_2 , T_3 and T_4 are not instantiated, so if neurons n_2 , n_3 and n_4 are not active (see $Conf_1$ of Table 1). This can be written as $x_1 = \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$.
- Tasks T_3 and T_4 are not instantiated, so if neurons n_3 and n_4 are not active (see $Conf_3$ of Table 1). This can be written as $x_1 = x_2 \cdot \overline{x_3} \cdot \overline{x_4}$.
- Tasks T_2 and T_4 are not instantiated, so if neurons n_2 and n_4 are not active (see $Conf_5$ of Table 1). This can be written as $x_1 = \overline{x_2} \cdot x_3 \cdot \overline{x_4}$.
- ...

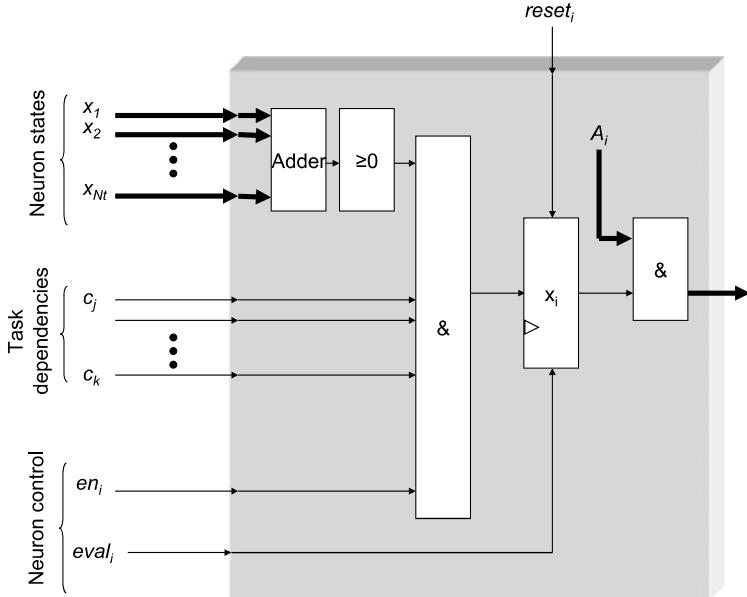


Fig. 10 First proposition for the internal structure of one neuron. The computation of the neuron state is based on the expressions (2) and (3). An adder and a comparator are then needed to evaluate the state of the neuron

Finally, the internal state x_1 of a neuron n_1 is determined by logical functions g'_1 as

$$\begin{aligned} x_1 &= g'_1(x_j)_{j \neq 1} \\ &= \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} + x_2 \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_2} \cdot x_3 \cdot \overline{x_4} + x_2 \cdot x_3 \cdot \overline{x_4} + \overline{x_2} \cdot \overline{x_3} \cdot x_4. \end{aligned} \quad (20)$$

All the neuron activation functions are defined from the scheduling configuration table (Table 1). Then, the complete expressions of these functions for the proof-of-concept application from Table 1 are

$$\begin{aligned} x_1 &= \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4} + x_2 \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_2} \cdot x_3 \cdot \overline{x_4} + x_2 \cdot x_3 \cdot \overline{x_4} + \overline{x_2} \cdot \overline{x_3} \cdot x_4, \\ x_2 &= \overline{x_1} \cdot \overline{x_3} \cdot \overline{x_4} + x_1 \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_1} \cdot x_3 \cdot \overline{x_4} + x_1 \cdot x_3 \cdot \overline{x_4}, \\ x_3 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_4} + x_1 \cdot \overline{x_2} \cdot \overline{x_4} + \overline{x_1} \cdot x_2 \cdot \overline{x_4} + x_1 \cdot x_2 \cdot \overline{x_4} + \overline{x_1} \cdot \overline{x_2} \cdot x_4, \\ x_4 &= \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot x_3. \end{aligned} \quad (21)$$

These simple equations are very easy to obtain, and the FPGA synthesis tools can greatly optimize the logic and thus limit their implementation cost. For example, the previous functions can be simplified as:

$$\begin{aligned} x_1 &= \overline{x_2} \cdot \overline{x_3} + \overline{x_4}, \\ x_2 &= x_1 \cdot \overline{x_4} + \overline{x_1} \cdot \overline{x_3} \cdot \overline{x_4} + \overline{x_1} \cdot x_3 \cdot x_4, \\ x_3 &= \overline{x_1} \cdot \overline{x_2} + \overline{x_4}, \\ x_4 &= \overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_2} \cdot \overline{x_3}. \end{aligned} \quad (22)$$

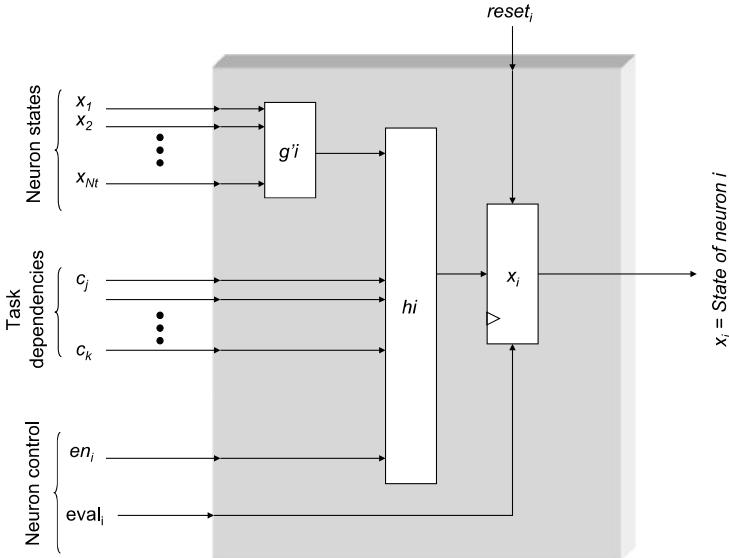


Fig. 11 Optimized internal structure of a neuron. In this structure, the MAC and the comparator have been replaced by the function g'_i which manages the configurations as defined in Table 1

In this case, the hardware implementation of the internal structure of a neuron can be simplified as described in Fig. 11. As the internal states of neurons are determined by logic functions, the neuron requires only LUTs to be implemented, i.e. no adder operators are required, so the basic neuron structure can be easily implemented in FPGAs. Furthermore, the neuron inputs and outputs can be now coded with only one bit, which drastically reduced the interconnection between neurons. As the logic complexity is not driven by the number of gates but by the number of inputs of the function, FPGAs are a good target for the RANN implementation.

For a large number of independent tasks, the determination of the possible configurations can become complex. However, several issues must still be discussed. Firstly, we assume that a reconfigurable resource cannot execute (at a given time) more than 20 simultaneous tasks as discussed in [28], but can execute several hundred tasks during the complete application execution. Despite our model can support more tasks, this limitation to 20 simultaneous tasks is realistic for applications implemented in nowadays SoCs. This allows to reduce the number of necessary neurons to solve the spatio-temporal scheduling problem. Secondly, targeted applications typically present dependencies between tasks which drastically reduce the logic expressions of activation functions and the set of feasible configurations. Therefore, the combinatorial problem can be limited by the intrinsic properties of the considered applications. Due to these two observations, the RANN complexity can be significantly limited.

For our experiments, we randomly generated several application task graphs containing 10 tasks. For these experiments, we also consider that tasks are dependent, with a *DependencyConstraint* equal to 0.5 (which corresponds to a classical

Table 3 Hardware implementation results on Virtex 5 XC5VLX30 FPGA for 10 simultaneous running tasks. The *DependencyConstraint* and *AreaConstraint* are fixed at value 0.5

Example number	Average area of tasks	Number of LUT	Estimated frequency [MHz]
1	5	10	944
2	10	10	944
3	10	10	944
4	15	14	461
5	60	20	492
6	100	20	492
7	30	31	376
8	45	32	413
9	38	35	456
10	34	39	356
11	50	40	416
12	50	40	354
13	17	46	350
14	25	51	293
15	19	55	345
16	22	65	314
17	30	67	339
18	20	69	315
19	15	77	324
20	21	85	322
Average		41	461

task dependency graph). The structure of the RANN was synthesized for Xilinx XC5VLX30 Virtex FPGA circuit. This circuit is a matrix of 2400 Configurable Logic Blocks (CLB) (one CLB comprises two slices, with each containing four 6-input LUTs and four Flip-Flops) and 32 RAM blocks (36 kbits each).

Table 3 shows a set of results for several RANN hardware implementations composed of 10 simultaneous tasks to schedule. The second column represents the average area occupied by implemented tasks out of the total area of the reconfigurable architecture. The average area of tasks represents the relative size of each task related to the overall resource area. Higher this number is, the more the task needs logical resources. The last two columns give the implementation results in terms of LUT and maximal frequency. In this table, we can see that the estimated frequency is linked with the number of LUTs necessary to implement the RANN. This number of LUTs directly depends on the task placement constraint, which depends on the number of logical function inputs. If the placement has few constraints (all the tasks can be mapped on the reconfigurable resource at a time or just one task can be mapped at a time) then the logical functions to compute the internal state of each neuron n_i (see (20)) are simple. Conversely, if the placement has lot of con-

straints (a large number of configurations exists for the task placement) then the logical functions to compute the internal state of each neuron are complex and need large number of LUTs. In this case, the logical function complexity limits the clock frequency of the system.

The average frequency is estimated at 461 MHz. The average number of LUT of the 10 neuron RANN structure remains low, and represents a very few percentage of the Virtex 5 FPGA, only 0.28% of the XC5VLX30 and 0.03% of the XC5VLX330.

8 Execution Performance Comparisons

Table 4 summarizes the number of cycles required to obtain the convergences of the classical ANN and the RANN. Results are given for independent tasks and for a number of tasks varying from 10 to 100, with area and execution time of the tasks randomly defined. The classical model is expressed in terms of neuron evaluations and results greatly depend on the implementation. For this model, the results are estimated with a fixed number of running tasks at each cycle. In this case, the classical model can also produce some solutions, but it requires a large number of simulation cycles.

For the RANN implementation, the reduction of the number of required neurons (the number of neurons is equal to the number of tasks) leads to an important reduction of convergence cycles. Furthermore, if we consider the dependencies between tasks and the total area constraint (which limit the simultaneous task instantiations), the number of cycles to converge remains small, approximately equal to the number of tasks.

These results show that our hardware RANN proposal is a good candidate for a hardware scheduler in the context of reconfigurable resources. Indeed, for large applications, the limited number of neurons to implement the scheduler ensures a large area for application tasks in the FPGA.

9 Conclusion

In this paper, a resource constraint scheduling service for reconfigurable hardware based on an ANN is presented. In this work, the variability of the number of scheduled tasks within a reconfigurable resource is taken into account. This variability is the result of the different area usage rates of tasks. Because the classical ANN approaches cannot schedule an unfixed number of tasks, these solutions are not satisfying for reconfigurable hardware which is limited by area usage rate. To solve this problem, we have proposed a Reconfigurable Artificial Neural Network (RANN) structure that ensures the management of tasks according to their area constraints. The main advantage of the RANN is its capacity to converge very quickly. This is due to the very limited number of neurons to model the resource constraint scheduling problem. The decomposition of the complete and complex scheduling into a

Table 4 Comparison of convergences between the classical model and the hardware implementation of the RANN. The number of time units N_c is set to 100

Nb. of tasks N_T	Classical ANN model		Our proposals hardware solution (RANN)		
	Nb. of neurons $N_T \times N_c$	Nb. of fired neurons for [μs]	Nb. of neurons N_t	Nb. of cycles for convergence	Execution time t_h [ns]
10	1000	$\simeq 10^3$	10	$\simeq 10$	25
20	2000	$\simeq 2 \cdot 10^3$	20	$\simeq 20$	50
40	4000	$\simeq 4 \cdot 10^3$	40	$\simeq 40$	100
60	6000	$\simeq 6 \cdot 10^3$	60	$\simeq 60$	150
80	8000	$\simeq 8 \cdot 10^3$	80	$\simeq 80$	200
100	10000	$\simeq 10 \cdot 10^3$	100	$\simeq 100$	250

sequence of small and simple scheduling steps also allows to limit the number of neurons. This decomposition is done through the definition of the Reconfigurable Schedule Tick (RST) which defines the tick interval between two reconfiguration steps. To precisely manage the reconfigurable resource, we propose an RST-by-RST adaptation of the RANN. We show that it is possible to manage a set of tasks at each tick by simple computations of neuron input values. We also show that our proposition supports task dependencies and limits the number of task switchings through the non-preemptive model of tasks.

Due to the current density of FPGA circuits and the dynamic reconfiguration feature, we have shown that the RANN structure is an efficient solution for the hardware spatio-temporal scheduling service. The dynamicity of FPGA permits to adapt the scheduler to a large number of applications, and the simplicity of the RANN (in terms of neuron implementation) ensures the management of large application task graphs. The comparison results of our hardware implementation with a classical solution show that the hardware implementation is really faster, with a lower implementation cost.

These contributions are important advances for our current works which consist in defining an efficient hardware implementation of the scheduling service in the context of Reconfigurable SoC. By limiting the number of task reconfigurations, the RANN limits the time overhead and the energy consumption of the overall application. These two parameters are important and require to be controlled in the context of RSoC.

Our future works concern the partial dependencies between tasks and a more precise management of the task reconfiguration. A partial dependency appears when a task can start its execution before a previous task has completely finished its own execution. In this case, the management of the dependencies needs some modifications. The second evolution concerns the decomposition of the task execution into two different steps: one for the configuration of the task and another one for the task execution. Because two executions of the same task can appear, the reconfiguration step can be optional and some optimizations can be developed. The third evolution concerns the extension of our technique to the 2D task placement. Indeed, current FPGA technologies enable task placement by reconfiguration of rectangular regions rather than full column configuration (e.g. Xilinx circuits offer this feature). We are currently working on this topic and we have studied several neural network solutions to support 2D placement. The main problem of these solutions is the definition of the energy function to ensure the neural network convergence. Another problem concerns the heterogeneity of the reconfigurable circuits and the placement position of each task within the reconfigurable region. While large number of papers have been published without considering this problem, we plan to manage these constraints by defining a very different neural network structure.

References

1. Mooney VJ, Blough DM (2002) A hardware-software real-time operating system framework for SoCs. *IEEE Des Test Comput*, Nov-Dec, pp 44–51

2. Baskaran K, Srikanthan T (2004) A hardware operating system based approach for run-time reconfigurable platform of embedded devices. In: Proc of the 6th real-time Linux workshop, 3–5 Nov
3. Richter K, Racu R, Ernst R (2003) Scheduling analysis integration for heterogeneous multi-processor SoC. In: Proceedings of the 24th international real-time systems symposium (RTSS '03), Cancun, Dec, pp 236–245
4. Nollet V, Coene P, Verkest D, Vernalde S, Lauwereins R (2003) Designing an operating system for a heterogeneous reconfigurable soc. In: IPDPS '03: proceedings of the 17th international symposium on parallel and distributed processing. IEEE Comput Soc, Washington, p 174.1
5. Levinson L, Manner R, Sessler M, Simmler H (2000) Preemptive multitasking on FPGAs. In: Proc IEEE symposium on field-programmable custom computing machines, Napa, California, Apr 17–19, pp 301–302
6. Lee Y-H, Chen C (2003) A modified genetic algorithm for task scheduling in multiprocessor systems. In: Proc of the ninth workshop on compiler techniques for high-performance computing (CTHPC '2003), Taipei, Taiwan, ROC, Mar
7. Catoni O (1998) Solving scheduling problems by simulated annealing. SIAM J Control Optim 36(5):1539–1575. [Online]. Available: <http://citeseer.ist.psu.edu/catoni96solving.html>
8. Abramson D, Smith K, Logothetis P, Duke D (1998) FPGA based implementation of a hop-field neural network for solving constraint satisfaction problems. In: Proc EUROMICRO conference, vol 2, pp 688–693
9. Izeboudjen N, Farah A, Titri S, Boumeridja H (1999) Digital implementation of artificial neural networks: from VHDL description to FPGA implementation. In: Engineering applications of bio-inspired artificial neural networks: artificial neural nets simulation and implementation. Lect Notes Comput Sci, vol 1607. Springer, Berlin, pp 139–148
10. Hamidzadeh B, Lilja D, Atif Y (1995) Dynamic scheduling techniques for heterogeneous computing systems. J Concurr Pract Exp 7:633–652, Oct
11. Noguera J, Badia RM (2004) Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. ACM Trans Embed Comput Syst 3(2):385–406, May
12. Cottet F, Delacroix J, Kaiser C, Mammeri Z (2002) Scheduling in real-time systems. Wiley, England
13. Anderson J, Srinivasan A (2000) PFair scheduling: beyond periodic task systems. In: Proc of the 7th international conference on real-time computing systems and applications, Cheju Island, South Korea, Dec, pp 297–306. [Online]. Available: <http://citeseer.ist.psu.edu/anderson00pfair.html>
14. Liu D, Lee Y-H (2004) PFair scheduling of periodic tasks with allocation constraints on multiple processors. In: Proc of the 18th international parallel and distributed processing symposium, vol 3, Los Alamitos, CA, USA, pp 119–126
15. Bazargan K, Kastner R, Sarrafzadeh M (2000) Fast template placement for reconfigurable computing systems. IEEE Des Test Comput 17(1):68–83, Jan–Mar. Special issue on reconfigurable computing
16. Steiger C, Walder H, Platzner M (2004) Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. IEEE Trans Comput 53(11):1393–1407, Nov
17. Ahmadiania A, Bobda C, Koch D, Majer M, Teich J (2004) Task scheduling for heterogeneous reconfigurable computers. In: SBCCI '04: proceedings of the 17th symposium on integrated circuits and system design, New York, NY, USA, pp 22–27
18. Yoo S-M, Youn HY, Choo H (2001) Dynamic scheduling and allocation in two-dimensional mesh-connected multicomputers for real-time tasks. IEICE Trans Inf Syst 84(5):613–622. [Online]. Available: <http://ci.nii.ac.jp/naid/110003210514/>
19. Cardeira C, Silva M, Mammeri Z (1997) Handling precedence constraints with neural network based real-time scheduling algorithms. In: Proc of the 9th euromicro workshop on real time systems, Toldeo, Spain, Jun, pp 207–214
20. Tagliarini G, Christ JF, Page WE (1991) Optimization using neural networks. IEEE Trans Comput 40(12):1347–1358, Dec
21. Hopfield JJ, Tank DW (1985) Neural computation of decisions in optimization problems. Biol Cybern 52:141–152

22. Cohen M, Grossberg S (1983) Absolute stability of global pattern formation and parallel memory storage by competitive neural networks. *IEEE Trans Syst Man Cybern* 13(5):815–826
23. Grossberg S (1988) Studies of mind and brain: neural principles of learning, perception, development, cognition and motor control. Boston Stud Philos Sci, vol 70. Reidel, Dordrecht
24. Chillet D, Benkermi I, Pillement S, Sentieys O (2007) Hardware task scheduling for heterogeneous SoC architectures. In: Proc of the European signal processing conference, Poznan, Poland, Sep 3–7, pp 1653–1657
25. Zhu J, Sutton P (2003) FPGA implementations of neural networks – a survey of a decade of progress. In: Lecture notes, field-programmable logic and applications, pp 1062–1066
26. Boumeridja H, Atencia M, Joya G, Sandoval F (2005) FPGA implementation of hopfield networks for systems identification. In: Computational intelligence and bioinspired systems, Lect Notes Comput Sci, vol 3512. Springer, Berlin, pp 582–589
27. Marchesi M, Orlandi G, Piazza F, Uncini A (1993) Fast neural networks without multipliers. *IEEE Trans Neural Netw* 4:53–62 Jan
28. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Proc design automation conference, pp 684–689. [Online]. Available: <http://citeseer.ist.psu.edu/dally01route.html>

A New Three-Level Strategy for Off-Line Placement of Hardware Tasks on Partially and Dynamically Reconfigurable Hardware

Ikbel Belaid, Fabrice Muller,
and Maher Benjema

Abstract The partially reconfigurable hardware devices are commonly used in real-time systems; these devices feature a high density of heterogeneous resources to enable multitasking and supply a reasonable flexibility with regard to application requirements. As a result, efficient management of hardware tasks and hardware resources is absolutely essential. Scheduling and placement methods suffer, however, from the issues of resource waste, task rejection and configuration overheads. This paper focuses on a new three-level placement strategy of hardware tasks on these prominent devices and aims at optimized use of the resources to target all the mentioned issues. Two complete methods are proposed in this chapter to efficiently solve the issue of reconfigurable area management. Experiments demonstrate improvement of up to 36% in resource utilization over the available reconfigurable resources, 43% in resource gain as compared to static implementation, and an overall configuration overhead of 11% from the total application running time.

1 Introduction

Scheduling of hardware tasks is highly dependent on placement, which focuses on allocation of hardware resources required by the scheduled hardware tasks. Thus, the scheduler decision should be taken in accordance with the ability of the placer to allocate free resources required by the elected task. The Field-Programmable Gate Array (FPGA) is the mostly widely used reconfigurable hardware device. In this work, we target the most recent heterogeneous Xilinx SRAM-based FPGA. The

I. Belaid (✉)

University of Nice-Sophia Antipolis/LEAT-CNRS, 250 rue Albert Einstein, 06560 Valbonne,
France

e-mail: Ikbel.Belaid@unice.fr

main feature of these devices is the use of the technique of dynamic partial reconfiguration that allows parts of the reconfigurable area to be configured without affecting the rest of the FPGA. Consequently, in spite of high configuration overhead, dynamic partial reconfiguration improves device utilization and application performance. Frequently, the existent methods of placement of hardware tasks on FPGAs face the problems of resource waste, task rejection and configuration overheads. It is for this reason that it is essential to define an efficient method of managing resource area by optimizing the placement quality. In general, the hardware task placement consists of two sub-functions: (1) *partitioning*, which handles the free space in the device and identifies the potential sites enabling execution of hardware tasks, and (2) *fitting*, which selects one from amongst several feasible placement solutions. In the context of the FOSFOR¹ project, we focus on the placement/scheduling problem in dynamically reconfigurable hardware devices, and in this work, basing on the physical features of the target technology, we introduce a new three-level strategy for off-line placement of hardware tasks on these devices by enhancing the quality of placement and by taking advantage of dynamic partial reconfiguration. Experiments conducted on heterogeneous-task-application show an improvement in placement quality marked by 36% of resource utilization from the reconfigurable area. This gain in resource utilization goes as high as 43% as compared to static design, and in the worst case, 11% of configuration overhead from the total application running time. The rest of the chapter is structured as follows: the next section presents the previous work in placement of hardware tasks. Section 3, Sect. 4 and Sect. 5 describe our three-level strategy for off-line hardware task placement. Section 6 covers our modeling of hardware task placement. Section 7 details the first complete exhaustive resolution of the placement problem and Sect. 8 deals with the second complete smart resolution. Section 9 presents the application and the placement quality evaluation. The conclusion and further research are depicted in Sect. 10.

2 Related Work

Several ways of performing the two sub-functions of hardware task placement are proposed. Bazargan, Kastner and Sarrafzadeh define in [1] two types of placement: on-line and off-line. In [1], the off-line placement, of better quality than the on-line placement, uses 3D templates for modeling of the tasks. It employs the greedy search and the simulated annealing approaches. In their 2D on-line placement, the fitting is essentially based on the bin-packing rules [2], and partitioning relies on two methods. The first method, called KAMER (Keeping All Maximal Empty Rectangles), searches all maximal empty rectangles which are not necessarily disjoined.

¹FOSFOR (Flexible Operating System FOr Reconfigurable platform) is a French national program targeting the most highly evolved technologies. Its main objective is to design a real-time operating system distributed on hardware and software execution units, which offers application tasks the required flexibility through run-time reconfiguration and homogeneous Hw/SW OS services.

The second method, named Keeping Non-Overlapping Empty Rectangles, keeps only the non-overlapping holes. Both methods are also evoked after each split or merge operation. Reference [3] deals with the on-line placement taking into account the precedence constraints of tasks. This work enhances the placement success by using a guarantee-based system and analyses the feasibility of non-preemptive scheduling as well as the feasibility of placement. In fact, it extends the Bazargan method of Keeping Non-Overlapping Empty Rectangles and defers split decision until the arrival of the next task, by means of On The Fly partitioning. Other extensions of Bazargan methods are also depicted in [4], which includes Intelligent Merging (IM) algorithm based on the dynamic combination of three new techniques: Merging Only if Needed, Partially Merging and Direct Combine. IM accelerates the Bazargan on-line algorithm by 3 times without losing placement quality. Reference [5] introduces the method of staircase to handle the free space during the first sub-function of the on-line placement. This method is considered efficient as it tries to cover the faults of the KAMER method proposed by Bazargan, in particular for task rejection. Unlike the previous work, [6] presents an approach of on-line placement by managing the occupied space instead of the free space, because of the hardness of managing empty space and the huge growth of the empty rectangles. Reference [6] proposes the Nearest Possible Position algorithm, which optimizes inter-task communication as well as external communication. In [7], handling of free space is performed through Scan Line Algorithm. Some works, as in [8], are realized in order to enhance quality placement and challenge the resource wastage and task rejection. In [8], task placement starts by an initial 2D partitioning of the free space on blocs of different sizes, according to application needs. Then the Immediate Fit algorithm is applied, using the operations of merging, splitting and recovering. By means of a slicing tree, recursive bi-partitioning is used in [9] to find the appropriate rooms in the reconfigurable hardware device for each task, according to the tasks' resources and inter-task communication. Once the rooms' topology is achieved, the sizing step is performed to compute the possible sizes for each room.

Lodi et al. in [10] propose different off-line approaches to resolve hardware task placement as 2D bin-packing problem, for instance, the Floor-Ceiling algorithm and the Knapsack packing algorithm in [11]. Fekete et al. describes in [12] an off-line approach through a graph-theoretical characterization to pack a set of items in a single bin. In [12], a feasible placement within a given container for a fixed scheduling is decided by the orthogonal packing problem. In [13], the approximate metaheuristic: genetic algorithm and the first fit strategy are adopted to solve the on-line placement of hardware tasks. By allowing task rotation, this approach identifies a feasible rearrangement and schedules the moves of executing tasks to attain a feasible placement for the pending task.

Most placement works are not guaranteed, since tasks are handled independently, which does not give the real free space, causes task rejection and fragmentation, and increases placement overhead. In addition, the proposed approaches are applicable in low-complexity applications where tasks are near-identical. The majority of these methods do not allow total flexibility, as the manipulated tasks are non-preemptive and non-replaceable. To the best of our knowledge, the realized works

in the placement domain are performed only on homogeneous devices; they could not therefore be applied in the recent heterogeneous technologies. In our work, we target the most recent heterogeneous Xilinx devices. We took our inspiration from the sub-functions of the generic placement and we adopt the 2D partitioning and exhaustive recursive resolution as well as complete smart resolution to improve the placement defects. Therefore, we propose an off-line strategy made up of three main levels. The first level of our strategy consists of an off-line pre-placement flow enabling task classification based on their resources. The second level searches all the possible physical locations partitioned on the device for the virtual execution units depicting the task classes provided by the first level. The third level takes decisions about the fitting of these virtual execution units in the physical blocs partitioned on the device and ensures task mapping to these fitted units. The second level and the third level represent the sub-functions of generic placement and their resolution is based on mathematical modeling.

3 Level 1: Off-Line Flow of Hardware Task Classification

In this flow, we propose taking into account two parameters: the hardware task resources and the heterogeneity of the device. We separate the hardware tasks from their Reconfigurable Physical Blocs (RPB), partitioned on the device by means of many types of Reconfigurable Zones (RZ). As RZs are tightly packed to the resource types of the hardware tasks, they ensure a trade-off between decreasing configuration overhead and resource efficiency. Besides these advantages, RZs make the platforms more dynamic and more flexible basing on partial run-time reconfiguration.

3.1 Flow Terminology

In this section, we present the keywords used in the off-line flow of hardware task classification which matches essentially the hardware tasks and the features of the device. We define few terms which are used to describe the flow: NT is the number of tasks, NR is the number of Reconfigurable Regions, NZ is the number of Reconfigurable Zones and NP is the number of resource types in the chosen technology. We define three levels of abstraction.

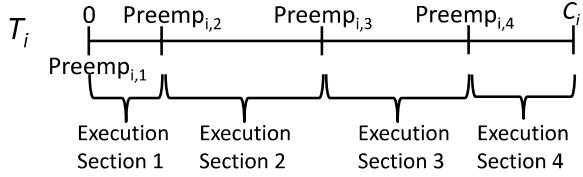
3.2 Application Level

According to resource types (r_k) provided in the target technology, each hardware task T_i is modeled by its physical model as shown in (1):

$$T_i_PHY = \{\alpha_{i,k} r_k\}, \quad 1 \leq k \leq NP, \quad 1 \leq i \leq NT \quad (1)$$

$\alpha_{i,k}$, k , i are Naturals, $\alpha_{i,k}$: Number of resource r_k in T_i

Fig. 1 The predefined preemption points in task T_i



In addition, every hardware task is characterized by its functional model which contains period (P_i), Worst-Case Execution Time (C_i), priority and a set of pre-emption points l ($Preemp_{i,l}$). The number of preemption points of T_i is denoted by $NbrPreemp_i$. The preemption points are instants in time taken throughout the worst-case execution time as shown in Fig. 1. These preemption points are selected by the designer so as to reduce data dependency between the execution sections within the task which are delimited by the preemption points. We resort to this method of predefining the preemption points in order to avoid the heavy classical method of load/readback bitstream, since it complicates the preemption, increases the overhead and requires a large memory space, as a new readback bitstream must be saved at each preemption. In our predefined preemptive modeling, by keeping always the same bitstream, as the finite state machine, which handles a set of registers and controls the task, is known, it is sufficient to provide the required memory and save or load the state of the task at the predefined preemption point when the task is preempted or resumed.

3.3 Physical Level

The device is partitioned into two regions.

Static region (SR): It includes the static part of the design. The static part might hold the static components of the device, such as configuration ports, clock managers or the reconfigurable units for I/O controlling. The static part (2) might also include Hardware Operating System services.

$$SR_PHY = \{\beta_k r_k\} \cup \{\text{static components}\}, \quad 1 \leq k \leq NP \quad (2)$$

β_k, k are Naturals, β_k : Number of resource r_k in SR

Reconfigurable region (RR): It is dedicated to the dynamic part of the design and it can be partially reconfigured at run-time. According to the size of the SR, we define the number and the limits of the RRs. Each RR is a set of heterogeneous resources; hence, it is characterized by its physical model in (3).

$$RR_i_PHY = \{\gamma_{i,k} r_k\}, \quad 1 \leq k \leq NP, \quad 1 \leq i \leq NR \quad (3)$$

$\gamma_{i,k}, k, i$ are Naturals, $\gamma_{i,k}$: Number of resource r_k in RR_i

Each RR includes several RZs: $\{N_j RZ_j\} \subset RR_i$. j depicts the type number of RZs mentioned by RZ_j and N_j is the number of each RZ_j in the corresponding RR_i . The RRs will be exploited later for hardware task placement. Therefore, our aim is to search an efficient method for managing RR resources.

Reconfigurable Zone (RZ): RZ is a virtual unit composed of heterogeneous resources. Each type of RZ (RZ_j) is specialized for a class of hardware tasks. RZ types are determined relying on the resources of hardware tasks in the step 1 of the flow. For each hardware task, we assign at least one type of RZ.

Reconfigurable Bloc (RB): As mentioned by (4), tasks, RRs and RZs are modeled by Reconfigurable Blocs (RB) in order to obtain their RB-models. This modeling is strongly linked with the chosen technology, because it is based on its reconfiguration granularity (the smallest reconfigurable bloc in the technology). Each type of RB (RB_k) is characterized by specified cost $RBCost_k$, which is defined according to its number in the device and the importance of its functionality. Unlike RR, in the RB-model of the task and of the RZ, the locations of RBs are not fixed, whereas they are determined later after searching the possible RPBs for each RZ.

$$\begin{aligned} T_i_RB &= \{X_{i,k} \text{ } RB_k\}, \quad 1 \leq i \leq NT, \quad 1 \leq k \leq NP \\ RR_i_RB &= \{Y_{i,k} \text{ } RB_k\}, \quad 1 \leq i \leq NR, \quad 1 \leq k \leq NP \\ RZ_i_RB &= \{Z_{i,k} \text{ } RB_k\}, \quad 1 \leq i \leq NZ, \quad 1 \leq k \leq NP \end{aligned} \quad (4)$$

where k refers the RB types mentioned by RB_k . $X_{i,k}$, $Y_{i,k}$ and $Z_{i,k}$ define the number of each RB_k respectively in T_i , RR_i and RZ_i . The number of RB types (RB_k) is equal to the number of resource types (NP) in the target technology. We use Xilinx Virtex 5 FPGA as a reference for the reconfigurable hardware device to lead our hardware resource management study. In Virtex 5 [14], the RB_k is a vertical stack including the same type of resources and matching the reconfiguration granularity. The RB_k is, therefore, either 20 CLBM (RB₁) or 20 CLBLs (RB₂) or 4 BRAMs (RB₃) or 8 DSP slices (RB₄).

The RB-model of task or of RR is obtained as follows (see (5) and (6)). Let us consider the set Div including the granularity factors in the target technology $Div = \{DIV_k\}$, $1 \leq k \leq NP$

$$X_{i,k} = \lceil \alpha_{i,k} / DIV_k \rceil, \quad 1 \leq i \leq NT \quad (5)$$

$$Y_{i,k} = \lceil \gamma_{i,k} / DIV_k \rceil, \quad 1 \leq i \leq NR \quad (6)$$

$Z_{i,k}$ in the RB-model of each RZ_i will be deduced from the RB-models of tasks by means of step 1 in the off-line flow of hardware task classification. Figure 2 illustrates an example of RB-model of a given T_i in Virtex 5 technology.

Reconfigurable Physical Bloc (RPB): The RB-model of RZs is fitted on RR within the Reconfigurable Physical Blocs (RPB). These 2D rectangular blocs represent the possible physical locations of RZs in the RRs.

Consequently, as illustrated by Fig. 3, the locations of RBs in the RB-models of RZs are determined after RB-models fitting. The partitioned RPBs for a given RZ might contain some RBs that are not required by RZ. For instance, RB_4 is inserted within RPB_3 but is not used by the corresponding RZ. This resource inefficiency is explained by the rectangular shape of the RPBs. Thus the number of RBs included in the RPB could exceed the required RBs in the RZ. The resource inefficiency is also due to the heterogeneity on the device; the partitioning could book some RBs which are not used by the RZ. The efficiency in RB utilization is an important metric

Fig. 2 Example of RB-model of T_i in Virtex 5

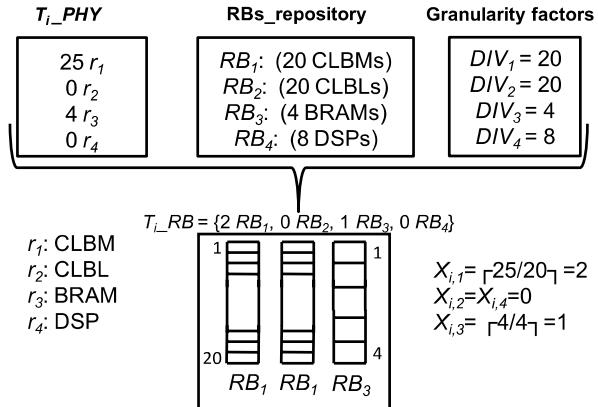
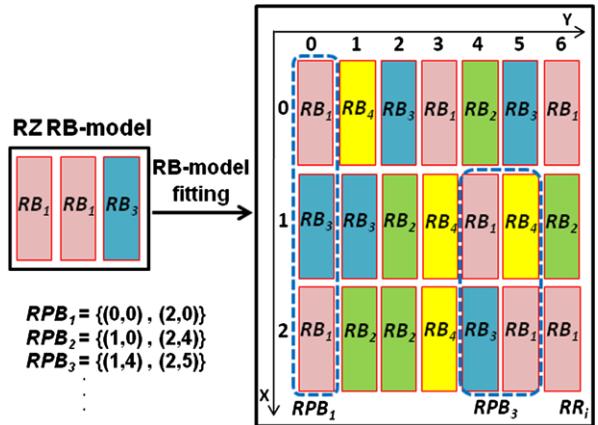


Fig. 3 Example of RPBs for RZ



to evaluate the placement quality. The RPBs are the final realistic representations of RZs on the device.

3.4 Flow Steps

The flow consists of three steps.

3.4.1 Step 1: RZ Types Search or Hardware Task Classes Search

Step 1 aims at defining the possible RZ types based on the RB-models of hardware tasks. Step 1 gathers the tasks sharing the same types of RBs under the same type of RZ and it is achieved by Algorithm 1 with worse-case complexity $O(NT \times NP \times NZ)$.

Algorithm 1: Search of RZ types

```

1: RZ-type = 0 // RZ types
2: List-RZ // list of RZ types
3: n // natural
4: for all tasks Ti do
5:   // Ti_RB = Xi,k RBk
6:   if ((RZ-type ≠ 0) and ( $\exists n, 1 \leq n \leq RZ\text{-type}$ ) /  $\forall k((X_{i,k} \neq 0 \text{ and } Z_{n,k} \neq 0) \text{ or } (X_{i,k} = 0 \text{ and } Z_{n,k} = 0))$ ) then
7:     // this test checks whether the task matches an RZ-type that already exists in list-RZ
8:     for all k do
9:        $Z_{n,k} = \max(X_{i,k}, Z_{n,k})$  // update RB number of RZn
10:    end for
11:   else
12:     Increment RZ-type
13:     RZRZ-type = Create new RZ(Xi,k) // new type of RZ, RZRZ-type = {Xi,k RBk}
14:     Insert(list-RZ, RZRZ-type)
15:   end if
16: end for

```

Fig. 4 Example of RZ types search

$$\begin{aligned}
T_1 &= \{25 \text{ RB}_1, 33 \text{ RB}_2\}, T_2 = \{4 \text{ RB}_3, 27 \text{ RB}_4\}, T_3 = \{18 \text{ RB}_1, 42 \text{ RB}_2\}, \\
T_4 &= \{52 \text{ RB}_2, 12 \text{ RB}_4\}, T_5 = \{46 \text{ RB}_3, 1 \text{ RB}_2, 1 \text{ RB}_3, 1 \text{ RB}_4\}, T_6 = \{36 \text{ RB}_2, 11 \text{ RB}_4\}, \\
T_7 &= \{15 \text{ RB}_2, 7 \text{ RB}_4\}
\end{aligned}$$

4 Classes of Hardware Tasks			
RZ_1 $Z_{1,1} = 25 (T_1)$ $Z_{1,2} = 42 (T_3)$ $Z_{1,3} = 0$ $Z_{1,4} = 0$	RZ_2 $Z_{2,1} = 0$ $Z_{2,2} = 0$ $Z_{2,3} = 4 (T_2)$ $Z_{2,4} = 27 (T_2)$	$RZ_3(T_6, T_7)$ $Z_{3,1} = 0$ $Z_{3,2} = 52 (T_4)$ $Z_{3,3} = 0$ $Z_{3,4} = 12 (T_4)$	RZ_4 $Z_{4,1} = 46 (T_5)$ $Z_{4,2} = 1 (T_5)$ $Z_{4,3} = 1 (T_5)$ $Z_{4,4} = 1 (T_5)$

The maximum number of RZ types is the number of hardware tasks. At the end of step 1, we obtain the task classes (RZ_i) as depicted by the example in Fig. 4. Algorithm 1 scans the RB-model of T_1 and checks whether there exists, in the list of RZ types $List\text{-RZ}$, a type of RZ that is already inserted and closely matches its required types of RBs. As $List\text{-RZ}$ is initially empty, the first type of RZ RZ_1 is created by T_1 . Similarly, as RZ_1 does not match the RB requirements of T_2 , step 1 produces the second type of RZ RZ_2 . Nevertheless, RZ_1 includes the RBs needed by T_3 , in this case, Algorithm 1 updates the number of RBs within RZ_1 by the maximum between the number of RBs in T_3 and that in RZ_1 . The required types of RBs in T_4 do not match any type of RZ included in $List\text{-RZ}$, Algorithm 1 decides the creation of a new type of RZ RZ_3 as required by T_4 and inserts it in $List\text{-RZ}$. In the same way, the fourth type of RZs RZ_4 is provided by T_5 . T_6 and T_7 need a type of RZ including RB_2 and RB_4 , which is closely represented by RZ_3 already inserted by means of the RB-model of T_4 . RZ_3 is not altered as the number of RBs in RZ_3 produced by T_4 exceeds the number of RBs within T_6 and T_7 . Four classes of hardware tasks are obtained, which are represented by RZ_1 , RZ_2 , RZ_3 and RZ_4 .

3.4.2 Step 2: Hardware Task Classification

Step 2 starts by computing costs D between hardware tasks and RZ types resulting from step 1. Based on RB-models of hardware tasks (T_i) and RZs (RZ_j), cost D is computed as follows, according to two cases.

We define through (7)

$$d_{i,j,k} = X_{i,k} - Z_{j,k}, \quad 1 \leq k \leq NP, \quad 1 \leq i \leq NT, \quad 1 \leq j \leq NZ \quad (7)$$

Case 1: $\forall k, d_{i,j,k} \leq 0$, RZ_j contains a sufficient number of each type of RB required by T_i , cost D is equal to the sum of differences in the number of each RB type between T_i and RZ_j weighted by $RBCost_k$ in (8).

$$D(T_i, RZ_j) = \sum_{1 \leq k \leq NP} RBCost_k \times |d_{i,j,k}| \quad (8)$$

Case 2: $\exists k, d_{i,j,k} > 0$ the number of RBs required by T_i exceeds the number of RBs in the RZ_j or T_i needs RB_k which is not included in RZ_j . In this case, the cost D between T_i and RZ_j is infinite (see (9)).

$$D(T_i, RZ_j) = \infty \quad (9)$$

The following example illustrates the computing of costs D between the seven tasks and RZ_3 . The lower the number of RB type on the device and the higher its functioning speed, the more its cost increases.

$$RZ_3 = \{0 RB_1, 52 RB_2, 0 RB_3, 12 RB_4\}$$

$$RBCost_1 = 20, \quad RBCost_2 = 80, \quad RBCost_3 = 192, \quad RBCost_4 = 340$$

$$D(T_4, RZ_3) = 0$$

$$D(T_1, RZ_3) = D(T_2, RZ_3) = D(T_3, RZ_3) = D(T_5, RZ_3) = \infty$$

$$D(T_6, RZ_3) = |36 - 52| \times 80 + 340 = 1620$$

$$D(T_7, RZ_3) = |15 - 52| \times 80 + |7 - 12| \times 340 = 4660$$

Step 2 assigns each task to the RZ type giving the lowest cost D with it and computes the workload for each RZ according to this assignment by using (10).

$$Load_RZ_j = \sum_{i \text{ in } RZ_j} C_i / P_i + (NbrPreemp_i \times Config_j) / P_i \quad (10)$$

$Config_j$ denotes the configuration overhead of RZ_j on the target technology. This overhead is computed by conducting the whole Xilinx partial reconfiguration flow from the floorplanning of RZ_j on the device up to partial bitstream creation and by taking into account the configuration frequency and the width of the selected configuration port. ($Config_j = \text{size of bitstream}/(\text{configuration frequency} \times \text{configuration port width})$.)

3.4.3 Step 3: Decision of Increasing the Number of RZs

Step 3 is performed when an overload ($>100\%$) is detected in some RZs during step 2. Step 3 aims to lighten the overload in RZs by conducting the migration of task execution sections to non-overloaded RZs before resorting to the solution of increasing the number of overloaded RZs. Hence, for each task, we search all the possible combinations of its execution sections. For each overloaded RZ, Algorithm 2 searches all the non-overloaded RZs that could accept at least one of its assigned tasks i.e. $D \neq \infty$. Then, Algorithm 2 makes a task-by-task check on the possibility of migration of an execution section or a combination of execution sections of the current task in order to reduce the overload of its RZ, respecting the workload of the non-overloaded receiving RZ. In the worst case, the complexity of Algorithm 2 is $O(M*N*NTM*TS)$, where M denotes the number of overloaded RZs, N is the number of non-overloaded RZs, NTM is the maximum number of tasks assigned to an overloaded RZ and TS is the maximum number of execution section combinations for a task assigned to an overloaded RZ. Step 3 groups the workloads of overloaded RZs in $L1$ (line 7) and the workloads of non-overloaded RZs in $L2$ (line 7). Step 3 goes throughout the RZs in $L1$ to resolve their overloads independently (line 11). Step 3 uses non-overloaded RZs in $L2$ to lighten the workloads of RZs in $L1$ (line 15). This step searches the non-overloaded RZ in $L2$ that gives finite cost D with at least one task assigned to the overloaded RZ during step 2 (line 19). Once step 3 finds the set of tasks that could be executed in the non-overloaded RZ, it balances the workloads between both RZs, respecting the task preemption points (line 22–line 37). If the overload persists in the RZ of $L1$, the algorithm decides to add other instances of this RZ up to $\lceil \text{workload of } RZ \rceil$ (line 42). When the processed non-overloaded RZ_n do not affect the added number of overloaded RZ_m , step 3 reinitializes their workloads to their values before dealing with the overload of RZ_m (line 43).

As output, level 1 provides a set of RZ types depicting the classes of hardware tasks and the number of their instances. The placement of hardware tasks is fulfilled by means of the two following levels.

4 Level 2: RPBs Partitioning on the Target Device

For each type of RZ provided by level 1, level 2 searches its potential physical locations on the target device by respecting its required RB types as well as the number of each RB type within the RZ. RPBs resulting from the partitioning must contain number of RBs greater than or equal to the number of RBs in RZ. Nevertheless, the partitioned RPB should be closest to the RB-model of RZs in order to ensure resource efficiency. Each RPB (RPB_i) is defined by its RB-model as expressed by (11).

$$RPB_i_RB = \{W_{i,k} RB_k\}, \quad 1 \leq k \leq NP \quad (11)$$

Algorithm 2: Decision of increasing the number of RZs

```

1:  $Load_m$ : the workload (%) of overloaded  $RZ_m$ 
2:  $Load_n$ : the workload (%) of non-overloaded  $RZ_n$ 
3:  $Load_{n,i}$ : the workload (%) of non-overloaded  $RZ_n$  after adding a section of execution of  $T_i$ 
4:  $Section_i$ : the list of possible execution sections of task  $T_i$  determined by its preemption
   points
5:  $Exe_i$ : execution section of  $T_i$ 
6:  $p, q, r, j, i, l$ : naturals
7:  $L1 = \{\text{workloads of overloaded } RZ_j\}; L2 = \{\text{workloads of non-overloaded } RZ_j\}$ 
8:  $L3$ : list of tasks
9: Sort  $L1$  in descending order
10: Sort  $L2$  in ascending order, in case of equality, Sort  $L2$  in ascending order according to
    configuration overhead
11: for  $p = 1$  to sizeof( $L1$ ) do
12:    $RZ_m = L1(p)$ 
13:    $Load_m = \text{workload}(RZ_m)$ 
14:    $q = 1$ 
15:   while  $q \leq \text{size of}(L2)$  and  $Load_m > 100$  do
16:      $RZ_n = L2(q)$ 
17:      $Load_n = \text{workload}(RZ_n)$ 
18:     // Search  $\{T_i\}$  from  $RZ_m$  to migrate to  $RZ_n$ 
19:     if  $\exists \{T_i\}$  assigned to  $RZ_m / D(T_i, RZ_n) \neq \infty$  then
20:       Sort  $\{T_i\}$  in ascending order according to  $D(T_i, RZ_n)$  in  $L3$ 
21:        $r = 1$ 
22:       while  $(r \leq \text{size of}(L3))$  and  $(Load_m > 100)$  do
23:          $T_i = L3(r)$ 
24:          $l = 1$ 
25:         // checking the possibility of relocation of the sections of  $T_i$  by respecting the
            workload of  $RZ_n$ 
26:         while  $l \leq \text{sizeof}(Section_i)$  and  $Load_m > 100$  do
27:           Select the first execution section  $Exe_i$  and discard it from  $Section_i$ 
28:            $Load_{n,i} = Load_n + Exe_i / P_i + Config_n / P_i$ 
29:           if  $Load_{n,i} \leq 100$  then
30:             // Migration of  $Exe_i$  from  $RZ_m$  to  $RZ_n$  is accepted
31:              $Load_m = Load_m - Exe_i / P_i - Config_m / P_i$  // Removing  $Exe_i$  from  $RZ_m$ 
32:              $Load_n = Load_{n,i}$  // Migration of  $Exe_i$  to  $RZ_n$ 
33:           end if
34:            $l++$ 
35:         end while
36:          $r++$ 
37:       end while
38:     end if
39:      $q++$ 
40:   end while
41:   if  $Load_m > 100$  then
42:     New  $RZ_m * (\lceil Load_m / 100 \rceil - 1)$  // Adding new  $RZ_m$ 
43:     Reinitialize the workload of  $\{RZ_n\}$  when it does not affect the number of added  $RZ_m$ 
44:   end if
45: end for

```

5 Level 3: Two-Level Fitting

Level 3 consists of two independent sub-levels. The first one ensures the fitting of RZs on the non-overlapped RPBs most suitable in terms of resource efficiency. The second sub-level performs the mapping of hardware tasks to RZs according to their predefined preemption points. Such mapping promotes the solutions giving the lowest configuration overhead and the lowest cost D . Task mapping guarantees an RZ for all the execution sections of the task, thus the problem of task rejection is discarded. Task mapping is strongly based on partial run-time reconfiguration that allows the dynamicity of execution on the same RZ without affecting the other running tasks on the distinct RZs. In addition, it increases the flexibility of the device as it enables the mapping of the task to several RZs.

