

LNCS 2913

**Timothy Mark Pink
Viktor K. Prasanna**

High Per Computi HiPC 200

**10th International Conference
Hyderabad, India, December 2003
Proceedings**

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2913

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Timothy Mark Pinkston Viktor K. Prasanna (Eds.)

High Performance Computing – HiPC 2003

10th International Conference
Hyderabad, India, December 17-20, 2003
Proceedings



Springer

Series Editors

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

Volume Editors

Timothy Mark Pinkston
Viktor K. Prasanna
University of Southern California
Department of Electrical Engineering
Los Angeles, CA 90089-2562, USA
E-mail: tpink@charity.usc.edu
prasanna@usc.edu

Cataloging-in-Publication Data applied for

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

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

CR Subject Classification (1998): D.1-4, C.1-4, F.1-2, G.1-2

ISSN 0302-9743

ISBN 3-540-20626-4 Springer-Verlag Berlin Heidelberg New York

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

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10971895 06/3142 5 4 3 2 1 0

Message from the Program Chair

Welcome to the proceedings of the 10th International Conference on High Performance Computing, HiPC 2003. This year, we were delighted to have 164 papers submitted to this conference from 20 different countries, including countries in North America, South America, Europe, Asia, and the Middle East. Of these, 48 papers from 11 different countries were accepted for presentation at the conference and publication in the conference proceedings. Less than 30% of the submitted papers were accepted this year, with each paper receiving a minimum of three reviews. Although the selection process was quite competitive, we were pleased to accommodate 10 (parallel) technical sessions of high-quality contributed papers. In addition to the contributed paper sessions, this year's conference also featured a poster session, an industrial track session, five keynote addresses, five tutorials and seven workshops.

It was a pleasure putting this program together with the help of five excellent Program Vice-Chairs and the 65-person Program Committee. Although the hard work of all the program committee members is deeply appreciated, I especially wish to acknowledge the dedicated effort made by the Vice-Chairs: Rajiv Gupta (Architecture), Jose Moreira (System Software), Stephan Olariu (Communication Networks), Yuanyuan Yang (Algorithms), and Xiaodong Zhang (Applications). Without their help and timely work, the quality of the program would not have been as high nor would the process have run so smoothly. I also wish to thank the other members of the supporting cast who helped in putting together this program, including those who organized the keynotes, tutorials, workshops, poster session, and industrial track session, and those who performed the administrative functions that were essential to the success of this conference. The work of Sushil Prasad in putting together the conference proceedings is also acknowledged, as well as the support provided by Jeonghee Shin in maintaining the CyberChair on-line paper submission and evaluation software. Last, but certainly not least, I express heartfelt thanks to our General Co-chair, Viktor Prasanna, for all his useful advice and for giving me the opportunity to serve as the program chair of this conference. This truly was a very rewarding experience for me.

I trust you find this proceedings volume to be as informative and stimulating as we endeavored to make it. If you attended HiPC 2003, I hope you found time to enjoy the rich cultural experience provided by this interesting city of Hyderabad, India!

December 2003

Timothy Mark Pinkston

Message from the Steering Chair

It was my pleasure to welcome attendees to the 10th International Conference on High-Performance Computing and to Hyderabad, an emerging center of IT activities in India.

We are indebted to Timothy Pinkston for his superb efforts as program chair in organizing an excellent technical program. We received a record number of submissions this year. Over the past year, I discussed the meeting details with Timothy. I am grateful to him for his thoughtful inputs.

Many volunteers helped to organize the meeting. In addition, I was glad to welcome Rajesh Gupta as Keynote Chair, Atul Negi as Student Scholarships Chair, and Sushil Prasad as Proceedings Chair. I look forward to their contributions for the continued success of the meeting series. Sushil Prasad did an excellent job in bringing out these proceedings. Kamal Karlapalem assisted us with local arrangements at IIIT, Hyderabad. Dheeraj Sanghi took on the responsibility of focussed publicity for the meeting within India.

Vijay Keshav of Intel India, though not listed as a volunteer, provided me with many pointers for bringing the India-based high-performance computing vendors to the meeting.

I would like to thank M. Vidyasagar for agreeing to host the meeting in Hyderabad and for his assistance with the local arrangements.

Continuing the tradition set at last year's meeting, several workshops were organized by volunteers. These workshops were coordinated by C.P. Ravikumar. He also volunteered to put together the workshop proceedings, and Sushil Prasad assisted him in this.

I would like to offer my special thanks to A.K.P. Nambiar for his continued efforts in handling financial matters in India. He has been associated with the meeting since its beginning. He has acted as Finance Cochair for 10 years and also provided me with invaluable inputs over the years in resolving meeting-related issues. He has expressed his desire to retire from his role as Finance Cochair.