6 Modeling of Placement Problem

The partitioning/fitting problem (level 2 and level 3) is a combinatorial optimization problem under constraints. It is characterized by an explosive space of admissible solutions. The placement problem is defined by the couple (S, F) , where S represents the set of the admissible solutions and $F(S \rightarrow R)$, R is a set of Reals) depicts the minimization objective function. The resolution of the problem consists of searching the solution s^* included in S where $F(s^*) \leq F(s)$ for each s in S . As all the constrained optimization problems, the problem of placement is defined by the quadruplets $\langle X, D, C, F \rangle$, where $X = \{X1, X2\}$ and $D = \{D1, D2\}$. $X1$ contains the first set of variables which consists of RPB coordinates, $X2$ contains the binary variables controlling whether the preemption points of each task is mapped to a given RZ. $D1$ and $D2$ represent respectively the finite domains of possible values of variables of $X1$ and $X2$. Hence, a potential solution for the problem consists of assigning each variable from $X1$ and $X2$ to a value from $D1$ and $D2$. C is a set of constraints that checks whether the combination of values is compatible with the variables. F is the minimization objective function that expresses the optimization criteria and enables the research of the optimal solution from the admissible ones. Consequently, we have associated the following mathematical model to the placement problem.

Constants

Task features, RZ features, RB features
 RR features: $width, height$, RB-model (RR_RB)

Variables (X)

$$X1 = \{(A_j, B_j), WRPB_j, HRPB_j, 1 \leq j \leq NZ\}$$

(A_j, B_j) : The coordinates of the upper left vertex of RPB_j on RR_RB

$WRPB_j$: The abscissa of the upper right vertex of RPB_j on RR_RB

$HRPB_j$: The ordinate of the bottom left vertex of RPB_j on RR_RB

$$X2 = \{PreempUnicity_{j,i,l}, 1 \leq i \leq NT, 1 \leq j \leq NZ, 1 \leq l \leq NbrPreemp_i\}$$

PreempUnicity_{j,i,l}: Binary variable checks whether the preemption point *Preemp_{i,l}* of task *T_i* is mapped to RZ *RZ_j*. It is equal to 1 when *Preemp_{i,l}* is fitted on *RZ_j*. For each possible solution, some other variables are also generated:

SumPreemp_{j,i}: This variable computes the sum of preemption points of *T_i* mapped to *RZ_j* by means of (12).

$$\text{SumPreemp}_{j,i} = \sum_{\substack{1 \leq l \leq \text{NbrPreemp}_i \\ \text{PreempUnicity}_{j,i,l} \neq 0}} 1, \quad 1 \leq i \leq NT, \quad 1 \leq j \leq NZ \quad (12)$$

OccupationRate_{j,i}: After preemption point mapping, the resulting occupation rate of each task *T_i* on each *RZ_j* is computed by (13). The occupation rate of the task *T_i* on *RZ_j* is the sum of its execution sections marked by the preemption points (*Preemp_{i,l}*) assigned to *RZ_j* (*PreempUnicity_{j,i,l}* ≠ 0)

$$\text{OccupationRate}_{j,i} = \begin{cases} \sum_{\substack{1 \leq l < \text{NbrPreemp}_i \\ \text{PreempUnicity}_{j,i,l} \neq 0}} (\text{Preemp}_{i,l+1} - \text{Preemp}_{i,l}) \\ + (\text{C}_i - \text{Preemp}_{i,\text{NbrPreemp}_i}), & \text{if } \text{PreempUnicity}_{j,i,\text{NbrPreemp}_i} \neq 0 \\ \sum_{\substack{1 \leq l < \text{NbrPreemp}_i \\ \text{PreempUnicity}_{j,i,l} \neq 0}} (\text{Preemp}_{i,l+1} - \text{Preemp}_{i,l}), \\ \text{otherwise} \end{cases} \quad (13)$$

AverageLoad: After preemption point mapping, the average of RZ workloads is calculated by (14).

$$\text{AverageLoad} = \frac{\sum_{\substack{1 \leq i \leq NT \\ 1 \leq j \leq NZ}} \frac{\text{OccupationRate}_{j,i}}{P_i} + \frac{\text{SumPreemp}_{j,i} \times \text{Config}_j}{P_i}}{NZ} \quad (14)$$

Domains (D)

D1 = {*da, db, dw, dh* are domains of naturals/*da* = *dw* = [1, *width*], *db* = *dh* = [1, *height*]}

D2 = {*d_{j,i,l}* is a domain of binaries, 1 ≤ *i* ≤ *NT*, 1 ≤ *j* ≤ *NZ*, 1 ≤ *l* ≤ *NbrPreemp_i* / *d_{j,i,l}* = {0, 1}}

Constraints (C)

Heterogeneity constraint (CPI): As RZs are fitted on RPBs, during RPB partitioning, the number of RBs within RPBs must be greater than or equal to those in RZs (*Z_{j,k}*) as formulated by (15) to satisfy RB requirements of RZs. Thus, RPBs must contain a sufficient number of each RB type needed by RZs. Because of the heterogeneity of RBs in the device and the rectangular shape of RPBs, the partitioned RPBs could include some RB types not required by RZs. Moreover, the number of

RB types in RPBs and included in RZs might exceed that required by RZs. This resource inefficiency is minimized by means of the objective function.

$$Z_{j,k} \leq \sum_{\substack{A_j \leq m \leq WRPB_j \\ B_j \leq n \leq HRPB_j}} \sum_{RR_RB[m][n] = RB_k} 1, \quad 1 \leq j \leq NZ \text{ and } 1 \leq k \leq NP$$

$RPB_j: (A_j, B_j, WRPB_j, HRPB_j); \quad RZ_j_RB = \{Z_{j,k} \text{ } RB_k\}$

(15)

Non-overlapping between RPBs (CP2): The constraint expressed by (16), restricts the fitting of RZs on non-overlapped RPBs.

$$\begin{aligned} A_q > WRPB_j \text{ or } A_j > WRPB_q \text{ or } B_q > HRPB_j \text{ or } B_j > HRPB_q \\ \forall 1 \leq j \neq q \leq NZ \end{aligned}$$
(16)

Non-overload in RZs (CM1): As noted in (17), during preemption point mapping for task T_i , none of the RZ RZ_j should be overloaded; the workload of RZ_j must not exceed 100%. As described by (17), the workload of each RZ takes into account the load produced by task execution as well as the configuration overhead of RZ by considering that the RZ is reconfigured at each preemption point.

$$\sum_{1 \leq i \leq NT} (OccupationRate_{j,i}/P_i + SumPreemp_{j,i} \times Config_j/P_i) \leq 100\% \quad (17)$$

Infeasibility of mapping for preemption points (CM2): The constraint expressed by (18) prohibits the mapping of preemption points of task T_i to RZ_j giving infinite D with the task. Effectively, as explained in step 2 of level 1, the infinite D between task and RZ means that there is a lack of RBs in RZ preventing task execution or an absence of RB types in RZ which are required by the task.

$$\begin{aligned} PreempUnicity_{j,i,l} = 0 & \quad \text{when } D(T_i, RZ_j) = \infty \\ \forall 1 \leq i \leq NT, \quad 1 \leq j \leq NZ \text{ and } 1 \leq l \leq NbrPreemp_i \end{aligned}$$
(18)

Uniqueness of preemption points (CM3): Each preemption point l of task T_i must be mapped to unique RZ_j (see (19)). Through this constraint, we guarantee also the completion of task execution as all the preemption points delimiting the execution sections of the task are fitted on RZs. Consequently, the problem of task rejection is eliminated.

$$\sum_{1 \leq j \leq NZ} PreempUnicity_{j,i,l} = 1, \quad \forall 1 \leq i \leq NT \text{ and } 1 \leq l \leq NbrPreemp_i \quad (19)$$

Minimization objective function (F)

In our modeling, the placement problem could be separated into two sub-problems: the partitioning of RPBs dealt simultaneously with RZ fitting and the fitting of tasks on RZs. The selection of the best solution for both sub-problems is guided by the objective function F that contains the optimization criteria (see (20)).

$$F = PlaceFunction + MappingFunction$$
(20)

PlaceFunction focuses on the fitting of RZs on the most suitable RPBs partitioned on the target device. The fitting must respect the heterogeneity constraint as

well as the non-overlapping between RPBs constraint. As expressed by (21), *PlaceFunction* evaluates the resource efficiency and promotes solutions that fit RZs on the closest RPB in terms of number and type of RBs.

$$\text{PlaceFunction} = \sum_{\substack{1 \leq j \leq NZ \\ 1 \leq k \leq NP}} \text{RBCost}_k \times (W_{j,k} - Z_{j,k}) \quad (21)$$

MappingFunction deals with the fitting of preemption points of hardware tasks on the RZs by respecting the three last constraints and by optimizing the three following criteria for measuring mapping quality (see (22)).

$$\text{MappingFunction} = \text{Map1} + \text{Map2} + \text{Map3} \quad (22)$$

The first expression of *Map1* evaluates whether the RZ is fully exploited; it evaluates to what degree the workload of RZ is close to 100%. The second expression checks whether the workloads of placed RZs are balanced; it entails checking whether the workloads of all RZs are near *AverageLoad* (see (23)).

$$\begin{aligned} \text{Map1} &= \sum_{1 \leq j \leq NZ} (100 - \text{Load_RZ}_j) + \sum_{1 \leq j \leq NZ} (\text{Load_RZ}_j - \text{AverageLoad})^2 / NZ \\ \text{Load_RZ}_j &= \sum_{1 \leq i \leq NT} \left(\frac{\text{OccupationRate}_{j,i}}{P_i} + \frac{\text{SumPreemp}_{j,i} \times \text{Config}_j}{P_i} \right) \end{aligned} \quad (23)$$

In (24), *Map2* computes the configuration overhead resulting from task mapping. *Map2* takes into account all the preemption points fitted on RZ, even the successive ones within the same task (*SumPreemp*_{j,i}), in order to obtain the worst-case configuration overhead. In fact, the scheduler could preempt a task on these successive preemption points in the same RZ in favor of a higher priority task. Minimizing *Map2* promotes the solutions of mapping tasks to RZs providing the lowest configuration overhead.

$$\text{Map2} = \sum_{\substack{1 \leq j \leq NZ \\ 1 \leq i \leq NT}} \text{SumPreemp}_{j,i} \times \text{Config}_j \quad (24)$$

As cost *D* reveals the resource waste when the task is mapped to RZ, *Map3* in (25) targets mapping tasks with high occupation rates to the RZs giving lowest cost *D* with them. In addition, more than resource waste, cost *D* also increases with the weight of each RB in terms of its frequency on the device and the importance of its functionality. Hence, the aim of *Map3* is the optimization of utilization of costly resources. Minimizing *Map3* ensures this optimization in resource use by mapping tasks with low occupation rates to RZs producing high costs *D* with them.

$$\begin{aligned} \text{Map3} &= \sum_{\substack{1 \leq j \leq NZ \\ 1 \leq i \leq NT}} (D(T_i, \text{RZ}_j)^2 \times \text{OccupationRate}_{j,i}^2 / 4 \\ &\quad - D(T_i, \text{RZ}_j) \times \text{OccupationRate}_{j,i}) \end{aligned} \quad (25)$$

7 Exhaustive Complete Resolution of Placement Problem

After selection of the necessary number of RZs from the off-line flow of hardware task classification and after fixing the set of preemption points for each hardware task, we start the partitioning/fitting resolution. Basing on recursive complete algorithms, we enumerate all possible assignments for variables of $X1$ and $X2$ until we find a solution satisfying the predefined constraints. All the found solutions are then evaluated with the objective function F in order to extract the optimal one. Algorithm 3 focuses on the partitioning/fitting RZs and Algorithm 4 resolves the sub-problem of task fitting on RZs.

These straightforward algorithms require going through the entire search space and do not finish in a reasonable time. Domains $D1$ and $D2$, with a finite number of values, define the size of search space for partitioning/fitting. To fit RZs on the RPBs

Algorithm 3: Exhaustive resolution of partitioning/fitting of RZs

```

1: Exhaustive_RZ_fitting(input  $E$ , input  $(X1, D1, C)$ , output  $solution$ )
2: if all variables of  $X1$  are assigned to a value in  $E$  then
3:   //  $E$  is a complete assignment: all RZs are fitted
4:   if  $E$  is with respect to constraints  $CP1, CP2$  then
5:     //  $E$  is a solution of RZ fitting
6:     Exhaustive_task_fitting( $E, (X2, D2, C), solution$ )
7:   end if
8: else
9:   //  $E$  is a partial assignment: some RZs are not yet fitted
10:  Choose a variable  $U_j$  from  $X1$  which is not assigned to a value in  $E$ 
11:  for all value  $V_j$  in a sub-domain of  $D1$  do
12:     $E_j = E \cup (U_j, V_j)$ 
13:    Exhaustive_RZ_fitting( $E_j, (X1, D1, C), solution$ )
14:  end for
15: end if
```

Algorithm 4: Exhaustive resolution of task fitting

```

1: Exhaustive_task_fitting (input  $E$ , input  $(X2, D2, C)$ , output  $solution$ )
2: if all variables of  $X2$  are assigned to a value in  $E$  then
3:   //  $E$  is a complete assignment: all tasks are mapped
4:   if  $E$  is with respect to constraints  $CM1, CM2, CM3$  then
5:     //  $E$  is a solution of partitioning/fitting
6:      $solution = solution \cup E$ 
7:   end if
8: else
9:   //  $E$  is a partial assignment: some tasks are not yet mapped
10:  Choose a variable  $U_i$  from  $X2$  which is not assigned to a value in  $E$ 
11:  for all value  $V_i$  in a sub-domain of  $D2$  do
12:     $E_i = E \cup (U_i, V_i)$ 
13:    Exhaustive_task_fitting( $E_i, (X2, D2, C), solution$ )
14:  end for
15: end if
```

having values of coordinates in $D1$, the number of assignments of these coordinates is defined by the search space $E1$:

$$E1 = \{(A_j, V_{11}), (B_j, V_{12}), (WRPB_j, V_{13}), (HRPB_j, V_{14}), 1 \leq j \leq NZ, V_{1n} \text{ is a value from a sub-domain of } D1\}.$$

The number of elements of this search space is:

$$|E1| = (|da| \times |db| \times |dw| \times |dh|)^{NZ} = (width^2 \times height^2)^{NZ}$$

The size of $E1$ increases exponentially with the number of RZs. Similarly, to map tasks to RZs by respecting their predefined preemption points, the number of assignments of the binary variables validating the mapping of preemption point to RZs is defined by search space $E2$:

$$E2 = \{(PreempUnicity_{j,i,l}, V_{j,i,l}), 1 \leq i \leq NT, 1 \leq j \leq NZ, 1 \leq l \leq NbrPreemp_i, V_{j,i,l} \text{ is a value from } d_{j,i,l}\}$$

Considering that the maximum number of preemption points within each task is equal to 5, the number of elements of this search space is:

$$\begin{aligned} |E2| &= |d111| \times |d112| \times |d113| \times |d114| \times |d115| \times \cdots \times |d1NT1| \times |d1NT2| \\ &\quad \times |d1NT3| \times |d1NT4| \times |d1NT5| \times \cdots \times |dNZ11| \times |dNZ12| \\ &\quad \times |dNZ13| \times |dNZ14| \times |dNZ15| \times \cdots \times |dNZNT1| \times |dNZNT2| \\ &\quad \times |dNZNT3| \times |dNZNT4| \times |dNZNT5| \end{aligned}$$

$$|E2| = \{0, 1\}^{NT \times NZ \times 5} = 2^{NT \times NZ \times 5}$$

For example, in Xilinx Virtex 5 SX50 technology, the width of its RB-model is 45 and its height is 6. To fit 6 RZs on this device, we must scan the space $E1$ of cardinal $(72\,900)^6$. For mapping 8 tasks to these fitted RZs, we must search within $E2$ of cardinal 10^{72} . Hence, the placement problem is an NP-complete problem as it has an exponential complexity in terms of its resolution. To avoid these complex prohibitive algorithms, we can use complete methods that intelligently scan the search space, thereby avoiding the assignment end up with non-admissible solutions. These methods employ efficient techniques, the most well-known one being Branch and Bound [15], described as the second method of resolution in the following section.

8 Non-Exhaustive Complete Resolution of Placement Problem

The method of Branch and Bound consists in enumerating all the possible solutions in an intelligent manner, relying on the features of the specified problem. This technique discards the partial solution that exceeds the last best bound function. In our problem, the bound function is calculated by the objective function F . The elimination of a partial solution not leading to the optimal solution considerably reduces the search space. Hence, the performance of Branch and Bound relies strongly on the quality of the bound function. Algorithm 5 describes the Branch and Bound

Algorithm 5: Branch and Bound resolution of placement problem [15]

```

1: c: natural // the counter on branched nodes
2: b: natural // the number of branched nodes provided by the current node
3: Best F = +∞,
4: Live = {(Node in root level, F(node in root level))} // the set of nodes to be processed
5: while Live ≠ ∅ do
6:   Select the node N from Live to be processed which gives the best F by using DFS
7:   Live = Live \ {(N, F(N))}
8:   if F(N) = F(X) for a feasible complete solution X and F(X) < Best F then
9:     Best F = F(X)
10:    Solution = X // A complete solution is founded
11:   else if F(N) ≥ Best F then
12:     Discard N from processing // partial candidate is rejected
13:   else
14:     // partial candidate is kept
15:     Branch on N generating N1, …, Nb by respecting the problem constraints
16:     for c = 1 to b do
17:       Bound Nc: compute F(Nc) // bound calculation for the node Nc
18:       Live = Live ∪ {(Nc, F(Nc))}
19:     end for
20:   end if
21: end while
22: Optimal Solution = Solution, Optimal Value = Best F

```

method applied on the placement problem by using the Depth First Search (DFS) strategy. For RZ fitting, as shown in Fig. 5, in each RZ-specific level of the search tree, a node depicts the RZ type fitted within potential RPB. For task fitting, as described by Fig. 6, in each task-specific level, each node denotes feasible mapping of its preemption points to RZs bringing finite *D* with it. Each iteration in Branch and Bound has three components: (1) selection of the node to process, (2) branching and (3) bound function calculation of the ramified nodes. The computing of bounds for nodes considers the best fitting for the remaining RZs or tasks without constraints. Whereas, only nodes respecting predefined constraints are ramified after node selection.

Figure 5 illustrates the resolution of the first sub-problem partitioning/fitting of RZs. We target to fit four RZs. The nodes with an X mark present the subsets of solutions that do not contain the optimal solution and with a √ mark present potential solutions. Initially, in RZroot level, only one subset of solutions exists, namely the root subset that contains all potential RPBs for *RZ*₁ with their bound functions. In RZroot level, the best node is selected, it is branched on partial solutions and the *PlaceFunction* of its ramified nodes are computed. The same processing is performed in *RZ*₂ level. As can be noticed, there are two best fittings processed in the *RZ*₃ level. The method starts by the first best node, i.e. *PlaceFunction3p* and processes this selected node. It is branched into two nodes by the RPBs of *RZ*₄. After computing the *PlaceFunction* for these ramified nodes, the *RPB*₁ – *RZ*₄ is selected and a complete solution is obtained. The last best *PlaceFunction* is *PlaceFunction41*. *RPB*₂ – *RZ*₄ is rejected as it does not optimize *PlaceFunction41*. Following

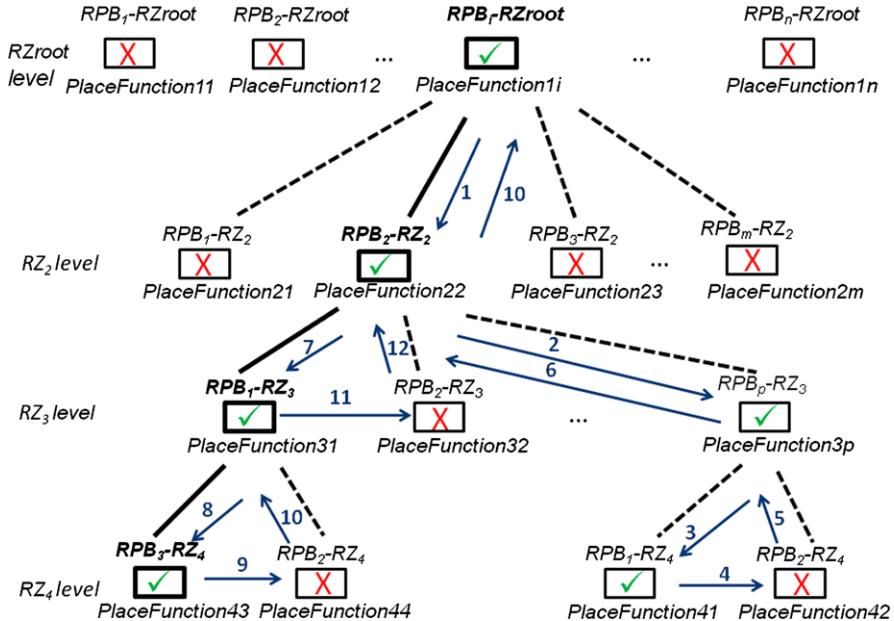


Fig. 5 Branch and Bound for partitioning/fitting of RZs

DFS, the next best node in the last visited level not yet processed is $RPB_1 - RZ_3$. The node is kept and branched into two RPBs on the RZ_4 level. $PlaceFunction43$ optimizes the last best $PlaceFunction$ and the solution is complete; thus, the optimal solution is obtained by $PlaceFunction43$. The next iteration processes the next best node in the last visited level; partial solution $RPB_2 - RZ_3$. This partial solution is rejected as $PlaceFunction32$ exceeds the last best $PlaceFunction$. The other nodes are not processed either, as their partial bound function exceeds the last best $PlaceFunction$. Finally, the optimal solution of partitioning/fitting of RZs is obtained by fitting RZ_{root} (RZ_1) on $RPB_i - RZ_{root}$, RZ_2 on $RPB_2 - RZ_2$, RZ_3 on $RPB_1 - RZ_3$ and RZ_4 on $RPB_3 - RZ_4$.

Figure 6 describes the progress of mapping three tasks to four RZs. The set of RZs giving finite cost D is provided for each task. In T_{root} level, the resolution selects the first best node i.e. $MappingFunction13$. After branching it and after computing the $MappingFunction$ for the partial solutions, the next iteration is released on T_2 level. In T_2 level, $MappingFunction23$ is the best partial solution; its node is ramified on T_3 level and the last best $MappingFunction$ is obtained by $MappingFunction31$. The process is repeated for the next best nodes in the last visited level. The nodes $MappingFunction21$ and $MappingFunction22$ are not processed since they exceed the last best $MappingFunction$. The method processes the next best node in the last visited level which is $MappingFunction11$. The branching of this node does not optimize the last best $MappingFunction$. Thus, its ramified nodes are not explored in T_2 level. Finally, the optimal solution of mapping hardware tasks to RZs is obtained by fitting the first preemption point P1 of T_1 on RZ_1 and its remain-

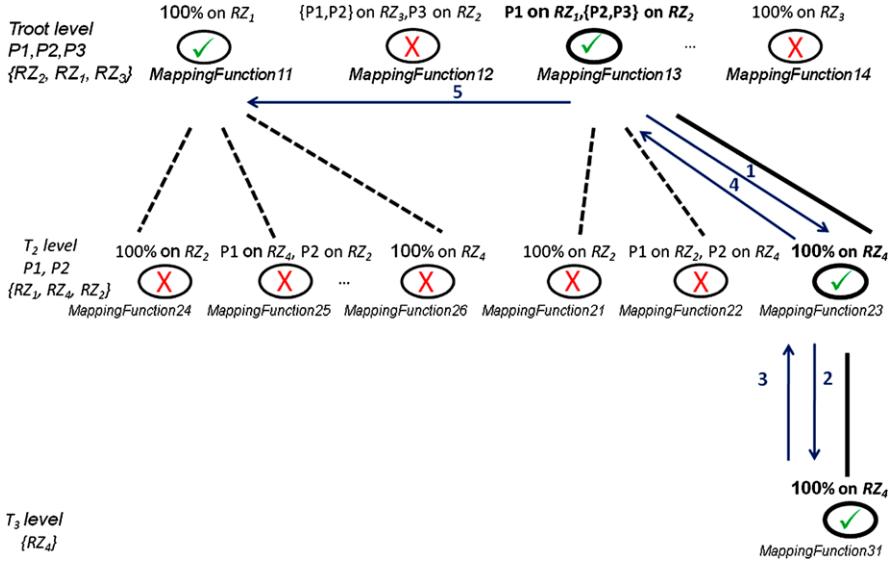


Fig. 6 Branch and Bound for task fitting

ing preemption points P_2 and P_3 on RZ_2 and by fitting all the preemption points of T_2 and T_3 on RZ_4 .

With both proposed methods, we ensure a full combination of RZ fitting as well as task mapping which solves the problem of task rejection confronted in the previous works of placement.

9 Application and Results

To investigate the influence of our three-level hardware task placement, we implemented an application composed of several hardware tasks taken from the Opencores website. The application in Fig. 7 contains heterogeneous tasks which are: microcontroller (T48), the master of the application that ensures the hardware task configuration as well as the data flow synchronization; FIR task, which computes 1000 FIR filters; MULTF, which performs 1000 floating point multiplications between two vectors of 8 bits; MDCT, which computes modified discrete cosine transform; AES, which performs encryption of blocs of 128 bits with 256 bit-key; DDS, which creates sinusoidal waves programmable with frequency and phase on-time; JPEG, which performs hardware compression of 24 frames per second and VGA, which drives VGA monitors with an 800×600 resolution.

The features of hardware tasks and their instances are presented in Table 1. At design time, we synthesized the hardware resources (RB-model) of these hardware tasks by means of ISE 11.3 Xilinx tool and we chose Xilinx Virtex 5 FX200 as reconfigurable hardware device. By considering the reconfiguration granularity, the

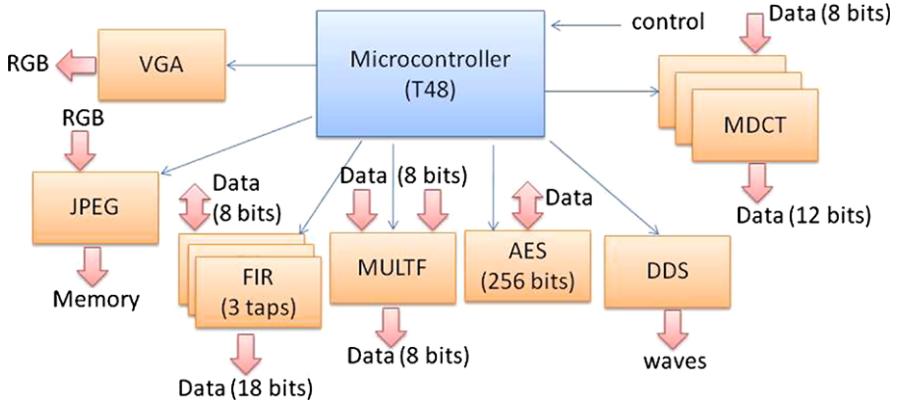


Fig. 7 Hardware tasks of the application

RBs in Virtex 5 are: RB_1 (20 CLBMs), RB_2 (20 CLBLs), RB_3 (4 BRAMs) and RB_4 (8 DSPs). We have assigned 20, 80, 192 and 340 as $RBCost$ respectively for RB_1 , RB_2 , RB_3 and RB_4 . Configuration overheads are determined by considering that each task defines an RZ by using a parallel 8 bit-width configuration port with 100 MHz as the configuration clock frequency. The partial reconfiguration flow dedicated by the PlanAhead 11.3 Xilinx tool enables the floorplanning of hardware tasks on the chosen device to create their bitstreams independently for estimating configuration overheads. Preemption points are determined arbitrarily according to the granularity of hardware tasks and their worst-case execution time (WCET). For all tasks, we consider that the first preemption point is equal to 0 μ s. The fifth column in Table 1 depicts the periods of hardware tasks.

As shown in Table 2, after completion of pre-placement analysis in level 1, step 1 results in the RZ types according to RB requirements in tasks and step 2 conducts the calculation of costs D between each task and each RZ type. Step 2 also computes the workloads of RZs after task assignment to RZs according to the lowest costs D mentioned by the bold numbers. An overload is detected in RZ_2 and RZ_6 . By means of step 3, the overload in RZ_6 is resolved by migration of two tasks among T_7 , T_{13} , T_{14} on RZ_3 on their second preemption points i.e. 120 μ s, since RZ_3 is the least loaded. Whereas, for RZ_2 , step 3 adds two other RZs having the same type of RZ_2 since the other RZs cannot totally resolve its overload. Consequently, the final number of RZs is equal to 8: RZ_1 , $3 \times RZ_2$ (RZ_2 , RZ_7 , RZ_8), RZ_3 , RZ_4 , RZ_5 , RZ_6 .

By means of AIMMS (<http://www.aimms.com>) environment that uses powerful solvers relying on Branch and Bound method, we have modeled partitioning/fitting of RZs as Mixed Integer Linear Programming model and the task fitting as Mixed Integer Non-Linear Programming model. By respecting the predefined constraints, Table 3 shows the obtained RZ fitting into RPBs on RR_RB . The RZ fitting is expressed by RPB coordinates: A, WRPB, B, HRPB as well as the RB-model of these RPBs. The costs Δ express the differences in RB_k between RZs and their associated RPBs and which evaluate the resource efficiency after RZ fitting.

Table 1 Hardware task features

Tasks	Instances	RB-model	WCET (μs)	Period (μs)	Configuration overhead (μs)	Preemption points (μs)
MDCT	$\{T_1, T_9, T_{10}, T_{11}, T_{12}\}$	$\{2RB_1, 12RB_2, 3RB_3, 0RB_4\}$	40 552	416 666	1856	10 000, 20 000, 30 000
AES	$\{T_2\}$	$\{4RB_1, 7RB_2, 1RB_3, 1RB_4\}$	51 540	100 000	2185	30 000, 40 000
DDS	$\{T_3\}$	$\{0RB_1, 1RB_2, 1RB_3, 1RB_4\}$	5000	12 000	432	none
T48	$\{T_4\}$	$\{5RB_1, 4RB_2, 0RB_3, 0RB_4\}$	20 000	50 000	605	800, 10 600, 16 300
JPEG	$\{T_5\}$	$\{8RB_1, 12RB_2, 0RB_3, 2RB_4\}$	350 000	416 666	2421	200 000, 300 000
MULTF	$\{T_6\}$	$\{1RB_1, 1RB_2, 0RB_3, 1RB_4\}$	5600	10 000	491	1400, 2350, 4020, 5170
FIR	$\{T_7, T_{13}, T_{14}\}$	$\{0RB_1, 1RB_2, 0RB_3, 1RB_4\}$	300	2000	112	120, 210, 255
VGA	$\{T_8\}$	$\{2RB_1, 4RB_2, 1RB_3, 0RB_4\}$	5000	10 000	681	1650, 2700

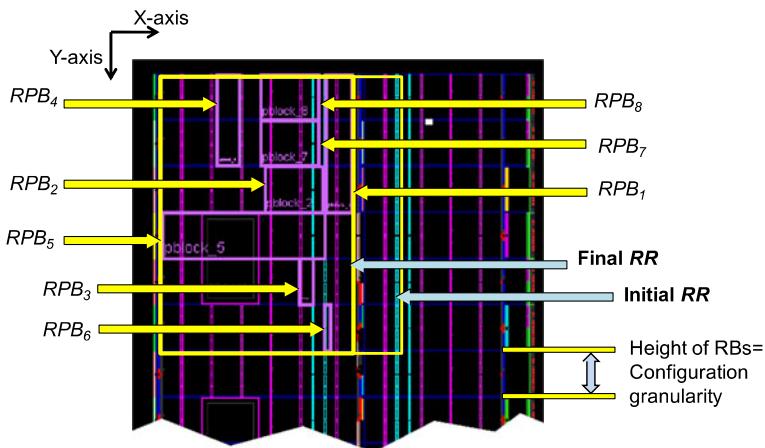
Table 2 The obtained results of level 1

RZ	MDCT (T_1, T_9, T_{10}, T_2)	AES (T_3)	DDS (T_4)	T48 (T_5)	JPEG (T_6)	MULTF (T_7, T_{13}, T_{14})	FIR (T_8)	VGA
RZ_1 (57%) $\{2RB_1, 12RB_2, 3RB_3, 0RB_4\}$	0	∞	∞	∞	∞	∞	∞	1024
RZ_2 (338%) $\{4RB_1, 7RB_2, 1RB_3, 1RB_4\} + 2 \times RZ_2(RZ_7, RZ_8)$ (step 3)	∞	0	560	∞	∞	732	752	620
RZ_3 (45%) $\{0RB_1, 1RB_2, 1RB_3, 1RB_4\}$	∞	∞	0	∞	∞	∞	192	∞
RZ_4 (44%) $\{5RB_1, 4RB_2, 0RB_3, 0RB_4\}$	∞	∞	∞	0	∞	∞	∞	∞
RZ_5 (85%) $\{8RB_1, 12RB_2, 0RB_3, 2RB_4\}$	∞	∞	∞	1380	0	1360	1380	∞
RZ_6 (112%) $\{0RB_1, 1RB_2, 0RB_3, 1RB_4\}$	∞	∞	∞	∞	∞	∞	0	∞

We observe high resource efficiency as the number of RBs within the RPBs is nearly equal to that of the RZs. The average of resource utilization is 36% of the available resources on the initial RR (55×6 RBs) delimited in Virtex 5 FX200 as shown in Fig. 8. We have created static design by floorplanning each instance of each hardware task on its RPB without using the concept of partial run-time reconfiguration. The obtained resource utilization of such static design is 63% of the

Table 3 Selected RPBs for RZ fitting on RB-model of reconfigurable device

RPB	A	WRPB	B	HRPB	RB-model	Δ
RPB_1	40	45	1	3	{3RB ₁ , 12RB ₂ , 3RB ₃ , 0RB ₄ }	1RB ₁
RPB_2	25	38	3	3	{4RB ₁ , 7RB ₂ , 2RB ₃ , 1RB ₄ }	1RB ₃
RPB_3	33	36	5	5	{0RB ₁ , 2RB ₂ , 1RB ₃ , 1RB ₄ }	1RB ₂
RPB_4	14	18	1	2	{6RB ₁ , 4RB ₂ , 0RB ₃ , 0RB ₄ }	1RB ₁
RPB_5	1	39	4	4	{8RB ₁ , 12RB ₂ , 3RB ₃ , 2RB ₄ }	3RB ₃
RPB_6	39	40	6	6	{0RB ₁ , 1RB ₂ , 0RB ₃ , 1RB ₄ }	0
RPB_7	24	37	2	2	{4RB ₁ , 7RB ₂ , 2RB ₃ , 1RB ₄ }	1RB ₃
RPB_8	24	37	1	1	{4RB ₁ , 7RB ₂ , 2RB ₃ , 1RB ₄ }	1RB ₃

**Fig. 8** Floorplanning of RPBs on Virtex 5 FX200

available resources on the initial RR. Therefore, the gain of configuration overhead in a static design is paid by the resource waste, which is 43% compared to our obtained results employing dynamic partial reconfiguration. The obtained RPBs are closely packed on the initial RR which avoids the resource waste and the external fragmentation on the device. For this reason, the initial RR could be resized to final RR in order to dedicate sufficient space for the remainder static part. The sub-problem of RZ partitioning/fitting was resolved after 2 hours and 30 minutes on CPU of 2 GHz with 2 GB of RAM.

The sub-problem of task fitting on RZs was solved within 9 seconds. The mapping of task preemption points of hardware tasks to RZs is detailed in Table 4. $T_{i,x}$ depicts the x -th execution section of T_i . $T_1, T_9, T_{10}, T_{11}, T_{12}$ are mapped to the unique RZ allowing their executions RZ_1 . T_7 and T_{14} start on RZ_6 , they are preempted after 85% of their worst-case execution time and continue their execution on RZ_3 . All the preemption points of T_{13} are fitted on RZ_6 . The biggest possible execution sections of T_7, T_{14} and T_{13} are mapped to RZ_6 . This mapping optimizes the

Table 4 Mapping of preemption points to RZs

RZs	Tasks	RZs	Tasks
RZ_1	100% of $T_1, T_9, T_{10}, T_{11}, T_{12}, T_{8,3}$	RZ_5	100% of T_5
RZ_2	$T_{6,4}, T_{6,5}, T_{8,1}$	RZ_6	$T_{7,1}, T_{7,2}, T_{7,3}, 100\%$ of $T_{13}, T_{14,1}, T_{14,2}, T_{14,3}$
RZ_3	$T_{7,4}, T_{14,4}, 100\%$ of T_3	RZ_7	$T_{6,3}, 100\%$ of T_2
RZ_4	100% of T_4	RZ_8	$T_{6,1}, T_{6,2}, T_{8,2}$

criteria of *MappingFunction*, particularly minimizing configuration overhead measured by *Map2* and the optimization of resource utilization evaluated by *Map3* as their costs D with RZ_6 is null. T_8 is mapped first to RZ_2 , it is stopped on RZ_2 on its second preemption point i.e. after 33% of execution and restarts on RZ_8 up to 54%. At this third preemption point, T_8 migrates to RZ_1 where it completes its execution. As RZ_2 and RZ_8 are of the same type, the high occupation rates of T_8 produced by the mapping of its preemption points to these RZs optimizes efficiently *Map3* as RZ_2 and RZ_8 give the lowest cost D (620) with T_8 . T_2, T_3, T_4 and T_5 are totally mapped respectively to RZ_7, RZ_3, RZ_4 and RZ_5 . The fitting of these previous tasks is the optimal solution for mapping their preemption points to RZs since it promotes both prominent criteria of measuring mapping quality: *Map2* and *Map3*. Task T_6 starts its execution on RZ_8 then is preempted on its third preemption point i.e. after 42% of execution. T_6 resumes its execution on RZ_7 till 72% of its execution. Hence, it is preempted again on RZ_7 on the fourth preemption point and restarts on RZ_2 where it is achieved. RZ_2, RZ_7 and RZ_8 are of the same type and give the best cost D (732) and the lowest configuration overhead for T_6 . After task mapping, *Map1* is highly optimized, effectively, the RZs are fully exploited and we have obtained an average load of 89%. In the worst case, the resulted preemption point fitting produces an overall configuration overhead of 72 959 μ s which is 11% of total running time. The obtained mapping guarantees an RZ for all execution sections for each task, consequently, there is no longer the problem of task rejection. During preemption point mapping, the tasks will be scheduled by respecting their deadlines which are equal to their periods and will be preempted on their predefined preemption points. Moreover, the temporal scheduling of preemption points of each task on RZs must respect the order of the corresponding execution sections.

10 Conclusion

An enhancement in placement quality is proved in this work. Our proposed three-level hardware task placement achieves a remarkably resource efficiency and minimization in overall configuration overhead. These obtained results are due to the optimization in resource use that relies on the physical features of target device, the functional features of hardware tasks and the advantages of partial run-time reconfiguration. Further research targets the placement/scheduling of task graph-based applications. We aim to extend our three-level method to take into account the

precedence and the deadline constraints by maintaining the quality of placement and scheduling especially execution span and overheads. The dependency between tasks should be investigated, especially in considering inter-task communication with the overall configuration overhead presented in this chapter. Inter-task communication will be an important criterion in deciding the optimal hardware task placement. In our FOSFOR project, the inter-task communication is studied by the management of an efficient communication network of type FAT-Tree as well as by the management of a shared memory between hardware tasks.

References

1. Bazargan K, Kastner R, Sarrafzadeh M (2000) Fast template placement for reconfigurable computing systems. *IEEE Des Test Comput* 17:68–83
2. Coffman EG Jr, Garey MR, Johnson DS (1997) Approximation algorithms for bin packing: a survey. PWS Publishing Company, Boston
3. Steiger C, Walder H, Platzner M, Thiele L (2003) Online scheduling and placement of real-time tasks to partially reconfigurable devices. In: International real-time systems symposium, pp 224–235
4. Marconi T, Lu Y, Bertels K, Gaydadjiev G (2008) Intelligent merging online task placement algorithm for partial reconfigurable systems. In: Design, automation and test in Europe, pp 1346–1351
5. Handa M, Vemuri R (2004) An efficient algorithm for finding empty space for online FPGA placement. In: Design automation conference, pp 960–965
6. Ahmadinia A, Bobda C, Bednara M, Teich J (2004) A new approach for on-line placement on reconfigurable devices. In: International parallel and distributed processing symposium, p 134
7. Cui J, Deng Q, He X, Gu Z (2007) An efficient algorithm for online management of 2D area of partially reconfigurable FPGAs. In: Design, automation and test in Europe, pp 129–134
8. Lu Y, Marconi T, Gaydadjiev G, Bertels K (2007) A new model of placement quality measurement for online tasks placement. In: Prorisc conference
9. Danne K, Stuehmeier S (2005) Off-line placement of tasks onto reconfigurable hardware considering geometrical task variants. *Int Fed Inf Process* 184:311–320
10. Lodi A, Martello S, Vigo D (1997) Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem. In: Meta-heuristics: advances and trends in local search paradigms for optimization. Kluwer Academic, Boston, pp 125–139
11. Lodi A, Martello S, Vigo D (1999) Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. *INFORMS J Comput* 11(4):345–357
12. Fekete SP, Kohler E, Teich J (2001) Optimal FPGA module placement with temporal precedence constraints. In: Design automation and test in Europe, pp 658–665
13. ElGindy H, Middendorf M, Schmeck H, Schmidt B (2000) Task rearrangement on partially reconfigurable FPGAs with restricted buffer. In: Field programmable logic and applications, pp 379–388
14. Xilinx (2007) Virtex-5 FPGA configuration user guide
15. Clausen J (1999) Branch and Bound algorithms-principles and examples. Denmark

End-to-End Bitstreams Repository Hierarchy for FPGA Partially Reconfigurable Systems

Jérémie Crenne, Pierre Bomel, Guy Gogniat,
and Jean-Philippe Diguet

Abstract This chapter presents an end-to-end hierarchy of bitstreams repository for FPGA-based networked and partially reconfigurable systems. This approach targets embedded systems with very scarce hardware resources taking advantage of dynamic, specific and optimized architectures. The hierarchy is based on three specific levels: FPGA local repository, local network repository and wide network repository. It allows the download of partial bitstreams depending on FPGA embedded resources and gives access to local or remote servers when a complete portfolio of bitstreams is needed. Based on real implementations and measurements, results show that the proposal is functional, uses a very little of hardware and software memory, and exhibits a download and reconfiguration time faster than state of the art solutions.

1 Introduction

FPGAs (Field-Programmable Gate Array) provide reconfigurable SoCs (System-On-Chip) with a way to build systems on demand. A single reconfigurable FPGA for many applications is a right answer to current critical issues in ASIC (Application Specific Integrated Circuit) design: the exploding design and production costs due to the continuous semiconductor technology density increase and the difficulty to upgrade and fix both hardware and software firmwares. Also, FPGAs hard blocks like processors, memories, DSPs (Digital Signal Processing) and high speed communication interfaces bring extreme flexibility at hardware and software levels, as well as at fine and coarse grain.

J. Crenne (✉)

LAB-STICC, Université Européenne de Bretagne, Lorient, France

e-mail: jeremie.crenne@univ-ubs.fr

In telecommunication industry, reconfigurable wireless Universal Terminal is now a well known idea which first appears in military area and became civilly popular in the 90s. This growing topic is a direct consequence on performances of FPGAs. This technology enables massive parallelism, enough computational power to realize DFE (Digital Front End) and the capability to be reconfigured at moderate power consumption [1]. Assuming that a device should support several digital mobile telephony services, digital broadcasting services, and/or digital data transfer services, it can take advantage of the partial reconfigurability. Current devices impose severe limited services due to inflexibility of their analog technology parts but tend to be bypassed by Software Defined Radio. SDR is a set of techniques that allows reconfiguration of a communication system without the need to physically change any hardware element. The underlying goal is to produce devices capable of supporting different services (multi-standard) with an adaptation of their hardware components in function of the wireless network such as GSM (Global System Mobile), GPRS (General Packet Radio Service), UMTS (Universal Mobile Telecommunications System) and WIMAX (Worldwide Interoperability for Microwave Access). In addition, they should be able to deal with wireless LAN (Local Area Network) standards like IEEE 802.11 known as WIFI (Wireless Fidelity). Delahaye et al. [2] prove the feasibility of dynamic partial reconfiguration on a heterogeneous SDR platform which provides a flexible way to build highly reusable systems on demand. Such devices require to dynamically adapt a subset of their functions in order to take all the variations in “real-time”. Thus these systems can take advantage of the dynamic partial reconfiguration (DPR) by swapping hardware resources on demand.

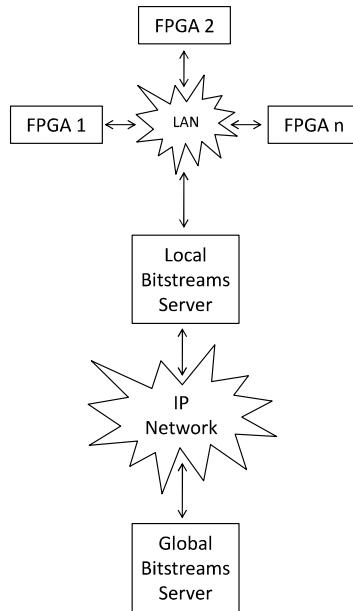
Xilinx’s Virtex FPGA reconfiguration can be exploited in different ways, partially or globally and externally (exo-reconfiguration) or internally (endo-reconfiguration). In this context Virtex’s dynamic and partial reconfiguration requires additional resources to store the numerous partial configuration bitstreams. Today, researchers exploit local FLASH memories as bitstreams repository and remote file servers accessed through standard protocols like FTP (File Transfer Protocol) or NFS (Network File System). Because memory is a scarce resource in low-cost, high-volume, embedded systems, we face the migration of silicon square mm from FPGAs to memories. Although low cost memories are in favor of this migration, there are some drawbacks:

- First, their reuse rate can be extremely low, since these memories could be used just once at reset in the worst case.
- Second, the balance in terms of global silicon square mm, component number reduction, PCB (Printed-Circuit-Board) area, power consumption and MTBF (Mean Time Between Failure), is negative.
- Third, for a single function to implement, the space of possible bitstreams can be huge and becomes bigger than the local memories. There are three combinatorial factors:
 - The FPGAs families with their increasing number of devices, sizes, packages and speed grades variations.
 - The number of possible configurations, unfortunately depending on spatial features like shape and location of IP (Intellectual Properties) area.

Table 1 Levels latencies and accesses types

Level	Latency	Access type
L1	1 ms–10 ms	local memory
L2	10 ms–100 ms	local server
L3	100 ms–1 sec	remote server

Fig. 1 LAN/WAN networking architecture. FPGAs are connected with an ad-hoc protocol to a local small bitstreams repository server. A larger global bitstreams repository server is used and can be queried with a standard IP protocol

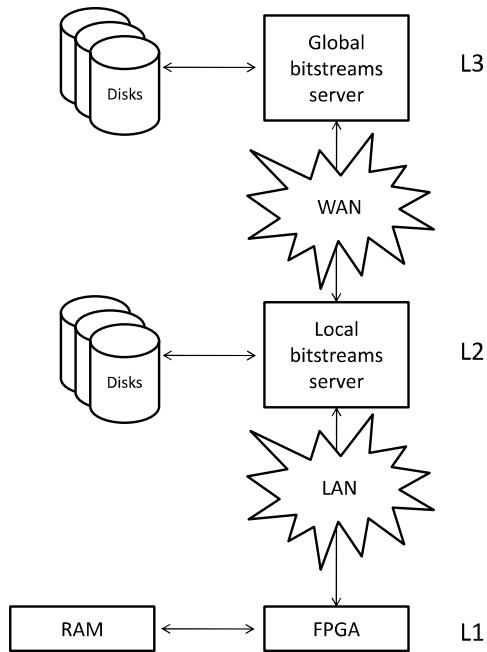


- The natural commercial life of the IPs producing regularly new versions and updates.

A bitstreams repository hierarchy becomes then necessary and must communicate through adapted physical channels and network protocols with the partially reconfigurable FPGAs. All FPGA system designers want the same: the best performances at the lowest cost to download partial bitstreams into FPGAs. The performances available today range from the ones provided by local memory where the latency is smaller to the ones of a remote file server where the latency is higher (Table 1). A bitstreams repository hierarchy delivers all versions of a single IP to all the portfolio targeted FPGAs. In a typical network topology (Fig. 1), this hierarchy is composed of three levels (Fig. 2):

- L1: a local bitstreams cache in memory.
- L2: a fast bitstreams server located in a dedicated LAN using a data link level protocol.
- L3: a standard global slower server located anywhere and accessed via TCP (Transmission Control Protocol) or UDP (User Datagram Protocol) based protocols.

Fig. 2 Bitstreams repository hierarchy levels $L1$, $L2$ and $L3$. Every level can be seen as a hierarchical memory. $L1$ is closer to the board and acts as a cache for FPGAs. $L2$ server is asked when the cache doesn't contain the required bitstream. If the $L2$ server is not able to find the required bitstream, then the last larger $L3$ server is queried



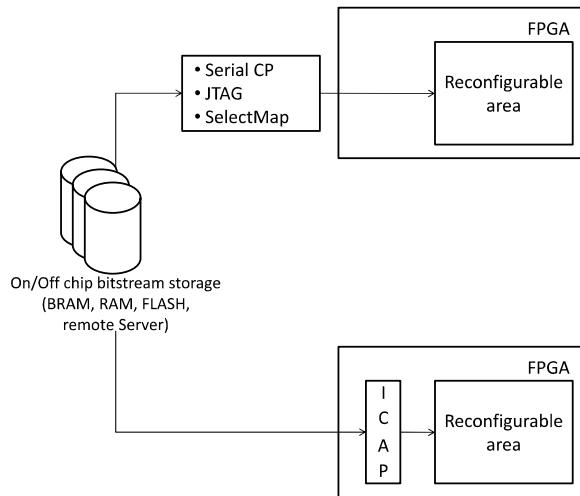
In the following, we present and describe each level in terms of software, hardware architectures, and communication protocols. We also intend to provide a specification and an optimized implementation of a minimal software layer abstracting the access to the involved hardware resources.

2 Hierarchy Level L1

Level L1 is the board level where designers glue together FPGAs and FLASH or RAM (Random Access Memory) memories. Bitstreams can be stored in memories and it is very common to use 512 MB FLASH ones. This is the most popular way to store bitstreams and build prototypes because there are many evaluation boards with FLASH readers at very low costs for Universities and researchers. But the less memories there are, the cheaper the system is to produce in high volume. L1 is geographically the closest repository to the FPGA, and the one with the smallest latency. Its latency depends, of course, on the memories and bus types used. It will always be the best, when compared with networking equipments.

The PR (Partial Reconfiguration) community agrees on the fact that, in applicative domains with strong real-time constraints, PR latency is one of the most critical aspect in its implementation. If not fast enough, the PR interest to build efficient systems can be jeopardized. Reconfiguration times will be highly dependent upon the size and organization of the partially reconfigurable regions inside an FPGA. Virtex-2 (V2) has column-wide frames embedded into partial bitstreams: hence

Fig. 3 Partial reconfiguration methods. The different exo-reconfiguration options are shown on the *top*, the endo-reconfiguration on the *bottom*



V2's bitstreams are bigger than necessary. Virtex-4s (V4) and Virtex-5s (V5) have relaxed this constraint, they now allow for arbitrarily-shaped regions. They have frames composed of 41×32 -bits words. The smallest V4 device, the LX15, has 3740 frames, and the largest V4 device has, the FX140, has 41152 frames. From Xilinx's datasheets, four methods of partial reconfiguration exist as shown in Fig. 3 and have different maximum downloading speeds:

1. externally (exo-reconfiguration):
 - a. Serial configuration port, 1 bit, 100 MHz, 100 Mb/s.
 - b. Boundary scan port (JTAG), 1 bit, 66 MHz, 66 Mb/s.
 - c. SelectMap port, 8 bits parallel, 100 MHz, 800 Mb/s.
2. Internally (endo-reconfiguration): Internal Configuration Access Port (ICAP) [3], V2, 8 bits parallel, 100 MHz, 800 Mb/s.

Of course, peak values are only objective and ICAP inside V4 and V5 have bigger word accesses formats (16 and 32 bits). Depending on the system designer's ability to build an efficient data pipeline from the bitstream storage (RAM, FLASH, or remote) to the ICAP, the performances will be close (or not) to the peak values. The good questions are "what latency is acceptable" for a given application and "what is its related cost" in terms of system cost (added memory/peripheral components). In particular, in the field of partial reconfiguration, to be able to compare contributions, we must identify what is the average size of a partial bitstream and what is its average acceptable reconfiguration latency. Finally, because systems can run at different frequencies, we must also integrate the system frequency in the numbers. Virtex V2p, Virtex V4, V5 and V6 series contain ICAP port and can be interfaced with hardware IPs or hard/soft processor cores such as PowerPC, Microblaze or OpenRISC. Maximum downloading speed rate announced by the fables in internal reconfiguration mode is capped to a maximum of 800 Mb/s when ICAP accesses are 8 bits wide. Entire cost for hardware its implementation is 150 slices and only a single BRAM.

2.1 Cache Architecture

The use of a FLASH memory via a mass storage card or an integrated on board memory is well known and use, at least for boot time. This kind of non-volatile storage is useful for maintaining a large range of bitstreams, and when the access time is not a constraint. Without talking about writing transactions, reading is close to 500 cycles for a single 32 bits word. This value is of course dependant on the flash technology, and the associated controller. With the use of a cache, designers are able to solve this “issue” by copying a bitstream into a faster memory which is located closer to the CPU. The problem here is that partial bitstreams are in a range of hundredths of KB when no compression is used like in [4]. Then, BRAMs memories are not the answer due to the overall available blocks which are much lower. BRAMs are a very scarce resources in FPGA. With this very short and basic analysis in mind, we propose the use of an SRAM for a cache memory. It is a tradeoff between the faster volatile memory (BRAM) and the lower non-volatile memory (FLASH). This complete software written cache, is efficient to speed up reconfiguration for some “critical” bitstreams at a low memory cost. Of course, we could expect a significant decrease in terms of reconfiguration time if a hardware IP is doing the same job. To avoid complex logic implementation, associativity is set to be direct mapped, which means that each line of the main memory can only be registered to only one address of the cache memory. The policy about memory usage is LRU (Least Recently Used) based: the less used bitstream will be replaced by the most used ones if there is not enough memory space to store all bitstreams. The difference between a regular memory cache and L1 is that the “line size” is rather big with L1. Xilinx’s product strategy has always been in favor of the reduction of the partial bitstreams size. Since Virtex V4 series, a real 2D partial reconfiguration can be applied and the column constraint of V2s is no more a bottleneck. Hence the average size of partial bitstreams is smaller. Anyway, the smaller the cacheline is, the less padding space will be lost when loading bitstreams which sizes are not pure multiple of this cacheline length. Defragmentation of free space in L1 can be done “off-line” while there is no bitstream transfer ongoing. Defragmentation is not detailed here, because it has no direct impact on downloading latencies.

2.2 Hardware Architecture

All the system is built using mature tools versions when used with dynamic partial reconfiguration. Both Xilinx’s tools EDK and ISE 9.2 as well as PlanAhead 10.1 for the partial workflow are used. The hardware architecture (Fig. 4) relies on a V2p 30 running at 100 MHz on a XUP evaluation board from Xilinx. A Power PC PPC405 core executes the PR software. We consider that dynamic IPs communicate with the FPGA environment directly via some pads. Thus, the FPGA is equivalent to a set of reconfigurable components able to switch rapidly from one

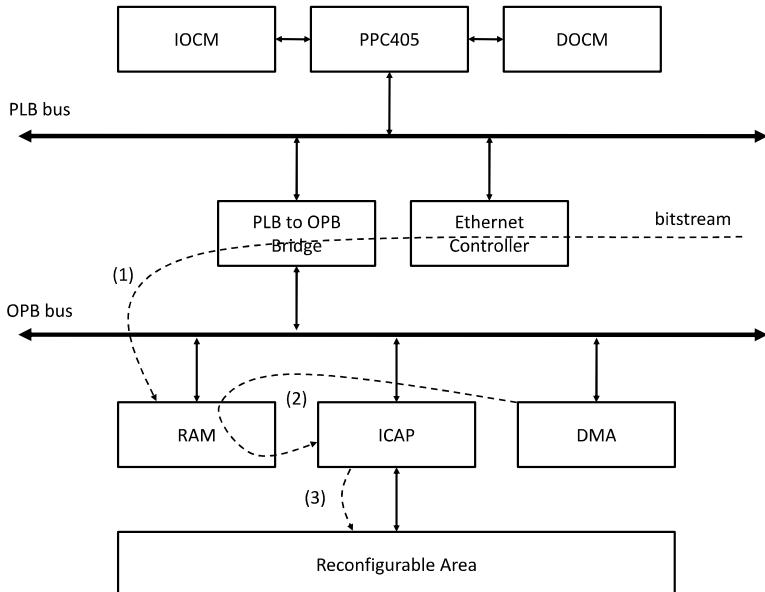


Fig. 4 L1 hardware architecture. When no bitstream in cache (RAM) is found, the incoming requested bitstream is sent by L2 repository and is copied from the Ethernet packet buffer to the cache (1). If the bitstream is present in cache, this one is copied to ICAP memory with the help of a DMA (2) and then written to reconfigurable area (3)

function to another. Communication with the PPC405 and inter-IPs communications are out of the scope of this chapter but can be implemented with Xilinx's and Hubner's bus macros [5] and OPB/PLB (On Chip Peripheral Bus)/(Processor Local Bus) wrappers for partially reconfigurable IPs as well as with an external crossbar like the Erlangen Slot Machine of Bodba et al. [6]. The design contains a PPC405 surrounded by its minimal devices set for PR. The PPC405 having a Harvard architecture, we add two memories to store the executable code and the data. These are respectively the IOCM (Instruction On Chip Memory) and the DOCM (Data On Chip Memory). The PPC405 communicates with its devices through two buses connected with a bridge. These are the PLB for faster devices and the OPB for slower devices. The Ethernet PHY controller is connected to the PLB and uses an integrated DMA (Direct Memory Access) to speed up transfer of incoming packets to the cache memory located in external memory. A second DMA is instantiated and managed by the PPC itself in order to copy a bitstream in cache to the ICAP removing PPC405 software copy bottlenecks. Finally the ICAP, connected to the OPB, manages the access and the downloading of bitstreams into the reconfigurable areas. The full exo-reconfiguration at reset is done using the external JTAG port while the endo-reconfiguration is dynamically done through the ICAP.

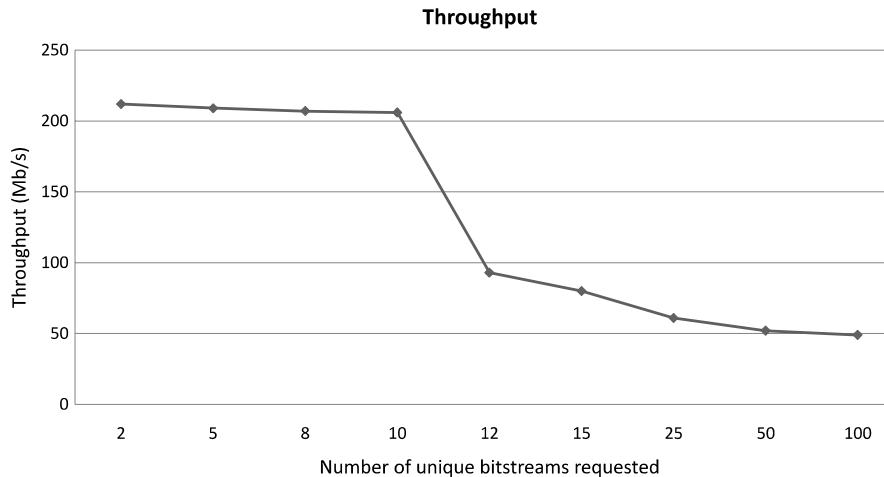


Fig. 5 Partial reconfiguration throughput. The effectiveness is related to cache capacity, replacement policy and associativity

2.3 Results

Our measures are based on the repetitive endo-reconfiguration of cryptography IPs like AES (Advanced Encryption Standard), DES (Data Encryption Standard) and 3DES. To obtain the following result (Fig. 5), the cache is configured to store 16 cachelines of 32 slots of 1496 bytes. With partial bitstreams of 74 KB, it is possible to store 10 bitstreams. The left side of the curve shows the sustained throughput when all bitstreams are present in cache. It exhibits an average download of bitstreams from the cache to the ICAP of about 2.1 Mb/(s MHz) be 210 Mb/s when the PPC405 is clocked at 100 MHz. As the cache is “only” able to store 10 bitstreams, the throughput dramatically decreases when the number of requested bitstreams is higher than the cache capacity. On the right side of the curve, we can underline that the more the number of unique requested bitstream is, the less is the throughput be a minimum of about 50 Mb/s. When no required bitstream is found in the local cache the hierarchy level L2 is automatically queried. This level is able to give access to a large number of partial bitstreams using a local server.

3 Hierarchy Level L2

Level L2 is the LAN level with a specific data link level protocol. It can provide a reconfiguration service with an average latency of 10 ms. Ethernet, in its simplest usage, is a medium sharing mechanism on top of which many protocols have been built. But it can also be seen as an excellent serial line. In terms of buying cost and ease of deployment it is a prime candidate to transfer bitstreams between close devices like our FPGA and the LAN bitstreams server.

Not strictly dedicated to DPR, the XAPP433 [7] application note from Xilinx, describes a system built around a Virtex4 FX12 running at 100 MHz. It contains a synthesized Microblaze processor executing the code of an HTTP server. The HTTP server downloads file via a 100 Mb/s Ethernet LAN. The protocol stack is Dunkel's lwIP [8] and the operating system is Xilinx's XMK (Xilinx MicroKernel). A 64 MB external memory is necessary to store lwIP buffers. The announced downloading rate is 500 KB/s, be 40 Kb/(s MHz) performances. This rate is 200 times less than the ICAP's one. Lagger et al. [9] propose the ROPES (Reconfigurable Object for Pervasive Systems) system, dedicated to the acceleration of cryptographic functions. It is built with a Virtex2 1000 running at 27 MHz. The processor is a synthesized Microblaze executing μ CLinux's code. It downloads bitstreams via Ethernet with HTTP and FTP protocols on top of a TCP/IP stack. For bitstreams of an average size of 70 KB, DPR latencies are about 2380 ms with HTTP and about 1200 ms with FTP. The reconfiguration speed is about 30 to 60 KB/s, be a maximum of 17 Kb/(s MHz). Williams and Bergmann [10] propose μ CLinux as a universal DPR platform. They have developed a device driver on top of the ICAP. This driver enables to download the content of bitstreams coming from any location because of the full separation between the ICAP accesses and the file system. Connection between a remote file system and the ICAP is done at the user level by a shell command or a user program. When a remote file system is mounted via NFS/UDP/IP/Ethernet the bitstreams located there, can naturally be downloaded into the reconfigurable area. The system is built with a Virtex2 and the processor executing the OS is a Microblaze. The authors agree that this ease of use has a cost in terms of performances and they accept it. No measures are provided. To have an estimation of such performances, some measures in a similar context have been done. A transfer speed ranging from 200 KB/s to 400 KB/s has been measured, representing a maximum performance of about 32 Kb/(s MHz).

This state of the art establishes that "Microblaze + Linux + TCP" dominates. Unfortunately, best downloading speeds are far below the ICAP and network maximum bandwidth. Moreover, memory needs are in the range of megabytes, thus requiring addition of external memories. Linux and its TCP/IP stack can't run without an external memory to store the kernel core and the communication protocols buffers. Secondly, the implementation, and probably the nature (specified in the 80s for much slower and less reliable data links) of the protocols, is such that only hundredths of Kb/s can be achieved on traditional LAN. Excessive memory footprint and maladjusted protocols are bottlenecks we intend to reduce.

3.1 Data Link over Ethernet 100 Mb/s

Ethernet standardized IEEE 802.3 [11], created by Metcalfe and Boggs at the Xerox Parc in the 70 s, is now a rich set of communication technologies to build cost effective LANs and to connect computers together. It is based on the diffusion of packets on a shared medium with collision detection (CSMA-CD). The insertion of

switches and hubs (multi-ports repeaters) to simplify cabling and to improve speed and quality of services, transforms the LAN into a set of point to point links connected through LAN-level routing equipments. With this topology, two equipments connected to the same switch communicate through a quasi-private link (excepted for broadcast packets). Ethernet's evolution is such that tenths, and even hundredths, of Mb/s are now available at very low costs with quasi-null error rates. With our repository hierarchy the bitstream server is connected to the same LAN than our system, it does not need level 3 routing toward any other LAN. Therefore we do not need IP routing and its companion protocols such as ICMP, ARP, TCP and UDP. The immediate drawback is that it does not allow the downloading of a bitstream from any other machine over Internet but remember, it will be the function of the L3.

To characterize in speed this LAN topology, a test must be done to define at which maximum speed Ethernet packets could be sent from a PC to the board. An application sends packets as fast as possible, with no specific protocol, no flow control and no error detection. Direct access to the Ethernet controller MAC level can be done easily thanks to the Linux raw sockets. This test demonstrates that a speed limit of almost 100 Mb/s is reachable. It depends only on the PC and switch performances. The absence of transmission errors during weeks of testing proves that, in such a context, the data link quality is so high that there is probably no need to implement a complex error detection. In line with Xilinx's strategy to reduce bitstreams, an error rates estimation can be done.

3.2 Error Rates

Partial bitstreams sizes are in the range of tenth of KBytes, say a maximum of 100 KB, be 800 Kbits. Each transmitted bit has a probability p of being erroneous. With today hubs these error rates are very small, at least in the magnitude of 10^{-9} . The probability to send n bits without error is obtained with the formulae $(1 - p)^n$, and the error rate is $1 - (1 - p)^n$ which gives the following values in (Table 2).

This shows error rates are very low for bitstreams. They are in favor of a very simple error detection and recovery strategy: a restart at bitstream level rather than at packet level. This is why the data link protocol described in [12] is a good candidate for such data transmissions. Moreover, when external perturbations occur, they will be concentrated on several adjacent packets. Their erroneous bits will have a higher correlation. Because we are not in field of RF transmissions, we do not need to decorrelate error bits with Reed–Solomon like interleavers. Here it is the opposite. The more concentrated the error bits will be, the better the protocol will be because it will reject all the bitstream file and then all the error bits. Thanks to the L1 cache at FPGA level, the bitstream transmission is only required when it is not present in the local memories and the bitstreams traffic is reduced.

Table 2 Estimated error rates for bitstreams downloading through network

n	p	$(1 - p)^n$	$1 - (1 - p)^n$
10^5	10^{-9}	0.9999	10^{-4}
10^5	10^{-10}	0.99999	10^{-5}
10^5	10^{-11}	0.999999	10^{-6}
10^6	10^{-9}	0.999	10^{-3}
10^6	10^{-10}	0.9999	10^{-4}
10^6	10^{-11}	0.99999	10^{-5}

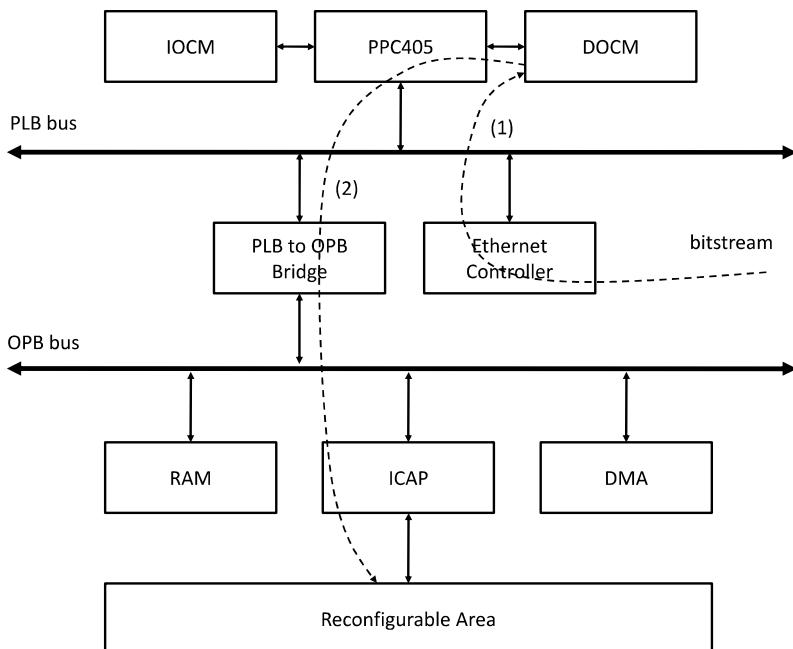


Fig. 6 L2 architecture without DMAs. The incoming bitstream is copied from Ethernet controller to DOCM memory (1) with the help of the Power PC processing time (2). The PPC writes the bitstream to the ICAP memory (2) ready to be written into the reconfigurable area

3.3 Hardware Architecture

The first proposed hardware architecture (Fig. 6) is similar to the one in Sect. 3 but without DMA instantiation saving a little of FPGA logic. The design exhibits a download of partial bitstreams at 400 Kb/(s MHz). Measurements show that the partitioning between hardware and software is not ideal. The bottleneck coming from software. Actually more than 90% of the processing time is spent in data transfers from Ethernet controller to the circular buffer and from the circular buffer to the ICAP controller.

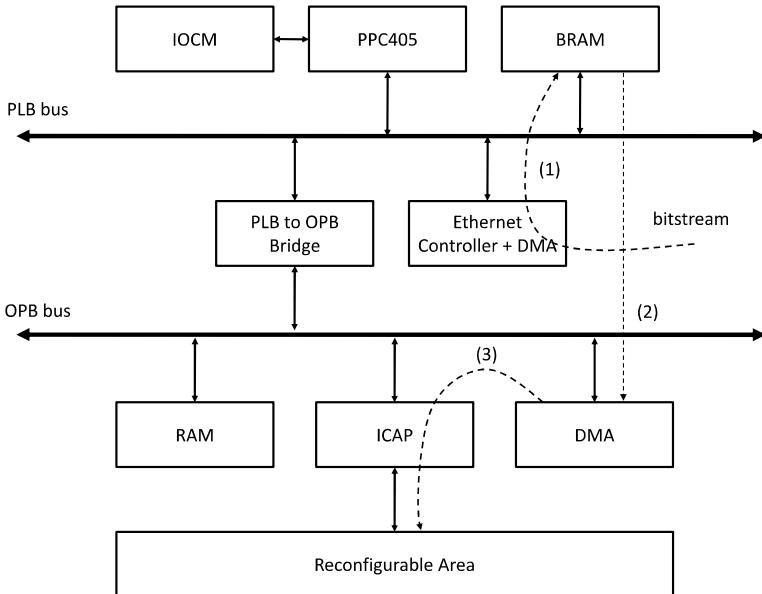


Fig. 7 L2 architecture with DMAs. The incoming bitstream is copied to a BRAM via a DMA integrated to the Ethernet controller (1). The content is then sent to the ICAP memory (2) with another DMA IP, and written into the reconfigurable area (3)

The second hardware architecture proposed (Fig. 7) relies on a V4 VFX 60 running at 100 MHz on a ML410 evaluation board from Xilinx. This FPGA has four embedded 10/100/1000 Mb/s Ethernet MAC controllers, among which only two are used on the ML410 board. Our architecture then used only one of these two, configured to communicate at 100 Mb/s. Instead of relying on pure software data transfer loops executed by the PPC, two DMAs are running in order to:

1. Transfer the data from the Ethernet controller to the circular buffer.
2. Transfer the data from the circular buffer to the ICAP.

The packets buffer, to be accessible by both DMAs cannot stay in DOCM (private to PPC) and must migrates in BRAMs located either on PLB or OPB bus. Because master accesses must be allowed for DMA, two bus bridges (PLB/OPB and OPB/PLB bridges) must be added to allow for such data transfers. After testing on V2/XUP and V4/ML410, we obtain similar results be a download of bitstreams at 800 Kb/(s MHz).

3.4 Software Architecture

The software architecture is based on three modules: the ICAP driver, the Ethernet driver and the PR protocol processing (Fig. 8). Then objective is to reduce the

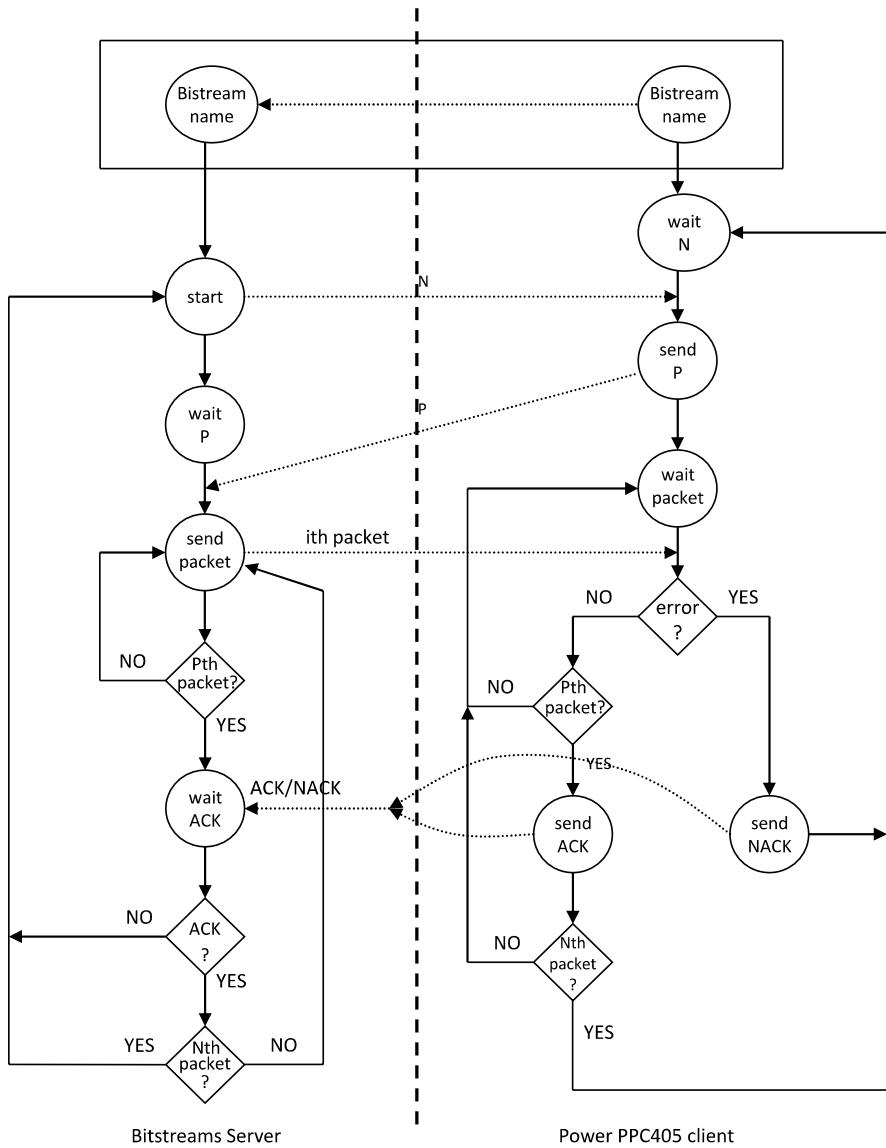


Fig. 8 Software dynamic partial reconfiguration protocol. Server and client software DPR protocol are respectively represented on the *left* and *right* side of the figure

number of software layers to cross when bitstreams are flowing from the Ethernet controller to the ICAP port. A time measurement module based on the internal hardware timer of the PPC405 and the access to the serial line via Xilinx's libc are also used and will not be commented as their use is marginal. This software establishes a data pipeline between the remote bitstreams server and the reconfigurable areas

in the FPGA. We can plan to reach the Ethernet maximum bandwidth of 100 Mb/s and today, with our 800 Kb/(s MHz), we reach a sustained rate of 80 Mb/s. To uncouple the ICAP downloading from the Ethernet packet reception we can design in software a producer–consumer paradigm: the producer being the Ethernet controller and the consumer being the ICAP port. A circular buffer is asynchronously fed with Ethernet packets by the Ethernet controller private DMA. Packets reception occurs by bursts: several packets are received without any data flow control feedback. The packet burst length (P) is less than or equal to the half capacity of the packets buffer. Each Ethernet packet has a maximum size of 1518 bytes and has a maximum payload of 1500 bytes of bitstream data. The PR protocol is executed concurrently with the Ethernet interrupt handlers. It analyzes the packets content and transfers the bit-stream data from the buffer to the ICAP port via the second DMA. The intermediate buffer sizing is a critical point in terms of performances. The bigger the burst is, the faster the protocol. The buffer size depends on the available memory at the reconfiguration time and this scarce resource can change in time. The protocol has been tailored to dynamically adapt its burst sizes to the buffer size, Ref. [12] gives its detailed specification.

3.5 Results

Results obtained (Fig. 9) depend also, as we could expect, on the producer–consumer packets buffer size allocated to the PR protocol. The curves at the top represent respectively from the bottom to the top, measured speeds for the first architecture for the second one. One can establish that, in all cases, when the packets burst has a size greater or equal to three packets ($P = 3$), maximum speeds of 400 Mb/(s MHz) (first architecture) and 800 Mb/(s MHz) (second architecture) are

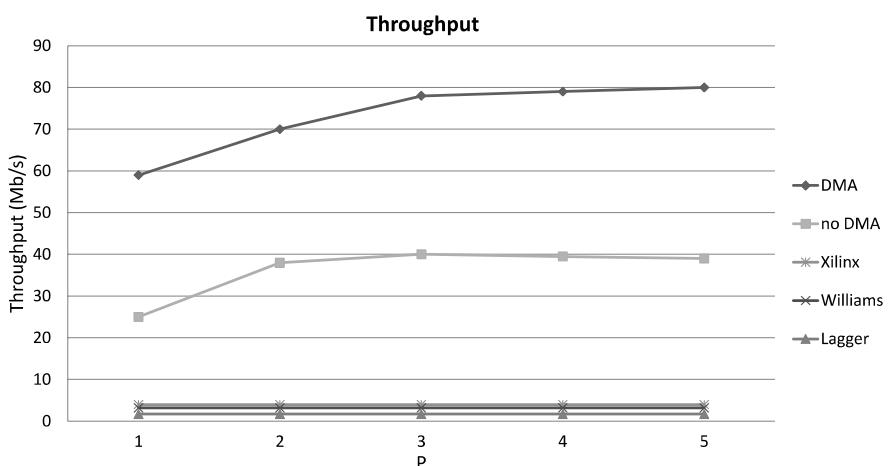


Fig. 9 Endo-reconfiguration vs P

reached and are stabilized. The size of the circular buffer being $2P$, it needs room for exactly six packets, be 9 KB (6×1.5 KB) only. Compared to usual buffer pools of hundredths of KB for standard protocol stacks, this is a very small amount of memory to provide a continuous PR service. Flat lines curves at the bottom, represent the average speeds reached by Xilinx, Lagger and probably Williams. Our PR protocol exhibits a reconfiguration speed of 80 Mb/s closer to our local 100 Mb/s Ethernet LAN limit. The gap between the reconfiguration speed and the ICAP speed is now about one order of magnitude instead of three orders of magnitude as previously. Finally, our PR software fits into 32 KB of data memory and 40 KB of executable code memory. When compared to related works, the endo-reconfiguration speed we have reached with our fast downloading is 20 times more efficient and needs less memory space.

4 Hierarchy Level L3

An ad-hoc specific data link is useful when no IP routing is required and only a little amount of hardware and software resources is available. However it is necessary to be able to download a bitstream from any machine. The use of the standard network architecture TCP/IP fits perfectly when a remote reconfiguration is necessary. Level L3 is the WAN (Wide Area Network) level where the latency is about 100 ms because of its geographic position which is the farthest.

4.1 Common Used Transport Protocols

Rind et al. [13] describe choices for TCP (Transport Communication Protocol) over UDP (User Datagram Protocol) and vice-versa related to speed, numbers of mobile devices and link capacity (bandwidth) metrics. Results are given in terms of throughput via a network simulator. It shows that TCP is giving better performance when minimum number of mobile devices are connected to a WLAN (Wireless LAN) and clearly setup that faster moving nodes are highly disturbing packets transmissions. UDP is found better if it is possible to bear little loss of packets. Consequently it is a first choice protocol for fast delivery of data. Uchida [14] presents a hardware-based TCP processor for Gigabit Ethernet which requires only one Ethernet PHY device. The circuit is small enough to be implemented on a single FPGA, with an announced 949 Mb/s throughput in both emission and reception directions. It has to be carefully compared with other systems. With this approach, no high traffic over the network is considered and TCP congestion control is well known to be designed and optimized for wired networks. International IEEE 802.11 [15] describes characteristics liable to a wireless LAN (WLAN). It makes possible to build broadband wireless local networks and in practice allows to link computers, laptops, PDAs, communicating devices and other peripherals, indoor or outdoor with ranges, speeds and modes depending on numerous revisions of the standard from 802.11a

to 802.11s. Wireless is sometimes the only possibility in applications where it is required to have a great mobility. In industry, reduction of wiring proves its pertinence: costly to install, to repair, imposing strict placements limitation. Compared to Bluetooth, the WIFI main strength stands in its higher throughput and range. As the system we target will be using a WIFI link and thus is limited to a much lower throughput, very high Gigabit transfer rate is oversized. Ploplys and Alleyne [16] perform a study where “wireless” UDP is used for real-time performance in control. Loss of data is well defined, explained and evaluated based on many factors such as range, environmental obstacles, computational loads and increased network traffic. Existing work establishes that TCP is vastly employed in LAN topology and UDP in WLAN. The use of UDP is natural when targeting wireless handled devices. UDP is also the most suitable standard for systems with a high latency and needs by nature, a shorter communication time.

4.2 TCP/IP Architecture Model

TCP/IP [17] (Transmission Control Protocol/Internet Protocol) is a networking reference framework used for developing networking applications. This model is usually described as a four-layered architecture as shown in Fig. 10. For a transmission, data are sent from layer 4 to layer 1, and vice-versa for a reception. In the following lines, we place the discussion on layer 1 and 3.

Today, IP protocols are adapted to low latency and high bandwidth data transfers. Therefore, this second point leads us to adapt our protocol to one of the most common transport protocol (layer 3). TCP [18] is used in many non-critical applications such as HTTP, FTP and SMTP. It is connection-based and guarantees that the

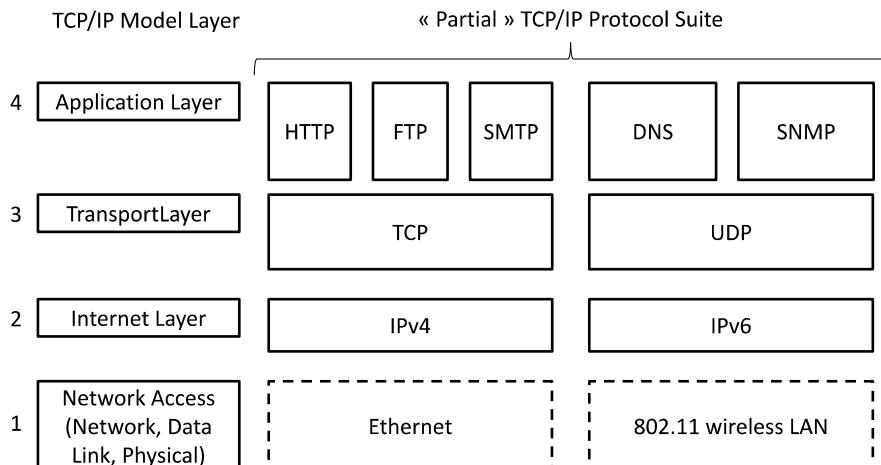


Fig. 10 The five-layered TCP/IP architecture model

Table 3 Comparison of TCP and UDP protocols [20]

Protocol	Complexity	Speed	Architecture	Caveats
UDP	Low	High	Broadcast client/server	Unreliable, string data
TCP	Average	Low	Client/server	String data

receiver will get exactly what the sender sent without any errors and in correct order. TCP resends the packet if it doesn't arrive correctly to the destination. To avoid congestion, TCP is cutting down speed whenever a packet is lost and re-increasing it slowly when packets are successfully transmitted. As for the most appropriate transmission protocol, we pinpoint that packets losses in TCP are attributed to congestion i.e. a high traffic. Hence for a wireless environment where bit error rate is high, TCP performances are highly degraded due to its window based congestion control mechanism. UDP [19] is similar to TCP and stands in the same TCP/IP layer. Known UDP applications are DNS and SNMP. Connectionless, its difference is located in the relationship between two parties. In other words, one can send data to another and that is all. UDP doesn't provide any reception reliability, thus, there is no guarantee that a packet will arrive. However if the transmission is correct, the packet will be received without any data corruption. UDP is faster than TCP as there is no extra overhead for error-checking above the packet level. A comparison between TCP and UDP is given in Table 3.

With this table and previous studied arguments, UDP has been found to be the most acceptable protocol for WLAN bitstream diffusion. From this point, we assume the use of UDP over the network.

4.3 Software Architecture

The architecture workflow differentiates the software running onto the server and the software onto the FPGA. On the left hand, the server executes a console application eventually hooked to a front-end executable. On the right hand, onto the client FPGA, the processor runs an executable built based on a TCP/IP stack.

4.3.1 lwIP as a TCP/IP Networking Stack

Instead of developping a networking library from scratch and in order not to reinventing the wheel, we choose an open source TCP/IP stack designed for embedded systems: lwIP. Directly available in EDK, lwIP [8] is an implementation under BSD license of the TCP/IP stack with RAM usage friendly in mind. It was initially written by Adam Dunkels of the Swedish Institute of Computer Science and is now maintained by several developers headed by Leon Woestenberg and hosted by Savannah. The porting proposed by Xilinx in EDK is robust (both Microblaze and PPC

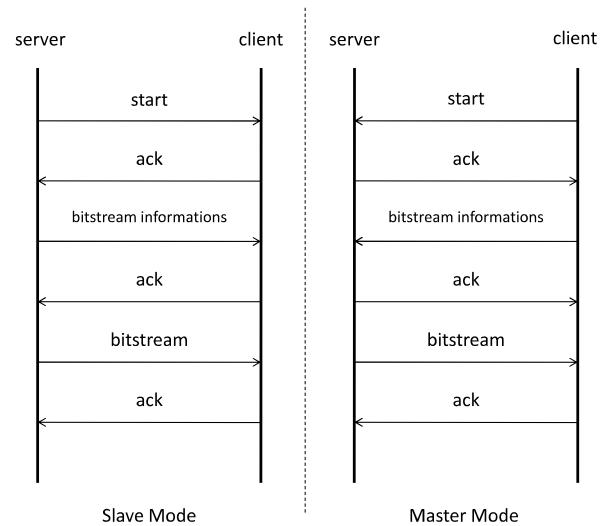
can be used without any problems), with a lot of parameters configurable at compile time that can be tuned to tailor an architecture according to the requirements. lwIP is also featuring a quite exhaustive characteristics list and can be run with an underlying OS or not.

Our first approach was to tailor lwIP to use UDP only as we don't need another protocol. To ensure packets producer-consumer paradigm, lwIP stack uses a pool of buffers. This pool is a critical point in terms of performances and has to be well scaled. Default setting value for this segment is close to 800 KB large be 512 packets of 1528 bytes. With that consideration we found native lwIP parameters to be oversized in EDK. The absence of transmission errors during days of testing, which consists of sending and receiving data as fast as possible and checking right packet order, proves that reducing it to 100 KB is pertinent without interfering overall performances. Next, we tried to unload the processor which is running the software from heavy computations. To achieve this goal, an option called UDP checksum offload could be set. It enables the network adapter to compute the UDP checksum on transmission and reception, saving the CPU time for doing it. The gain in terms of throughput is significant ($\times 2$) when the PPC is clocked at 100 MHz and can't handle too much computation.

4.3.2 Software DPR Protocol

We are now describing our protocol by first introducing the chosen frame's structure. It is always constituted of two fields. The first one is reserved for the frame number and is 4 bytes long. The second is the data, i.e. the bitstream. The allowable packet size relies directly to the frame format we use: DIX Ethernet Version II frame format. The total minimum size of one Ethernet frame is 1528 bytes. It includes 12 bytes of inter-frame spacing and 8 bytes of preamble. From these 1518 bytes, 4 are for Frame Check Sequence (FCS). Another 6 are for destination Ethernet address, 6 are source Ethernet address, and 2 are the type, leaving 1500 bytes for user data in each frame. Lastly, 20 bytes go to an IP header, and 8 to the UDP header, leaving a maximum of 1472 bytes for data in each frame. Bitstreams are generally bigger than this maximum so it has to be fragmented before emission. Packet's transmissions are synchronized using a classic end-to-end handshake for ensuring correct data transmission. The protocol is able to work in two modes: slave and master (Fig. 11). In master mode, the FPGA is responsible for asking the LAN server a partial bitstream. In slave mode, it reacts to the server requests and is forced to update itself. Obviously, obtaining a maximal reconfiguration throughput must be considered with care. Safety concerning the write of a partial bitstream to the reconfigurable area is necessary in partial reconfiguration context. A loss of a packet will result in an incomplete form of data reception, so on an impossibility of writing the complete partial bitstream into the reconfigurable area. Manifestly, it will lead to an unpredictable behavior. To avoid this, a frame number helps to know if a packet is missing or a wrong received order reception occurs. In addition, before writing to ICAP, a CRC (Cyclic Redundancy Check) is done to be sure that everything was fine during the transmission and every packets was received correctly any error.

Fig. 11 Slave and master mode protocol description



4.4 Hardware Architecture

The hardware architecture is completely similar to the one in Sect. 3. However the dynamic partial reconfiguration protocol as to be detailed in depth. A packet reception is divided into four steps (Fig. 12). It goes through internal Ethernet FIFO to the reconfigurable area:

1. First, an Rx interruption occurs. Received packet is stored into an internal Ethernet Controller FIFO.
2. Second, FIFO buffer is copied to RAM where memory pool (circular buffer) allocated by lwIP is available.
3. Third, Memory pool content is transferred to ICAP BRAM.
4. Finally, ICAP BRAM is written to the reconfigurable area.

To ensure that our software/hardware partitioning is the best, some evaluations are done, based on four basic architectural options:

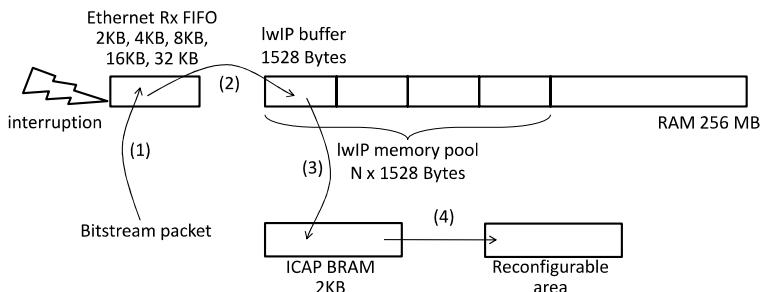


Fig. 12 Bitstream path from Ethernet to ICAP

Table 4 Hardware/software partitioning options results

	Cache enable	Cache disable	Cache disable + ICAP DMA	Cache enable + both DMAs
Throughput	–	+	–	+
Hardware memory footprint	+	+	–	–
Software memory footprint	+	+	–	–
Overall	++	+++	–	+

1. With processor data cache disable.
2. With processor data cache enable.
3. With processor data cache disable and DMA used to transfer memory pool content to ICAP BRAM.
4. With processor data cache enable and DMA for transferring.

In all these cases, the PPC405 instruction cache is activated and set to 16 KB large (8 BRAMs). When enabled the data cache is also 16 KB large. Table 4 demonstrates that software data copy with data cache enable is the best setup. This can be explained because EDK’s DMA engine has no internal buffering, and doesn’t do burst transfers. For processor without instruction cache, it might make sense to add a DMA, otherwise the inner loop of the optimized memory copy would be in cache and be executed at 2 cycles per instruction. The limiting factor will become the OPB latency (reading/writing from/to the OPB RAM). Indeed, when a data cache is enabled and as the processor exhibits cache coherency anomalies, it has to remain clean. It is the responsibility of the developer to ensure that any buffers in cache which are passed to the DMA are flushed from it. In addition the cache has to be invalidated prior to using any buffers resulting of a DMA operation. That is why we decided to rely on pure software concerning all data transfers and mainly between lwIP memory pool buffers and ICAP’s BRAM. Moreover, this saves some hardware resources and decreases significant overhead due to the OPB to PLB bridge as well as avoiding additional managements by the PPC.

4.5 Results

One can establish that the optimum packet size given is always close to the Maximum Ethernet Transmission Unit (MTU), 1500 bytes. Burst size is set to the maximum, be the total number of frames needed to send one bitstream depending on its size. The Ethernet controller FIFO is fixed to 8 KB. The server is running the application protocol in slave mode so it is initiating the transfer to the FPGA. Figure 13 sums up throughputs results as a function of the packets size and according to the network configurations given in Fig. 14.

The LAN configuration must be first envisaged to ensure the global effectiveness of the protocol. It linked directly server and client machine via a crossover cable.

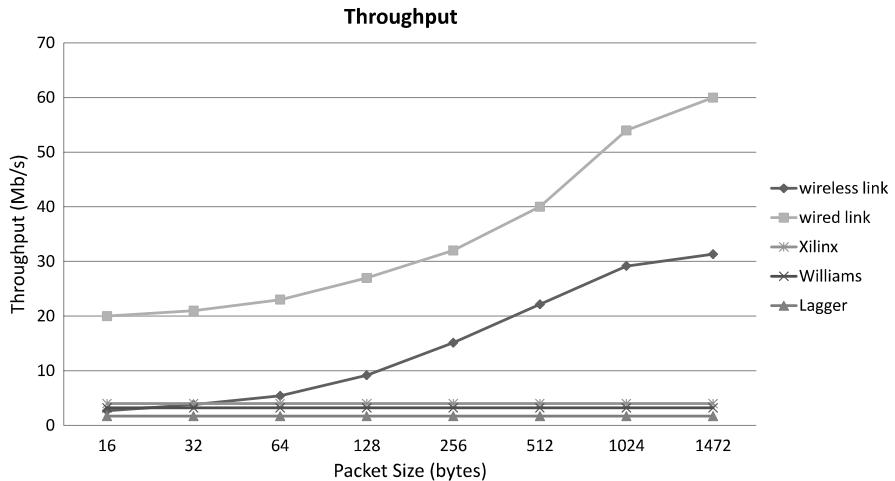


Fig. 13 Throughput curves for LAN and WLAN

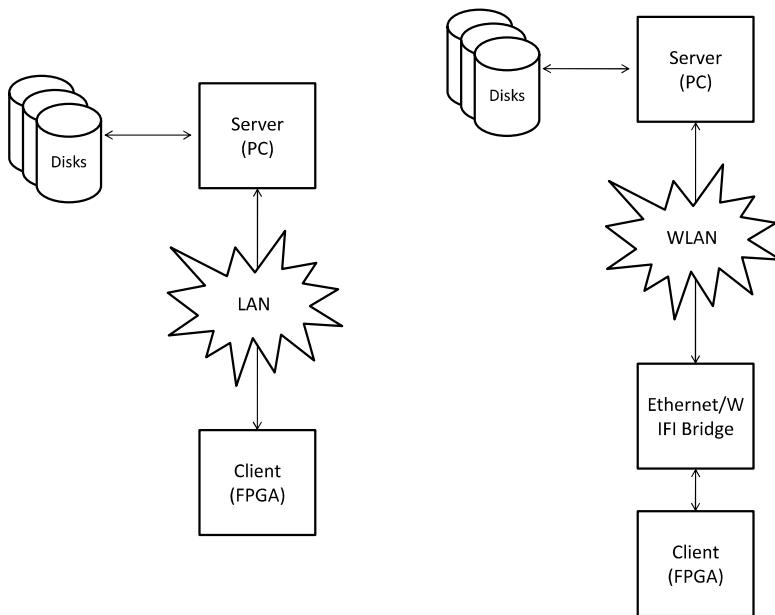


Fig. 14 LAN (on the left) and WLAN (on the right) protocol performance rating configurations

Results obtained depend as we could expect, on transmitted packets size. We obtain a sustainable 60 Mb/s throughput with an average packet size of 1472 bytes. This high transfer throughput matches with WIFI WLAN rate where 30 Mb/s is reachable.

The WLAN configuration describes the server which is connected to the client with a wireless link. FPGA is connected to an Ethernet/WIFI bridge removing the need of specific drivers. WIFI network type is set to ad-hoc allowing two or more wireless clients to be connected each other without any central controller. In this context, a constant 30 Mb/s throughput is reachable with the same LAN scenario software memory usage. This is the maximum physically achievable as the 54 Mb/s possible by the 802.11g standard is pure theory. To our best knowledge, no other works have proposed such a deal with dynamic and partial reconfiguration.

In terms of software, 100 KB are dedicated to executable code memory and 100 KB are allocated for data memory. This is 4 times less software memory compared to known previous works. In terms of hardware, all software instruction and data codes fits into BRAMs and then doesn't need any additional DDR RAM controller. Thus only 8% of the V2p chip, 2524 over 32383 slices, is occupied. In a Virtex V2p a slice consists of two 4-input look-up tables and two flip-flops. This size is sufficient to implement user circuits with the protocol on a one single FPGA. This brings our entire design to a lightweight implementation.