B. Ramachandra of Software Technology Park, Bangalore has graciously agreed to take on the responsibility of Finance Cochair for 2004.

Major financial support for the meeting was provided by several leading IT companies. I would like to thank the following individuals for their support:

N.R. Narayana Murthy, Infosys; Venkat Ramana, Hinditron Infosystems; Shubhra Roy, Intel India; and Uday Shukla, IBM India.

The meeting has very limited financial resources. I would like to thank the keynote speakers for their efforts to come to the meeting in spite of our limited financial support.

Finally, I would like to thank Henryk Chrostek, Sumit Mohanty, and Animesh Pathak at USC and Rohini Bhide at the Taj Krishna for their assistance over the past year.

December 2003

Viktor K. Prasanna

Message from the Vice General Chair

It was a pleasure to invite attendees to Hyderabad – the City of Pearls – and the 10th International Conference on High-Performance Computing. It was an honor and a pleasure to be able to serve the international community by bringing together researchers, scientists, and students, from academia and industry, to this meeting in Hyderabad, the capital city of Andhra Pradesh and a city that is fast emerging as India's information technology center.

First let me recognize **Manish Parashar** for his help publicizing this conference, and **Sushil K. Prasad** for serving as the publications chair. **Srinivas Aluru** did an excellent job organizing the tutorials presented by leading experts. HiPC 2003 included seven tutorials in areas likely to be at the forefront of high-performance computing in the next decade, such as mobile, ad hoc, and sensor networks, information security, and application areas in sensors, multimedia, and iterative methods.

I wish to thank all of the conference organizers and volunteers for their contributions to making HiPC 2003 a great success. I would especially like to thank the general co-chairs, **Viktor K. Prasanna** and **M. Vidyasagar**, for their enormous contributions steering and organizing this meeting. Their leadership and dedication is remarkable. It is to their credit that this meeting has become the premier international conference for high-performance computing. Special thanks are also due to the program chair, **Timothy Pinkston**, for his hard work assembling a high-quality technical program that included contributed and invited papers, an industrial track, keynote addresses, tutorials, and several workshops.

December 2003

David A. Bader

Conference Organization

General Co-chairs

Viktor K. Prasanna, University of Southern California
M. Vidyasagar, Tata Consultancy Services

Vice General Chair

David A. Bader, University of New Mexico

Program Chair

Timothy Pinkston, University of Southern California

Program Vice-chairs

Algorithms

Yuanyuan Yang, State University of New York at Stony Brook

Applications

Xiaodong Zhang, National Science Foundation

Architecture

Rajiv Gupta, University of Arizona

Communication Networks

Stephan Olariu, Old Dominion University

Systems Software

José E. Moreira, IBM T.J. Watson Research Center

Steering Chair

Viktor K. Prasanna, University of Southern California

Workshops Chair

C.P. Ravikumar, Texas Instruments India

Poster/Presentation Chair

Rajkumar Buyya, The University of Melbourne

Scholarships Chair

Atul Negi, University of Hyderabad, India

Finance Co-chairs

Ajay Gupta, Western Michigan University
A.K.P. Nambiar, Software Technology Park, Bangalore

Tutorials Chair

Srinivas Aluru, Iowa State University

Awards Chair

Arvind, MIT

Keynote Chair

Rajesh Gupta, University of California, San Diego

Industry Liaison Chair

Sudheendra Hangal, Sun Microsystems

Publicity Chair

Manish Parashar, Rutgers, State University of New Jersey

Publications Chair

Sushil K. Prasad, Georgia State University

Steering Committee

Jose Duato, Universidad Politecnica de Valencia, Spain

Viktor K. Prasanna, University of Southern California, Chair

N. Radhakrishnan, US Army Research Lab

Sartaj Sahni, University of Florida

Assaf Schuster, Technion, Israel Institute of Technology, Israel

Program Committee

Algorithms

Mikhail Atallah, Purdue University

Michael A. Bender, State University of New York at Stony Brook

Xiaotie Deng, City University of Hong Kong

Ding-Zhu Du, National Science Foundation

Qianping Gu, Simon Fraser University

Hong Jiang, University of Nebraska-Lincoln

Ran Libeskind-Hadas, Harvey Mudd College

Koji Nakano, Japan Advanced Institute of Science and Technology

Yavuz Oruc, University of Maryland at College Park

Arnold L. Rosenberg, University of Massachusetts at Amherst

Christian Scheideler, Johns Hopkins University

Jinwoo Suh, University of Southern California/ISI

Albert Y. Zomaya, University of Sydney

Applications

Srinivas Aluru, Iowa State University

Randall Bramley, Indiana University

Jack Dongarra, University of Tennessee
Craig Douglas, University of Kentucky and Yale University
Ananth Grama, Purdue University
David Keyes, Old Dominion University
Xiaoye Li, Lawrence Berkeley National Laboratory
Aiichiro Nakano, University of Southern California
P.J. Narayanan, Intl. Institute of Information Technology, Hyderabad
Yousef Saad, University of Minnesota
Eric de Sturler, University of Illinois at Urbana-Champaign
Xian-He Sun, Illinois Institute of Technology
Xiaoge Wang, Tsinghua University
Li Xiao, Michigan State University