5 Conclusion and Perspectives

The proposed bitstreams repository hierarchy shows there is still opportunities to improve cache-level, LAN level and IP-level, caching strategies and protocols in order to provide an efficient and remote reconfiguration service over a standard network. Never fully taken into consideration in previous works, our PR platform takes advantage of the three specific levels L1, L2 and L3. For L1, where the reconfiguration throughput is 200 Mb/s at 100 MHz and is already in a very good shape for futures enhancements. For L2, the reconfiguration throughput is 80 Mb/s be a multiplication factor of 20 compared to other works. For L3, the wireless technology is used with its high 30 Mb/s throughput which is the physical maximum achievable. Moreover, when used in a LAN topology, the implementation exhibits an order of magnitude gain ($\times 15$) compared to the best previous work, be 60 Mb/s. The underlying protocol is also simple and could be reused “as is” with a low software memory cost: 200 KB for data and instructions, and only 8% of the V2p chip is needed for the biggest design. Table 5, sums up the respective speeds and memory footprints for every level architectures presented in this chapter.

From here we could target other implementations and optimizations for reconfigurable embedded systems. First, IP addresses are assigned statically. One could plan to do it more automatically and dynamically by implementing a DHCP (Dynamic

Table 5 Comparative architecture throughputs and software memory footprints (PPC at 100 MHz)

	[9]	[10]	[7]	L1 Arch1	L2 Arch1	L2 Arch2	L3 LAN	L3 WLAN
Throughput (Mb/s)	1.7	3.2	4	200	40	80	60	30
Memory (bytes)	>1 M	>1 M	<100 K	<100 K	<100 K	<100 K	>200 K	>200 K

Host Configuration Protocol) on the server depending of the context. This can allow them to be added to a network without manual intervention. Next, when targeting systems without PPC405 hard cores, it might be welcome to port the application to a synthesizable Microblaze soft core at a cost of significant slices loss unfortunately. Moreover, it is worth noting that there is no security guarantee when exchanging data between the server and the client. Confidentiality, data integrity and authenticity (secure transaction in a word) are not addressed here, but this is anyway a good path to follow. In addition, even if not a standard, RUDP (Reliable UDP) should be investigated for being a protocol vastly employed for real time performances. As Virtex V2p is now considered as an efficient but sometimes deprecated part due to tools incompatibility, migrating to a Virtex V5 or V6 is becoming a must be. These should deliver smaller partial bitstreams, thus smaller reconfiguration times, as well as a reduction of FPGA slices consumption. Last and not least when talking about network, Quality of Service (QoS) is essential for both LAN and WLAN. Number of works [21], [22] include an additional software or hardware reliability layer over UDP to ensure correct data by implementing some simple algorithms. It could be a great idea to focus onto such methods where hostile environments could lead to packet losses.

References

1. Becker M, Hubner M, Ullmann M (2003) Power estimation and power measurement of Xilinx Virtex FPGAs: trade-offs and limitations. In: Proceedings of the 16th symposium on Integrated circuits and systems design table of contents. IEEE Comput Soci, Washington, p 283
2. Delahaye JPh, Gogniat G, Roland C, Bommel P (2004) Software radio and dynamic reconfiguration on a DSP/FPGA platform. Frequenz J Telecommun 58:152–159
3. Xilinx ICAP. http://forums.xilinx.com/xlnx/attachments/xlnx/elinux/494/1/opb_hwicap.pdf
4. Hubner M, Ullmann M, Weissel F, Becker J (2004) Real-time configuration code decompression for dynamic FPGA self-reconfiguration. In: 18th international parallel and distributed processing symposium (IPDPS '04), workshop 3 IEEE Comput Soc, Los Alamitos
5. Hubner M, Becker T, Becker J (2004) Real time LUT-based network topologies for dynamic and partial FPGA self-reconfiguration. In: Proceedings of the 17th symposium on integrated circuits and system design. ACM, New York
6. Bodba C, Majer M, Ahmadiania A, Haller T, Linarth A, Teich J (2007) Increasing flexibility in FPGA-based reconfigurable platforms: the Erlangen slot machine. In: Proceedings of the conference on field-programmable technology (FPT), pp 37–42
7. Xilinx (2006) Xapp433. Web server design using microblaze soft processor
8. Dunkels A. lwIP. <http://www.sics.se/~adam/lwip/>
9. Lagger A, Upéguí E, Sanchez E (2006) Self reconfigurable pervasive platform for cryptographic application. In: International conference on field programmable logic and applications, FPL '06
10. Williams J, Bergmann N (2004) Embedded Linux as a platform for dynamically self-reconfiguring systems-on-chip. In: Proceedings of the international conference on engineering of reconfigurable systems and algorithms. CSREA Press, USA
11. Ethernet. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer. IEEE Standard 802.3
12. Bommel P, Gogniat G, Diguet JPh (2008) A networked lightweight and partially reconfigurable platform. Patent FR 08 50641 – N/R BFF 08P0055

13. Rind AR, Shahzad K, Qadir MA (2006) Evaluation and comparison of TCP and UDP over Wired-cum-Wireless LAN. In: IEEE multitopic conference, INMIC '06
14. Uchida T (2008) Hardware-based TCP processor for gigabit ethernet. *IEEE Trans Nucl Sci* 55(3):1631–1637
15. WIFI. Wireless LAN medium access control (MAC) and physical layer (PHY) specifications
16. Ploplys NJ, Alleyne AG (2003) UDP network communication for distributed wireless control. In: Proceedings of the ACC, Denver, CO, pp 3335–3340
17. RFC1122 (1989) Requirements for internet hosts – communication layers
18. RFC793 (1981) Transmission control protocol
19. RFC768 (1980) User datagram protocol
20. National Instruments (2009) Building networked applications with the LabWindows/CVI UDP support library
21. Grewal J, DeDoure JM (2004) Provision of QoS in wireless networks. In: Annual conference on communication networks and services research. IEEE Comput Soc, Los Alamitos
22. Raknet. <http://www.jenkinssoftware.com>

Part 3

Embedded Systems Design

SystemC Multiprocessor RTOS Model for Services Distribution on MPSoC Platforms

Benoît Miramond, Emmanuel Huck,
Thomas Lefebvre, and François Verdier

Abstract This paper addresses the problem of image processing algorithms and architecture adequacy by considering the need of custom operating systems in the context of multiprocessor Systems-on-Chip. We adopt a high-level approach for modeling the system that brings together the application, operating system and architecture in a single framework for the design space exploration. We have developed a configurable and modular SystemC model of multiprocessor-distributed operating systems. This model brings about new modeling properties at high-level in comparison with existing modeling frameworks. We also developed a graphical modeling environment that makes use of the modularity of the model to facilitate the design and provide automatic code generation. Thanks to this approach, both the algorithm-architecture matching and the RTOS service distribution can be jointly and easily explored at an early stage in the design flow. We present experimental results in the context of artificial vision for mobile robots and show that our method produces a good trade-off between estimation accuracy and simulation time.

Keywords Embedded vision systems · System-on-Chip · Design methodologies · Real-time operating systems modelisation · SystemC simulation · Multiprocessor architectures

1 Introduction

With the increasing complexity of embedded applications, hardware platform architectures include more and more parallel execution units. At the same time the current

B. Miramond (✉)

ETIS Laboratory, UMR CNRS 8051, Université de Cergy-Pontoise/ENSEA, 6, avenue du Ponceau, 95014 Cergy-Pontoise, France
e-mail: miramond@ensea.fr

technology is able to integrate 1 billion transistors into a single chip. Systems-on-Chip (SoC) now comprise heterogeneous processors with memory hierarchy, DMA, I/O and hardware accelerators, possibly reconfigurable, that communicate over a shared bus or a Network-on-Chip (NoC). In a real-time context controlling these complex systems is often devoted to one or more RTOS.¹ These can be deployed as software or hardware, either partially or completely depending on the non-functional constraints imposed by the global system. As a result, one must take new design decisions regarding the implementation of specific distributed RTOS.

The exploration of hardware/software systems was a research topic in the last decades, yielding a unified modeling and simulation environment, namely the SystemC language [1] which is now widespread used in the community. However, efficient and accurate modeling solutions for embedded software and real-time mechanisms in a multiprocessor context are still lacking.

In this paper we propose a method that tackles this design challenge by introducing a high-level RTOS model for custom system design. Working at a high level of abstraction allows the designer to jointly explore the parallel SoC architecture and the distribution of the RTOS in terms of custom services. Both dynamic behavior control and embedded constraints satisfaction problems can thus be solved by a single approach, early in the design flow. In the current work we thus propose a distributed and modular RTOS SystemC model. The model follows a Transaction Level Modeling (TLM) approach and introduces a Service Accurate level: it enables both functional and timed verifications without the need to model processing resources. By working at this level of abstraction, an early exploration of the architecture dimensioning is also possible.

This work is part of OveRSoC [2], a project which aims at developing a methodology for the design and the evaluation of specific OSes targeting Reconfigurable System-on-Chip (RSoC).

The remaining of this paper is organized as follows: Sect. 2 presents related works; Sect. 3 introduces the abstract modular RTOS model; Sect. 4 describes the method for the distribution of services onto MPSoC platforms. Section 5 presents the DOGME tool that implements the method. Experimental results are provided in Sect. 6. Finally, we conclude and discuss future works.

2 Related Work

Several approaches have been developed to deal with the exploration and validation of embedded software at high-level. He et al. in [3] classifies research on RTOS modeling and simulation into three categories: System Call translation, Native OS and Virtual OS. The latter corresponds to abstract models that facilitate exploration. In this context, SystemC [1] was mainly selected as the underlying modeling language for the developed executable models. A first step in our work was then to

¹Real-Time Operating System.

extend SystemC to embedded software modeling features which the current version of the language still does not support. The works presented in [4–6] and [3] are examples of simulation environments dealing with this challenge. In these works similar solutions are provided to model the mechanisms of target RTOS such as scheduling of multithreaded applications, preemption and synchronization. Indeed, a way to implement several scheduling policies in SystemC is to assign static or dynamic priorities to the simulation processes and to control which processes are declared ready in the simulation kernel. Scheduling task executions on a sequential processing unit thus consists in maintaining only one process unblocked. The API provided by these models must be as generic as possible in order to comply with most existing operating systems in the market. This in turn facilitates exploration. For example the SPACE project [7] encapsulates existing RTOS into a common API. As a consequence the entire code of the operating system must be available, since the RTOS and the application software are cross-compiled, and the resulting binary code is executed by an ARM ISS.² In order to be as generic as possible we based our approach on a modular API and the associated embedded software implementation. The API is generated according to the services needed inside each RTOS instance on the multiprocessor platform. In this respect, it is similar to [8] where authors propose to automatically extract the services required by system calls in the application code.

Most of the proposed abstract RTOS models target monoprocessor architectures. Yet increasing complexity of embedded system requires parallel and heterogeneous platforms. Madsen et al. in [9] proposed a framework that supports modeling of multiple RTOS. Application software is modeled as a set of tasks executing on a multiprocessor platform. The approach does not take the exact functionality of tasks into account. As said by Hwang et al. in [10] a trade-off is needed to determine the best abstraction level of simulation models. To be exact, the dynamic behavior of distributed real-time applications can only be reached with functional models by using either ISS approaches [11] or annotated TLM models [10] as is the case in this paper.

3 RTOS Modeling

This section presents the main mechanisms needed to jointly model and simulate hardware/software tasks and the RTOS in SystemC.

In order to model complex embedded platforms made up of multiple parallel and heterogeneous resources, it is crucial that we modeled not only the functional software, and the underlying hardware but also the bindings between them, which are generally composed of RTOS instances.

In order to explore the design solution space as efficiently as possible, we decided to model the system at a high level of abstraction where the hardware is par-

²Instruction Set Simulator.

tially hidden. Our modeling methodology focuses on the services provided by the platform.

The core element of our distributed architecture model is a high-level functional RTOS model written in SystemC. Since SystemC does not support OS modeling facilities in its current version, we first extended SystemC with embedded software modeling features, which was the subject of a previous work [12].

In this paper, we address the SAT level of abstraction: Service Accurate plus Time. In comparison with lower detailed levels of abstraction, this allows us to drastically reduce the simulation time of the behavior of the application. This approach is very different from Donlin's CP+T level (Communicating Processes with Time) [13], which mainly focuses on hardware modeling but does not include RTOS services.

This level of modeling implies that (i) the architecture is not modeled explicitly, (ii) all the application tasks are functional, annotated with approximated or measured execution times, and (iii) all the RTOS services are explicit and timed.

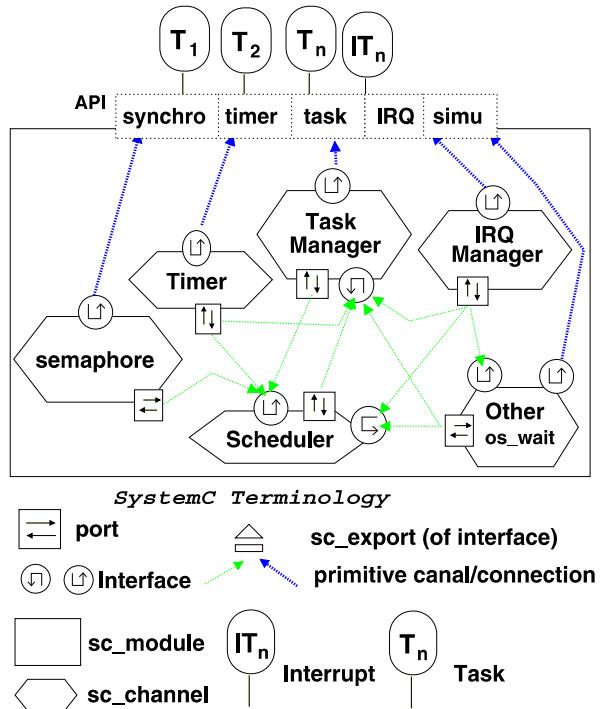
The proposed RTOS model in [12] acts as a *Service Accurate + Time* model for a virtual Processing Element (PE): all the necessary services of an embedded RTOS are modeled as independent modules each with its own behavior and timing. A PE can represent either a classical embedded processor or a dedicated accelerator. The RTOS model is built upon a collection of *service modules* that are implemented in the form of hierarchical sc_modules; this fosters high level exploration of custom architectures. The main RTOS model instantiates all of its modules and uses sc_export to provide a global API to the application code, as illustrated in Fig. 1. Each *service module* has its own interface that provides the embedded application with the corresponding services functions. This model includes mechanisms to model the dynamic creation of tasks, task preemption and interrupt handling [12]. Fig. 1 exemplifies the hierarchical structure of the SystemC RTOS model composed of the following service modules:

- a task manager that keeps track of the information and properties of each task according to its implementation (either software or hardware): namely its state, its context, its priority, timings, its area, software or hardware resources in use...
- a scheduler that implements a specific algorithm, e.g. EDF [14], HPF [14], horizon [15],....
- A synchronization service based on semaphores.
- A time management service that keeps track of time, timeouts, periods, deadlines....
- An interrupt manager that renders the system reactive to external or internal events.
- A specific simulation service that periodically increases the time variable.

Each service module is modeled as a SystemC hierarchical sc_channel and is represented in Fig. 1 using the usual SystemC representation [1]. A service module thus provides several service functions through its interface.

For instance, the task manager provides the following functions: the dynamic creation of a task, the deletion of a task, the retrieval of the state of a task, the modification of the state of a task.... The task creation function associates a simulation

Fig. 1 The modular RTOS model and its composed API. Each OS service exports its own interface to the application. Services are connected together to ensure the global OS coherency and behavior



process (and thus concurrency) to one of the pure C functions present at the application layer.

Some service functions are accessible from the application layer through the OS API. Those are called *external service functions*. Others are only accessible from the other service modules through a SystemC port to establish inter-module communications and are called *internal service functions*.

In this layer, timed simulations of the application use a specific simulation call (called OS_WAIT()), associated with each block of task code between two system-calls. This service, makes it possible for each function to progress in time and to manage interruption and preemption (see Fig. 1). In addition, each OS service function within the OS itself may also be annotated with timing information (depending on the processor) making a timed simulation of a realistic system possible. The evaluation of execution time is always in terms of a number of processor cycles. It can be either a rough estimation or dynamically computed by a SystemC ISS connected to our RTOS model.

The system library currently provides a set of basic generic services, including interrupt management, timer management, inter-tasks synchronization, and memory management. Other works [16] have extended the library to manage reconfigurable architectures. The extended library provides hardware and software specific services such as the management of software or hardware tasks, software scheduling policies and hardware placement algorithms.

4 MPSOC Modeling and RTOS Distribution

4.1 Distant Communications and Services Requests

We now present the model extension for distributed multiprocessor architecture exploration. The distributed model makes the following assumptions: the application is decomposed into multiple threads sharing the same address memory space, the application is statically partitioned onto multiple processing nodes, each processor has its own scheduling strategy (policy, task priorities etc). All inter-processor communications are modeled using SystemC transactions with respect to TLM methodology [1]. A unique `transport` method is used for both requests and replies. All communications are currently performed instantaneously but the transactional approach enables future communication refinements (4).

Our approach for modeling distributed OS services is inspired from the middleware philosophy which consists in using proxy and skeleton services. A proxy service provides a local entry point to a distant service accessible through an interconnection infrastructure. This approach adds dedicated ports and interfaces to the RTOS and to the service modules.

Figure 2 illustrates transactions between two local semaphore services (proxies) and a shared distant semaphore implementation (skeleton). Get and release

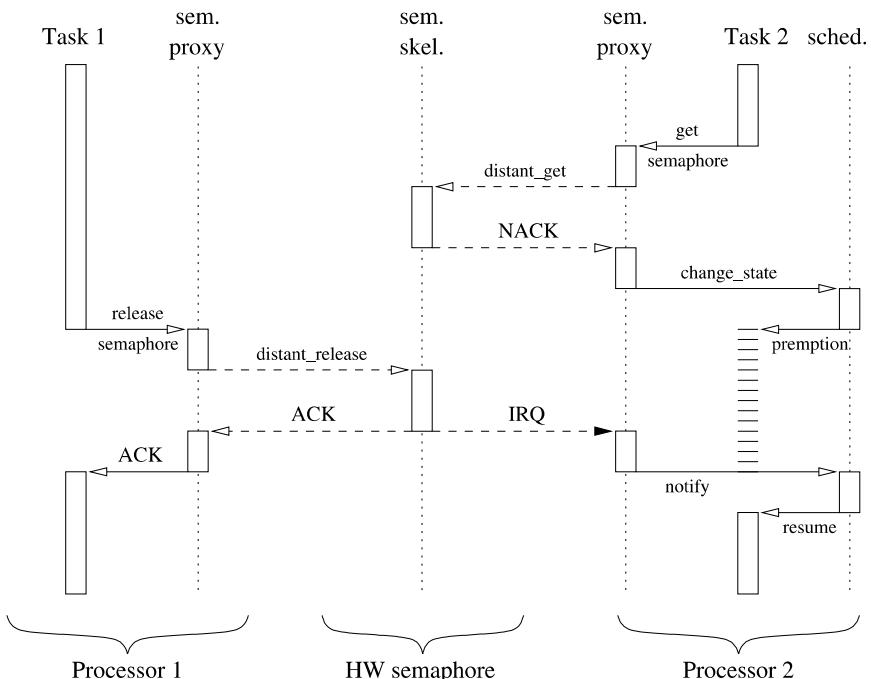


Fig. 2 Activity diagram of local/distant calls to a shared semaphore proxy/skeleton between two OS models

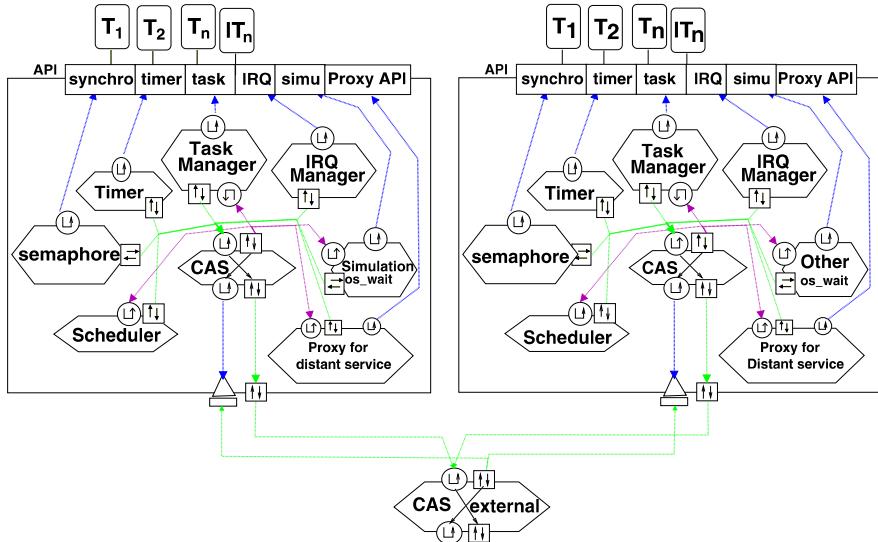


Fig. 3 Distributed services communicate through the CAS abstract channel. Model of MPSoC RTOSes with a hardware shared semaphore service. Each RTOS has a local *Proxy* service which forwards a (semaphore) request to an external device (the *skeleton*) that processes the real service, as an RPC (Remote Procedure Call)

semaphore invocations are performed locally to the proxy which forwards transactions to the distant service. By using a simple *transport* method, all distant calls put the caller tasks into an active waiting state. In case of access conflicts, the shared service has its own arbitration policy. Then, replies are sent back to the caller at the end of the service execution.

Communication from a distant service to local proxies are performed by using signals similar to interrupt requests, that are managed by local proxies. Suspended tasks may then be resumed by their own schedulers depending on local policies.

Based on this distant service invocation, we can easily imagine and construct a model of a shared distant synchronization service (implemented in software or in hardware), like a semaphore. Then it allows to quickly map the application onto a multiprocessor platform and evaluate the potential acceleration that distribution of computations could potentially permit, as shown in Fig. 3. The gain depends both on the granularity of the functional application partitioning and on its potential parallelism (Amdahl's law).

4.2 CAS Model

The goal is to easily assemble and connect library services in order to build a custom distributed operating system. Each processing node executes a part of the application and its own subset of RTOS services. The application tasks make system calls

to the local RTOS which look at the localization of the asked service. If the service corresponds to a local implementation, the service is classically called. If it is implemented on a distant node, the local proxy transfers the request through a Calling Abstraction Service (CAS). The CAS thus constitutes the elementary structure of the distributed RTOS. An instance of the CAS is present in each RTOS kernel. Its role is to determine the localization of the services, to build a request packet and to route the packet to its destination.

The CAS is an abstract model of inter processor communication channel. It represents a kind of software bus such as those known as ORB (Object Request Broker) in the object-oriented philosophy of CORBA [17]. At our level of modeling, the concept of software bus satisfies all our objectives: modularity support, abstraction of the communication media and implementation transparency. Indeed, the RTOS provides a unique and global programming interface. The application tasks execute and communicate whatever their mapping onto the different processors. So, programming an application onto a heterogeneous MPSoC platform becomes more natural. The homogeneous programming model brings exploration facilities for the application-architecture matching.

In order to determine the localization of the service provider, the CAS builds at boot time a table of the processing nodes and assigns a routing ID to each node. These ID are a modeling artifact and do not necessarily correspond to addresses of memory mapped components. Each packet sent onto the CAS thus contains the following information:

- the node ID of the calling task,
- the node ID of the service provider,
- the type of service,
- the specific service parameters,
- the status of the packet.

The programmer specifies the node ID at the creation of each service. The node ID are then stored in the descriptors of the corresponding services (mutex, semaphore, task...).

At this high level of abstraction, the system model executes following a shared memory paradigm. According to this abstraction strategy, applications execute in a single memory space and all the service descriptors are accessible by all the processing nodes. This first assumption serves as a simplification of the model for the exploration and can be removed during the refinement process. So, once the service created, it can be used by any task on any processor. The first argument of the system call specifies the address pointer of the descriptor in the shared memory. The node ID of the service provider is then determined dynamically by the CAS from the service descriptor. Once again, this modeling approach does not mean any assumption on the refined RTOS model. The service localization information would be for example stored in a static table copied on each PE in the same way as the MPCI³ layer of RTEMS [18].

³Multi Processor Communication Interface.

As depicted in Fig. 3 the CAS structure consists in two types of new building blocks. The internal CAS module serves as a local router for local and distant requests. Inside an RTOS instance, each service is connected to the CAS module for send (port) and for receive (interface) operations. The external CAS module serves as a global router between RTOS instances.

Each transfer is implemented as a blocking call for the application task as explained in Fig. 2. For the shared services, the access policy (as semaphore in Fig. 2) can be customized by the designer. In the current version of the tool, we provide FIFO and HPF policies.

So the exploration of the application mapping becomes easily accessible to the system designer that can program its application on top of a real RTOS API, deploy its application onto multiple processors as easily as a single processor and automatically generate simulation code. The results of the simulations are a functional executable specification of the system and a set of estimated metrics. The estimated metrics depend on the refinement step of the processing elements. We already presented, in the context of reconfigurable computing [16], multiple metrics such as filling ratio of FIFO based communication, FPGA occupation and Gantt charts.

5 A Tool for Specific OS Definition

5.1 Goal of the Tool

Due to the complexity of the exploration process, the HW/SW designer needs tools to apply the OveRSoC methodology. The DOGME (Distributed Operating system Graphical Modeling Environment) software provides an integrated graphical environment to model, simulate, and validate the distribution of OS services onto MPSoC platforms. The goal of the tool is to ease the use of the exploration methodology and to generate automatically a complete executable model of the MPSoC platform (hardware and software). The automation is based on the flexible SystemC model of RTOS described in Sects. 3 and 4. The tool enables to build an application specific RTOS by assembling generic and custom OS service basic blocks using a graphical editor [19]. The application is linked to the resulting OS thanks to a standard POSIX API. Finally, the entire platform is simulated using the SystemC kernel.

5.2 Presentation of the DOGME Tool

The developed tool follows five main design steps represented in Fig. 4:

- **Design of the platform:** the design phase consists in choosing and instantiating toolbox components into the graphical workspace editor in order to assemble the OS services and distribute them onto the SoC processing elements. Figure 5

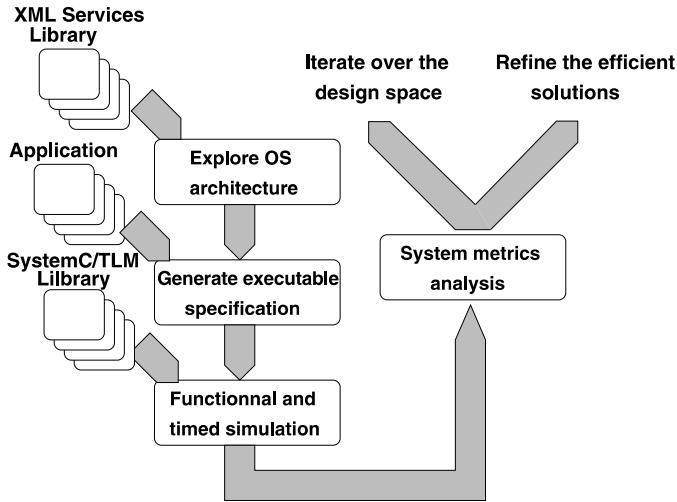


Fig. 4 The DOGME tool brings facilities to manipulate the components of the library. These components model RTOS services for the control of a SoC platform. In the library the services are described both by a SystemC generic source code and an XML exchange file. The designer graphically instantiates the components, then the tool automatically adds debug components for metric evaluation into the specification and generates the code of the corresponding platform. The platform is compiled and linked with the SystemC libraries and simulated with the help of graphical interfaces. The designer can finally evaluate the metrics of the platform and take decisions about exploration or refinement

shows an example of RTOS composition including services like task management, scheduling, semaphore, IRQ controller....

At this step the designer will successively take decisions about:

- mapping of functions into threads,
- hardware/software partitioning,
- instantiation of the required services,
- distribution of the services onto the PEs.

- **SystemC source code generation:** after interconnecting all components and verifying the bindings between services, the structural source code of all the objects that are instantiated into the platform is automatically generated.
- **Compilation and simulation of the platform:** to complete the design of the platform, the parametrized structural SystemC description is combined with the behavioral source code of the components provided by the user. The global SystemC description is compiled and simulated.
- **Analysis of the simulation results:** graphical diagrams are produced to visualize the evolution of the system metrics during the simulated time. This step helps the designer to evaluate the current design quality.

We are currently implementing the DOGME tool as a stand-alone application based on an Eclipse Rich Client Platform [20]. Typical project management functions like importation of platforms or components into the standard library are sup-

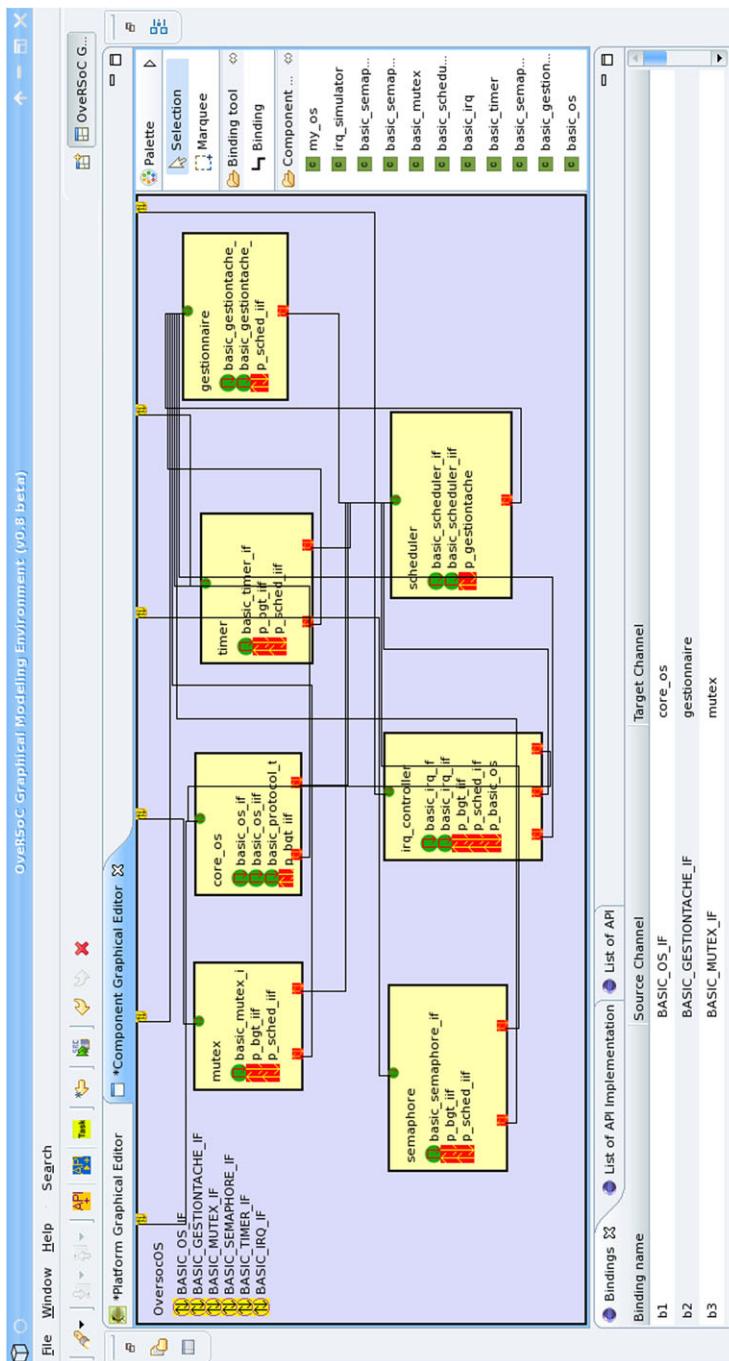


Fig. 5 The DOGME tool represents a distributed RTOS through hierarchical views: the Component Graphical Editor, where the services are organized inside each PE, and the Platform Graphical Editor, where the groups of services are composed according to the number and type of PEs into the SoC platform. Here the Component Graphical Editor is shown. It uses toolbox components to specify and customize the services of a dedicated group. Each service component provides several service functions

ported as well as the creation of new platform models. Re-usability is achieved in the tool by adding the newly created platform to the standard library. All data manipulated by DOGME are loaded and stored using a proprietary XML format dedicated to embedded software modeling.

6 Experiments and Results

The efficiency of the present approach has been evaluated by exploring the architecture of a SoC dedicated to the artificial vision of a mobile robot. The studied image processing application exhibits dynamic properties and specific real-time constraints that need a dedicated embedded control system. The distributed RTOS model was used to explore and define the control strategy of the application tasks.

6.1 A Robotic Vision System

The studied visual system implements a multi-scale approach to extract visual primitives in a camera frame. The visual system provides a local characterization of the keypoints detected in the image flow of an 8-bits gray-scale CCD-camera (382×286 pixels). This local characterization feeds a neural network which associates motor actions with visual information: for example, this neural network can learn the trajectory of the robot as a function of the scene recognition. The studied visual system can be divided into two main modules:

- a multi-scale mechanism for characteristic points research (keypoints detection),
- a mechanism supplying a local feature of each keypoint.

The multi-resolution approach is now well known in the vision community. A wide variety of keypoint detectors based on multi-resolution mechanisms can be found in the literature. Amongst them are the Lindeberg interest point detector [21], the Lowe detector [22] – based on local maxima of the image filtered by Difference of Gaussians (DoGs) – or the Mikolajczyk detector [23] where keypoints are provided by the computation of a 2-D Harris function and fit local maxima of the Laplacian over scales.

The used detector identifies keypoints as sharp corners. More precisely, the keypoints correspond to the local maxima of the gradient magnitude image filtered by DoGs (Fig. 6). Moreover, the detector remains computationally reasonable and exhibits a good stability. It also automatically sorts the keypoint lists of each resolution it studies.

Keypoints are detected in a sampled scale space based on an image pyramid. Pyramids are used in multi-resolution methods to avoid expensive computations due to filtering operations. The algorithm used to construct the pyramid is detailed and evaluated in [24]. The pyramid is based on successive image filtering with two dimensions Gaussian kernels. Keypoints detected on the images are the first N local

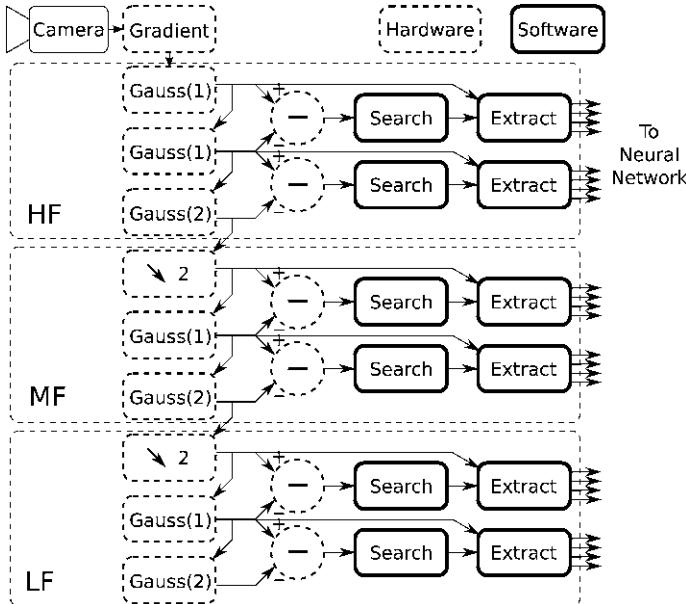


Fig. 6 General view of the application tasks in the multi-resolution approach: High, Medium and Low Frequencies (resolutions)

maxima existing in each DoG image of the pyramid. Thus the keypoint research algorithm sorts the local maxima according to their intensities and extracts the coordinates of the N first ones. The shape of the neighborhood for the research of maxima is a circular region with a radius of 20 pixels around the keypoint.

The number N which parametrizes the algorithm corresponds to a maximal number of detections. Indeed the robot may explore various visual environments (indoor vs. outdoor) and particularly more or less cluttered scenes may be captured, as illustrated in Fig. 7. The full description of our application is out of the scope of this paper and we refer interested readers to [25].

6.2 Deployment Exploration

We specified the application as a set of 30 dynamic communicating tasks. Their degree of parallelism and their execution time vary according to input data, namely the number of interest points in the input camera frames.

In this context we made the profiling of the entire application on an embedded platform, namely an ALTERA DE2 FPGA board, with a Cyclone II FPGA configured as a 32 bit softcore processor (Nios II at 100 MHz). We also built the profile of the OS services used in μ C/OS-II [26] (a deterministic RTOS). For the purpose of the exploration we targeted a multiprocessor architecture (MPSoC). The execution

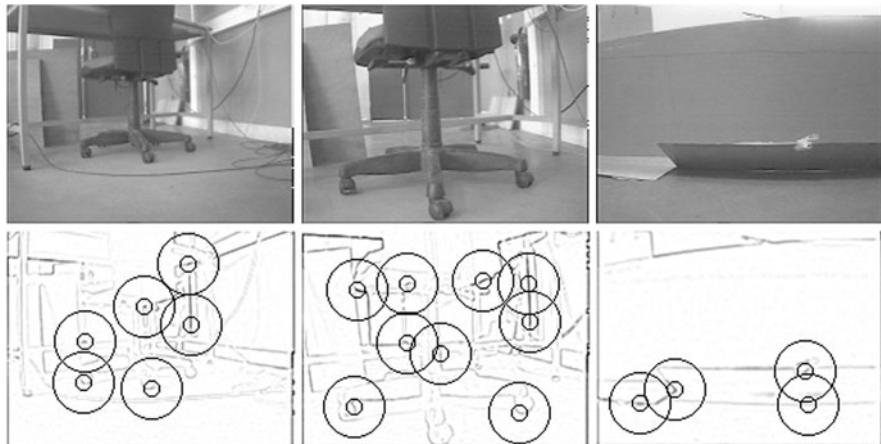


Fig. 7 Keypoints detected on different scenes. The same parameters (N, γ) are used. Keypoints from only one half-octave are shown (middle half-octaves here). *Left*: Average scene (6 keypoints detected). *Center*: Same scene but closer, more cluttered (9 keypoints). *Right*: Less cluttered scene. (4 keypoints)

time at task level was measured and back-annotated into the high-level model, in order to explore and evaluate the architecture dimensioning and the implementation strategies: tasks distribution, service distribution, scheduling algorithms....

To evaluate the efficiency of our modeling approach, we performed two sets of experiments. First we evaluated the model accuracy by comparing the simulated execution time to actual board measurements for multiple implementations. The average application times are depicted in Table 1. The high level simulation error stands within 3–4%.

Then we evaluated the simulation time of the application on top of our RTOS model, and compared it to a purely functional description. We explored and simulated the deployment of the application tasks on architectures composed of 1 to 6 Nios-II processors. Tasks execute and communicate in the same way on board and in simulation, through a single shared memory space protected with hardware semaphores. Table 2 shows the scalability of our model up to 6 processors, and indicates the average simulation overhead for different platform sizes. Simulations were made on a Linux workstation equipped with a 1.66 GHz Intel DualCore with 2 GB of RAM. For a monoprocessor platform the RTOS model does not significantly impact the execution time of the simulation since the overhead is only 9.8% over the purely functional application description. Results indicate that the overhead is around 22% more per simulated RTOS. However the execution times of the sim-

Table 1 Average application execution time in ms

Measured on board	Simulated on RTOS model	Error in %
29 268,5 ms	28 369,6 ms	3–4

Table 2 Simulation overhead of the following number of RTOS for the vision application

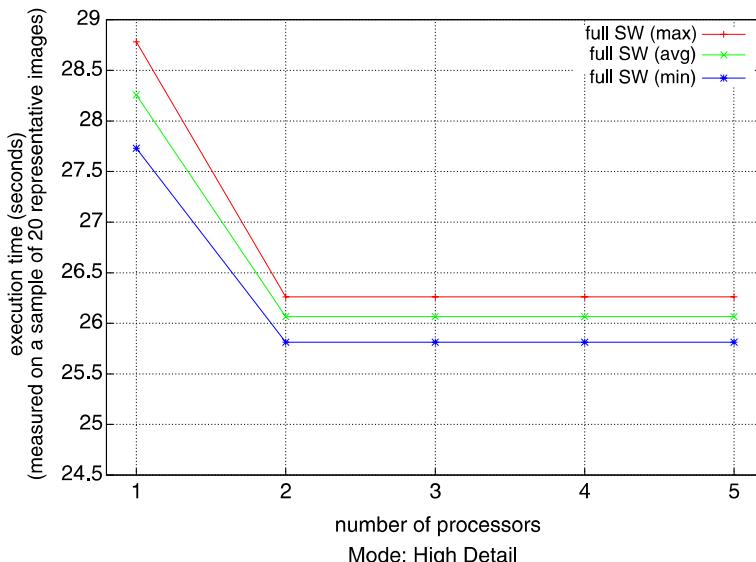
Number of RTOS instances	0	1	2	3	4	5	6
Simulation time (second)	5.5	6.0	7.4	8.6	9.8	11.1	12.8
Overhead (%)	-8.9	0.0	22.5	43.2	63.2	84.2	112

ulations are kept in a reasonable range even for a 6 processor architecture where the number of system calls and preemptions becomes very important.

6.3 Results

We used the modularity advantage of the OS model to evaluate the gain due to parallelism on the execution time of the partitioned application on a multiprocessor SoC, using a shared semaphore module for synchronization. Figure 8 shows the potential gain using multiple processors (from 1 to 5) for the High-detail mode of the robot (all tasks shown in Fig. 6 are executed). The modes of the robot are out of the scope of this paper, but are described in [25]. We can see that the execution time remains steady beyond 2 processors. Indeed, the Gaussian filters represent the critical part of the application and the current task granularity limits the gain of parallelism on the global performance.

According to the results of this first exploration phase we identified critical and regular processes (gradient, Gaussian filters, and DoGs), which represent more than

**Fig. 8** Execution times for different numbers of processors, for a full SW implementation

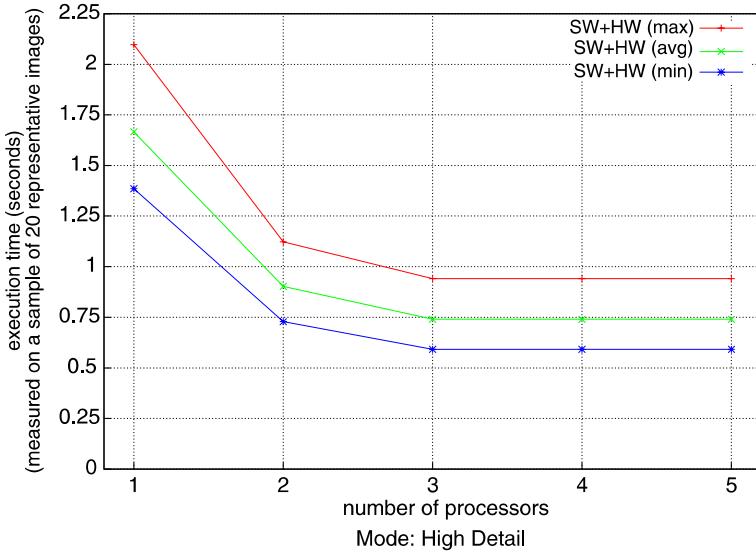


Fig. 9 Application execution times according to the number of processors on our hybrid HW/SW architecture

80% of the total execution time. These processes are good candidates for a static hardware implementation as dedicated accelerators.

These results conduct to a first refinement process. Indeed, in order to evaluate the acceleration we developed a VHDL description of the selected tasks. The temporal characteristics reported by the hardware synthesis tool (Altera's Quartus II in our case) were integrated into the model. The corresponding hardware tasks were modeled as independent and concurrent SystemC threads with the back-annotated execution times. In the model, each hardware block provides its software controller an interruption line for synchronization when data are available.

From the identified Hw/Sw partitioning we led a second set of experiments resulting on the performance evaluation depicted in Fig. 9. By comparing the results of Figs. 8 and 9 we found a speed up factor of 17 thanks to the hardware acceleration. The hardware implementation of the pyramid also makes the parallelization effects more significant for the three processor architecture.

With these results the targeted FPGA will be configured with 3 Nios II processors and the discussed hardware blocks (accelerators and semaphores). Each processor will run an instance of a custom RTOS refined from the OS model. The inter-processor synchronization will be realized through the shared hardware semaphore service. The entire application can thus be implemented on a single chip (SoC) as depicted on Fig. 10.

The software tasks will be partitioned as 6 sets of tasks (one search task and its following extraction and formatting task), i.e. a set for each half-scale (see Fig. 6).

Simulations have shown that running the MF and LF scale sets (4 sets) of tasks on a single processor is faster than running only one HF half-scale group. Thus, in

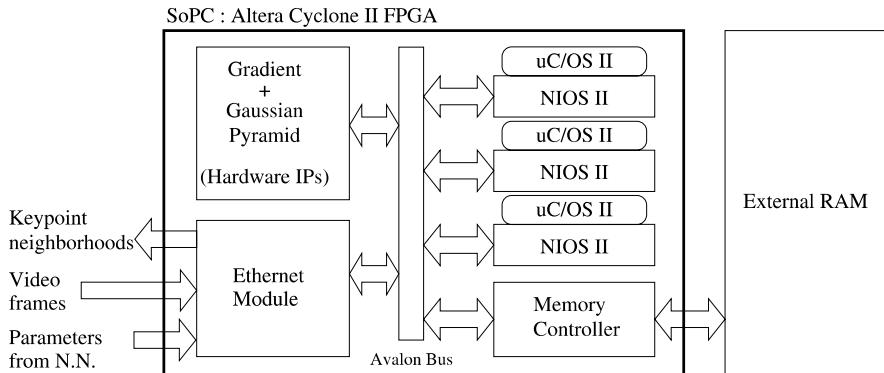


Fig. 10 The SoC architecture obtained after exploration will be embedded into the mobile robot

high-detail mode, the groups will be implemented on the 3 processors as follows: HF groups on two processors, and all the remaining tasks (MF and LF scales) on a third one.

7 Conclusion

We have presented in this paper a high-level exploration method for the mapping of digital signal and image processing applications onto heterogeneous multiprocessor architectures. This method proposes to take the problem of application specific operating system definition into account during the exploration. The RTOS is thus defined and distributed on the platform according to the application requirements. So, the main contribution of this work is to propose an abstract, generic and scalable multiprocessor high level RTOS model. With this model, the designer can evaluate the impact of different service implementations on the real-time performances of the system. The presented methodology and the related DOGME tool are composed of four main design steps: composition of services, distribution onto the architecture, generation of the executable specification, and validation at high-level of the real-time behavior of the application. This methodology was used to define the architecture of a SoC dedicated to artificial vision in the context of mobile robotic platforms. Exploration results show the accuracy of the simulations and the scalability of the model to complex multiprocessor platforms. We are currently applying the method to others embedded applications. With each new application, we are extending an execution time data base for elementary treatments. Perspectives of this work concern on the one hand the integration of new synchronization mechanisms and scheduling algorithms in the service library, and on the other hand the automatic refinement of the high-level model down to the embedded platform.

References

1. OSCI, IEEE 1666TM standard SystemC language. Available at: <http://www.systemc.org>
2. Benkermi I, Benkhelifa A, Chillet D, Pillement S, Prevotet J, Verdier F (2005) System-level modelling for reconfigurable SoCs. In: Design of circuits and integrated systems (DCIS '05)
3. He Z, Mok A, Peng C (2005) Timed RTOS modeling for embedded system design. In: IEEE real time on embedded technology and applications symposium, pp 448–457
4. Desmet D, Verkest D, Man HD (2000) Operating system based software generation for Systems-on-Chip. In: Conference on design automation, pp 396–401
5. Hastono P, Klaus S, Huss S (2004) Real-time operating system services for realistic SystemC simulation models of embedded systems. In: Forum on specification and design languages
6. Posadas H, Adamez J, Villar E, Blasco F, Escuder F (2005) RTOS modeling in SystemC for real-time embedded SW simulation: a POSIX model. Des Autom Embed Syst 10(4):209–227
7. Chevalier J, Benny O, Rondonneau M, Bois G, Aboulhamid EM, Boyer FR (2003) SPACE: a hardware/software SystemC modeling platform including an RTOS. In: Forum on design languages
8. Gauthier L, Yoo S, Jerraya A (2001) Automatic generation and targeting of application specific operating systems and embedded systems software. In: Design automation and test in Europe (DATE), pp 679–685
9. Madsen J, Virk K, Gonzalez M (2003) Abstract RTOS modeling for multiprocessor System-on-Chip. In: Symposium on System-on-Chip, pp 147–150
10. Hwang Y, Abdi S, Gajski D (2008) Cycle-approximate retargetable performance estimation at the transaction level. In: Design automation and test in Europe (DATE)
11. Chung MK, Yang S, Lee SH, Kyung CM (2005) System-level HW/SW co-simulation framework for multiprocessor and multithread SoC. In: Int symposium on VLSI technology systems and applications, pp 177–180
12. Huck E, Miramond B, Verdier F (2007) A modular SystemC RTOS model for embedded services explorations. In: Conference on design and architectures for signal and image processing, Grenoble
13. Donlin A (2004) Transaction level modeling: flows and use models. In: Hardware/software codesign and system synthesis conference (CODES + ISSS), Stockholm, Sweden, pp 75–80
14. Ramamritham K, Stankovic J (1994) Scheduling algorithms and operating systems support for real-time systems. Proc IEEE 82(1):55–67
15. Hsiung PA, Huang CH, Chen YH (2009) Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC. J Embed Comput 3(1):53–62
16. Miramond B, Huck E, Verdier F, Benkhelifa MEA, Granado B, Aichouch M, Prvotet JC, Chillet D, Pillement S, Lefebvre T, Oliva Y (2009) OveRSoC: a framework for the exploration of RTOS for RSoC platforms. Int J Reconfigurable Comput 2009(450607):1–18. Dec. [Online]. Available: <http://publi-etis.ensea.fr/2009/MHVBGAPCPLO09>
17. OMG CORBA specification. Available at: <http://corba-directory.omg.org/>
18. RTEMS, Real-time operating system for multiprocessor systems. Available at: <http://www.rtems.com/>
19. Miramond B, Verdier F, Aichouch M DOGME Distributed operating system graphical modeling environment. Videos available at: <http://oversoc.ensea.fr/oversoc-graphical-modeling-environment-1>
20. Eclipse rich client platform. Available at: <http://eclipsercp.org/>
21. Lindeberg T (1998) Feature detection with automatic scale selection. Int J Comput Vis 30(2):79–116
22. Lowe D (2004) Distinctive image features from scale-invariant keypoints. Int J Comput Vis 2(60):91–110
23. Mikolajczyk K, Schmid C (2004) Scale and affine invariant interest point detectors. Int J Comput Vis 60(1):63–86
24. Crowley J, Riff O, Piater J (2002) Fast computation of characteristic scale using a half-octave pyramid. In: Proceedings of the cognitive vision workshop (Cogvis), Zurich

25. Verdier F, Miramond B, Maillard M, Huck E, Lefebvre T (2008) Using high-level RTOS models for HW/SW embedded architecture exploration: case study on mobile robotic vision. EURASIP J Embed Syst 2008:349465 Special issue on design and architectures for signal image processing
26. Labrosse J (2002) MicroC/OS-II, the real-time kernel. CMP Books, Lawrence

A List Scheduling Heuristic with New Node Priorities and Critical Child Technique for Task Scheduling with Communication Contention

Pengcheng Mu, Jean-François Nezan,
and Mickaël Raulet

Abstract Task scheduling is becoming an important aspect for parallel programming of modern embedded systems. In this chapter, the application to be scheduled is modeled as a Directed Acyclic Graph (DAG), and the architecture targets parallel embedded systems composed of multiple processors interconnected by buses and/or switches. This chapter presents new list scheduling heuristics with communication contention. Furthermore, we define new node priorities (*top level and bottom level*) to sort nodes, and propose an advanced technique named *critical child* to select a processor to execute a node. Experimental results show that the proposed method is effective to reduce the schedule length, and the runtime performance is greatly improved in the cases of medium and high communication. Since the communication cost is increasing from medium to high in modern applications like digital communication and video compression, the proposed method is well-adapted for scheduling these applications over parallel embedded systems.

1 Introduction

The recent evolution of digital communication and video compression applications has dramatically increased complexities of both the algorithm and the embedded system. To face this problem, System-on-Chip (SoC), which embeds several cores (e.g. multicore DSPs) and several hardware accelerators (e.g. Intellectual Properties), becomes the basic element to build complex multiprocessor embedded systems. This kind of system is also known as Multiprocessor System-on-Chip (MPSoC) [13]. Meanwhile, the language used for multiprocessor programming is

P. Mu (✉)

Ministry of Education Key Lab for Intelligent Networks and Network Security, School of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, P.R. China
e-mail: pengchengmu@gmail.com

changing from traditional C to dataflow language such as SystemC¹ and CAL [3]. This kind of program can also be modeled as dataflow graph like the Synchronous Dataflow (SDF) graph [12] and the Boolean Dataflow (BDF) graph [2] during the compilation of multiprocessor programming [11].

Task scheduling of a dataflow description over a multicomponent embedded system is becoming more and more important due to the strict real-time constraints and the growing complexities of applications. It usually consists of assigning tasks to components, specifying the order in which tasks are executed on each component, and specifying the time at which they are executed. However, task scheduling is not straightforward; when performed manually, the result is usually a suboptimal solution. Scheduling on general parallel computer architectures has been actively researched, but task scheduling on parallel embedded systems is different from the general scheduling problem [19]. Communications between components have a very important impact on the scheduling performance and the hardware resources' utilization. Therefore, it is necessary to find new task scheduling methodologies which produce optimal results for programming on parallel embedded systems.

This chapter aims at tackling the task scheduling problem for programming on parallel embedded systems. The program to be scheduled is represented as a task graph modeled by Directed Acyclic Graph (DAG) [14, 19], where nodes represent tasks (i.e. computations) and edges represent data flows (i.e. communications) between tasks. If a program is described by other dataflow graph models (e.g. the SDF graph) which usually consist of iterative computations, these graphs should firstly be transformed to DAGs which will be used for the task scheduling.

The objective of task scheduling is to respectively assign computations and communications to processors and buses (communication links) of the target system in order to get the minimum schedule length (makespan). The scheduling could be done at compile time (namely static scheduling) or at run time (namely dynamic scheduling). Generally, static scheduling is more suitable than dynamic scheduling for deterministic applications in parallel embedded systems by leading to lower code size and higher computation efficiency, where an application is deterministic if the parameters like the data size and the execution time of each part of the application are known at the compile time. These applications usually include the digital communication applications and some image processing applications with fixed parameters. If these parameters vary during the execution of the application, it is necessary to use the dynamic scheduling, but the performance is usually not optimal. This chapter focuses on the static scheduling; all the task scheduling heuristics in the following parts are done at compile time.

The general task scheduling problem is proven to be NP-hard [4, 14]; therefore, many works try to find heuristics to go up to the optimal solution. Early task scheduling heuristics as in [1, 7] do not consider communication costs. As the communication increases in modern applications, many scheduling heuristics [6, 9, 14, 22, 23] have to take into account communications between tasks. Most of these heuristics

¹<http://www.systemc.org/>.

use fully connected topology network in which all communications can be concurrently performed. Different arbitrary processor networks are then used in [5, 8, 15, 18, 21] to accurately describe real parallel systems, and the task scheduling takes into account communication contentions on communication links.

Most of the above heuristics are based on the approach of list scheduling. Some basic techniques are given in [16] for list scheduling with communication contention. This chapter will provide more advanced techniques. Firstly, three new groups of node priorities will be defined and used to sort nodes in addition to the two existing groups; secondly, a technique of using a node's *critical child* will be proposed to improve the performance of selecting a processor for this node. This chapter will finally combine these two techniques giving efficient runtime performances.

The rest of this chapter is organized as follows: Sect. 2 firstly introduces necessary models and definitions, then the task scheduling problem with communication contention is described in this section. The node levels used for sorting nodes are defined in Sect. 3. Our advanced list scheduling heuristic is proposed in Sect. 4, and its time complexity is analyzed in Sect. 5. Section 6 gives experimental results, and the chapter is concluded in Sect. 7.

2 Models and Definitions

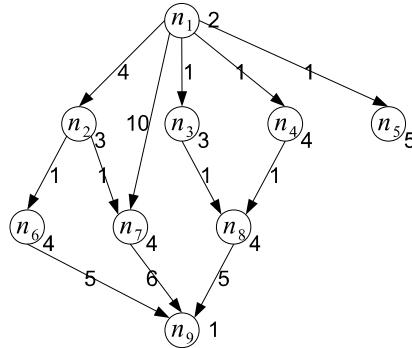
The program to be scheduled is called an algorithm and is modeled as a DAG in this chapter. The multiprocessor target system is called an architecture and is modeled as a topology graph. These two models are detailed as follows.

2.1 DAG Model

A DAG is a directed acyclic graph $G = (V, E, w, c)$ where V is the set of nodes and E is the set of edges. A node represents a computation meaning that a node in the graph can be a subprogram specified in another language (C, assembly language, ...). Between two nodes $n_i, n_j \in V$, e_{ij} denotes the edge from the origin node n_i to the destination node n_j and represents the communication between these two computations. The weight of node n_i (denoted by $w(n_i)$) represents the computation cost; the weight of edge e_{ij} (denoted by $c(e_{ij})$) represents the communication cost. In this model, the set $\{n_x \in V : e_{xi} \in E\}$ of all the direct predecessors of node n_i is denoted by $\text{pred}(n_i)$; the set $\{n_x \in V : e_{ix} \in E\}$ of all the direct successors of node n_i is denoted by $\text{succ}(n_i)$. A node n with $\text{pred}(n) = \emptyset$ is named a source node, where \emptyset is the empty set. Meanwhile, a node n with $\text{succ}(n) = \emptyset$ is named a sink node.

The execution of computations on a processor is sequential, and a computation cannot be divided into several parts. A computation cannot start until all its input communications are finished, and all its output communications cannot start until

Fig. 1 A DAG example with 9 nodes and 12 edges



this computation is finished. In general, communications are sequentially scheduled on a communication link between two processors; however, when a switch is used, communications can be simultaneously scheduled respecting the input and output constraints above.

Figure 1 gives a DAG example which consists of 9 nodes and 12 edges, weights of nodes and edges are also shown in this figure. This DAG has been used in [10] to illustrate performances of different scheduling heuristics, and it will also be used in Sect. 6.1 to show the performance of our method.

2.2 Topology Graph Model

A topology graph $TG = (N, P, D, H, b)$ has been used to model a target system of multiple processors interconnected by communication links and switches [18]. N is the set of vertices, P is a subset of N , $P \subseteq N$, D is the set of directed edges, H is the set of hyperedges, and b is the relative data rate of edge. The union of the two edge sets D and H designates the link set L , $L = D \cup H$, and an element of this set is denoted by l , $l \in L$.

The topology graph is denoted as $TG = (N, P, L, b)$ in this chapter, and directed edges are not used in a target system. A vertex $p \in P$ represents a processor, and a vertex $n \in N, n \notin P$ represents a switch. Since directed edges are not used, a link $l \in L$ is actually a hyperedge h , which is a subset of two or more vertices of N , $h \subseteq N$, $|h| > 1$. A hyperedge connects multiple vertices and represents a half-duplex multidirectional communication link (e.g. a bus). The positive weight $b(l)$, associated with a link $l \in L$, represents its relative data rate.

Differing from the vertex of processor, a switch is a vertex used only for connecting communication links, and no computation can be executed on it. Switches are assumed to be ideal.

Ideal Switch: For a switch s , let l_1, l_2, \dots, l_n be all the communication links connected to s . If two links l_{i_1} and l_{i_2} of them are not used for the moment, a communication can be transferred on l_{i_1} and l_{i_2} without any impact from/to communications on other communication links connected to s .

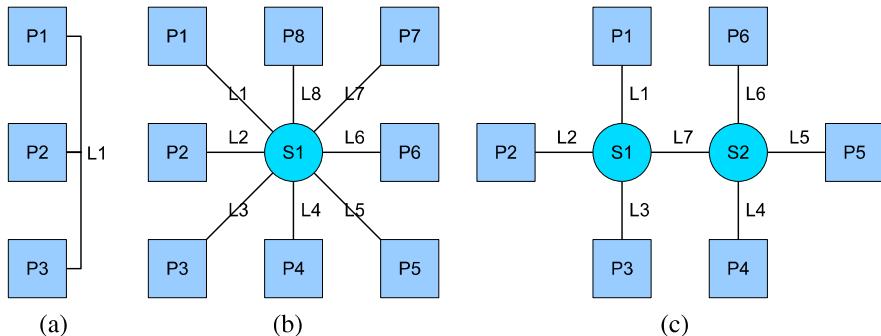


Fig. 2 Architecture examples: (a) Three processors sharing a bus; (b) Eight processors connected to a switch by eight buses; (c) Six processors interconnected by buses and switches

Switches are contention-free according to the above description. Separate communication links connected to the same switch can be used for different communications at the same time; however, a new communication could not begin on a link if this link is busy. Communication links are considered homogeneous in this chapter, but processors can be heterogeneous. Therefore, the relative data rate is assumed to be 1 for all the links, $b(l) = 1, \forall l \in L$, but a computation usually needs different execution durations on different types of processors.

Figure 2(a)–(c) gives three architecture examples: (a) three processors sharing a bus; (b) eight processors connected to a switch by eight buses; and (c) six processors interconnected by buses and switches. The architecture in Fig. 2(c) models the C6474 Evaluation Module² which includes two C6474 multicore DSPs.

A route is used to transfer data from one processor to another in the target system. It is a chain of links connected by switches from the origin processor to the destination processor. For example, $L1 \rightarrow L7 \rightarrow L4$ is a route from $P1$ to $P4$ in Fig. 2(c). Routing is an important aspect of task scheduling. Since the scheduling is static, a route between two processors is also considered as static and is determined at compile time. It is possible to determine routes once and to store them in a table, then the routing during the scheduling becomes looking up the table.

2.3 Task Scheduling with Communication Contention

A schedule of a DAG is the association of a start time and a processor with each node of the DAG. When the communication contention is considered, a schedule also includes allocating communications to links and associating start times on these links with each communication. A communication needs the same duration on each link because of the homogeneity of links. However, a computation usually needs

²<http://focus.ti.com/docs/toolsw/folders/print/tmdxevm6474.html>.

different durations on different processors because processors are heterogeneous. The average duration of a computation on different types of processors is used to represent the node weight.

Following terms describe a schedule S of a DAG $G = (V, E, w, c)$ over a topology graph $TG = (N, P, L, b)$. The start time of a node $n_i \in V$ on a processor $p \in P$ is denoted by $t_s(n_i, p)$; the finish time is given by $t_f(n_i, p) = t_s(n_i, p) + w(n_i, p)$, where $w(n_i, p)$ is the execution duration of n_i on p . A node can be constrained to some processors of the target system. The set of processors on which n_i can be executed is denoted by $Proc(n_i)$, and the processor on which n_i is actually allocated is denoted by $proc(n_i)$. The finish time of a processor is the maximum finish time among all the nodes allocated on this processor, $t_f(p) = \max_{proc(n_i)=p} \{t_f(n_i, proc(n_i))\}$, and the schedule length of S is the maximum finish time among all the processors in the system, $sl(S) = \max_{p \in P} \{t_f(p)\}$.

The communication represented by an edge exists only when its origin node and destination node are not allocated on the same processor. The start time of an existing edge $e_{ij} \in E$ on a link $l \in L$ is denoted by $t_s(e_{ij}, l)$; the finish time of e_{ij} is given by $t_f(e_{ij}, l) = t_s(e_{ij}, l) + c(e_{ij})$. A node (computation) can start on a processor at the time when all the node's input edges (communications) finish. This time is called the Data Ready Time (DRT) and is denoted by $t_{dr}(n_j, p) = \max_{e_{ij} \in E} \{t_f(e_{ij}, l)\}$, where l is a link on which e_{ij} is allocated. The DRT is the earliest time when a node can start. If n_j is a node without input edge, $t_{dr}(n_j, p) = 0, \forall p \in P$.

Node Scheduling Condition: For a node n_i , let $[A, B], A, B \in [0, \infty]$ be an idle time interval on the processor p . n_i can be scheduled on p within $[A, B]$ if

$$\max\{A, t_{dr}(n_i, p)\} + w(n_i, p) \leq B$$

The start time of n_i on p is given by

$$t_s(n_i, p) = \max\{A, t_{dr}(n_i, p)\}$$

Communications are handled in the way of cut-through on a route because of the circuit switching. Therefore, an edge e_{ij} is aligned on all the links of the route $l_{R_1} \rightarrow l_{R_2} \rightarrow \dots \rightarrow l_{R_k}$ with $t_s(e_{ij}, l_{R_1}) = t_s(e_{ij}, l_{R_2}) = \dots = t_s(e_{ij}, l_{R_k})$. The start time and finish time of e_{ij} on all the links of the route are uniformly denoted by $t_s(e_{ij})$ and $t_f(e_{ij})$ with $t_f(e_{ij}) = t_s(e_{ij}) + c(e_{ij})$.

Edge Scheduling Condition: For a DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$, let $l_{R_1} \rightarrow l_{R_2} \rightarrow \dots \rightarrow l_{R_k}$ be a route for an edge $e_{ij} \in E$ and let $[A, B], A, B \in [0, \infty]$ be a common idle time interval on all the links of this route. e_{ij} can be scheduled on this route within $[A, B]$ if

$$\max\{A, t_f(n_i, proc(n_i))\} + c(e_{ij}) \leq B$$

The start time of e_{ij} on this route is given by

$$t_s(e_{ij}) = \max\{A, t_f(n_i, proc(n_i))\}$$

3 Node Levels with Communication Contention

The top level and bottom level are usually used as node priorities which are important for DAG scheduling as shown in [15, 17]. The top level of a node is the length of the longest path from any source node to this node, excluding the weight of this node; the bottom level of a node is the length of the longest path from this node to any sink node, including the weight of this node.

3.1 Existing Node Levels

Two groups of top and bottom levels have been used in task scheduling heuristics, which are respectively named as the computation top/bottom levels (tl_{comp} and bl_{comp}) and the top/bottom levels (tl and bl). Figure 3(a)–(b) illustrates the dependences between nodes for the two existing groups of top levels and bottom levels, where the red dotted nodes and edges are used to recursively define the top levels and bottom levels of n_i .

- **Computation top level and bottom level** (Fig. 3(a))

The computation top level of a node is the length of the longest path from any source node to this node including only the weights of nodes; the computation bottom level of a node is the length of the longest path from this node to any sink node including only the weights of nodes. The weights of edges are not taken into account in the computation top level and bottom level. They are recursively defined as follows:

$$tl_{comp}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max_{n_k \in pred(n_i)} \{tl_{comp}(n_k) + w(n_k)\}, & \text{otherwise} \end{cases}$$

$$bl_{comp}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max_{n_k \in succ(n_i)} \{bl_{comp}(n_k)\} + w(n_i), & \text{otherwise} \end{cases}$$

- **Top level and bottom level** (Fig. 3(b))

The top level and bottom level additionally take into account the weights of edges on the path by contrast with the computation top level and bottom level. They are recursively defined as follows:

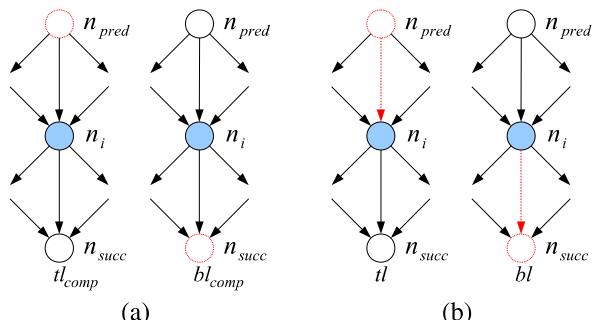


Fig. 3 Two existing groups of node levels: (a) tl_{comp} and bl_{comp} ; (b) tl and bl

$$tl(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max_{n_k \in pred(n_i)} \{tl(n_k) + w(n_k) + c(e_{ki})\}, & \text{otherwise} \end{cases}$$

$$bl(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max_{n_k \in succ(n_i)} \{bl(n_k) + c(e_{ik})\} + w(n_i), & \text{otherwise} \end{cases}$$

3.2 New Node Levels

Besides the two existing groups of node levels, this chapter proposes three new groups. The dependences between nodes for the three new groups of top levels and bottom levels are shown in Fig. 4(a)–(c). The formalized definitions of these top levels and bottom levels are given as follows.

- **Input top level and bottom level** (Fig. 4(a))

The input top level and bottom level take into account weights of nodes on the path as well as weights of all the input edges of a node on the path. They are recursively defined as follows:

$$tl_{in}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max_{n_k \in pred(n_i)} \{tl_{in}(n_k) + w(n_k)\} + \sum_{e_{li} \in E} c(e_{li}), & \text{otherwise} \end{cases}$$

$$bl_{in}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max_{n_k \in succ(n_i)} \{bl_{in}(n_k) + \sum_{e_{lk} \in E} c(e_{lk})\} + w(n_i), & \text{otherwise} \end{cases}$$

- **Output top level and bottom level** (Fig. 4(b))

The output top level and bottom level take into account weights of nodes on the path as well as weights of all the output edges of a node on the path. They are recursively defined as follows:

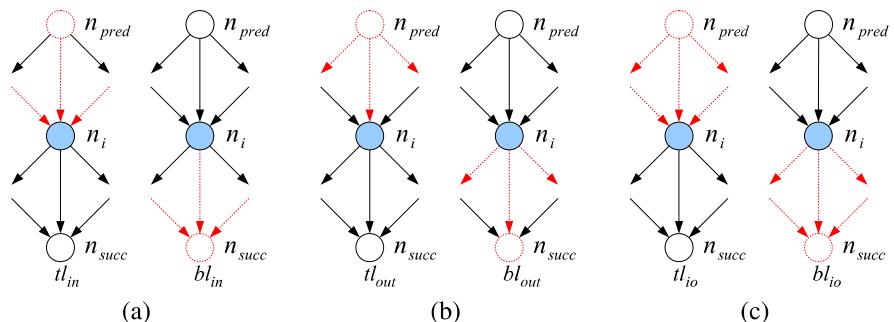


Fig. 4 Three new groups of node levels: (a) tl_{in} and bl_{in} ; (b) tl_{out} and bl_{out} ; (c) tl_{io} and bl_{io}

$$tl_{out}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max_{n_k \in pred(n_i)} \{tl_{out}(n_k) + w(n_k) + \sum_{e_{kl} \in E} c(e_{kl})\}, \\ & \text{otherwise} \end{cases}$$

$$bl_{out}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max_{n_k \in succ(n_i)} \{bl_{out}(n_k)\} + \sum_{e_{il} \in E} c(e_{il}) + w(n_i), \\ & \text{otherwise} \end{cases}$$

- **Input/output top level and bottom level** (Fig. 4(c))

The input/output top level and bottom level take into account weights of nodes on the path as well as weights of all the input and output edges of a node on the path. They are recursively defined as follows:

$$tl_{io}(n_i) = \begin{cases} 0, & \text{if } n_i \text{ is a source node} \\ \max_{n_k \in pred(n_i)} \{tl_{io}(n_k) + w(n_k) + \sum_{e_{kl} \in E} c(e_{kl}) - c(e_{ki})\} \\ & + \sum_{e_{li} \in E} c(e_{li}), \\ & \text{otherwise} \end{cases}$$

$$bl_{io}(n_i) = \begin{cases} w(n_i), & \text{if } n_i \text{ is a sink node} \\ \max_{n_k \in succ(n_i)} \{bl_{io}(n_k) + \sum_{e_{lk} \in E} c(e_{lk}) - c(e_{ik})\} \\ & + \sum_{e_{il} \in E} c(e_{il}) + w(n_i), \\ & \text{otherwise} \end{cases}$$

The three new groups of node levels take into account the communication contention between nodes in comparison with the two existing groups, and Table 1 gives all the five groups of top levels and bottom levels for the DAG given in Fig. 1. Since node levels are usually used as node priorities to sort nodes in the list scheduling heuristic, the three new groups of node levels gives more possibilities to generate different node lists which will be shown in Sect. 4.1.

Table 1 Different node levels for the DAG in Fig. 1

	tl_{comp}	bl_{comp}	tl	bl	tl_{in}	bl_{in}	tl_{out}	bl_{out}	tl_{io}	bl_{io}
n_1	0	11	0	23	0	41	0	35	0	55
n_2	2	8	6	15	6	35	19	16	19	36
n_3	2	8	3	14	3	26	19	14	19	26
n_4	2	9	3	15	3	27	19	15	19	27
n_5	2	5	3	5	3	5	19	5	19	5
n_6	5	5	10	10	10	21	24	10	24	21
n_7	5	5	12	11	20	21	24	11	34	21
n_8	6	5	8	10	9	21	24	10	25	21
n_9	10	1	22	1	40	1	34	1	54	1

Algorithm 1: List_Scheduling(G, TG)

Input: A DAG $G = (V, E, w, c)$ and a topology graph $TG = (N, P, L, b)$
Output: A schedule of G on TG

```

1 NodeList  $\leftarrow$  Sort_Nodes( $V$ );
2 for each  $n \in NodeList$  do
3    $p_{best} \leftarrow$  Select_Processor( $n, P$ );
4   Schedule_Node( $n, p_{best}$ );
5 end

```

4 List Scheduling Heuristic

Algorithm 1 gives the commonly used static list scheduling heuristic. This algorithm is composed of three procedures of `Sort_Nodes()`, `Select_Processor()` and `Schedule_Node()`. This section describes improvements for the first two procedures compared with the classic methods given in [18].

4.1 Sorting Nodes with Five Groups of Node Priorities

Nodes are firstly sorted into a static list by the procedure of `Sort_Nodes()` in the heuristic. Since the order of nodes in the list affects the final schedule result, many different priority schemes have been proposed to sort nodes [9, 15]. Experiments in [17] show that list scheduling with static list sorted by bottom level outperforms other compared contention aware algorithms. Our list scheduling heuristic uses the bottom level and the top level to sort nodes, the procedure of `Sort_Nodes()` sorts nodes into a list of *NodeList* according to the following rule:

Rule for Sorting Nodes: Nodes are sorted by the decreasing order of their bottom levels; if two nodes have equal bottom levels, the one with greater top level is placed before the other; if both the bottom level and the top level are equal, these nodes are sorted randomly.

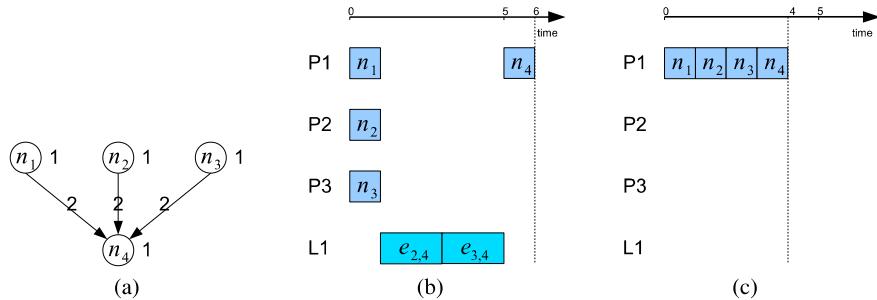
According to the five groups of top levels and bottom levels given in Table 1, the resulting static lists are shown in Table 2, where three different node lists are generated.

4.2 Processor Selection

The classic list scheduling heuristic selects the processor allowing the earliest finish time for a node. This rule probably gives a locally optimized result. In fact, this rule usually gives bad results for the join structure of a DAG especially in the case of great communication cost and communication contention. Figure 5(a) shows such an example; Figure 5(b) gives the schedule result with the classic processor selection

Table 2 Different static node lists according to the top levels and bottom levels in Table 1

Node priority	Static node list	No.
$bl_{comp} \& tl_{comp}$	$n_1, n_4, n_3, n_2, n_8, n_7, n_6, n_5, n_9$	Node list 1
$bl \& tl$	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	Node list 2
$bl_{in} \& tl_{in}$	$n_1, n_2, n_4, n_3, n_7, n_6, n_8, n_5, n_9$	Node list 2
$bl_{out} \& tl_{out}$	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	Node list 3
$bl_{io} \& tl_{io}$	$n_1, n_2, n_4, n_3, n_7, n_8, n_6, n_5, n_9$	Node list 3

**Fig. 5** A join DAG and two different schedule results: (a) A join DAG; (b) Schedule result with the classic processor selection method; (c) The best schedule result

method, which selects a new processor for each one of n_1, n_2 and n_3 to provide the earliest finish time. Therefore, the execution of node n_4 has to wait until the communications from n_2 and n_3 finish, and the schedule length is 6 at last. By contrast, the schedule of all nodes on the same processor is shown in Fig. 5(c) and has a schedule length of 4.

In [9], the critical child of a node is defined as one of its successors that has the smallest difference between the absolute latest possible start time (ALST) and the absolute earliest possible start time (AEST). This critical child is used for scheduling in the case of unbounded number of processors and without communication contention. This chapter uses the concept of critical child for list scheduling in the case of bounded number of processors and with communication contention. The critical child is differently defined as follows.

Critical Child: Given a static node list $NodeList$, the critical child of node n_i is denoted by $cc(n_i)$ and is one of n_i 's successors which emerges firstly in $NodeList$.

According to this definition, the critical child of n_i may be different if $NodeList$ differs. This is the major difference between our critical child and that in [9]. Table 3 shows the critical children according to the different static node lists given in Table 2.

Using critical child makes the processor selection take into account not only the predecessors of a node, but also its most important successor. Our method of using the critical child to select processor is given in Algorithm 2. Since it is possible

Table 3 Critical children according to the different static node lists in Table 2

No.	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9
Node list 1	n_4	n_7	n_8	n_8	null	n_9	n_9	n_9	null
Node list 2	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null
Node list 3	n_2	n_7	n_8	n_8	null	n_9	n_9	n_9	null

Algorithm 2: Select_Processor(n_i, P)

Input: A node $n_i \in V$ and the set P of all processors
Output: The best processor p_{best} for the input node n_i

```

1 Choose the critical child  $cc(n_i)$ ;
2  $BestFinishTime \leftarrow \infty$ ;
3 for each  $p \in Proc(n_i)$  do
4    $FinishTime \leftarrow Schedule\_Node(n_i, p)$ ;
5    $MinFinishTime \leftarrow \infty$ ;
6   if  $cc(n_i) \neq null$  then
7     for each  $p' \in Proc(cc(n_i))$  do
8        $FinishTime \leftarrow Schedule\_Node(cc(n_i), p')$ ;
9       if  $FinishTime < MinFinishTime$  then
10          $MinFinishTime \leftarrow FinishTime$ ;
11       end
12     Unschedule the input edges of  $cc(n_i)$ ;
13     Unschedule  $cc(n_i)$  from  $p'$ ;
14   end
15 else
16    $MinFinishTime \leftarrow FinishTime$ ;
17 end
18 if  $MinFinishTime < BestFinishTime$  then
19    $BestFinishTime \leftarrow MinFinishTime$ ;
20    $p_{best} \leftarrow p$ ;
21 end
22 Unschedule the input edges of  $n_i$ ;
23 Unschedule  $n_i$  from  $P$ ;
24 end

```

that $cc(n_i)$ is not a free node with all its predecessors scheduled during the processor selection for n_i , the scheduling of $cc(n_i)$ only takes into account its scheduled predecessors in the procedure of `Select_Processor()` for n_i .

4.3 Node and Edge Scheduling

The method of scheduling a node n_i onto a processor p is given in Algorithm 3, and Algorithm 4 gives the method for edge scheduling. Since an edge e_{ij} is scheduled only when its origin node n_i has been scheduled, the scheduling of this edge addi-

Algorithm 3: Schedule_Node(n_i, p)

Input: $n_i \in V$ and a processor $p \in P$
Output: The finish time of n_i on p

```

1 for each  $n_l \in pred(n_i)$ ,  $proc(n_l) \neq p$  do
2   | Schedule_Edge( $e_{li}, p$ );
3 end
4 Calculate the DRT of node  $n_i$ ;
5 Find the earliest idle time interval for node  $n_i$  on processor  $p$  respecting the node scheduling
  condition;
6 Calculate the finish time of  $n_i$  on  $p$ ;
```

Algorithm 4: Schedule_Edge(e_{ij}, p)

Input: $e_{ij} \in E$ and a processor $p \in P$ on which the node n_j is to be scheduled
Output: None

```

1 if  $n_i$  is scheduled then
2   | if  $proc(n_i) \neq p$  then
3     | | Determine the route  $R$  from  $proc(n_i)$  to  $p$ ;
4     | | Find the earliest common idle time interval on all the links of  $R$  respecting the
      | | edge scheduling condition;
5   | end
6 end
```

tionally needs to know the processor p on which the destination node n_j of e_{ij} will be scheduled.

5 Analysis of Time Complexity

In [16], the time complexity of the classic list scheduling heuristic is given as $O(PE^2O(routing) + V^2)$, where P , V and E are the number of processors, the number of nodes and the number of edges, respectively. The time complexity increases by a factor of P with the critical child, but the combination with different node priorities does not increase the time complexity. The time complexity of our proposed list scheduling heuristic is analyzed as follows.

The route can be determined (calculated or looked up) in $O(1)$ time in the procedure Schedule_Edge() for static routing. If the route contains $O(routing)$ links, it takes $O(E O(routing))$ time to find the earliest common idle time interval on all the links of the route. Thus, the complexity of Schedule_Edge() is $O(E O(routing))$.

The procedure Schedule_Node() firstly needs to use $O(\frac{E}{V})$ times of the procedure Schedule_Edge() on average, then it takes $O(\frac{E}{V})$ time to calculate the DRT, and it takes $O(\frac{V}{P})$ time to find an idle time interval for a node on average. At last, it takes $O(1)$ time to calculate the finish time of the node. Therefore, the

total complexity of the procedure `Schedule_Node()` is $O(\frac{E^2 O(\text{routing})}{V} + \frac{V}{P})$ on average.

As to the procedure `Select_Processor()`, it firstly takes $O(V)$ time to find the critical child $cc(n_i)$. When $cc(n_i)$ is found, given a specific processor p , it needs at most $O(P)$ times of `Schedule_Node()` for the scheduling of n_i and $cc(n_i)$. Hence, the complexity in the outer for-loop is $O(P(\frac{E^2 O(\text{routing})}{V} + \frac{V}{P}))$, and the total complexity of `Select_Processor()` is $O(P(\frac{PE^2 O(\text{routing})}{V} + V))$.

In Algorithm 1, sorting nodes has the complexity of $O(V \log V + E)$ (computing node levels in $O(V + E) +$ sorting in $O(V \log V)$). Our new definitions of top level and bottom level do not change the complexity of computing node levels; therefore, the complexity of sorting nodes is always $O(V \log V + E)$. Since the procedure `Select_Processor()` is more complicated than the procedure `Schedule_Node()`, the complexity in the for-loop is equal to that of the procedure `Select_Processor()`. Finally, the total complexity of the proposed list scheduling heuristic is given by $O(P(PE^2 O(\text{routing}) + V^2))$.

6 Experimental Results

This section gives experimental results of the proposed list scheduling heuristic in comparison with the classic one given in [18]. The architecture in Figs. 2(a) and 2(b) are used for the comparison in Sects. 6.1 and 6.2, respectively.

6.1 Comparison with an Example

The DAG given in Fig. 1 is used in this section to show that using the critical child and different node priorities improves the schedule performance. Table 1 has given all the five groups of top levels and bottom levels; the resulting static lists are given in Table 2; and the critical child for each node is obtained according to these static lists in Table 3.

Figure 6 gives the schedule result of the classic heuristic with nodes sorted by bl and tl , where the schedule length is 21.

Using the critical child technique with the three different node lists in Table 2 gives different schedule results. The schedule result for the node list (Node list 1) sorted by bl_{comp} & tl_{comp} is shown in Fig. 7(a) with the schedule length of 18. Since the node list (Node list 2) sorted by bl & tl is same as that sorted by bl_{in} & tl_{in} , the same schedule result is obtained and shown in Fig. 7(b) with the schedule length of 18. Figure 7(c) gives the schedule result for the same node list (Node list 3) sorted by bl_{out} & tl_{out} and by bl_{io} & tl_{io} . The schedule length is 17 and is better than the two former schedule lengths of 18. All the three schedule results with the critical child technique are better than that of the classic heuristic. In addition, though this example shows that the node list sorted by bl_{out} & tl_{out} and by bl_{io} & tl_{io} outperforms

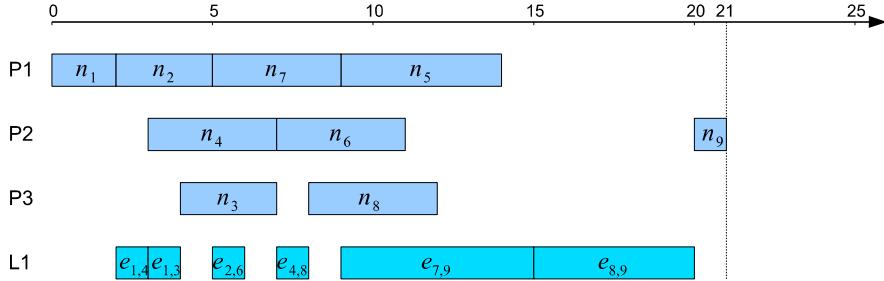


Fig. 6 Schedule result of classic heuristic for the DAG in Fig. 1 and the architecture in Fig. 2(a)

others, it is not usually true when changing the DAG. Therefore, all the five groups of node priorities should be used for scheduling a DAG, and the best result is chosen among them at last.

6.2 Comparison with Random DAGs

Random graphs are commonly used to compare scheduling algorithms in order to get statistical results which are more persuasive than the result for a particular graph. We implement a graph generator based on SDF³ [20] to generate random SDF graphs except that the SDF graphs are constrained to be DAGs (same rate between two operations, no cycles). A random DAG is described in five aspects: (1) the number of nodes, (2) the average in degree, (3) the average out degree, (4) the random weights of nodes, and (5) the random weights of edges. The average in degree and out degree are assumed to be same in this chapter. The weights of nodes vary randomly from w_{\min} to w_{\max} . The communication to computation ratio (*CCR*) is used to generate random weights of edges. The *CCR* is defined as the average weight of edges divided by the average weight of nodes in this chapter, that is $CCR = \frac{1}{|E|} \sum_{e \in E} c(e) / \frac{1}{|V|} \sum_{n \in V} w(n)$. The weights of edges are generated randomly from $w_{\min} \times CCR$ to $w_{\max} \times CCR$. The *CCR*'s typical values of 0.1, 1 and 10 represent the low, medium and high communication situations, respectively.

A list scheduling heuristic can use all the five groups of node priorities to get different results. We combine the five groups of node priorities with a heuristic and choose the best result; the whole process is called a combined heuristic. The schedule length of the combined heuristic is compared to the classic list scheduling heuristic with nodes sorted by *bl* & *tl*. The acceleration factor (*acc*) is defined as $acc = sl_{classic} / sl_{compared}$ to show the speed-up of the compared heuristic.

Figure 8 gives the average *acc* of the combined heuristic with critical child. Weights of nodes are generated randomly from 100 to 1000, and 1000 random DAGs for each group are tested to obtain the statistical results.

The average *acc* increases as *CCR* increases, and the schedule result is sped up by using the combined heuristic in the cases of $CCR = 1$ and $CCR = 10$. The average

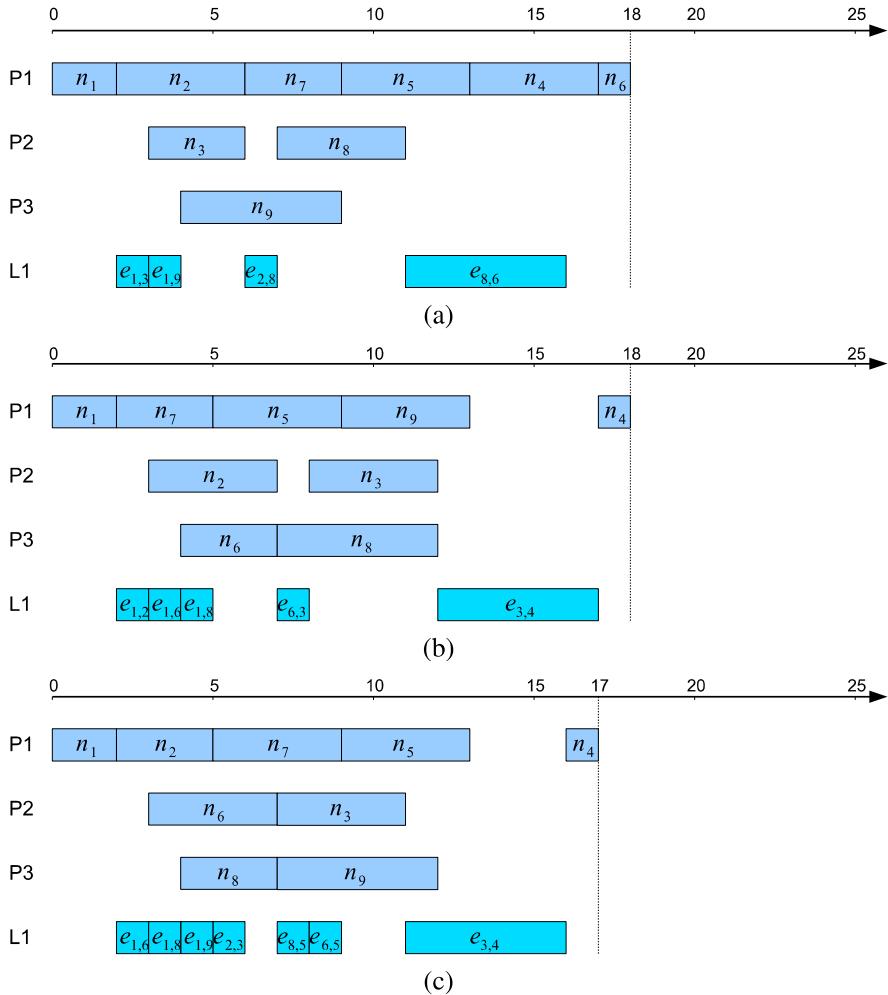


Fig. 7 Schedule results with critical child for the DAG in Fig. 1 and the architecture in Fig. 2: (a) Node list 1, schedule length = 18; (b) Node list 2, schedule length = 18; (c) Node list 3, schedule length = 17

acc also increases as the number of average in/out degree increases when $CCR = 10$. The reason for this phenomenon is that the critical child technique helps to select better processors for nodes with multiple predecessors. The greater the in/out degree is, the better the critical child works. However, when $CCR \leq 1$, the communication is low in comparison with the computation, the communication contention is hence weak even if the in/out degree increases. Since the techniques proposed in this chapter mainly aims at the communication contention, the improvement in the cases of $CCR = 0.1$ and $CCR = 1$ is not as great as that in the case of $CCR = 10$. The modern applications like digital communication and video compression usually

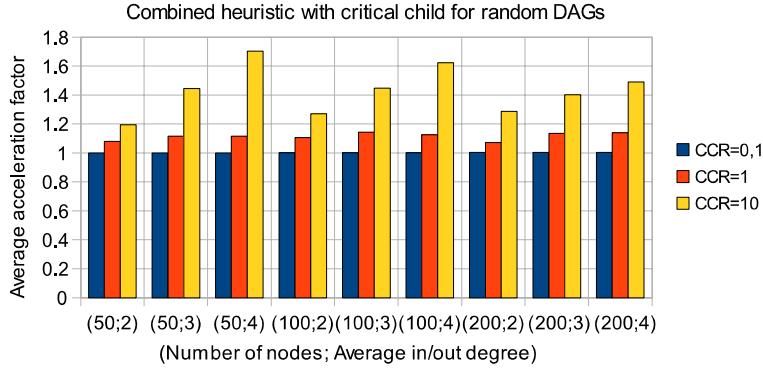


Fig. 8 Average *acc* of combined heuristic with critical child for randomly generated DAGs

have $CCR > 1$; therefore, our method is useful for scheduling these applications on parallel embedded systems.

6.3 Time Complexity

Figure 9(a)–(b) shows the time used to schedule different sizes of DAGs on architectures with different numbers of processors by our combined heuristic. All the DAGs have the average in/out degree of 4, and all the processors are connected to a switch. As shown in Fig. 9(a) and Fig. 9(b), the time increases with the square of V and also with the square of P . We run our heuristic on a Pentium Dual-Core PC at 2.4 GHz, and it takes about 3 minutes to schedule a DAG with 500 nodes on an architecture of 16 processors.

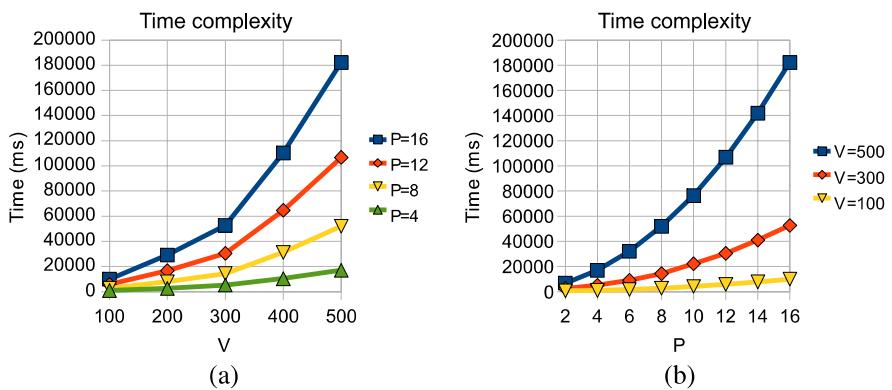


Fig. 9 Time complexity of the proposed heuristic: (a) Time increases with the square of V ; (b) Time increases with the square of P

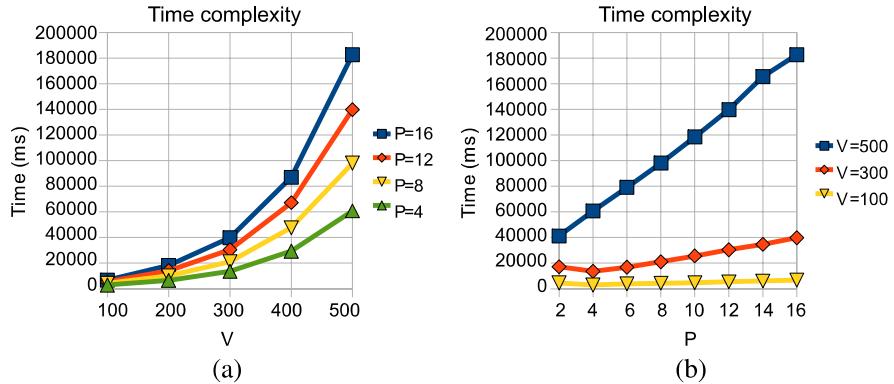


Fig. 10 Time complexity of the classic heuristic: (a) Time increases with the square of V ; (b) Time increases linearly with P

By contrast, Fig. 10(a)–(b) shows the time used by the classic heuristic. As shown in Fig. 10(a) and Fig. 10(b), the time increases with the square of V , but it increases linearly with P .

In fact, a complicated embedded application usually has less than 500 nodes in models of coarse and medium grain, and P is usually much smaller than V and E in a parallel embedded system. Therefore, the increase of time complexity is reasonable and acceptable for rapid prototyping.

7 Conclusions and Prospects

This chapter presents three new groups of node priorities (top level and bottom level) and a technique of critical child for list scheduling with communication contention. The new node priorities take into account the communication contention and are used to sort nodes in order to get different node lists. The technique of critical child helps to select a better processor for a node. The combination of different node lists and the critical child technique gives different schedule results for a specific DAG, and the best one is chosen at last. Experimental results show that using different node lists and the critical child technique is effective to shorten the schedule length for most of the randomly generated DAGs in the cases of medium and high communication.

The architecture model in this chapter is relatively simple especially in the aspect of routing; therefore, a more detailed architecture model will be researched in the future work to describe actual parallel embedded systems with multiple processors and heterogeneous communication links. However, the techniques proposed in this chapter will always be useful even if the architecture model becomes complex. Since the communication cost is increasing from medium to high in modern digital communication and video compression applications, our method will work well for scheduling these applications on parallel embedded systems.

References

1. Adam TL, Chandy KM, Dickson JR (1974) A comparison of list schedules for parallel processing systems. *Commun ACM* 17(12):685–690. <http://doi.acm.org/10.1145/361604.361619>
2. Buck JT (1993) Scheduling dynamic dataflow graphs with bounded memory using the token flow model. Ph.D. thesis, EECS Department, University of California, Berkeley
3. Eker J, Janneck JW (2003) CAL Language Report. Tech. Rep. ERL Technical Memo UCB/ERL M03/48, University of California at Berkeley
4. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York
5. Grandpierre T, Lavarenne C, Sorel Y (1999) Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In: Proceedings of 7th international workshop on hardware/software co-design, CODES '99, Rome, Italy. URL: <http://www-rocq.inria.fr/syndex/pub/codes99/codes99.pdf>
6. Hwang JJ, Chow YC, Anger FD, Lee CY (1989) Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J Comput* 18(2):244–257. <http://dx.doi.org/10.1137/0218016>
7. Kasahara H, Narita S (1984) Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans Comput* 33(11):1023–1029. <http://dx.doi.org/10.1109/TC.1984.1676376>
8. Kwok YK, Ahmad I (1995) Bubble scheduling: a quasi dynamic algorithm for static allocation of tasks to parallel architectures. In: SPDP '95: proceedings of the 7th IEEE symposium on parallel and distributed processing. IEEE Computer Society, Washington, p 36
9. Kwok YK, Ahmad I (1996) Dynamic critical-path scheduling: an effective technique for allocating task graphs onto multiprocessors. *IEEE Trans Parallel Distrib Syst* 7(5):506–521. [10.1109/71.503776](http://dx.doi.org/10.1109/71.503776)
10. Kwok YK, Ahmad I (1999) Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput Surv* 31(4):406–471. <http://citeseer.ist.psu.edu/kwok99static.html>
11. Lee E, Parks T (1995) Dataflow process networks. *Proc IEEE* 83(5):773–801. [10.1109/5.381846](http://dx.doi.org/10.1109/5.381846)
12. Lee EA, Messerschmitt DG (1987) Synchronous data flow. *Proc IEEE* 75(9):1235–1245
13. Martin G (2006) Overview of the MPSoC design challenge. In: Proceedings of the 43rd annual conference on design automation, San Francisco, CA, USA
14. Sarkar V (1989) Partitioning and scheduling parallel programs for multiprocessors. MIT Press, Cambridge
15. Sih G, Lee E (1993) A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Trans Parallel Distrib Syst* 4:175–187. [10.1109/71.207593](http://dx.doi.org/10.1109/71.207593)
16. Sinnen O (2007) Task scheduling for parallel systems. Wiley, New York
17. Sinnen O, Sousa L (2004) List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Comput* 30(1):81–101
18. Sinnen O, Sousa L (2005) Communication contention in task scheduling. *IEEE Trans Parallel Distrib Syst* 16(6):503–515
19. Sriram S, Bhattacharyya SS (2000) Embedded multiprocessors – scheduling and synchronization. Dekker, New York
20. Stuijk S, Geilen M, Basten T (2006) SDF³: SDF for free. In: Application of concurrency to system design, 6th international conference, ACSD 2006, proceedings. IEEE Comput Soc, Los Alamitos, pp 276–278. [10.1109/ACSD.2006.23](http://dx.doi.org/10.1109/ACSD.2006.23). URL: <http://www.es.ele.tue.nl/sdf3>
21. Tang X, Li K, Padua D (2009) Communication contention in APN list scheduling algorithm. *Sci China Ser F* 52(1):59–69

22. Wu MY, Gajski D (1990) Hypertool: a programming aid for message-passing systems. IEEE Trans Parallel Distrib Syst 1(3):330–343. URL: <http://citeseer.ist.psu.edu/wu90hypertool.html>
23. Yang T, Gerasoulis A (1994) DSC: scheduling parallel tasks on an unbounded number of processors. IEEE Trans Parallel Distrib Syst 5(9):951–967. [10.1109/71.308533](https://doi.org/10.1109/71.308533)

Multiprocessor Scheduling of Dataflow Programs within the Reconfigurable Video Coding Framework

Jani Boutellier, Christophe Lucarz,
Victor Martin Gomez, Marco Mattavelli,
and Olli Silvén

Abstract The new Reconfigurable Video Coding (RVC) standard of MPEG marks a transition in the way video coding algorithms are specified. Imperative and monolithic reference software is replaced by a collection of interconnected, concurrent functional units (FUs) that are specified with the actor-oriented CAL language. Different connections between the FUs lead to different decoders: all previous standards (MPEG-2 MP, MPEG-4 SP, AVC, SVC, ...) can be built with RVC FUs. The RVC standard does not specify a schedule or scheduling heuristic for running the decoder implementations consisting of FUs. Previous work has shown a way to produce efficient quasi-static schedules for CAL actor networks. This paper discusses the mapping of RVC FUs to multiprocessor systems, utilizing quasi-static scheduling. A design space exploration tool has been developed, that maps the FUs to a multiprocessor system in order to maximize the decoder throughput. Depending on the inter-processor communication cost, the tool points out different mappings of FUs to processing elements.

1 Introduction

The effort of designing the Reconfigurable Video Coding (RVC) standard [1] is motivated by the intent to describe already existing video coding standards with a set of common atomic building blocks (*e.g.*, IDCT). Under RVC, existing video coding standards are described as specific configurations of these atomic blocks, also known as functional units (FUs). This greatly simplifies the task of designing future multi-standard video decoding applications and devices by allowing software and hardware reuse across standards.

J. Boutellier (✉)

Computer Science and Engineering Laboratory, University of Oulu, Oulu, Finland
e-mail: jani.boutellier@ee.oulu.fi

The functional units are described in RVC with a dataflow/actor object-oriented language named CAL that allows concise description of signal processing algorithms. For this reason, CAL has been chosen as the language for the reference software of the standard. With CAL, decoders are described as a set of atomic blocks in a way that exposes parallelism between the computations. However, the abstract and high-level CAL models require a systematic implementation methodology and tools to implement these CAL models into real systems. One of the implementation problems is the assignment of RVC FUs to the processing elements (PEs) available in the underlying system, as well as generating efficient schedules for the FU actions.

In previous work [5], a methodology has been designed for transforming RVC CAL networks into a set of homogeneous synchronous dataflow (HSDF) [12] graphs that enable efficient quasi-static scheduling. The work presented in this paper takes as an input the set of HSDF graphs produced by the previous work, and tries to find an optimal mapping of RVC FUs to the PEs in the system. This paper describes a design space exploration (DSE) tool and as an example, shows the mapping of the RVC MPEG-4 Simple Profile (SP) decoder to a homogeneous multiprocessor system. The number of processors is not limited by the approach, but the combinatorial explosion forces us to limit the search space. Finally, the results provided by the DSE tool are discussed. The issue of mapping RVC decoders to a multicore platform has previously been discussed in [3].

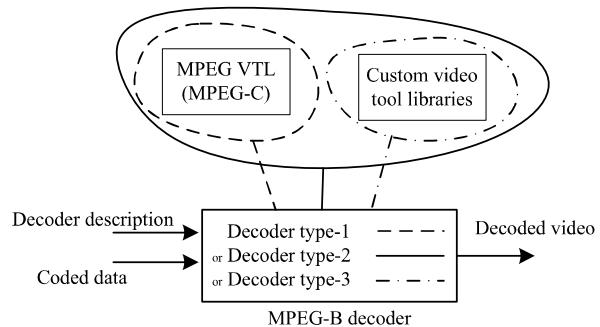
This paper is an extended version of the one presented at the DASIP 2009 conference [6]. The main differences of this extended version are that previously the case study only encompassed six functional units, whereas it now encompasses ten. In the previous paper version, the interprocessor communication overhead was computed assuming that the shared memory has equally many access ports as there are processors. In this paper version the communication takes place through one or two ports and the communication model is exact. Finally, the best solution is now chosen based on weighted results. In the conference paper, the results were regarded with equal weights on every graph. These changes are explained more precisely later in the article.

The paper is organized as follows. Section 2 explains the main concepts in the Reconfigurable Video Coding framework. Section 3 explains the scheduling approach used. Section 4 illustrates the methodology on a real-life application (MPEG-4 Simple Profile decoder). Section 5 concludes the paper.

2 Concepts of the Reconfigurable Video Coding Framework

The MPEG RVC framework aims to offer a more flexible and fast path to innovation of future video coding standards. The RVC framework also provides a high level specification formalism that establishes a starting point model for direct software and hardware synthesis. Moreover, the RVC framework intends to overcome the lack of interoperability between various video codecs that are deployed into the market. Unlike previous standards, RVC does not itself define a new codec. Instead,

Fig. 1 The RVC decoder can be instantiated from standard or custom FUs



it provides a framework to allow content providers to define a multitude of different codecs, by combining together FUs from the Video Tool Library (VTL). Such a possibility clearly simplifies the task of designing future multi-standard video decoding applications and devices by allowing software and hardware reuse across video standards.

The main strength of RVC is that unlike the traditional video coding standards, where decoders used to be rigidly specified, a description of the decoder is associated to the encoded data, enabling a reconfiguration and instantiation of the appropriate decoder at the video data receiver. Figure 1 illustrates how the decoders can be constructed within the RVC framework. The MPEG RVC framework defines two standards: MPEG-B, which defines the overall framework as well as the standard languages that are used to describe different components of the framework, and MPEG-C, which defines the library of video coding tools employed in existing MPEG standards [14].

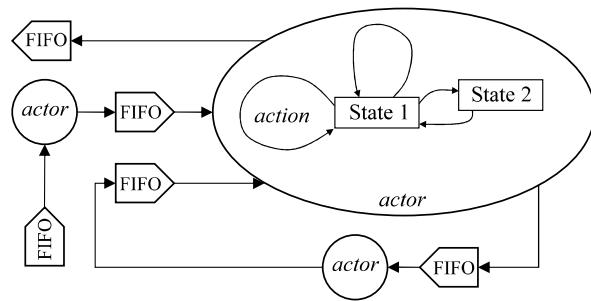
MPEG VTL is normatively specified using RVC-CAL. An appropriate level of granularity for blocks within the standard library is important, to enable efficient reuse within the RVC framework. If the library is too coarse, modules will be too large to allow reuse between different codecs. On the other hand, if the granularity is too fine, the number of modules in the library will be too large for an efficient and practical reconfiguration process, and may obscure the desired high-level description and modeling of the RVC decoder. Prior to RVC, reuse of components across applications has been done, *e.g.*, in multi-mode systems [15].

Besides the specification of RVC video decoders, the CAL language has also been used to describe an OFDM inner receiver [16].

2.1 The CAL Language

CAL is a dataflow and actor oriented language that has been recently specified as a subproject of the Ptolemy project [9] at the University of California, Berkeley. The final CAL language specification has been released in December 2003 [8]. CAL models different algorithms by using a set of interconnected dataflow components called actors (see Fig. 2).

Fig. 2 An arbitrary CAL actor network



An actor is a modular component that encapsulates its own state. The state of any actor is not sharable with other actors. Thus, an actor cannot modify the state of another actor. Interactions between actors are only allowed through input and output ports. The behavior of an actor is defined in terms of a set of actions. The operations an action can perform are to (1) consume input tokens, to (2) modify internal state and to (3) produce output tokens. The actors are connected to each other through FIFO channels and the connection network is specified with XML Dataflow (XDF). The action executions within one actor are purely sequential, whereas at the network level, the actors can work concurrently. CAL allows also hierarchical system design: each actor can contain a network of actors.

A CAL actor can also be interpreted as an Extended Finite State Machine (EFSM) [7], and the actions as EFSM state transitions (see Fig. 3). The state-space of an EFSM is much greater than that of a regular FSM, because EFSMs (CAL actors) may contain variables. The state transitions in the CAL actors cannot take place freely: four types of control mechanisms [13] define which action is going to execute next. (1) Availability of input tokens, (2) value of input tokens, (3) actor state and (4) action priorities. These control mechanisms increase the expressiveness of the CAL language, but unfortunately produce an overhead at run-time: actors are constantly checking the status of control mechanisms. The quasi-static scheduling approach used in this work [5] minimizes this run-time overhead by analyzing the

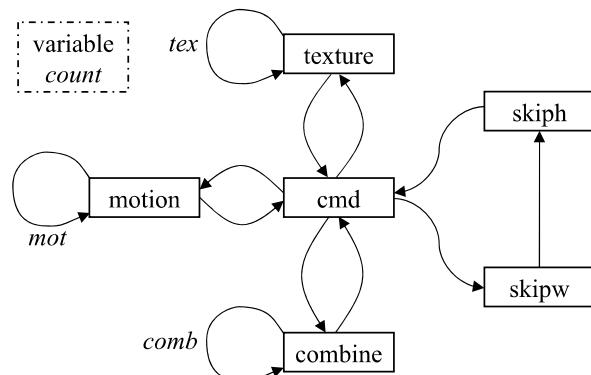


Fig. 3 The CAL actor *add* interpreted as an extended finite state machine. The EFSM state represents a CAL actor state and the EFSM state transition corresponds a CAL action

CAL network at design-time, leaving only the necessary control mechanisms active at run-time.

3 The Scheduling Approach

Nowadays, a significant amount of video decoding takes place on mobile devices that have strict power and performance constraints. Thus, also the scheduling used in mobile video decoders must be done efficiently.

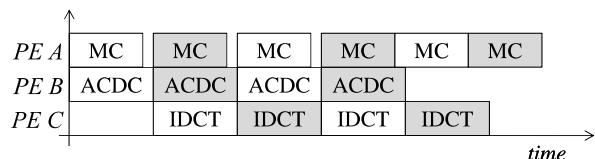
In quasi-static scheduling [11], most of the scheduling effort is done off-line and only some infrequent data-dependent scheduling decisions are left to run-time. The off-line determined schedule parts are collected to a repository that is used by the run-time system, which selects entries from the repository on demand and appends them to the ongoing program execution. This approach limits the number of run-time scheduling decisions and improves the efficiency of the system.

Quasi-static scheduling fits very well to the context of video decoding. The video decoding process of hybrid block-based decoders (such as MPEG-4 SP) consists of the decoding of *macroblocks* that consist of several *blocks*. For example, in MPEG-4 SP the blocks are of size 8×8 pixels, and six blocks form a macroblock that produces the information that can be seen on the screen in a 16×16 pixel area. The decoding process varies block-by-block, so the static schedule pieces in quasi-static scheduling should be of a granularity of one block. Furthermore, quasi-static scheduling assumes that the scheduled tasks have been pre-assigned to processing elements at design time, which also reduces the run-time overhead of scheduling. Quasi-static scheduling for multiprocessor platforms has been extensively discussed in [4].

Figure 4 sketches a quasi-static schedule as described above. The decoding of even-numbered blocks is depicted with white tasks in the Gantt chart, whereas the odd-numbered blocks are gray. Blocks 4 and 5 only require Motion Compensation (MC) for decoding, whereas blocks 0 through 3 require also AC/DC prediction and IDCT. The figure simplifies the real-world computations: in reality each block (MC, AC/DC, IDCT) would consist of hundreds of CAL actions. The figure is simplified so far that it only discriminates the tasks on different PEs and different schedule parts. The detailed action schedules that are not shown, are computed at design time and stored for run-time use. The run-time system then selects the appropriate schedule part for decoding each block.

In our previous work [5], we have explained a procedure to transform RVC CAL networks into homogeneous static synchronous data flow (HSDF) graphs that can

Fig. 4 A quasi-static 3-PE macroblock decoding schedule consisting of 6 parts



be quasi-statically scheduled. The quasi-static scheduling algorithm takes the CAL actors and their interconnecting networks as an input, and produces a set of HSDF graphs as an output. The number of produced HSDF graphs depends on the number of *modes* that the CAL network has; the different decoder modes represent the various decoding approaches of 8×8 pixel blocks.

In each produced HSDF graph, one HSDF vertex represents an instance of a CAL action. For example, if the RVC *add* (see Fig. 3) actor executes the *tex* action 64 times, there will be 64 HSDF vertices representing that action in the HSDF graph. Note that if a CAL actor is active in several different network modes M , the same HSDF vertices will appear in each graph that represents those modes M .

The quasi-statically schedulable RVC MPEG-4 decoder was created from the C code that was produced by the Open RVC-CAL Compiler (ORCC) [10]. The transformation steps to produce quasi-statically schedulable code from unscheduled code have mostly been automated, and work is in progress to complete the automation.

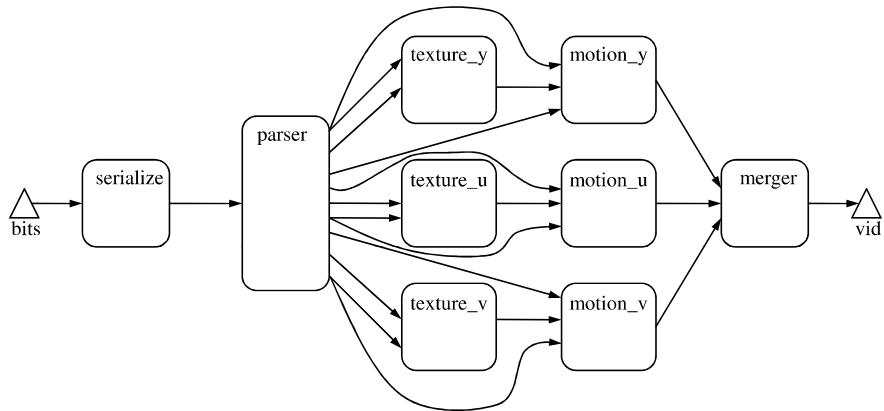
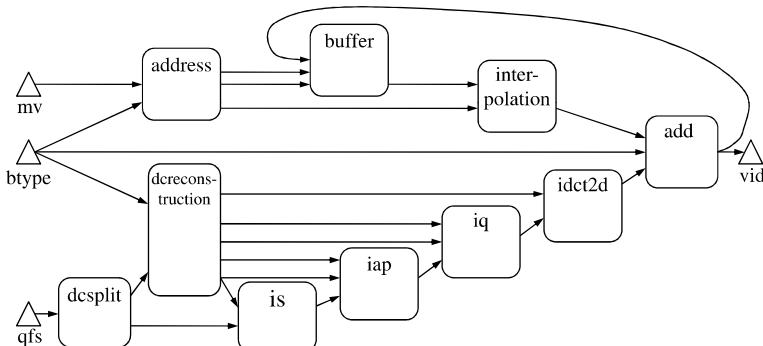
Quasi-static scheduling of the RVC MPEG-4 decoder is very beneficial: the quasi-statically scheduled decoder runs 80% faster than the unscheduled decoder on a single-processor system. Quasi-static scheduling creates long sequences of code that behaves in a predictable fashion, which again enables the compiler to optimize to code way more efficiently than unscheduled code. This is the reason for the dramatic improvement in execution speed.

Besides the code optimization potential, there is an additional reason to use quasi-static scheduling in multiprocessor systems. Quasi-static scheduling assumes that the actions executed have a deterministic execution time. This enables creating the inter-processor communication patterns at system design time, which greatly simplifies the run-time system. In our case study that is explained next, this design-time planning of inter-processor communication is used as well.

4 Case Study: MPEG-4 SP Decoder

The behavior of the MPEG-4 SP decoder (see Fig. 5) is controlled to a great extent with the *btype* token that is created in the *parser* actor and affects the behavior of the *texture* and *motion* networks that do the main decoding effort. For our work, the *btype* token was analyzed and it was discovered that it defines five major operations modes for the *texture* and *motion* networks. The combined *motion* and *texture* networks are depicted in Fig. 6.

With the information about the different decoder modes induced by the *btype* token, the MPEG-4 SP decoder was profiled extensively to get the number of CAL actor action executions for each mode. This profiling produced as a result a list of actor activities that is depicted in Table 1. The table simplifies the profiling results in the way that any activity in the actor respective to the mode is marked with an *x*, independent of the number of actions executed. The total number of actions executed in each mode is described in Table 2.

**Fig. 5** A high-level view of the RVC MPEG-4 SP decoder**Fig. 6** Motion compensation and texture decoding of RVC MPEG-4 SP**Table 1** Activity of the MPEG-4 SP actors in different operation modes

Actor	New frame	Inter block	ZMV block	Intra block	Hybrid block
Address	x	x	x	x	x
Buffer		x	x	x	x
Interpol.		x	x		x
DCSplit				x	x
DCRec.	x	x	x	x	x
IS	x	x	x	x	x
IAP	x	x	x	x	x
IQ				x	x
IDCT2D				x	x
Add	x	x	x	x	x

Table 2 Number of HSDF vertices in the five mode graphs, and the graph frequency

	New frame	Inter block	ZMV block	Intra block	Hybrid block
Number of HSDF vertices in graph	13	514	512	674	920
Graph frequency in test clip foreman	0.002	0.501	0.088	0.026	0.383

4.1 Design Space Exploration

The focus of this work is to explore the mapping of the MPEG-4 SP actors to a multiprocessor system. The number of mapping alternatives is considerable and requires an automated approach.

The target architecture consists of homogeneous PEs that are connected over a shared bus, as depicted in Fig. 7. We define the task as a design space exploration (DSE) problem that has three parameters: (a) mapping of each FU to one of the processing elements and (b) determining the priorities between FUs and (c) setting the cost of inter-processor communication (IPC). In situations, where several actions could potentially fire, those vertices that belong to a higher-priority FU, are fired before the vertices that belong to lower-priority FUs. Each FU is constrained to run completely on one PE, but one PE can be responsible for any number of FUs. For these experiments, the number of PEs was restricted to four to avoid the explosion of the number of possible solutions.

Pino *et al.* [17] have considered a related problem of *clustering* SDF graphs on multiprocessors. In a clustering problem, an arbitrary graph is given, and the vertices must be grouped as clusters that are then assigned to processors. The clustering problem is essentially about discovering the sets of vertices that should belong together to one cluster. In our mapping problem the clustering step can be omitted, because the original CAL model defines the clusters: the HSDF vertices belonging to one RVC FU form one cluster.

Our design-space exploration software takes as an input the set of HSDF graphs produced by our previous work [5], the number of clock cycles consumed by each action of CAL actors, and the IPC cost, which is a simple integer constant. The DSE software searches the combination of $FU \Rightarrow PE$ mappings and FU priorities that produces the minimal combined makespan (schedule length) for all HSDF graphs. The combination of makespans is computed by a weighted sum of graph makespans.

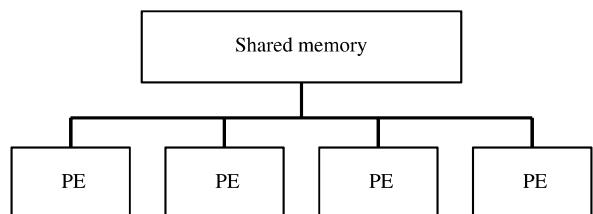


Fig. 7 The processor-memory interconnect assumed in this article

The weights are determined by the graphs' appearing frequencies, which are shown in Table 2. The weighting of graphs ensures that the throughput is optimized considering the whole training video sequence. Weighting is important, because one set of parameters (PE mapping and priority) might work really well for one graph, but produce a bad result for another graph. By weighting, it is made certain that the most often appearing graphs also execute in the shortest possible time. A successful use of this approach naturally requires that the training video sequence is as universal as possible, not to optimize the system to a specific kind of video.

The priorities between FUs in the design-time scheduling slightly affects the resulting schedule makespans. An optimal solution would be to assign a different priority to each HSDF vertex, but that would make the search space unfeasible. Thus, the priorities are assigned to the complete clusters (FUs).

The CAL actions in RVC are generally very fine grained and one action usually only modifies the value of a single pixel. We used the C language version of MPEG-4 SP (acquired from the ORCC compiler) to measure the latency of each action on an Altera Cyclone III FPGA running the Nios II/f soft processor [2]. The program was executed from an external DDR SDRAM and the processor had 4 KB of instruction cache and 2 KB of data cache. Figure 8 shows the average measured latencies for each action. However, the quasi-static scheduling approach requires the use of worst-case execution times to allow the design-time planning of inter-processor communication.

In the first phase of the DSE tool's operation, inter-processor communication graphs (IPC graphs) (see [18] for explanation) are generated from the application graphs (which describe the five different operation modes of MPEG-4 SP). The generation of IPC graphs is mapping dependent, which means that this has to be done again for each FU \Rightarrow PE mapping alternative, see Fig. 9. After generating an IPC graph, the schedule makespan for that graph is computed and eventually the FU \Rightarrow PE mapping producing the best result is selected and fixed. In the second phase, the DSE software evaluates the different FU priority alternatives, and computes the schedule makespan for each. In the third phase the FU \Rightarrow PE mapping and FU priorities are both fixed and a schedule is generated for visual inspection through a Gantt-chart.

The transformation tool that produces the HSDF graphs for our DSE tool, generates the graphs of all CAL actors in Fig. 6, which leaves the *parser*, *serialize* and *GEN_mgnt_Merger420* actors outside the quasi-static scheduling. The actions of these actors are executed in the default (unscheduled) way.

The search space of solutions with 10 FUs and 4 processors is vast, and must be restricted in some fashion. For the generation of a representative selection of FU \Rightarrow PE mappings, an algorithm based in integer partitions (see [19] for an explanation) was developed. This algorithm produced 3852 different mappings that were evaluated by the software tool. For the different FU priorities, 100 randomly generated alternatives were generated. It is not guaranteed that this procedure produces absolutely minimal makespans, because only a fraction of the search space is explored. Even though the first phase of the optimization produces considerably larger variations in makespan than the second phase (determination of priorities),

Actor	Action	Ex. time	Freq.	Weight
address	address_cmd_noMotion	430	10	0.0
	address_init	2110	594	0.4
	address_write_addr	970	38016	12.6
	address_done	150	594	0.0
	address_cmd_motion	330	519	0.1
	address_getmvx	600	519	0.1
	address_getmvy	1930	519	0.3
	address_read_addr	540	47288	8.7
	address_cmd_neither	1370	65	0.0
	address_newvop	450	1	0.0
buffer	address_getw	530	1	0.0
	address_geth	1180	1	0.0
interpolation	buffer_read	960	47288	15.5
	buffer_write	540	38016	7.0
interpolation	interpolation_row_col_0	380	9925	1.3
	interpolation_other	680	37363	8.7
	interpolation_start	700	584	0.1
	interpolation_done	0	584	0.0
add	add_cmd_motionOnly	210	362	0.0
	add_motion	380	23165	3.0
	add_cmd_textureOnly	220	10	0.0
	add_texture	200	653	0.0
	add_cmd_other	210	224	0.0
	add_combine	1240	14198	6.0
	add_newvop	210	1	0.0
	add_done	120	594	0.0
IDCT2D	broadcast_add_VID_do	920	38016	11.9
	IDCT_row	3480	1856	2.2
	transpose	7350	232	0.6
	IDCT_column	2390	1856	1.5
	retranspose	5540	232	0.4
	clip_read_signed	890	232	0.1
IQ	clip_limit	1040	14851	5.3
	IQ_ac	670	14619	3.3
	IQ_get_qp	1510	232	0.1
IAP	IQ_done	10	232	0.0
	IAP_newvop	610	1	0.0
	IAP_skip	400	362	0.0
	IAP_start	790	232	0.1
IAP	IAP_advance	170	594	0.0

Fig. 8 The measured latency of each action. Weight stands for the total computation time taken by the CAL action in the training video sequence

Actor	Action	Ex. time	Freq.	Weight	
IS	IAP_copy	350	14619	1.7	
	IS_skip	0	0	0.0	
	IS_start	590	232	0.0	
	IS_done	320	464	0.1	
	IS_read_only	460	14619	2.3	
DCSplit	DCsplit_dc	1420	232	0.1	3.5
	DCsplit_ac	670	14619	3.3	
DCReconstruction	addressing_start	580	1	0.0	0.9
	addressing_getw	640	1	0.0	
	addressing_geth	320	1	0.0	
	addressing_read_intra	920	10	0.0	
	addressing_read_other	870	584	0.2	
	addressing_advance	190	594	0.0	
	addressing_predict	3120	10	0.0	
	invpred_start	1760	1	0.0	
	invpred_skip	80	2	0.0	
	invpred_read_intra	8310	10	0.0	
	invpred_getdc_intra	660	10	0.0	
	invpred_read_inter_ac	3700	222	0.3	
	invpred_getdc_inter	700	222	0.1	
	invpred_read_other	1090	362	0.1	
	invpred_advance	240	594	0.0	
	invpred_sat	1500	232	0.1	
Total		avg = 739		$\sum = 100\%$	

Fig. 8 (Continued)

(IPC cost)/100	1	2	3	4	5	6	7	8	9	10	11	12	13	15	17	19	21	23	25	27	29	31	33	36	39	42	45
address	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	A	
buffer	A	C	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	A	
interpolation	B	A	C	C	C	C	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	A	
add	C	A	C	C	C	D	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	A	
DCSplit	C	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
DCRec.	A	A	A	A	A	A	C	C	C	C	C	C	C	C	C	A	A	A	A	A	A	A	A	A	A	A	
IS	A	A	D	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
IAP	A	A	D	D	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
IQ	D	A	A	A	D	D	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
IDCT2D	D	D	A	A	D	D	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	
# Processors	4	4	4	4	4	4	4	3	3	3	3	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2	1

Fig. 9 The FU \Rightarrow PE mapping as a function of IPC cost for a 2-port shared memory. The unit of IPC cost is a clock cycle

this optimization approach might get trapped to local minima. Acquiring better solutions can be accomplished by either investing more time in the optimization or by applying a more sophisticated optimization approach.

4.2 The Results

Each IPC cost that was imposed on the system resulted in a different FU \Rightarrow PE mapping and FU priority. In the experiments we searched solutions for IPC costs between 100 and 4500 (in clock cycles). To give a meaning to the IPC cost, it is worthwhile to mention that the average action latency was found out to be 740 (weighted average considering the frequency of each action).

With the IPC cost 100, the DSE algorithm mapped the tasks to four processors (see also Fig. 10, topmost chart). When the IPC cost was increased, the four-processor mapping was used until IPC cost became 800, after which the mapping shifted to three processors. Taking a closer look at the mapping for three processors reveals that one single FU has been mapped to processor C, and this processor is responsible for less than 1% of the total computational load (according to the measurements that are shown in Fig. 8).

Visually, this schedule induced by this mapping can be seen in Fig. 11 in the topmost Gantt chart. This behavior is simply a consequence of the optimization objective: minimizing the schedule makespan. If solutions like this would like to be avoided, should an additional optimization constraint of processor utilization be imposed. However, there is no universal way to define the weight of this additional

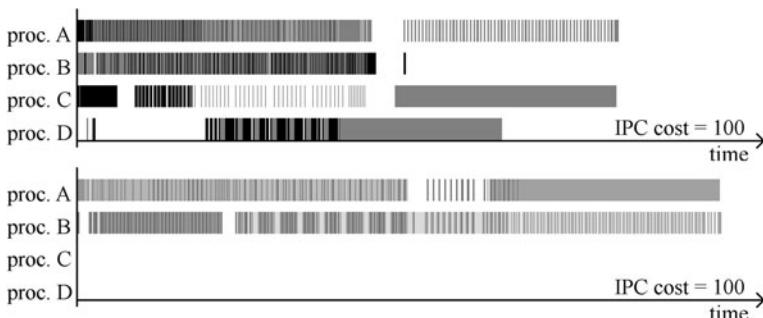


Fig. 10 Schedules for the *hybrid block* operation mode

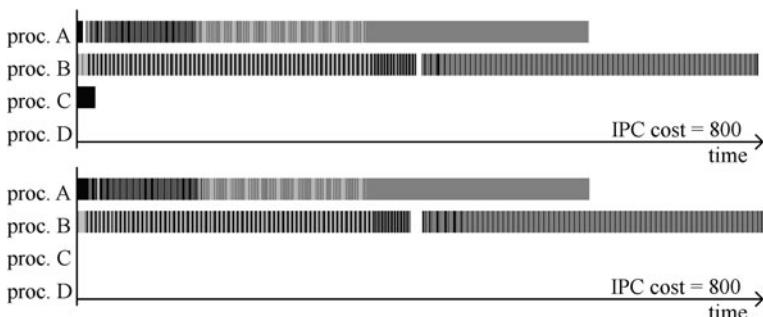


Fig. 11 Schedules for the *hybrid block* operation mode

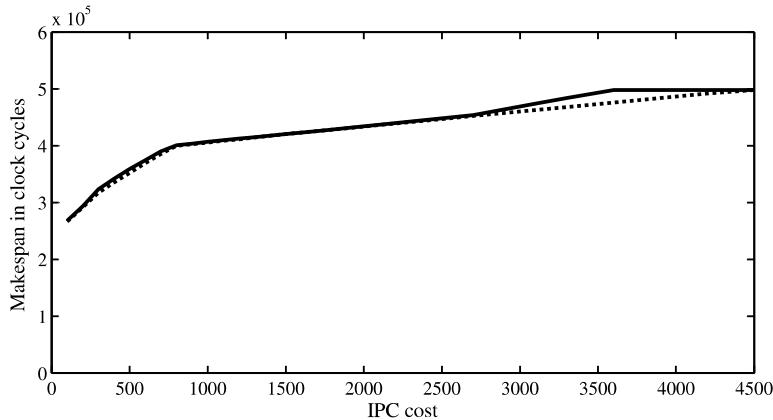


Fig. 12 Total weighted schedule makespan for the five mode graphs, as a function of IPC cost

constraint, because it rises the question about the cost of an extra processor. The lowermost schedule in Fig. 11 shows the mapping of IPC cost 800 with two processors: the makespan is increased by 0.5%, but only two processors are used. The FU that was on processor C is now mapped to processor A.

A fair question to ask is why the DSE algorithm chooses to map the most lightweight FU to processor C, instead of another one. The reason is in the IPC cost and the amount of communication: all other FUs transmit and receive approximately 64 tokens for each graph invocation. The lightweight FU *DCReconstruction*, that was mapped to processor C, is an exception: it receives and transmits less than six tokens. Since the IPC cost is a function of the number of tokens, this FU is the only one that could have been mapped to processor C without extending the total schedule makespan because of IPC activity.

After increasing the IPC cost beyond 1700, the computations were mapped on two PEs. Curiously, this mapping reflects the same motion compensation/texture decoding division that is also present in the original CAL models, but which has disappeared from the HSDF graphs that are used as input to the DSE tool. Finally, when the IPC cost becomes higher than 4200, a uniprocessor implementation provides the best throughput: everything is computed on PE A.

Figure 12 shows the total weighted makespan sum as a function of IPC cost. The performance advantage offered by a multiprocessor solution quickly starts to vanish as the IPC cost increases. When the IPC cost is less than 300, the performance advantage is between 45% and 35%, which is reasonably good. The two-processor solutions with IPC cost from 1700 to 4200 offer an advantage less than 15%, which is generally not interesting. The reason for this poor benefit lies again in the token rates: even with the best 2-processor mapping, 64 tokens must be transmitted over the IPC channel for each graph invocation. Even with the IPC cost 1700 this results in $1700 * 64 = 108800$ cycles lost to communication on *each* processor. The solid line shows the makespan with a system that has a single-port shared memory, and the dashed line depicts a system with a 2-port memory. A respective curve was also

computed for a non-blocking memory, but the differences to the 2-port memory results were negligible.

Another direction to approach the mapping problem is to fix the number of PEs to a certain number, and see what is the best attainable schedule makespan for the IPC cost of the system. The lowermost chart in Fig. 10 shows the best 2-processor mapping with IPC cost 100. The FUs *address*, *buffer* and *IDCT2D* reside on processor B, and the rest of the FUs on processor A. The total weighted makespan sum of this 2-processor solution is 20% worse than that of the 4-processor solution with the same IPC cost. In systems where the best possible performance is not required, this 2-processor solution is certainly more feasible than the 4-processor alternative.

Finally, the reader must be reminded about the applicability of these results: the solutions computed by the DSE tool depend on the latencies computed with the NIOS II processor and the specific C code implementation of the MPEG-4 decoder. Regarding the C code implementation, it is necessary to mention that the communication between the FUs is rather inefficient. This can be seen, for example, in the *broadcast_add_VID* action appearing under the *add* FU. In reality that action only reads one token and broadcasts it to two other FUs, and still uses 920 clock cycles for this. The reason behind this is that *broadcast_add_VID* is connected to the rest of *add* over a FIFO channel and transmits the broadcasted tokens over two FIFO channels as well. The handling of FIFOs is rather time-consuming in this C implementation, and streamlining it would certainly re-distribute the computational load to some extent.

5 Conclusion

In this paper we have presented the results of design space exploration for finding mappings of RVC MPEG-4 Simple Profile decoder functions to a multiprocessor system. Depending on the magnitude of the inter-processor communication cost, the design space exploration software maps the functions from one to four processors.

Prior to mapping the functions to processors, the CAL language specification of the RVC MPEG-4 decoder has been transformed to a set of quasi-statically schedulable HSDF graphs. The mapping of functions to processors has been done for these HSDF graphs.

The experiments show that the RVC MPEG-4 SP decoder can very well take advantage of a multiprocessor system: With reasonable IPC costs, two cores can provide computational speedup as the texture decoding and motion compensation parts of decoding can be computed relatively independent of each other. Mapping the decoder to a higher number of processors is only feasible if the IPC cost is low.

In this article we have presented a systematic approach for acquiring inter-processor communication-conscious quasi-static multiprocessor schedules for HSDF graphs created from CAL language programs. A graphical user interface is currently being constructed for the DSE tool and it has been integrated to the transformation tool that produces HSDF graphs from CAL networks. As future work, the multiprocessor schedules and mappings pointed out by the DSE tool are to be tested on a real multiprocessor system.

References

1. Lucarz C, Amer I, Mattavelli M(2009) Reconfigurable video coding: objectives and technologies. In: IEEE international conference on image processing, Cairo, Egypt. IEEE, Cairo
2. Altera (2009) Altera corp. Nios II processor reference handbook. http://www.altera.com/literature/hb/nios2/n2sw_nii5v2.pdf
3. Amer I, Lucarz C, Roquier G, Mattavelli M, Raulet M, Nezan JF, Deforges O (2009) Reconfigurable video coding on multicore: the video coding standard for multi-core platforms. IEEE Signal Process Mag, Special issue on multicore platforms 26(6):113–123
4. Boutellier J (2009) Quasi-static scheduling for fine-grained embedded multiproCESSing. PhD thesis, Department of electrical and information engineering, University of Oulu, Finland
5. Boutellier J, Lucarz C, Lafond S, Martin Gomez V, Mattavelli M (2009) Quasi-static scheduling of CAL actor networks for reconfigurable video coding. J Signal Process Syst. doi:[10.1007/s11265-009-0389-5](https://doi.org/10.1007/s11265-009-0389-5)
6. Boutellier J, Martin Gomez V, Lucarz C, Mattavelli M, Silvén O (2009) Multiprocessor scheduling of dataflow models within the reconfigurable video coding framework. In: Conference on design and architectures for signal and image processing (DASIP), Sophia Antipolis, France
7. Cheng KT, Krishnakumar AS (1993) Automatic functional test generation using the extended finite state machine model. In: DAC '93: proceedings of the 30th international design automation conference. ACM, New York, pp 86–91. doi:<http://doi.acm.org/10.1145/157485.164585>
8. Eker J, Janneck JW (2003) CAL language report. Technical Report UCB/ERL M03/48, UC Berkeley
9. Eker J, Janneck JW, Lee EA, Liu J, Liu X, Ludvig J, Neuendorffer S, Sachs S, Xiong Y (2003) Taming heterogeneity – the Ptolemy approach. Proc IEEE 91(1):127–144
10. Gorin J, Nezan JF, Raulet M, Wipliez M (2009) Open RVC-CAL compiler. <http://sourceforge.net/projects/orcc>
11. Lee EA (1988) VLSI signal processing III: recurrences, iteration, and conditionals in statically scheduled block diagram languages. IEEE Press, New York, pp 330–340
12. Lee EA, Messerschmitt DG (1987) Synchronous data flow. Proc IEEE 75(9):1235–1245
13. Lucarz C, Mattavelli M, Wipliez M, Roquier G, Raulet M, Janneck JW, Miller ID, Parlour DB (2008) Dataflow/actor-oriented language for the design of complex signal processing systems. In: Conference on design and architectures for signal and image processing, Bruxelles, Belgium, pp 168–175
14. MPEG (2008) ISO/IEC FCD 23002-4 Information technology – MPEG video technologies – part 4: video tool library
15. Oh H, Ha S (2002) Hardware-software cosynthesis of multi-mode multi-task embedded systems with real-time constraints. In: CODES '02: proceedings of the tenth international symposium on hardware/software codesign, Estes Park, CO, pp 133–138. doi:<http://doi.acm.org/10.1145/774789.774817>
16. Olsson T, Carlsson A, Wilhelmsson L, Eker J, von Platen C (2010) A reconfigurable OFDM inner receiver implemented in the CAL dataflow language. In: Proc IEEE international symposium on circuits and systems (ISCAS), Paris, France
17. Pino JL, Bhattacharyya SS, Lee EA (1995) A hierarchical multiprocessor scheduling system for DSP applications. In: Asilomar conference on signals, systems and computers, vol 1, Pacific Grove, CA, pp 122–126. doi:[10.1109/ACSSC.1995.540525](https://doi.org/10.1109/ACSSC.1995.540525)
18. Sriram S, Bhattacharyya SS (2000) Embedded multiprocessors: scheduling and synchronization. Marcel Dekker, New York
19. Wilf HS (2000) Lectures on integer partitions. <http://www.math.upenn.edu/~wilf/pims/pimslectures.pdf>

A High Level Synthesis Flow Using Model Driven Engineering

Sebastien Le Beux, Laurent Moss,
Philippe Marquet, and Jean-Luc Dekeyser

Abstract This chapter presents a High Level Synthesis (HLS) flow dedicated to intensive signal processing applications. Model Driven Engineering (MDE) is the skeleton of this flow. The benefits of extending this software technology to hardware design are used to solve major difficulties encountered by usual HLS flows. Both *users* and *designers* of the flow take advantage of the MDE methodology, leading to a concrete and effective advancement in the HLS research domain. The flow is automated from UML specifications to VHDL code generation. It has been successfully evaluated for the design of a hardware accelerator dedicated to signal processing.

Keywords High Level Synthesis · Hardware accelerators · Model Driven Engineering · Intensive signal processing

1 Introduction

Intensive Signal Processing (ISP) applications handle large amounts of data and are characterized by hierarchical and data parallel tasks, which manipulate multidimensional data arrays according to complex data dependencies. Performance requirements often preclude ISP applications from being implemented purely in software and instead call for using custom and efficient hardware accelerators. A hardware accelerator is an electronic design dedicated to the execution of a specific application. Its hardware architecture can be designed for a maximal parallelization of the algorithm needed to execute its application and for optimal execution support

S. Le Beux (✉)

Institut des Nanotechnologies de Lyon, Ecole Centrale de Lyon, 36, Avenue Guy de Collongue, 69134 Ecully Cedex, France
e-mail: Sebastien.Le-Beux@ec-lyon.fr

for regular and repetitive tasks. However, the complexity of hardware accelerators makes them difficult to manipulate at low abstraction levels (in a Hardware Description Language (HDL) for instance). The description of complex ISP applications is also error prone and tedious when using tools that constrain the number of dimensions of data arrays.

High Level Synthesis (HLS) seeks to simplify the design of hardware accelerators by describing applications at a high abstraction level and by generating the corresponding low level implementation. Application specification is easier at a high abstraction level since hardware designers do not need to handle all low level implementation details. HLS thus aims to achieve algorithm-architecture matching by construction, through the automated synthesis of a hardware architecture for an application specified at a high level. The automatic generation of low level implementations drastically reduces non-recurring engineering costs and the time to market compared to hand-tuned implementations in HDL. For these reasons, HLS tools have been increasingly successful among the hardware designer community. This trend is followed by the continual integration of new capabilities and functionality in the tools. Therefore, successful HLS has to support rapidly evolving technologies and be maintainable in order to capitalize on efforts. We present some design challenges faced by HLS and how model-driven engineering can meet them.

1.1 Design Challenges

This section presents some critical design challenges faced by both HLS tool *users* (i.e. hardware designers) and HLS tool *designers*.

1.1.1 HLS Tool User

From the tool user's point of view, the specification's abstraction level is sometimes not high enough to be really independent of low level implementation considerations: each particular implementation of a same application requires a particular specification. Such specifications are generally done in C or C-like syntax (e.g. Handel-C or SystemC) [15, 16, 30]. Unfortunately, such textual low level descriptions do not provide the opportunity to immediately extract specific information such as data dependencies, data parallelism and hierarchy. Conversely, a graphical representation associated to a factorized expression of the potential data parallelism and a powerful expression of data dependencies can solve the difficulties faced by HLS tool users. Moreover, a standard representation will considerably enhance discussions between the different field experts who take part in the specification of an application.

1.1.2 HLS Tool Designer

The gap between high abstraction levels and low abstraction levels is often bridged with one or several Internal Representations (IR) [15, 16, 19] in HLS tools. The set of concepts associated to an IR is generally difficult to handle due to the lack of formal definition of these concepts and of the relations between them. Therefore, IR extension and maintenance (necessary for the development and evolution of the tool) rely on new specifications of the IR itself. Conversely, with a formal definition, extensions and maintenance are supported by the addition of new concepts and new relations. This ensures a high extensibility and maintainability of the IR, and consequently of the tool itself. Furthermore, the clear identification of concepts and relations in an IR allows a compilation process based on *concept to concept* translations to take care of the relations between these concepts. The consequences of the introduction of new concepts or relations in the source or target IR are then localized in the compilation (i.e. translation) process.

At the level of the design flow, a clear separation of the compilation phases implies a clear identification of the concepts, which helps to capitalize on the tool designer's efforts. Such tool development requires a strong methodology well-suited to designer habits and a stable and advanced technology to ensure the reuse, extension and maintainability of designer developments.

1.2 Proposed HLS Flow

This chapter presents an HLS flow dedicated to massively parallel ISP applications. The input is a graphical UML model of such an application. This model is at a high abstraction level: it is independent from any implementation technology. The output is an hardware accelerator able to execute the corresponding application. The generation of a hardware accelerator from the input model is handled by three successive transformations, see Fig. 1. The first transformation generates an internal representation of ISP applications and keeps only useful concepts necessary to represent them. The second transformation refines ISP models into RTL models; an RTL model represents an hardware accelerator able to execute the corresponding ISP application. The last transformation ensures the generation of the VHDL code corresponding to the hardware accelerator. Usual Electronic Design Automation (EDA) tools are then used to synthesize the resulting VHDL code to either an FPGA or an ASIC.

Our flow is entirely built with the Model Driven Engineering (MDE) methodology [32]: abstraction levels are defined in metamodels and refinements are done by model transformations. By allowing a clear specification of concepts and relations between concepts, MDE helps to reduce designer difficulties in developing and maintaining HLS tools. MDE also eases extensions of the proposed HLS flow:

- A *fine grain* extension extends the purpose of the HLS flow, for instance to manage control flow applications. This is successfully accomplished with the addition of new concepts in the metamodels and new rules in model transformations.

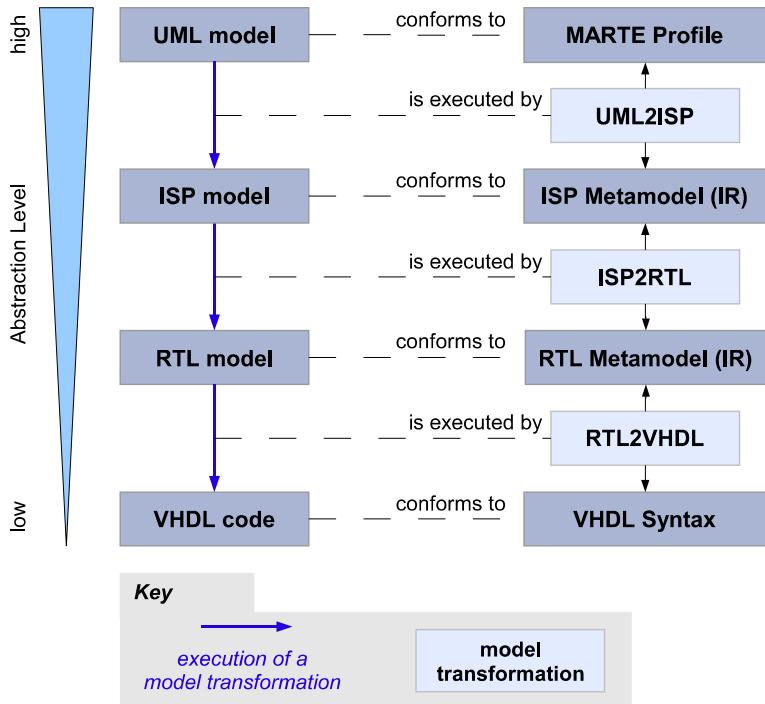


Fig. 1 The proposed HLS flow for ISP applications

- A *coarse grain* extension consists of a modification of the design flow itself for new purposes. For instance, a model transformation could be added to generate Verilog [34] code from the RTL metamodel or to create an RTL model from another metamodel (a metamodel used in another tool for instance).

By successfully meeting the design challenges enumerated above, our HLS flow suitably solves the major difficulties encountered by both tool users and designers.

This chapter is organized as follow. Related works are presented in Sect. 2. Section 3 introduces MDE. We present our flow in Sects. 4 and 5. Experimental results are presented in Sect. 6. The last section concludes this work.

2 Related Works

During the last few years, the trend in HLS research [12, 15, 16, 30] and in commercial state-of-the-art HLS tools [7, 8] has been to generate HDL code from C/C++ or C-like languages such as Handel-C or SystemC. Using C code to generate hardware designs allows working with a well-known language and at a higher abstraction level than RTL. However, since C is a sequential imperative language and neither

an HDL nor a parallel programming language, C by itself is not well-suited to describing hierarchical applications that manage both task and data parallelism. Users of such a tool must strictly adhere to its coding guidelines both to make sure that their C-based input falls within the tool’s synthesizable subset and to allow the tool to infer hierarchy and parallelism from the sequential code. System-level design languages such as SystemC help in specifying basic connections between the RTL blocks generated by HLS, but face the same challenge than C when it comes to actually performing HLS for each of these blocks. These works are compared to our flow in the following paragraph.

HLS tools usually infer data parallelism from loops with bounded indexes. However, the extraction of data dependencies across loop indexes from user-provided code is both tedious and error-prone: its complexity dramatically increases with the number of dimensions and the shape of the pattern. In our flow, the expression of data dependencies relies on *tilers* [5]. Tilers do not share these drawbacks since data dependencies are expressed explicitly and independently of each other through a matrix-vector expression. Each tiler has Origin, Paving and Fitting attributes which express how a data pattern is built from an array. The origin vector specifies the origin of the reference pattern in the array. The paving and fitting matrices respectively specify how an array is covered by patterns and how the patterns are constructed with array elements. A formal description of tilers is given in [5].

Some other approaches aim to specify application with code that follows the polyhedral model. This implies stricter user code restrictions than conventional HLS tools while offering more opportunities for optimizations in the IR using existing libraries [3, 11, 14]. Such approaches share the drawbacks previously identified for conventional HLS tools.

Another high level text-based approach would be for the user to specify the application as a mathematical formula. In [21], linear signal transforms are formally specified in the SPIRAL language [26] as products of structured sparse matrices and several RTL implementations, with different degrees of parallel execution, can be generated for a given user-provided transform. This formalism is well-suited to transforms which can be defined recursively, such as a Discrete Fourier Transform (DFT). However, this formalism is not well-suited to specifying a complete ISP application which involves several different transforms with complex data dependencies. A graphical formalism such as UML is better suited to representing the hierarchy, data dependencies and task parallelism of such an application. Furthermore, the use of MDE in our flow allows greater opportunities for extensions by tool designers while the input language of [21] is narrowly domain-specific. Tools based on the polyhedral model or mathematical formulas could also be used to generate elementary components to be integrated in our flow, so these works are complementary to ours.

MDE has been increasingly adopted in the design of embedded systems in general [31]. The basic modeling formalism is the general purpose language UML, which offers attractive graphical representations. Because of its generality, UML is refined by the notion of *profile* to address domain-specific problems. There are currently several profiles for the design of embedded systems such as SysML [25],

UML SPT [24], UML-RT [33], TUT Profile [17], ACCOR/UML [18] and Embedded UML [20]. Because all these profiles may potentially overlap, significant standardization efforts have been recently undertaken by the OMG, resulting in the single unified and effective MARTE standard profile [23], on which our HLS flow relies. MARTE stands for Modeling and Analysis of Real-Time Embedded systems. Among other things, MARTE provides mechanisms to express in a factorized way the potential parallelism available in applications. MARTE is thus well-suited to the design of intensive signal processing applications and is used to model input applications in our HLS flow.

While these profiles allow one to specify a system with high level models, refinements from such models towards low level models have to be achieved. Some proposals use specific notations, defining a fully *executable* model semantics [1, 22, 28]. Such expressive notations allow one to define models with sufficient information so that the specified system can be completely generated. However, code is directly generated from the specifications, without any intermediary representation. The same is observed in previous works on VHDL code generation from UML [4, 9, 29, 35], where code is obtained directly by mapping UML concepts on VHDL syntax. These tools focus on finite state machines, so they do not address ISP applications. Furthermore, the absence of successive refinements leads to a lack of flexibility when targeting new abstraction levels or new languages. While these approaches rely on an abstraction of the system by using high level models, they only exploit a little of its benefits by being directly dependent on target languages or abstraction levels. Compared to these works, the high level synthesis flow we propose considers intermediate abstraction levels. This allows a smooth refinement from high abstraction level descriptions to low level implementations. This eases extensions of the flow (e.g. to target dynamically reconfigurable architectures [27] or to generate Verilog).

3 Model Driven Engineering

Complex systems can be easily understood via abstract and simplified representations: *models*. A model highlights the intent of a system without describing the implementation details. Several methodologies sought to manipulate models in the past decades, starting with Chen [6], and MDE [32] is one of these methodologies. It has been oriented towards the modeling of software engineering systems. Since the resulting models must be comprehensive and machine-readable, MDE also covers code generation. In this way, MDE stands apart from other model-based methodologies. This section details the major aspects of MDE that are *models*, *metamodel* and *model transformations*. General mechanisms are introduced and their relevance to ISP applications is highlighted and discussed.

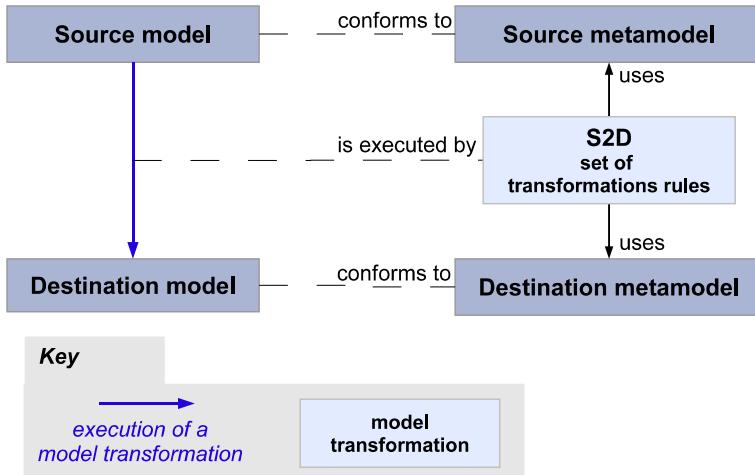


Fig. 2 A model transformation

3.1 Model and Metamodel

A model is an abstraction of reality. Models can be graphically observed from different points of view in order to highlight specific aspects of a given reality. Models focusing on aspects such as data parallelism and task parallelism can represent ISP applications well. A metamodel gathers the set of concepts and relations between the concepts used to describe a model according to a particular purpose (e.g. according to a given abstraction level). A model is then said to *conform to* a metamodel. Generally speaking, a metamodel defines the syntax of its models, like a grammar defines its language. A metamodel dedicated to the modeling of ISP applications at a given abstraction level thus gathers the corresponding set of concepts and relations. Such metamodel is assimilated to an IR in the HLS tool.

3.2 Model Transformations

A model transformation [10] is a compilation process which transforms a *source* model into a *target* model, as illustrated in Fig. 2. The source and target models respectively conform to the source and target metamodels. A model transformation relies on a set of small *rules*. According to such a decomposition, particular and specific attention can be provided to the concepts or set of concepts handled by a given rule. For instance, data parallelism and task parallelism can be transformed with the specific attention they require.

Figure 3 illustrates a graphical representation of a simple rule used to transform components at a high abstraction level (c:Component) into components at a lower abstraction level (tc:Component). Each rule is divided into three parts: the *rule input pattern*, the *signature* and the *rule output pattern*.

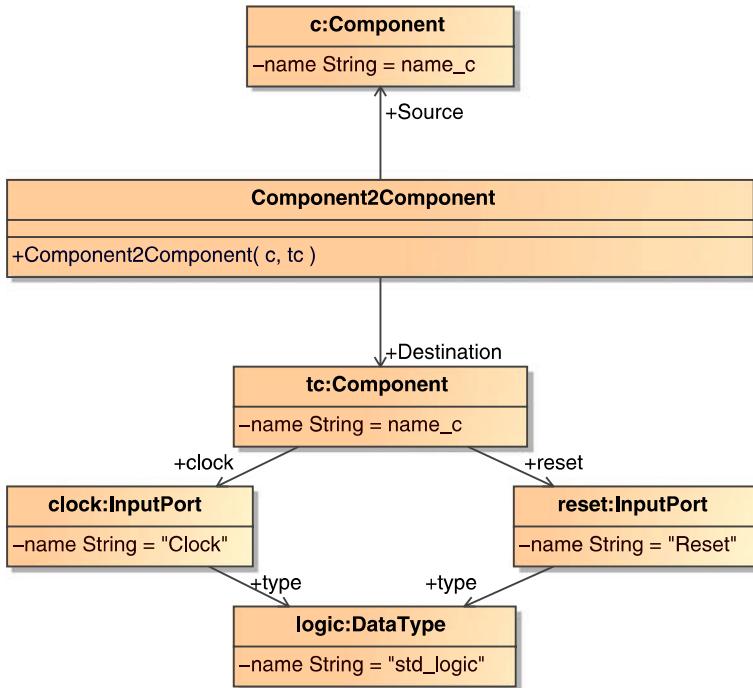


Fig. 3 Graphical representation of a transformation rule

- The rule compares the input pattern to the source model in order to detect a concept or a set of concepts which trigger an execution. Such a condition is illustrated on the top part of the figure. In this example, the rule input pattern is very simple and contains the single concept **c:Component**.
- The signature of a rule is represented in the center of the graphical representation, it corresponds to the **Component2Component** concept. The signature allows the identification of the rule input and output patterns by the source and destination relations. During the transformation's execution, the signature identifies the set of concepts matching the rule input pattern, stores the information associated to these concepts and potentially calls other transformation rules (so-called sub-rules).
- The rule output pattern, illustrated on the bottom part of the figure, corresponds to a set of concepts in the target model that are created during a rule execution. In the example, it includes four concepts. **tc:Component** is the main one since it is directly linked to the signature. Such a rule allows adding information during the transformation. For instance, clock and reset ports are attached to the transformed components (concepts **clock:InputPort** and **reset:InputPort**).

Model transformations are well-suited to performing refinements from high abstraction level specifications to code generation. For this purpose, model transfor-

mations add implementation details all along the compilation process. The code generation is a *model to text* transformation. Unlike model to model transformations, they are made of templates. However, the transformation principle remains the same.

4 High Level Specification Models

This section presents the high level models used in our design flow.

4.1 UML Model

Applications are modeled in UML, which is an OMG standard commonly used by the MDE community. We use the MARTE *profile* (i.e. extension) and its mechanisms to represent data parallelism, task parallelism and data dependencies. The concepts present in such UML models include high-level components with their inputs and outputs and how these components are instantiated, assembled and connected together to model the application. These concepts are illustrated here through the modeling of a matrix multiplication example.

Figure 4 represents the UML model of a matrix multiplication example which multiplies matrix MA and MB in order to produce a matrix MC. MatrixMultiplication is a hierarchical task. The data consumed and produced by this task are respectively represented by the input ports MA and MB and the output port MC. In this example, each port corresponds to a matrix and the dimensions of each port are those of the corresponding matrix: 5×3 for MA, 2×5 for MB and 2×3 for MC. In such UML models, input and output data are represented as multidimensional arrays. There are no restrictions on the number of dimensions of data arrays. This allows

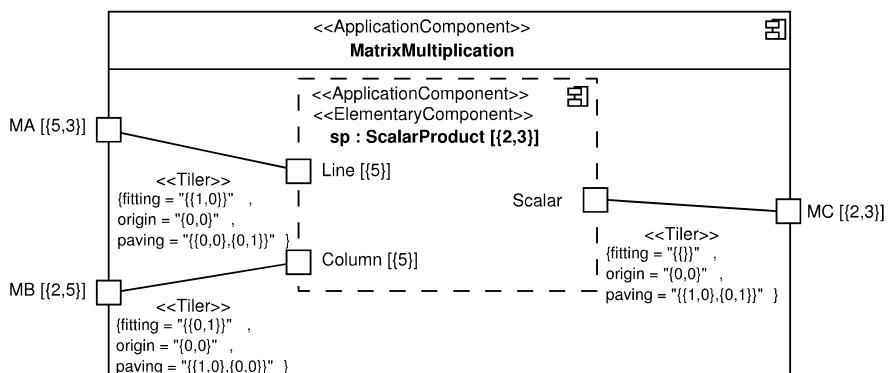


Fig. 4 UML model of the matrix multiplication example

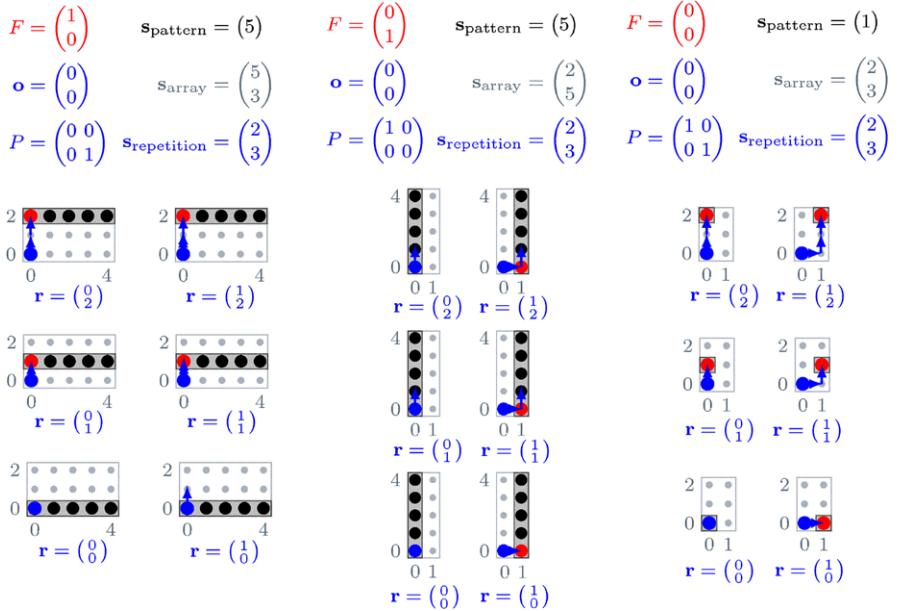


Fig. 5 The data dependencies expressed by the tilers of the MatrixMultiplication task

the modeling of multidimensional data manipulations typical of ISP applications. For instance, video processing applications handle data over two spatial and one temporal dimensions, whereas sonar chains process data over spatial, temporal and frequency dimensions.

The multiplicity {2, 3} of the component instance `sp` of task `ScalarProduct` indicates that it is a data parallel task. In this example, the `ScalarProduct` task is repeated 2×3 times. Each iteration in the repetition space consumes input data patterns and produces output data patterns. *Tiler* connectors model the data dependencies used to generate these patterns. Each tiler models data dependencies linking an M -dimension data array to an N -dimension pattern. These data dependencies are not limited to compact and axis-aligned patterns.

Figure 5 represents the data dependencies expressed by the tilers used in the matrix multiplication example. The left-hand side of Fig. 5 represents the tiler that links `MA` with `Line`, the center corresponds to the second input tiler and the right-hand side illustrates the output tiler. This figure represents the data consumed and produced in the data arrays (i.e. `MA`, `MB` and `MC`). For instance, in the first iteration on the repetition space $r = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, the first line of data array `MA` and the first column of `MB` are read.¹ This line and column are used in the first iteration of task `sp` to produce the first data (i.e. the data at (0, 0) in output data array `MC`). In iteration

¹The line and the column are constructed through the fitting field.

$r = \binom{1}{0}$, the first line of **MA** is read again while the second column of **MB** is used.² The data at position (1, 0) in **MC** is computed and so on and so forth: by iterating over the whole repetition space, the whole output data array **MC** is produced.

The task **sp** consumes two input patterns (Line and Column) and produces the output pattern Scalar which corresponds to the result of a scalar product of a line and a column. **ScalarProduct** is an elementary task, i.e. a leaf in the application model. Its behavior is provided by the deployment part of the UML model [2] which links each elementary component to a given IP available in a library. Hence, the application model remains independent from the implementation target.

4.2 ISP Model and UML2ISP

Generally speaking, the ISP metamodel includes the interesting subset of MARTE dedicated to the description of ISP applications. Additional features allow to modify the *hierarchy* and to set the *data parallelism execution partitioning*. The hierarchy in ISP models can be modified through the loop transformations proposed in [13]. These loop transformations can modify, create or delete the hierarchy and move the data parallelism inside this hierarchy. Since the result of a loop transformation is an ISP model, successive loop transformations can be applied. The ISP metamodel allows specifying a data parallel execution for each hierarchical task. Thus, part of the data parallelism can be executed sequentially while the other part is executed in parallel. To satisfy constraints of the RTL metamodel (which will be further detailed in Sect. 5), the following rules must be respected: the top level tasks are sequentially executed, and the lowest ones are executed in parallel. The set of specified executions defines the data-parallelism partitioning.

Specifying parallel or sequential execution for a set of hierarchical tasks is similar to the operations of allocation, scheduling and binding performed by conventional HLS tools [7]. Thus, specifying a given data parallel execution means allocating a given set of computing units, scheduling tasks to given clock cycles and binding tasks to the allocated computing units. However, conventional HLS tools typically allocate only simple fine-grained components, such as adders and multiplexers, contained in a fixed RTL library. On the other hand, the computing units allocated by our HLS flow can be pre-defined library components, user-defined components, or a hierarchical composition thereof, and our flow can thus allocate computing units covering a large spectrum of complexity and granularity.

UML2ISP ensures the transformation of a UML model into an initial ISP model. The hierarchy of the resulting ISP model matches the hierarchy defined in the UML model.

²The shift of the line and the column are constructed by the paving.

5 Implementation at a Low Level

5.1 RTL Model

The RTL metamodel gathers the set of concepts used to describe hardware accelerators at the RTL level. Such hardware accelerators can execute the targeted ISP applications according to a specific execution model.³ This execution model is data flow oriented and handles, among others, hierarchy, multidimensional data dependencies, data parallelism and task parallelism. The following provides an overview of the RTL metamodel.

In order to model hierarchical and well-structured hardware accelerators, the RTL metamodel relies on a component based approach. Communications between components go through interfaces, which are composed of ports. There are input and output ports: a component can receive or send data. The shape and type of each port can also be specified. The RTL metamodel gathers the set of concepts used to implement parallel and sequential execution. These concepts are illustrated with the matrix multiplication example. The repetition space around the `ScalarProduct` task is {2, 3}. This task can be executed in parallel or sequentially:

- With a parallel execution, this task is instantiated 2×3 times, as illustrated in Fig. 6(a). The six filled boxes represent the instances. Each instance computes a separate given line-column scalar product through connections to the customized data paths.
- With a sequential execution, `ScalarProduct` is instantiated only once, as illustrated in Fig. 6(b). The overall computation is coordinated by a controller (represented with a lozenge) using multiplexers and demultiplexers (latches are also used in order to store data, they are not drawn in the figure in order to keep it readable). The controller iterates over the repetition space and, by controlling the multiplexers, sends the right data (i.e. the right line and the right column for this example) to the single computing unit. Conversely, the demultiplexers send the right data to the output tiler. A constraint we have is that the most top level task has to be sequentially executed.

The mixed parallel/sequential execution relies on a combined use of these concepts. Thus, the hierarchy of the accelerator can be modified and the data parallelism moved through this hierarchy. The data parallelism included in each hierarchical component is then executed independently from each other.

Data dependencies are implemented through data paths that are composed of connectors, buffers, latches, multiplexers and demultiplexers. These data paths can implement simple data array dependencies used in task parallelism as well as complex multidimensional data array dependencies used in data parallelism.

³The word *model* is different from the term *model* used in MDE. In order to avoid any confusion, the term *execution model* is used when dealing with the way an application is executed.

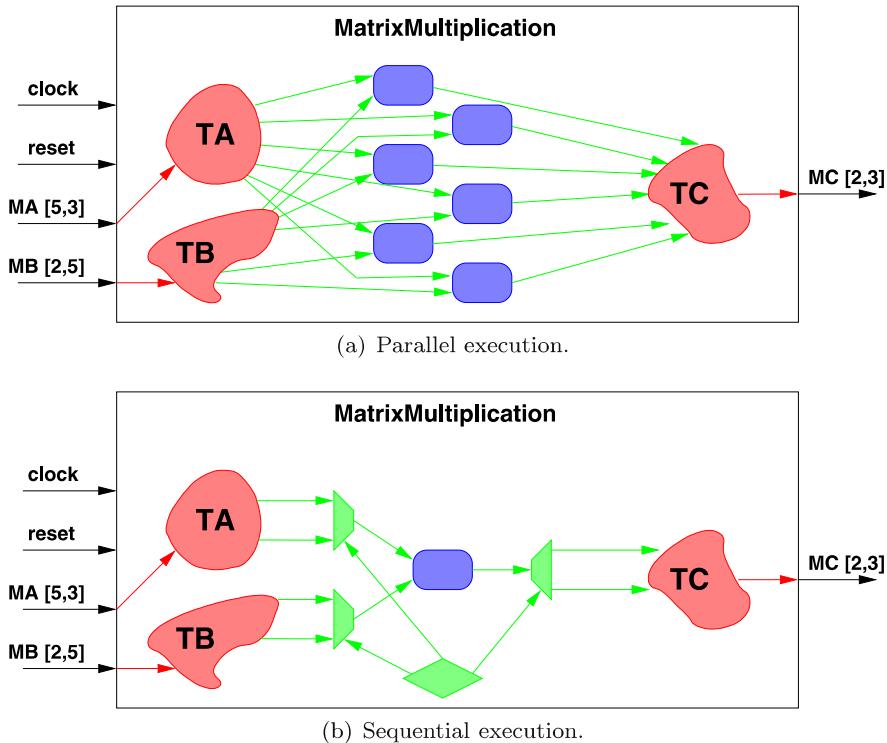


Fig. 6 Hardware execution of the data parallelism in RTL models

5.2 ISP 2RTL Transformation

ISP models are independent from any implementation technology. Their automated implementation in either FPGA or ASIC technologies thus requires very specific refinements assumed by the ISP 2RTL transformation. ISP 2RTL is composed of rules. While some rules are very simple (such as the so-called one-to-one rules), some others are more tedious. For instance, the creation of a customized data path starting from a pure expression of data dependencies relies on a quite complex set of rules. In this set of rules, Tiler2InputTiler generates the data path and Tiler2InputTilerInstance interconnects the resulting data path into the component. The execution of Tiler2InputTilerInstance is triggered each time a part of ISP model matches the rule input pattern, as illustrated in Fig. 7(a). This rule input pattern identifies the source and target of the tiler's connectors, the shape of the source port, the repetition space of the repeated task, etc. When Tiler2InputTilerInstance is triggered, the Tiler2InputTiler blackbox rule is automatically triggered; it analyzes the tiler's attributes (i.e. Origin, Paving and Fitting) in order to generate the right data path. For this purpose, Tiler2InputTiler computes, for each data in the pattern and for each iteration in the repetition space, the data read in the input array. Basically, the computation is done in two successive steps:

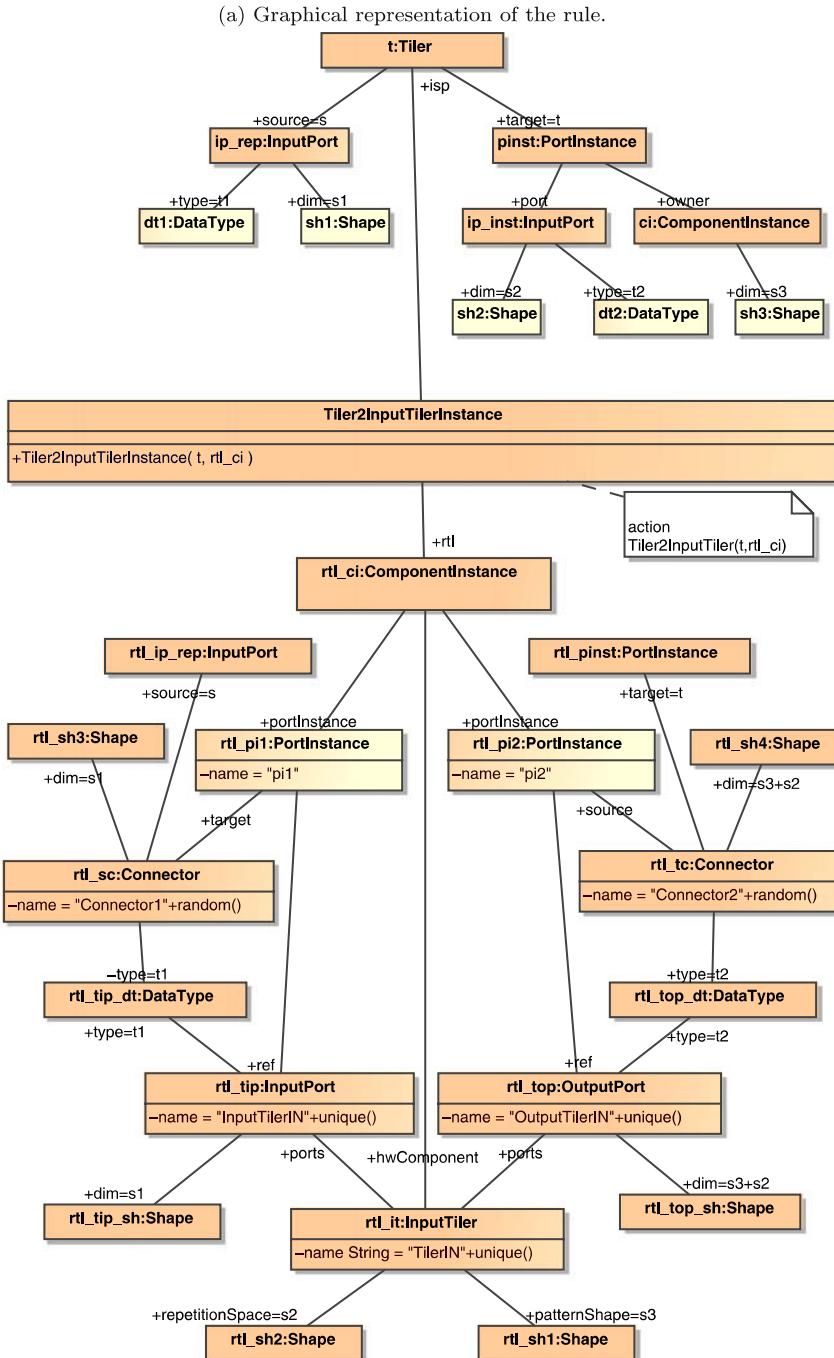


Fig. 7 The Tiler2InputTilerInstance rule transforms the tilers into customized data paths

(b) Execution of the rule.

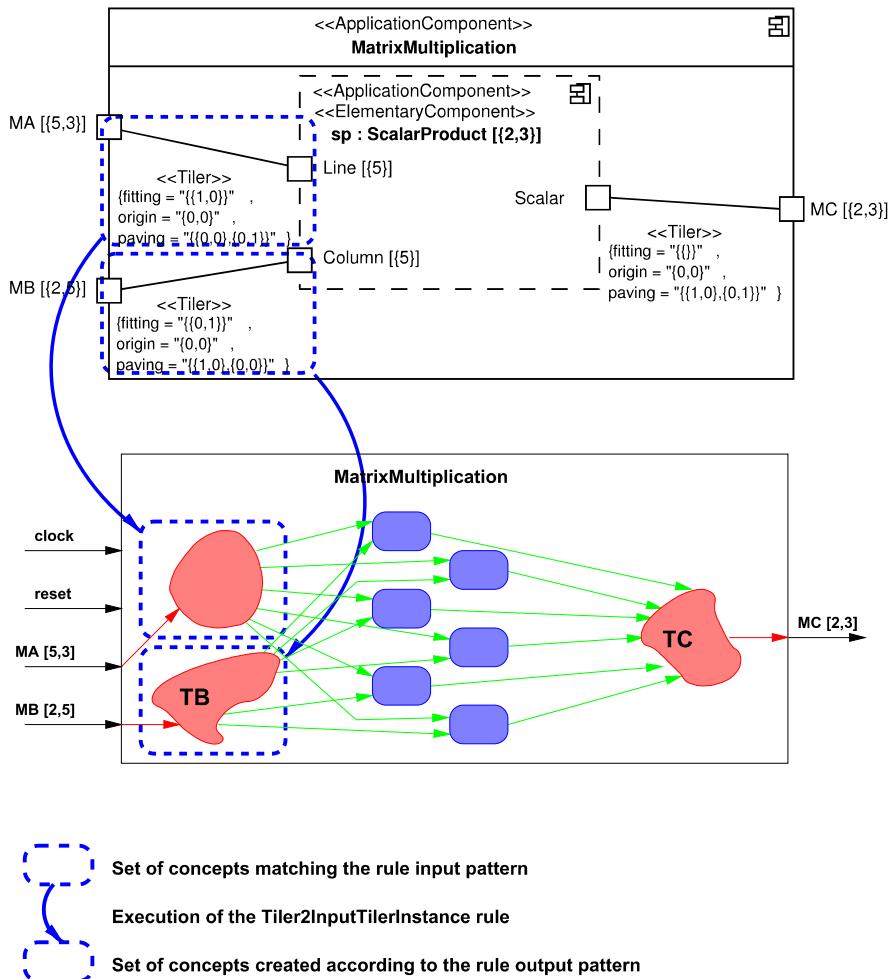


Fig. 7 (Continued)

- based on the iteration in the repetition space, the origin coordinates in the input array are computed;
- based on the data in the input pattern and the origin coordinates, the coordinates of the data read in the input array are computed.

As a result, we obtain a set of one-to-one links between the input pattern and the input array. From these links, wires and buffers are allocated.

Figure 7(b) represents the execution of these rules onto an ISP model. Two inputs tilers, highlighted through the dashed shapes, match the Tiler2InputTilerInstance rule

```
(a) Textual representation of a rule.
ENTITY <%=element.getName()%> IS
PORT (
<%=ts.generate(element.getClock())%>;
<%=ts.generate(element.getReset())%>;
<%for (Port p : (List<Port>)
    element.getPorts())
{%
<%=ts.generate(p)%><%
}>);
END <%=element.getName()%>;
```

```
(b) Resulting VHDL code for the example.
ENTITY MatrixMultiplication IS
PORT (
clock : IN Std_Logic;
reset : IN Std_Logic;
MA : IN Type_5_3_Integer;
MB : IN Type_2_5_Integer;
MC : OUT Type_2_3_Integer);
END MatrixMultiplication;
```

Fig. 8 A rule transforming RTL components into VHDL entities

input pattern.⁴ This rule is thus triggered twice and Tiler2InputTiler is subsequently triggered, resulting in TA and TB. Due to the simplicity of the example, the generated data paths (not detailed in the figure) only include wires.

5.3 RTL2VHDL Transformation

The RTL metamodel is independent from any HDL syntax, but is low level enough to allow code generation through the RTL2VHDL transformation. VHDL code generation from the RTL metamodel is performed through templates that navigate the RTL model to find their associated concepts and print them in a VHDL syntax. Figure 8(a) presents the template associated to the Component concept in the RTL metamodel. Figure 8(b) shows excerpts of the generated code for the MatrixMultiplication component. Special attention was given to keep multidimensional data arrays and data parallelism factorized. The generated code can be directly synthesized (e.g. on FPGA) with standard logic synthesis tools.

6 Case Study

This section illustrates the correctness and efficiency of our flow dedicated to intensive signal processing applications. For this purpose, a correlation algorithm is studied. This algorithm is well known and frequently used in intensive signal processing. Equation (1) gives its mathematical formulation, which is composed of a set of multiplications and additions that can be executed in parallel.

$$C_{cy}(j) = \sum_{i=0}^{1023} c(i) \cdot y(i + j) \quad (1)$$

⁴The figure represents the UML model, which is very close to the ISP one.

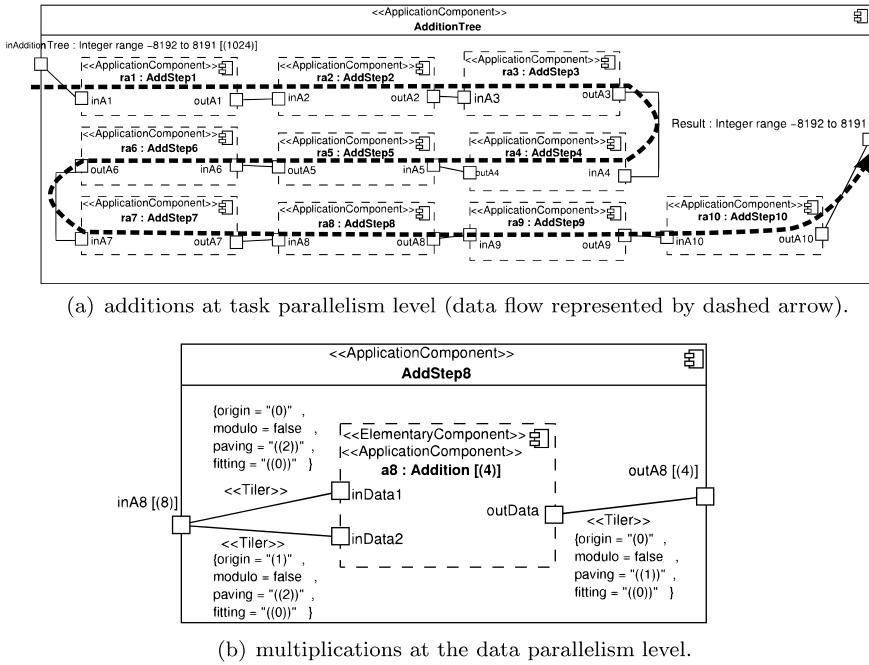


Fig. 9 UML model of the correlation algorithm

6.1 UML Model

The correlation algorithm application has been modeled in UML and independently from any implementation. All the data parallelism and task parallelism are extracted so that they can be used to generate an efficient implementation. In the following, representative parts of the UML are presented.

Figure 9(a) illustrates component *AdditionTree* that realizes the sum in the correlation algorithm. The input port *inAdditionTree* is composed of 1024 data (i.e. the data to sum) and the output port is a scalar value (i.e. the result of the sum). The 10 component instances represent the 10 pipeline stages of the tree topology used to realize the sum and the data flow is represented by the dashed arrow. Figure 9(b) represents the data parallel task *AddStep8*, which is the 8th component instantiated in the pipeline stage. Its input port *inA8* and its output port *outA8* are respectively composed of 8 data and 4 data. The elementary task *a8* is repeated 4 times to realize the necessary computation.

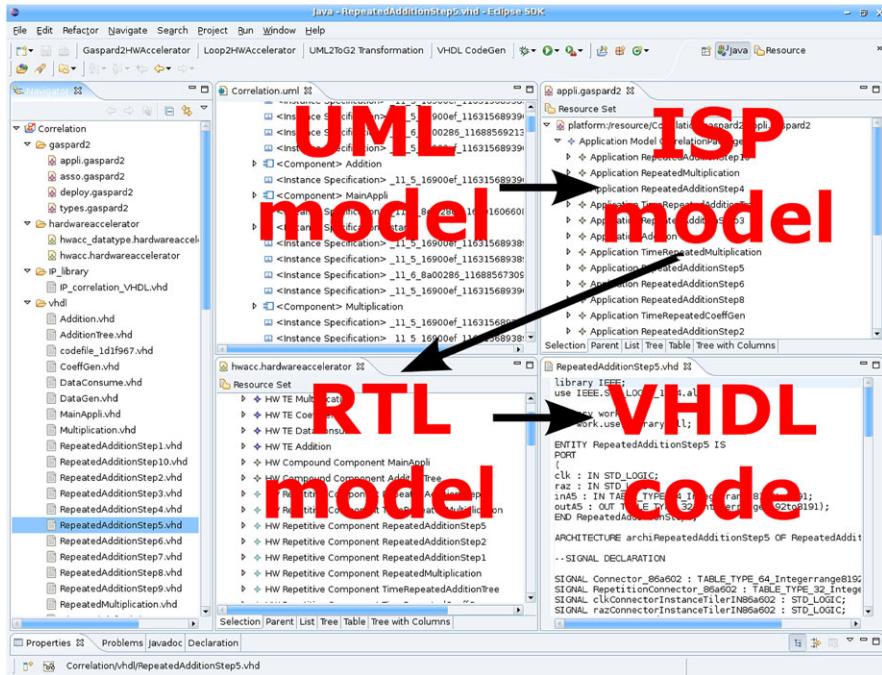


Fig. 10 Generating VHDL code from UML model

6.2 Generated Hardware Accelerator

From the UML model presented above, our flow automatically generates a hardware accelerator able to execute the correlation algorithm. For this purpose, an ISP model, an RTL model and a VHDL code are successively generated, as illustrated in Fig. 10.

The resulting VHDL code was synthesized for an Altera Stratix 2S60 FPGA using the Quartus tool from Altera. Figure 11(a) illustrates the 6 last stages of the tree topology and the corresponding reduction of data arrays from a pipeline stage to another. Figure 11(b) represents the synthesis results for the 8th pipeline stage. In the figure, marks 1 and 5 correspond to the input and the output ports, marks 2x and 4 point out the generated components that resolve data dependencies initially expressed with tilters. Finally, marks 3x represent the four elementary tasks that realize parallel execution of additions. By expressing the data dependencies through tilters in UML, our flow finds that data dependencies are efficiently implemented in hardware with shift register, as illustrated in Fig. 11(c).

The relevance of the HLS flow is evaluated through a comparison between a manually implemented hardware accelerator and an automatically generated one. Synthesis results are summarized in Table 1. As a first result, the latency of both accelerators remains strictly the same. The maximum frequency of the automatically generated hardware accelerator is 1.9% higher compared to the manually implemented one. The number of resources required to implement the accelerators are

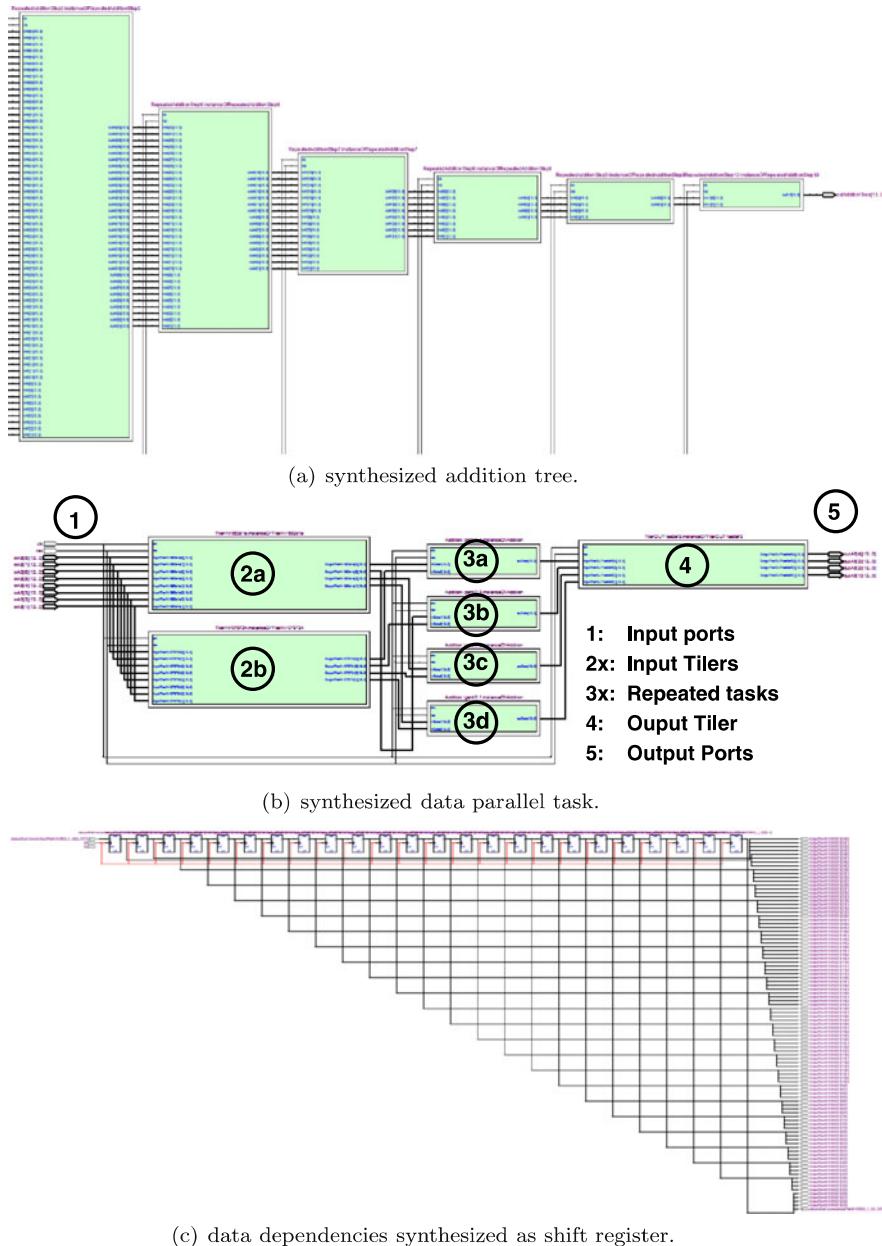


Fig. 11 Synthesis results of the generated hardware accelerator

also close to each other since only 10.7% additional resources are necessary for the automatically generated hardware accelerator. Whereas manually implementing the hardware accelerator takes weeks, modeling the UML model of the application only

Table 1 Synthesis results of the manually and automatically generated hardware accelerators

Version	Max frequency (in MHz)	Latency (cycles)	Used resources (ALUTs)	Development time
Manual	213	11	17006	weeks
Automated	217	11	18834	hours

takes a few hours. Hence, development time can be greatly shortened with the HLS flow at the price of using additional hardware resources.

7 Conclusion

This chapter advocates the use of the MDE methodology for high level synthesis. In order to demonstrate the benefits of MDE, we developed a model-based HLS flow. This flow relies on a precise definition of features, such as data parallelism and data dependencies, that are important for intensive signal processing applications. We have also shown that MDE provides key benefits to both users and designers of our HLS flow: users work in a standardized unified graphical environment and designers can easily extend and maintain the flow.

From applications modeled at a high abstraction level in UML, the flow automatically performs successive refinements and generates the corresponding VHDL code. Such refinements rely on a clear identification of concepts in the different abstraction levels and on a suitable decomposition of the model transformations into rules. We have validated the relevance of our HLS flow for correlation algorithms. The quality of results achieved by our HLS flow is almost as good (same latency with 10.7% more hardware resources) than that achieved with hand-coded VHDL. The flexibility and productivity advantages of high-level specifications and automated refinements more than outweigh the small degradation in the quality of results. Also, the performance of such an application-specific hardware accelerator is generally much higher than that achieved by software running on a (non-application-specific) processor.

MDE could also enable extensions to the flow to target other types of applications, other implementation languages or other abstraction levels.

References

1. Alanen M, Lilius J, Porres I, Truscan D, Oliver I, Sandstrom K (2006) Design method support for domain specific soc design. In: Proceedings of the fourth workshop on model-based development of computer-based systems and third international workshop on model-based methodologies for pervasive and embedded software (MBD-MOMPES '06), pp 25–32
2. Atitallah RB, Piel E, Niar S, Marquet P, Dekeyser J-L (2007) Multilevel MPSoC simulation using an MDE approach. In: IEEE international SoC conference (SoCC 2007), Hsinchu, Taiwan

3. Bastoul C (2004) Code generation in the polyhedral model is easier than you think. In: PACT'13 IEEE international conference on parallel architecture and compilation techniques, Juan-les-Pins, France, pp 7–16
4. Björklund D, Lilius J (2002) From UML behavioral descriptions to efficient synthesizable VHDL. In: Proceedings of the 20th IEEE Norchip conference
5. Boulet P (2007) Array-OL revisited, multidimensional intensive signal processing specification. Research report, RR-6113, INRIA
6. Chen P-S (1976) The entity-relationship model – toward a unified view of data. ACM Trans Database Syst 1(1):9–36
7. Coussy P, Gajski DD, Meredith M, Takach A (2009) An introduction to high-level synthesis. IEEE Des Test Comput 26(4):8–17
8. Coussy P, Morawiec A (eds) (2008) High-level synthesis: from algorithm to digital circuit. Springer, New York
9. Coyle FP, Thornton MA (2005) From UML to HDL: a model driven architectural approach to hardware-software co-design. In: Information systems: new generations conference (ISNG), pp 88–93
10. Czarnecki K, Helsen S (2003) Classification of model transformation approaches. In: Proceeding of OOPSLA workshop on generative techniques in the context of model driven architecture
11. Devos H, Beyls K, Christiaens M, Van Campenhout J, Stroobandt D (2006) From loop transformation to hardware generation. In: Proceedings of the 17th ProRISC workshop, Veldhoven, pp 249–255
12. Frigo J, Gokhale M, Lavenier D (2001) Evaluation of the streams-C C-to-FPGA compiler: an applications perspective. In: Proceedings of the 2001 ACM/SIGDA ninth international symposium on field programmable gate arrays (FPGA), pp 134–140
13. Glitia C, Boulet P (2008) High level loop transformations for multidimensional signal processing embedded applications. In: International symposium on systems, architectures, modeling, and simulation (SAMOS VIII), Samos, Greece
14. Guillou A-C, Quinton P, Risset T (2003) Hardware synthesis for multi-dimensional time. In: IEEE 14th international conference on application-specific systems, architectures and processors (ASAP '03), The Hague, The Netherlands, pp 40–51
15. Guo Z, Buyukkurt B, Najjar W, Vissers K (2005) Optimized generation of data-path from C codes for FPGAs. In: DATE '05: proceedings of the conference on design, automation and test in Europe. IEEE Comput Soc, Washington, pp 112–117
16. Gupta S, Dutt N, Gupta R, Nicolau A (2003) SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In: Intl conf on VLSI design, pp 461–466
17. Kangas T, Kukkala P, Orsila H, Salminen E, Hännikäinen M, Hämäläinen TD, Riihimäki J, Kuusilinna K (2006) UML-based multiprocessor SoC design framework. ACM Trans Embed Comput Syst 5(2):281–320
18. Lanusse P, Gérard S, Terrier F (1998) Real-time modeling with UML: the ACCORD approach. In: UML 98: beyond the notation, Mulhouse, France
19. Lo J, Eggers S, Levy H, Tullsen D (1996) Compilation issues for a simultaneous multithreading processor. In: Proceedings of the first SUIF compiler workshop, pp 146–147
20. Martin G, Lavagno L, Louis-Guerin J (2001) Embedded UML: a merger of real-time UML and co-design. In: Proceedings of the 9th international symposium on hardware/software codesign (CODES), pp 23–28
21. Milder P, Franchetti F, Hoe JC, Puschel M (2008) Formal datapath representation and manipulation for implementing DSP transforms. In: 2008 45th ACM/IEEE design automation conference, Piscataway, NJ, USA, pp 385–90
22. Nguyen KD, Sun Z, Thiagarajan PS, Wong W-F (2004) Model-driven SoC design via executable UML to SystemC. In: RTSS '04: proceedings of the 25th IEEE international real-time systems symposium (RTSS '04). IEEE Comput Soc, Washington, pp 459–468
23. Object Management Group (2007) A UML profile for MARTE. <http://www.omg.org/marte.org>
24. Object Management Group, Inc. (ed) (2005) (UML) profile for schedulability, performance, and time, version 1.1. <http://www.omg.org/technology/documents/formal/schedulability.htm>

25. Object Management Group, Inc. (ed) (2006) Final adopted OMG SysML specification. <http://www.omg.org/cgi-bin/doc?ptc/06-0504>
26. Püschel M, Moura JMF, Johnson J, Padua D, Veloso M, Singer B, Xiong J, Franchetti F, Gacic A, Voronenko Y, Chen K, Johnson RW, Rizzolo N (2005) SPIRAL: code generation for DSP transforms. Proc IEEE 93(2):232–275
27. Quadri IR, Meftali S, Dekeyser J-L (2009) From MARTE to dynamically reconfigurable FPGAs: introduction of a control extension in a model based design flow. Technical report, DART – INRIA Lille – Nord Europe – INRIA – CNRS: UMR8022 – Université des Sciences et Technologies de Lille - Lille I. <http://hal.archives-ouvertes.fr/inria-00365061/PDF/RR-6862.pdf>
28. Riccobene E, Scandurra P, Rosti A, Bocchio S (2006) A model-driven design environment for embedded systems. In: DAC '06: proceedings of the 43rd annual conference on design automation. ACM, New York, pp 915–918
29. Rieder M, Steiner R, Berthouzoz C, Corhay F, Sterren T (2007) Synthesized UML, a practical approach to map UML to VHDL. Rapid integration of software engineering techniques. Springer, Berlin
30. Rinker R, Carter M, Patel A, Chawathe M, Ross C, Hammes J, Najjar WA, Böhm W (2001) An automated process for compiling dataflow graphs into reconfigurable hardware. IEEE Trans Very Large Scale Integr Syst 9(1):130–139
31. Schmidt DC (2006) Model-driven engineering. IEEE Comput 39(2):41–47
32. Seidewitz E (2003) What models mean. IEEE Softw 20(5):26–32
33. Selic B (1998) Using UML for modeling complex real-time systems. In: LCTES '98: proceedings of the ACM SIGPLAN workshop on languages, compilers, and tools for embedded systems. Springer, London, pp 250–260
34. Thomas DE, Moorby PR (1998) The Verilog hardware description language, 4th edn. Kluwer Academic, Norwell
35. Vidal J, de Lamotte F, Gogniat G, Soulard P, Diguet J-P (2009) A co-design approach for embedded system modeling and code generation with UML and MARTE. In: design, automation and test in Europe conference and exhibition (DATE)

Generation of Hardware/Software Systems Based on CAL Dataflow Description

**Richard Thavot, Romuald Mosqueron,
Julien Dubois, and Marco Mattavelli**

Abstract This chapter presents a new development of rapid prototyping tools for system design based on data-flow specifications. In this context, the efficiency of tools for the automatic translation from the data-flow programs to C and/or HDL are assessed by means of two design cases. The chapter also introduces the new concept of the automatic synthesis of interfaces. Such generic interfaces are implemented by using an embedded microprocessor, which can support a large variety of interfaces already available as native IP libraries in the case of FPGA. The two design cases described here have been developed, tested and validated on different implementation platforms. The results of the assessment show that flexibility, genericity and generality are attractive features of the proposed interface implementation methodology approach.

1 Introduction

Nowadays, heterogeneous embedded systems platforms are composed by a variety of different types of computational units, digital signal processor (DSP), graphics processing unit (GPU), Central Processing Unit (CPU), dedicated co-processor, custom acceleration logic units or generic field-programmable gate array (FPGA) just to mention the most common building blocks. Heterogeneous platforms are more and more frequently used to support the implementation of complex processing applications. Nowadays, the tasks of developing, optimizing and mapping such complex application algorithms on heterogeneous platforms become more and more challenging and require new flexible and efficient methodologies. A key issue is therefore to provide new design methodologies for quick architecture prototyping.

R. Thavot (✉)

SCI-STI-MM, Ecole Polytechnique Fédérale de Lausanne, CH 1015 Lausanne, Switzerland
e-mail: richard.thavot@epfl.ch

In this perspective several steps need to be accomplished, and several issues need to be addressed by the design process, they can be summarized as: (1) define which partition of the algorithms will result in a more efficient implementation on a component of the heterogeneous platform, (2) find which partitioning schemes satisfy the constraints of the application and (3) how to translate the algorithm partitions into the form (i.e. language) compatible with the corresponding platform component. Several works have already addressed some of such issues [1–7]. However, only a few of them propose a complete solution for the design of heterogeneous platforms including software components, hardware components and their different interfaces. All approaches mentioned above propose to address the first issue by using a unique language, allowing for system specification at a high level of abstraction for both SW and HW components. For several reasons, an approach based on a data-flow language is proposed by the authors. This form of programming language presents several advantages versus the classical imperative sequential languages employed so far in System design. Data dependencies, task concurrency and parallelism are explicitly and directly represented by a graph in a visual and intuitive form where algorithms are encapsulated in data-flow components. For such propose the CAL actor language [1] is used. CAL is based on the asynchronous data-flow computation model. It provides many interesting features particularly attractive and appropriate for system modeling, the most important as mentioned above are encapsulation, explicit concurrency and composability. The second issue can be addressed by using a tool environment that supports different design space exploration stages and yields efficient mapping and partitioning of the high level algorithm specification on each component of the heterogeneous platform. An essential element of the tool environment is the inclusion, at the level of the unified computation model, of the architectural components of the heterogeneous platforms and of native library/IP components. A unique description allows the task partition to be delayed. A data-flow program written in CAL is composed of a set of independent actors, which consume data tokens from input ports, consume these tokens and generate tokens on output ports. Connections between output and input ports of actors are represented by oriented edges that represent the direction of the tokens flowing between two actors ports. CAL language explicitly shows how a complete design can in principle be partitioned on different platform components as well as the type of communication between them. Generic drivers have been designed to easily convert the program by including with the appropriate interfaces. A generic interface architecture has to be able to connect to different communication controllers. These drivers could be connected with their sub-layers to every IP-interface defined within the component library. Two types of drivers are necessary: one for communication interfaces and another for handling the communication with memories. In this chapter, a description of the communication interface driver for hardware components is provided, with a focus on the case of an FPGA with a soft-core. The chapter shows that the driver architecture is generic and can be used with different communication interfaces. The chapter is organized as follows: Sect. 2 presents the main objective, the language and the methodology for system design. Differences and similarities between the different approaches are discussed in this section. Section 3 highlights

the efficiency of the CAL synthesis to SW and HW languages by means of two design cases (an MPEG-4 simple profile decoder and a bar code decoder). Section 4 describes the role and functionality of the interface drivers. In Sect. 5 some examples of interface drivers are provided. Finally, Sect. 6 concludes this chapter.

2 Objectives and principles

This section describes the main objectives of this work and highlights the differences with some other approaches such as:

- UGH [5],
- CatapultC [4],
- GAUT [6],
- generation of embedded hardware/software from SystemC [7].

UGH methodology presents a similar approach to CAL data-flow to generate a hardware co-processor description by using a high level description language. However, this approach does not provide a full heterogeneous platform. Moreover, UGH requires three inputs: the algorithm description, a draft of data-paths and finally the required system's frequency. Unfortunately, interfaces between different coprocessors are not supported. Our methodology needs only two inputs: the algorithm description, hardware information(hardware/software partition, kind of interfaces, required system's frequencies). In our approach, draft of data-paths are intrinsically included in the algorithm specification. Catapult^{TMC} uses a unique formalism to describe the entire algorithm. Unfortunately, Catapult^{TMC} has a conceptual limitation. Models implemented in languages such as C that use a sequential programming approach, exhibit inherent limitations with data dependencies and task parallelisms. Moreover, the management of the interfaces is not supported. Similarly, GAUT methodology has the same limitation that Catapult^{TMC} to describe algorithm. GAUT exhibits data dependencies between operations via a derived from C/C++ compiler. As explained previously, our methodology does not need data dependencies exploration on an algorithm, because it is directly provided by the data-flow representation itself. This approach manages the point to point links via the LIS communication theory [8]. Nevertheless, GAUT approach is too dedicated to DSP applications.

Finally, “generation of embedded hardware/software from SystemC” approach is the closest methodology compared to the CAL actor approach. This methodology includes generation for software and hardware with management of interfaces. The SystemC description has to be refined at a bit accurate level to enable high synthesis performances to be obtained. The interface control must be fully detailed in SystemC whereas a CAL dataflow program hides any sort of low-level detail and controls. Moreover, the management of interfaces in SystemC provides point to point links whereas the CAL actor approach provides multi-point links. The CAL actor methodology [1] and management of interfaces are described below so as to highlight the advantages of such approach versus the state of the art approaches.

2.1 CAL Actor Language

A data-flow program written in CAL is composed by a set of independent “actors” [1, 2] and by the topology of their connections which constitute a network (see Fig. 1) and is specified using an XML dialect. An actor is a standalone entity which has its own internal state represented by a set of state variables. It performs computations by executing actions and it must have, at least, one action to perform computations. An action execution is modeled as a sequential atomic component which means that once started it cannot be interrupted until it finishes and no other action, of the same actor, can execute at the same time. An action is executed on the basis on the internal state of the actor and on the availability and values assumed by the tokens at the input ports. Each “actor” has a set of input and output ports through which it communicates with other actors by passing data tokens. In summary, an “actor” may consume tokens from inputs, may change the internal state and may produce tokens at the outputs by firing actions that execute one after the other inside each actor. As shown Fig. 1, an actor can be represented in two different ways: (1) a network which is a set of actors and a set of connections; (2) a set of actions, where only one action is selected to be executed according to the internal state machine. CAL actor language provides different mechanism that can be used for the scheduling and control the execution order of actions inside an actor. The XML based description of network of actors supports hierarchical system specifications. A network is simply the specification of the input and output ports connections defining the communication of data tokens between actors. The communication channels are constituted by FIFOs components of theoretically infinite size that preserve the order of the tokens without loss of data. By writing CAL networks, designers can only focus on the modeling of the dataflow system and do not need to care much about the low level of details to implement the communication between actors. This is done by the tools that synthesize SW and HW implementations. Such tools provide to the designer some control over communication parameters such as length of queues and the types of exchanged data in the final implementation. When a data-flow program is developed, it can be simulated using the OpenDataflow simulator [9] to check for the correct functionality.

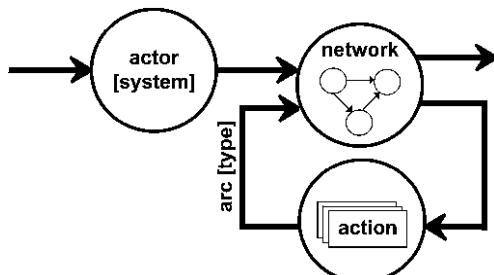


Fig. 1 CAL principles

2.2 Objectives: Unified Specification Formalism

The main objective of the approach based on writing CAL data-flow programs is to define and develop a methodology with a unique unified specification formalism for software and hardware components that can be mapped onto heterogeneous multi-component embedded platforms by direct synthesis of SW and HW. In a CAL-based design flow, the whole system is modeled and implemented in CAL. The partitioning between hardware and software can be easily modified since the same source is used for generating both components. Figure 2 details the components of the design flow: a CAL program which is the application algorithm is one input, whereas the architecture of the heterogeneous platform is another input of the design flow. First, the CAL program is validated by means of a behavioral simulation. Then, the second step is a pre-partitioning stage. This step defines, in accordance with the constraints and the architecture (component and interface), which actors will be mapped as hardware or software components. The methodology and optimization criteria for the partitioning of a network are not addressed in this work. The focus here is in the automatic synthesis of the interfaces and drivers that yield a correct and efficient implementation of any partitioning of a network onto a heterogeneous system. Given a CAL data-flow program partitioned on architecture components, attributes are added to each actor corresponding to the physical component and attributes are added at each arc connecting two different partitions and corresponding to the physical interface of the heterogeneous system. The attributes are represented in brackets in Fig. 1 and this one does not change the CAL dataflow program and its behavior. Using such information CAL synthesizers can include driver synthesis into the design according to the physical interfaces present in the platform. However, on the hardware side, the synthesis and integration of interfaces and their drivers is much more difficult to be obtained because there is no operating system (OS) that can manage interfaces automatically. Moreover, many edges connecting two partitions may share the same physical interface. An example of a partition that presents this case is shown Fig. 3. Once the choice of the partition of a network is done, each partition is translated into the appropriate implementation language by two synthesis tools: CAL2HDL [10] for FPGAs components and CAL2C for processors [11].

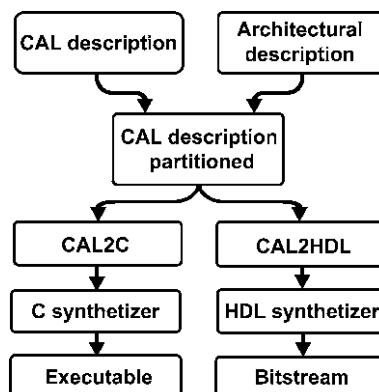
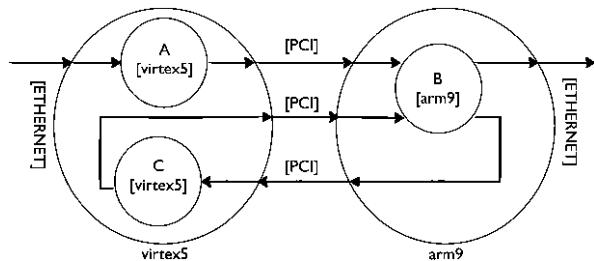


Fig. 2 Overview of the design flow based on CAL description

Fig. 3 Example of partitioning



A further step of the implementation process is to synthesize and to implement some actors by using specific IPs components that might be present as native optimized components libraries. The objective of this chapter is to describe how the automatic synthesis of interfaces and drivers has been implemented and with which results.

2.3 The Global Interfaces Methodology

One of the key issues in system design is to be able of producing working prototypes so as to validate system implementation architectures that satisfy the application requirements. The CAL approach with the synthesis tools enables to describe and to implement a complete design chain. The initial CAL data-flow program representing the behavior processing of the application, as explained in the previous section, can be partitioned into several components of type SW and HW that can be automatically translated to C and VHDL codes respectively, and assigned to physical SW and HW components of a heterogeneous platform. The efficiency of the automatic synthesis will be discussed in Sect. 5 with the results of two design cases. Two specific classes of actors can be defined to represent respectively the external interfaces and the external memory. The two resulting models can be used at different stages of the design validation. The model can be used to validate: (1) the functional CAL, (2) the CAL description obtained after merging with the architecture definition. For instance, the external interfaces can be described with a simple description: (1) bandwidth, (2) temporal interruption (period or randomly generated). Obviously, the model can be completed to be more conformat to the real physical interfaces. The external interfaces are directly exchanged with the physical link (for instance ETHERNET, RS232, PCI, ...). A completed automatic implementation requires handling the control of the different interfaces. A technological solution is proposed for the two partitions. The processor in charge of the SW partition can easily handle the control of the interface with a C driver. For the HW partition, a controller, as well as the driver, must also be generated to connect the interface with the HW partition. A unique driver structure is proposed to handle interfaces. These latter enables a large variety of interfaces to be integrated. Hardware driver is composed of a generic part to handle multi-connectivity between CAL actor partition and an interface then a peripheral IP specific to each physical interface. The structure of the driver is based on a micro-controller (Xilinx-Microblaze

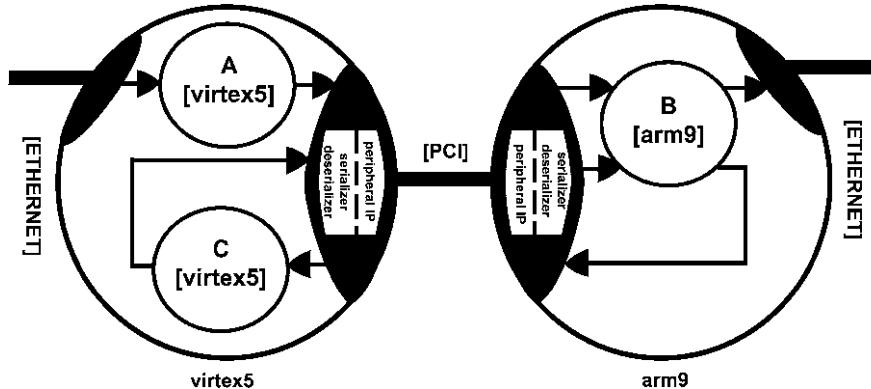


Fig. 4 Example of partitioning with the addition of the communication driver

or Altera-NiosII), which nowadays proposes a large variety of interface controllers. For instance, as presented in Fig. 4, the different drivers should be added for each physical external interface. These drivers are defined with two sub-layers: (1) serializer/deserializer, (2) Peripheral IP. The serializer/deserializer offers the possibility to connect several arcs for using a single interface. The peripheral IP is the controller of the physical interface. In summary, for the SW partition, a C driver is generated with the CAL2C tool. For the HW partition, a specific driver based on embedded micro-controller is generated with the CAL2HDL tool. The micro-controller program is dedicated to a specific interface via libraries.

3 Effectiveness of CAL2C and CAL2HDL

Translators have been tested by two different signal processing applications. The first is the MPEG-4 SP decoder [10–13], while the second is the code bar decoding [14–17] in postal sorting. Both applications focus on the flexibility of a CAL program specification and on the interests of high level of abstraction for a complete application specification.

3.1 First Design Case: MPEG-4 SP Decoder

MPEG-4 is a suite of standards composed of several “parts”, where each part standardizes various entities related to multimedia, such as audio, video, and file formats. MPEG-4 contains a number of features that allow it to compress video much more effectively than older standards and to provide more flexibility. Figure 5 shows the MPEG-4 Part 2 SP decoder which has been described via a data-flow model using CAL. This decoder is composed of three distinct functional components. The

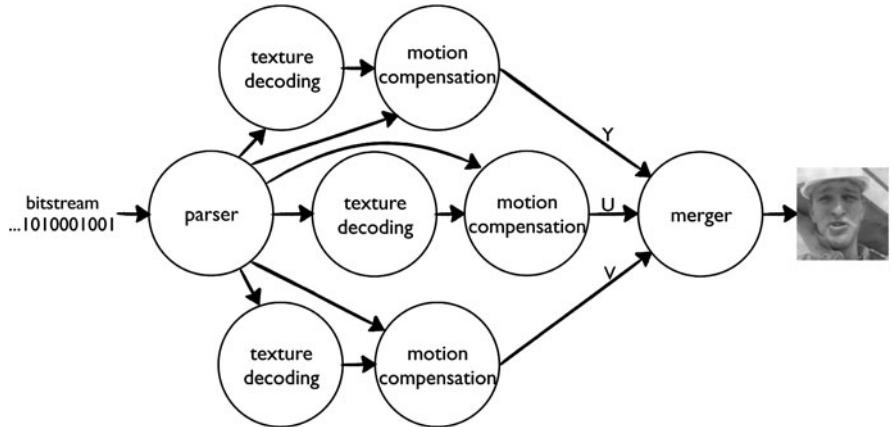


Fig. 5 MPEG-4 SP decoder described in CAL actor language

first includes the parser and merger actors. The parser partition the bitstream video in Y, U, V syntax elements streams, MB type, texture coefficients and associated motion vectors. The merger recomposes the video picture. The second part is used to decode the texture, and then the third part computes the motion compensation on the decoded texture. The MPEG-4 SP data-flow program is composed of 42 actor instantiations. Figure 6 compares the entire MPEG-4 SP decoder written in CAL and the same decoder directly described in HDL files. This graph shows a relevant advantage in term of development time and code size description for the MPEG-4 SP decoder CAL specification compared with the manually written HDL program (normalize to 1). This graph also shows an advantage for CAL program synthesized to HDL in term of area used by the FPGA and the obtained throughput. The entire MPEG-4 decoder can be directly generated from the CAL program using CAL2C [12] synthesis tool. Table 1 shows the different performances in terms of throughput between three MPEG-4 SP decoder implementations [10].

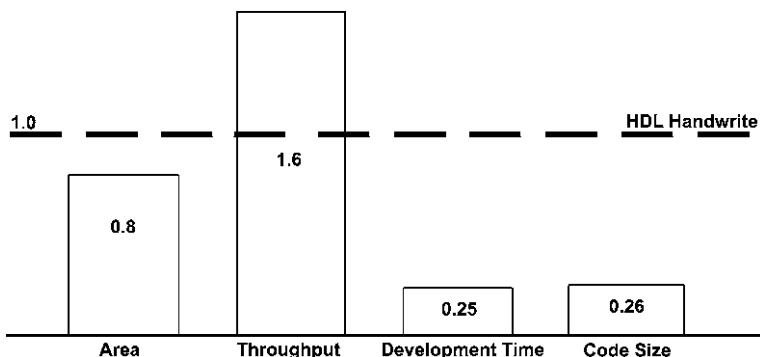


Fig. 6 Comparison of hardware performances between CAL generation and HDL handwrite for the MPEG-4 SP decoder

Table 1 Performances of the MPEG-4 decoder described in CAL, C and HDL generated

MPEG4SP decoder	Speed (kMB/S)	Code size (kSLOC)
CAL simulator	0.015	3.4
CAL2C	2	10.4
CAL2HDL	290	4

3.2 Second Design Case: the Code Bar Decoder

The goal of this application is to detect and decode bar codes on letters to enable automatic sorting at different stages of the logistic postal letter handling. Figure 7 shows the code bar decoder processing which has been specified in CAL. This processing system is composed by three distinct parts: the preprocessing, two processing (blobbing and code bar decoder), and then the manager stage. The first part applies some filtering operations so as to improve the picture quality and to identify the useful area for the bar code decoding. The third component is in charge of managing the flow of information among the different components. Figure 8 compares two architectures, one is synthesized from the CAL program, whereas the other is the manually-written HDL program (normalized to 1). The results show that the development time and the code size of the CAL data-flow program have a factor of four of advantage compared with the handwritten HDL. The figure also shows that in term of area the difference is minor, but the throughput is substantially different even if both satisfy the requirements of the application.

These two design cases show the effectiveness of the CAL synthesis to SW and HDL languages (CAL2HDL and CAL2C). Therefore, the HW/SW partitioning can be delayed to the last stage of the design flow. Consequently, the automatic insertions of interface controllers represent a key-point for efficient rapid prototyping.

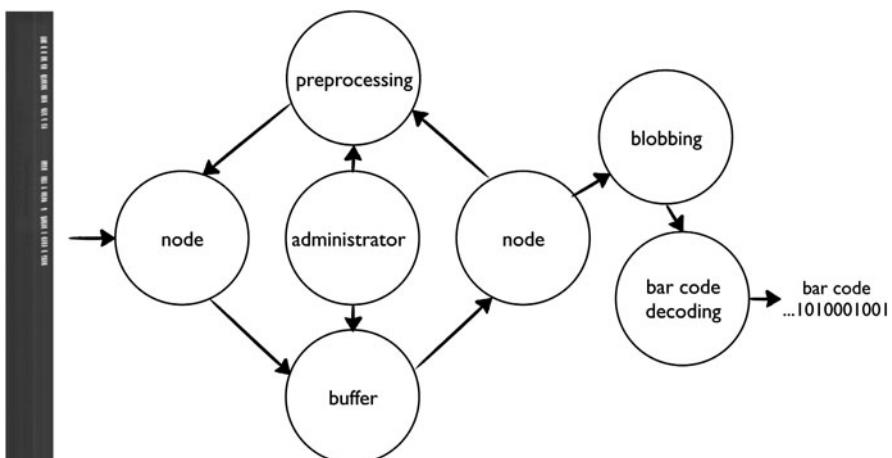


Fig. 7 Code bar decoder describes in CAL actor language

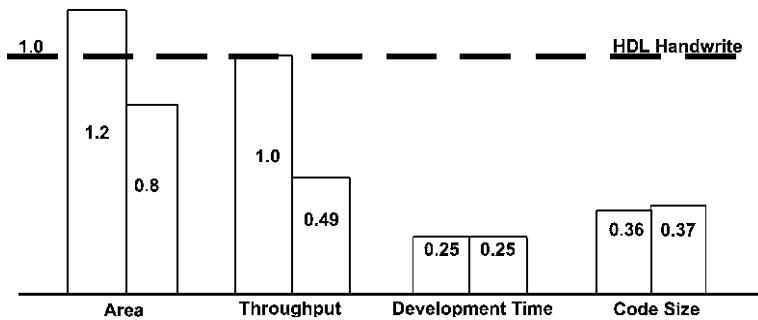


Fig. 8 Comparison of hardware performances between two CAL generation and HDL handwrite for the bar code decoder

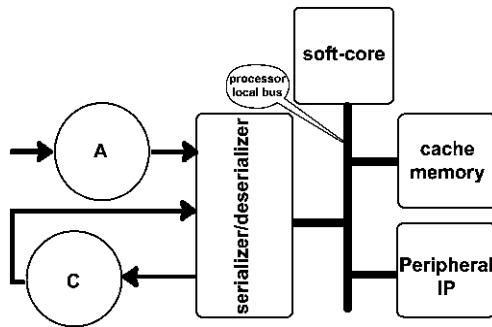
The final design results present more or less the same features, particularly the amount of silicon FPGA area used. This latter is closed to the handwritten description. But the development time and the size of the programs have been improved by about a factor of four.

4 Interfaces Driver Generation for Implementation

4.1 Driver Architecture Overview

A driver is used to directly connect an interface. The architecture is automatically generated to connect and handle virtually an unlimited number of devices. In the partitioned CAL specification, only parameters change to configure the right adapter as explained in Sect. 2.3. Figure 9 represents the driver on the hardware side. The generic driver is composed by a peripheral IP, a serializer/deserializer and a controller. The generic device connection allows connecting many communications with the serialization and the deserialization. A microprocessor or controller is required to enable and to support a large number of different interfaces. The soft-core microprocessor is an RISC architecture. It provides flexibility and scalability and it is customizable and fully implemented in programmable logic. This solution has already been proposed, for instance in [7, 18], to define and manage interfaces, nevertheless this approach has never been implemented from a dataflow description. The advantage of using a microprocessor is that most dedicated communication IPs already have been created and optimized by manufacturers. Moreover, this solution aims at suppressing or decreasing a loss of performances that has already been noticed for instance in [18] with PCI express. Therefore, our approach has been designed to support different manufacturer soft-processors: microBlaze for Xilinx and/or NiosII for Altera. Hence two different implementations using different FPGA technologies and interfaces are reported in this chapter and detailed in Sect. 5. The driver architecture is designed around the soft-core processor. Each component is

Fig. 9 Driver architecture
functional view



accessible by the Processor Local Bus which is managed by the soft-core. The microprocessor is connected on a multi-master bus to access the IP slaves. The soft-core uses robust and light-weight OS which are based on the pre-emptive real time multi-tasking operating system kernel for microprocessors. This type of OS allows having several concurrently running tasks called “threads” and allows having event flags, which suspend or run the thread according to their status. The architecture aims at supporting a large range of peripheral interfaces, therefore memory elements should be available. The architecture supports one internal soft-core memory and/or an external cache memory. The architecture is generated according to the selected peripheral. The architecture has a peripheral IP controller that is either built by the manufacturers or self-described. This approach, in our opinion, saves a large amount of the prototyping time nevertheless it retains the manufacturer’s IP performances. Our specific flow translator converts the tokens into a pile of data and vice versa. Piles of data are accessible from the processor local bus addresses and also its pile statuses. This conversion is realized by means of FIFOs for both directions. The driver adapter has also the possibilities of checking both FIFO statuses. A key contribution of our architecture is the ability of automatically handling data-flow multi-connectivity on the same peripheral. Indeed, several “CAL actor language arcs” can use one driver, if only one interface can be used for many tasks or transfers. For instance, and has shown previously in Fig. 4, the driver should manage two output arcs and an input arc. For this reason, the component named serializer/deserializer has a key position in our architecture. The driver must be able to serialize data from channels and must be able to deserialize data to the proper channel. Moreover the driver should manage deadlock e.g. if arcs resources are not available on the reception side then the respective arcs should be disconnected of the serializer/deserializer to keep a correct functionality of the system. For this reasons, next subsection focus on the description of the serializer/deserializer process. Two algorithms are tested within the serializer/deserializer module. Then the generic device connection is explained and the resulting efficiency and implementations resources are compared.

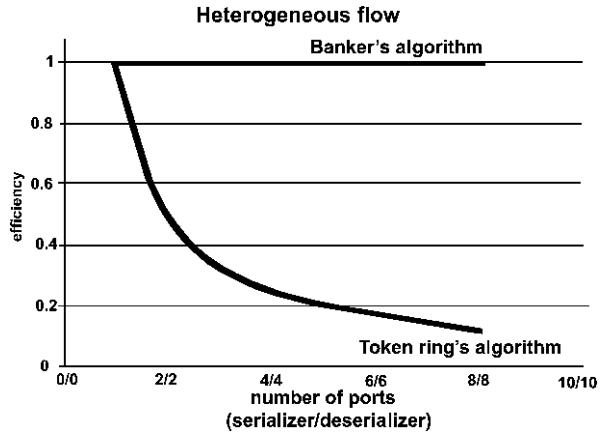
4.2 *Serialization and Deserialization Process*

Drivers are used to convert a token from any channel into a sequence of data transmitted across a network connection link. This allows automatic generation of the driver according to the number of “arcs” connected to the device. Even if it is written in CAL, it remains completely transparent because it will be instantiated during the pre-partitioning and the partitioning steps. Another advantage of developing a CAL data-flow program is that the successive re-design and improvements stages result easier. The serialization driver should add data such that the deserialization driver is able to redistribute the tokens on the right channels. Moreover, the arcs connected to the driver are not necessarily active at the same time, so then the driver must be able to adjust its consumption and its production of tokens automatically according to the arc activities. To solve this problem of random token presence on the arcs, two algorithms have been assessed to consume tokens according priorities. Both algorithms have the same functionality i.e. one algorithm could be on a partition and the second algorithm on the second partition. Then both algorithms have a feedback loop which handles deadlocks. The numbers of messages sent which manage deadlocks are calculated according the size of each FIFO of reception defined by a CAL actor arc. Smaller the FIFO size of reception is smaller the numbers of messages must be and vice-versa. The FIFO size is defined by attributes on the arc. The sent back message rules are each FIFO of reception sends back a message when the numbers of data consumed and produced by it is equal to the FIFO size. The First algorithm is the “Token ring’s algorithm”. This algorithm is based on the traditional “Round robin” i.e. each communication channel is on a turnstile. When a channel is selected and both FIFO of reception is not full and FIFO of emission is not empty. Then a message is sent with a command data and all the data available according the two FIFOs. Then the status of the FIFO sent are updated and the turnstile is pushed. When one of the two FIFOs of a channel is empty then nothing is sent and the turnstile is pushed. The second algorithm is the “Banker’s algorithm”. This algorithm is based on the Banker’s algorithm [19] used by operating system developed by Edsger Dijkstra. This one will directly select the channel with the highest priority. The priority is calculated according to the occupancy of the FIFO of emission, the vacancy of the FIFO of reception and the size of each FIFO. As the previous algorithm, the message is built with the same behavior.

4.2.1 Comparison of the Efficiency

The efficiency of both algorithms is different according the numbers of ports serialized, deserialized and the flow applied on each port (see Fig. 10). In this experiment, the resulting global bandwidth of the different ports should not obviously exceed the bandwidth of the physical interface. The efficiency is considered as optimum when no latency is observed on any port. The presented results have been obtained with the two following interfaces: PCI and Ethernet. The used CAL design bandwidth is always under the interface bandwidth so as to not perturb the measure of efficiency

Fig. 10 Efficiency versus the numbers of ports and the number of connections applied on each ports



of each algorithm. Figure 10 shows that with a homogeneous data-flow on each port then both algorithms have the same efficiency equal to 1 and do not depend of the port number. Contrariwise, with heterogeneous data-flow the efficiency of Token ring's algorithm decreases according to the numbers of ports. The features of the Banker's algorithm allow for maximum efficiency for any number of ports. However, the implementation complexity of the Banker's algorithm is higher than the Token ring's one.

4.2.2 Comparison of the Hardware Implementation

In this section, both algorithms of the serializer/deserializer have been synthesized to determine the numbers of slices and the maximum frequency available on two different XILINX FPGAs and an ALTERA FPGA (see Tables 2, 3, 4). In terms of slices and frequency, the token ring's algorithm achieves very good result. The slice doubles when the number of ports quadruples with a frequency more or less equivalent. Contrariwise the Banker's algorithm does not present the same performance. The number of slices increases linearly with the number of ports. Moreover the frequency for a high number of ports decreases sorely. Figure 11 represents the trend of each table according the number of ports. Despite a hardware resource overhead, the Banker's algorithm presents higher performances for a large number of connections (≤ 4). With a heterogeneous data-flow, this implementation can be considered

Table 2 Hardware performance for both algorithms on a Virtex 5

Algorithm	Numbers of ports	1/1	2/2	4/4	8/8
Token ring	Slices	354	421	750	819
	Frequency (MHz)	229	215	203	191
Banker	Slices	391	522	784	1284
	Frequency (MHz)	218	207	199	159

Table 3 Hardware performance for both algorithms on a Spartan 3

Algorithm	Numbers of ports	1/1	2/2	4/4	8/8
Token ring	Slices	396	500	659	954
	Frequency (MHz)	100	86	82	80
Banker	Slices	367	522	903	1907
	Frequency (MHz)	86	79	75	60

Table 4 Hardware performance for both algorithms on a Cyclone II

Algorithm	Numbers of ports	1/1	2/2	4/4	8/8
Token ring	Slices	606	807	963	1391
	Frequency (MHz)	97	79	67	49
Banker	Slices	481	730	1222	2497
	Frequency (MHz)	87	76	61	37

as an efficient alternative to less sophisticated solutions, such as the Token ring's algorithm. Both algorithms have been implemented to control the multi-connectivity on one interface.

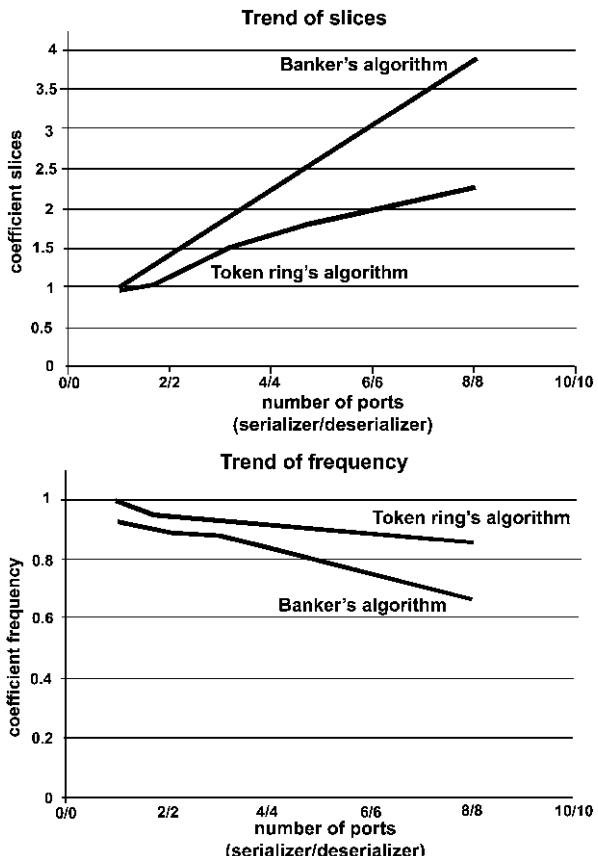
4.2.3 Algorithm Synthesis

Both algorithms are available in the prototyping framework and algorithms have been implemented to control the multi-connectivity on same interface. Until 4 ports, the Banker's algorithm represents an efficient solution to deal with heterogeneous data flow. The resource requires to implement this protocol is slightly similar to traditional algorithm as the Token ring algorithm. Beyond of 4 ports, the token ring's algorithm is a good way in term of slices and frequencies but this algorithm is completely inefficient if a heterogeneous flow is applied on the serializer/deserializer. Probably the designer is in charge to select the well algorithm according to his application requirements.

5 Design Cases with Interfaces Driver Generation

The drivers are used with simple CAL examples (at the level of partitioned CAL model), which sends and receives data from and to the FPGA. Examples use different interface with multi-connectivity to prove the flexibility and the genericity on this methodology. Examples follow the steps from partitioned CAL to bitstream generation. These examples have been tested and validated on different platforms. To verify the flexibility and the smooth operation of the generic hardware interface model, several implementations with communication bus has been performed.

Fig. 11 Slices and frequency trend as function of the numbers of ports



5.1 Ethernet Link

In this example a peripheral that is dedicated for the Ethernet protocol has been tested. To compare the flexibility of the driver interface, the system is implemented on two different FPGAs from two different manufacturers. The first example has been designed targeting the Altera family associated with an SMSC component. The same case can be applied to the Xilinx family. Nevertheless, Virtex 5 from Xilinx includes a specific core dedicated for Ethernet communication, which is the second case. Both designs need an external memory to use the light-weight implementation of the stack TCP/IP.

5.1.1 Ethernet on Cyclone II

The 32-bit embedded-processor NIOS II [20] is connected on a local processor bus named “Avalon”. The Avalon interface family defines interfaces for usage in both high-speed streaming and memory mapped applications. In the studied case, the

peripheral controller connected to this bus is the SMSC LAN91C111 device controller [21]. This external device is designed to facilitate the implementation of a third generation of Fast Ethernet connectivity solutions for embedded applications.

5.1.2 Ethenet Link on Virtex 5

The 32-bit Harvard RISC microprocessor that is used with the Virtex 5 is the microblaze [22]. The microblaze is connected on processor local bus named “PLB”. The device controller connected to the PLB is the TEMAC (Tri-Mode Ethernet MAC). The TEMAC is a configurable core ideally suited for using in networking equipment such as switches and routers. The customizable TEMAC core enables system designers to implement a broad range of integrated Ethernet designs, from low cost 10/100 Ethernet to higher performance 1 Gigabit ports.

5.2 PCI Link

In this example a peripheral that is dedicated for the PCI protocol has been tested. The proposed implementation is based on the driver that is described in Sect. 4. This implementation is compared with the direct implementation of a specific PCI controller. As described in Sect. 4, the driver is based on a microblaze microprocessor. The target technology is the Virtex 2 Pro FPGA from Xilinx. The peripheral connected on the PLB bus is the PLBV46 PCI Full Bridge. The proposed implementation is obtained with the automatic translation of the driver dedicated to this PCI configuration. The performance is identical for both designs, and therefore represents another important step to obtain an efficient rapid prototyping tool based on CAL data-flow programs.

6 Conclusion

This chapter has proposed a new methodology to generate system interfaces from a data-flow description. The methodology may be considered as a solution to yield system architecture exploration results as well as a key contribution to quick prototyping. Indeed the system’s architecture can be defined in CAL data-flow language and automatically translated into an efficient implementations. The generic driver proposed in this study enables the implementation of different interfaces. Moreover the generic driver can handle multi-connectivity employing two different algorithms. These algorithms can virtually manage up to 256 input/output “CAL actor arcs” with for each FIFO a depth range of 1 to 16,777,216 data. The structure of this driver enables to deal with bandwidth repartition of the different external interfaces. Generic driver methodology provides a high-degree of flexibility and robustness. The proposed methodology facilitate the design-flow of typical embedded systems.

In case of the occurrence of a lack of performance, a designer may use a native peripheral IP, and then include it in the generic peripheral driver. Moreover, the proposed driver methodology enables an easy changing of interfaces for exploring several mapping and design exploration options.

References

1. Eker J, Janneck J (2003) CAL language report. Tech. Rep. ERL Technical Memo UCB/ERL M03/48. University of California at Berkeley, December
2. Lucarz C, Mattavelli M, Wipliez M, Roquier G, Raulet M, Janneck JW, Miller ID, Parlour DB (2008) Dataflow/actor-oriented language for the design of complex signal processing systems. In: Workshop on design and architectures for signal and image processing (DASIP08), Bruxelles, Belgium, November
3. Bhattacharyya SS, Brebner G, Janneck JW, Eker J, von Platen C, Mattavelli M, Raulet M (2008) OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. ACM SIGARCH Comput Archit News Arch 36(5), Ronneby, Sweden, December
4. Bollaert T (2008) Catapult synthesis: a practical introduction to interactive C synthesis. In: High-level synthesis from algorithm to digital circuit. Springer, Berlin. 978-1-4020-8587-1 (HSL chap. 3)
5. Augé I, Pétrot F (2008) User guided high level synthesis. In: High-level synthesis from algorithm to digital circuit. Springer, Berlin. 978-1-4020-8587-1 (HSL chap. 10)
6. Coussy P, Chavet C, Bomei P, Heller D, Senn E, Martin E (2008) GAUT: A high-level synthesis tool for DSP applications. In: High-level synthesis from algorithm to digital circuit. Springer, Berlin. 978-1-4020-8587-1 (HSL chap. 9)
7. Ouadjaout S, Houzet D (2006) Generation of embedded hardware/software from SystemC. EURASIP J Embed Syst 2006:1–11, June
8. Carloni LP, McMillan KL, Sangiovanni-Vincentelli AL (2001) Theory of latencyinsensitive design, IEEE Trans Comput-Aided Des Integr Circuits Syst 20(9):18
9. Open DataFlow Sourceforge Project. <http://opendf.sourceforge.net/>
10. Janneck JW, Miller ID, Parlour DB (2008) Profiling dataflow programs. In: Reconfigurable video coding and processing (ICME08), Germany, Hannover, June
11. Roquier G, Wipliez M, Raulet M, Janneck JW, Miller ID, Parlour DB (2008) Automatic software synthesis of dataflow program: an MPEG-4 simple profile decoder case study. In: IEEE workshop on signal processing systems (SIPS08), Washington, USA
12. Janneck JW, Miller ID, Parlour DB, Mattavelli M, Lucarz C, Wipliez M, Raulet M, Roquier G (2008) Translating dataflow programs to efficient hardware: an MPEG-4 simple profile decoder case study. In: Design automation and test in Europe (DATE08), Munich, Germany
13. Lucarz C, Mattavelli M, Thomas-Kerr J, Janneck JW (2007) Reconfigurable media coding: a new specification model for multimedia coders. In: Signal processing systems (SIPS07), October
14. Thavot R, Mosqueron R, Alisafaei M, Lucarz C, Mattavelli M, Dubois J, Noel V (2008) Dataflow design of a co-processor architecture for image processing. In: Workshop on design and architectures for signal and image processing (DASIP08), Bruxelles, Belgium, November
15. Mosqueron R, Dubois J, Mattavelli M (2007) High performance embedded coprocessor architecture for CMOS imaging systems. In: Workshop on design and architectures for signal and image processing (DASIP07), Grenoble, France, November
16. Mosqueron R, Dubois J, Mattavelli M (2008) Smart camera with embedded coprocessor: a postal sorting application. In: Optical and digital image conference, Strasbourg, France, April, Proc SPIE, vol. 7000. SPIE, Bellingham
17. Dubois J, Mattavelli M (2003) Embedded coprocessor architecture for CMOS based image acquisition. ICIP IEEE Int Conf Image Process 2:591–594

18. Nassar F, Haase J, Grimm C, Nachtnebel H, Ghameshlou M (2008) Design and simulation of a PCI express based embedded system. In: IEEE Austrian workshop on microelectronics (AUSTROCHIP08), Linz, Austria, October
19. Dijkstra EW EWD108-Een algorithme ter voorkoming van de dodelijke omarming (in Dutch; algorithm for the prevention of the deadly embrace). <http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.pdf>
20. NIOS II Processor Reference Handbook. <http://www.altera.com/>
21. 10/100 Non-PCI Ethernet Single Chip MAC+PHY. <http://www.smsc.com/>
22. MicroBlaze Processor Reference Guide. <http://www.xilinx.com/>

Index

1D and 2D task placements, 121
1D placement of tasks, 130
1D task placement, 120, 124

A

ables, 124
acceleration factor, 231
accelerators, 270
adaptation, 81
adaptive grids, 36
algorithm-architecture adequacy, 197, 204, 206, 209, 210, 212, 213
algorithms and architecture adequacy, 197 alternative, 245
ANN for optimization problems, 121
ANNs, 121
anti-blooming system, 92
architecture exploration, 197, 198, 200, 202, 209–213
area constraint, 135, 136, 139
area fragmentation, 121
artificial neural networks (ANNs), 118, 119
automatic synthesis of interfaces, 275

B

Banker’s algorithm, 286–288
bitstream, 173, 174
bootstrap, 65
bottom level, 223–226, 230, 234
Branch and Bound, 161, 162, 165

C

CAL, 238
CAL actor language, 276, 278, 282, 283, 285
CCR, 231
classes of hardware tasks, 154
CMOS, 81–85, 87–90

code generation, 258, 260, 261, 268
combinatorial optimization, 156
communication contention, 217, 219, 221, 225–227, 232, 234
communication to computation ratio (*CCR*), 231
compositing, 30
configuration overhead, 145, 146, 153, 156, 159, 165, 167, 168
constrained optimization problems, 156
convergence, 132
convergences of the classical ANN and the RANN, 140
correlation algorithm, 268, 269
critical child, 217, 219, 227, 229–232, 234
Cube, 31
Cube-3, 31
Cube-4, 31

D

DAG, 218–223, 225, 226, 230, 231, 233, 234
data dependencies, 257, 262
data parallel, 262
data parallelism, 257, 263
data path, 264, 265, 268
data ready time (DRT), 222
data-flow language, 276, 290
dependency constraint, 135, 136, 138, 139
descriptor, 58
design space exploration, 238
device, 149
digital differential analyzer (DDA), 29, 30
directed acyclic graph (DAG), 217–219
distributed operating system, 197, 205
distributed real-time applications, 199
distributed RTOS, 197
DRA, 120

- DRT, 222, 229
 DSX, 62
 dynamic and partial reconfigurations, 118
 dynamic partial reconfiguration, 146, 167
 dynamic reconfiguration, 121
 dynamically reconfigurable accelerators, 117
 dynamically reconfigurable architecture, 131
- E**
 edge scheduling, 222, 228
 efficiency in RB utilization, 150
 embedded applications, 117
 embedded systems, 275, 290
 energy function, 121
 error rates, 180
 Ethernet, 179
 Ethernet communication, 289
 execution model, 264
 exploration of the architecture, 198
 exploring the architecture, 208
- F**
 field-programmable gate array (FPGA), 145
 fitting, 156, 158, 160, 162, 165
 FPGA, 171
- H**
 hardware accelerator, 253, 254, 264, 270, 271
 hardware architecture, 3, 5, 6, 16, 25
 hardware implementation of ANNs, 124
 hardware resource management, 150
 hardware task classes search, 151
 hardware task classification, 148, 153
 hardware task placement, 146, 149
 heterogeneity, 158
 heterogeneity of RBs in the device, 157
 heterogeneity on the device, 150
 heterogeneous, 164
 heterogeneous multiprocessor, 118
 heterogeneous multiprocessor architecture, 123, 124
 heterogeneous resources, 150
 heterogeneous Xilinx devices, 148
 hierarchy, 263
 high level of abstraction, 276, 281
 high level synthesis, 254
 high-level model, 213
 high-level modelling, 197–200, 204, 210, 213
 HLS, 254, 255
 homogeneous multiprocessor architectures, 123
 Hopfield artificial neural network, 121
 Hopfield model, 117, 121, 122, 124, 132
- I**
 implementation results of the RANN, 136
 input pattern, 260
 integrated circuit, 86
 integration time, 81–85, 88–90, 92
 intensive signal processing applications, 268
 intensive signal processing (ISP), 253
 interband, 3–7, 9–13, 16, 25
 interface controllers, 281, 283
 interface implementation methodology approach, 275
 internal representations, 255
 internal structure of a neuron, 138
 internal structure of one neuron, 137
 interprocessor communication, 238
- K**
 Kahn process networks, 54
 kd-tree, 28
- L**
 LAN, 178
 list scheduling, 217, 219, 225–227, 229–231, 234
 list scheduling heuristic, 231
 loop transformations, 263
 lossless compression, 4, 5
 low-cost, 81, 84
 lwIP, 187
 Lyapunov function, 121
- M**
 mapping, 163
 mapping of hardware tasks, 156
 marching cubes, 31
 MARTE, 258
 MARTE profile, 261
 MARTE standard profile, 258
 mathematical model, 156
 memspaces, 65
 metamodel, 258, 259
 micro-controller, 280, 281
 minimizing configuration, 168
 mobile robot application, 197, 208, 209, 211, 213
 model, 258, 259
 model and metamodel, 259
 model driven engineering, 255, 258
 model transformation, 258–260
 modeling of placement problem, 156
 monoprocessor architecture, 122, 123
 MPEG, 238
 MRI, 47

multi-connectivity, 280, 288, 290
multi-resolution, 28
multi-writer/multi-reader, 54
multiprocessor, 238
multispectral image, 3, 4, 7, 16

N

nD-AP cache, 32, 33
network-on-chip, 57
node priorities, 219, 223, 225, 229–231, 234
node priorities (top level and bottom level),
 217
node scheduling, 222

O

octree, 36
off-line strategy, 148
operating system, 118
optimization, 81
optimization problems, 119
optimized internal structure of a neuron, 138
OS model, 211, 212

P

parallel embedded system, 217, 218, 233, 234
parallelism, 238
partial reconfiguration, 165
partial run-time reconfiguration, 148, 156, 168
partially reconfigurable hardware devices, 145
partitioning, 154, 158, 276, 279, 283, 286
partitioning/fitting, 160, 162, 165
partitioning/fitting problem, 156
partitioning/fitting resolution, 160
PCI protocol, 290
PET, 47
PFair, 120, 121
PFair algorithm, 120
phase-locked propagation, 40
placement, 145
placement of hardware tasks, 146, 154
placement problem, 156, 158, 161, 162
pre-fetching, 45, 46
problem of placement, 156
profile, 257

Q

quasi-static scheduling, 238

R

RANN, 142
RANN architecture, 130
RANN convergence, 131, 134
RANN convergence complexity, 135
RANN hardware implementations, 139

RANN implementation, 140
RANN model, 131
RANN principle, 127
RANN structure, 127, 129, 130
rapid prototyping, 275, 283, 290
ray, 28
 shooting, 28
 tracing, 28
real-time systems, 120
reconfigurable ANN, 124
reconfigurable artificial neural network
 (RANN), 117, 118, 120, 140, 142
reconfigurable hardware device, 150, 164
reconfigurable region, 149
reconfigurable schedule tick, 130, 142
reconfigurable SoC, 142
reconfigurable system-on-chip (RSoC), 118
reconfigurable video coding, 237
reconfiguration, 172, 175
reconfiguration granularity, 150, 164
reconfiguration schedule tick, 134
recursive complete algorithms, 160
resolution of placement problem, 160, 161
resource constraint scheduling, 118
resource efficiency, 154, 159, 166, 168
resource inefficiency, 150
resource utilization, 146, 168
resource waste, 145, 146, 159, 167
RISC, 284, 290
Round robin, 286
route, 221, 222, 229
RSoC platform, 119
RTOS are modeled, 200
RTOS distribution, 198, 200, 202–208, 210,
 213
RTOS model, 197–202, 204–206, 208, 210,
 213
RTOS service distribution, 197
rule, 259, 265
rule input pattern, 259
rule output pattern, 259, 260

S

schedule and placement, 118
schedule length, 217, 222, 227, 230, 234
scheduling, 230
SDF, 218, 231
SDR, 172
sequential atomic component, 278
serializer/deserializer, 281, 284, 285, 287, 288
signature, 259, 260
sinogram, 47
soft-core, 276, 284, 285

- spatio-temporal, 124
spatio-temporal scheduling, 118, 119, 125, 138
static scheduling, 121
synchronous dataflow (SDF), 218, 238
synthesizers, 279
system-on-chip, 117
SystemC, 197, 198
SystemC kernel, 205
SystemC modelling, 197–206, 212
SystemC transactions, 202
- T**
task and communication graph, 56
task dependencies, 127, 130
task migrations, 121
task placement, 117
task scheduling, 117, 217–219, 221, 223
TCP, 185
TCP/IP, 186
temporal and spatial task scheduling, 120
three-level hardware task placement, 164, 168
tiler, 257, 262
time complexity, 219, 229, 234
TLM, 199
token ring’s algorithm, 286–288
top level, 223–226, 230, 234
- top/bottom levels, 223
topology graph, 219, 220, 222
transaction level modelling (TLM), 198, 202, 203
transformation rules, 260
transformations, 255
two-level fitting, 156
- U**
UDP, 185
UML, 257, 258, 261
UML model, 261, 270
- V**
vision sensors, 87
voxel, 28
VoxelCache, 31, 48
- W**
WCETs, 123
worst case execution time, 123
- X**
Xilinx partial reconfiguration, 153
Xilinx SRAM-based FPGA, 145
Xilinx Virtex 5 FPGA, 150