Architecture

Prith Banerjee, Northwestern University
Sandhya Dwarkadas, University of Rochester
Manoj Franklin, University of Maryland
Kanad Ghose, State University of New York, Binghamton
Daniel Jimenez, Rutgers University
Mahmut Kandemir, Pennsylvania State University
Olav Lysne, University of Oslo
Avi Mendelson, Intel, Israel
Dhabaleswar Panda, Ohio State University
Fabrizio Petrini, Los Alamos National Laboratory
Antonio Robles, Polytechnic University of Valencia
Andre Seznec, IRISA, France
Per Stenstrom, Chalmers University of Technology
David Whalley, Florida State University
Jun Yang, University of California, Riverside

Communication Networks

Marco Conti, CNUCE/CNR Pisa
Abhay Karandikar, Indian Institute of Technology, Mumbai
Victor Leung, University of British Columbia
Cauligi Raghavendra, University of Southern California
Dheeraj Sanghi, Indian Institute of Technology, Kanpur
Pradip Srimani, Clemson University
Mani Srivastava, University of California, Los Angeles
Ivan Stojmenovic, University of Ottawa
Jie Wu, Florida Atlantic University
Jingyuan Zhang, University of Alabama
Albert Y. Zomaya, University of Sydney

Systems Software

Gianfranco Bilardi, University of Padua
Rahul Garg, IBM India Research Laboratory
Hironori Kasahara, Waseda University
Barney Maccabe, University of New Mexico
Rajib Mall, Indian Institute of Technology, Kharagpur
Sam Midkiff, Purdue University
Edson Midorikawa, University of Sao Paulo
Michael Phillipsen, Friedrich-Alexander-University
Lawrence Rauchwerger, Texas A&M University
Dilma Da Silva, IBM T.J. Watson Research Center
Anand Sivasubramaniam, Pennsylvania State University
Yanyong Zhang, Rutgers University

National Advisory Committee

R.K. Bagga, DRDL, Hyderabad
N. Balakrishnan, SERC, Indian Institute of Science
Ashok Desai, Silicon Graphics Systems (India)
Kiran Deshpande, Mahindra British Telecom
H.K. Kaura, Bhabha Atomic Research Centre
Hans H. Kafka, Siemens Communication Software
Ashish Mahadwar, PlanetAsia
Susanta Misra, Motorola India Electronics
Som Mittal, Digital Equipment (India)
B.V. Naidu, Software Technology Park, Bangalore
N.R. Narayana Murthy, Infosys Technologies
S.V. Raghavan, Indian Institute of Technology, Chennai
V. Rajaraman, Jawaharlal Nehru Centre for Advanced Scientific Research
S. Ramadorai, Tata Consultancy Services, Mumbai
K. Ramani, Future Software
S. Ramani, Hewlett-Packard Labs India
Karthik Ramarao, Hewlett-Packard (India)
Kalyan Rao, Satyam Computer Services
S.B. Rao, Indian Statistical Institute
H. Ravindra, Cirrus Logic
Uday S. Shukla, IBM Global Services India
U.N. Sinha, National Aerospace Laboratories

Workshop Organizers

Workshop on Bioinformatics and Computational Biology

Co-chairs

Srinivas Aluru, Iowa State University
M. Vidyasagar, Tata Consultancy Services

Workshop on Cutting Edge Computing

Co-chairs

Uday S. Shukla, IBM Software Lab
Rajendra K. Bera, IBM Software Lab

Workshop on Soft Computing

Chair

Suthikshn Kumar, Larsen and Toubro Infotech

Trusted Internet Workshop

Co-chairs

G. Manimaran, Iowa State University
C. Siva Ram Murthy, Indian Institute of Technology, Chennai

Workshop on Autonomic Applications

Co-chairs

Manish Parashar, Rutgers University
Salim Hariri, University of Arizona

Workshop on E-Science (Grid Computing and Science Applications)

Co-chairs

Dheeraj Bhardwaj, Indian Institute of Technology, Delhi
Simon C.W. See, Sun Microsystems, and Nanyang Technological University

Workshop on Embedded Systems for Media Processing

Co-chairs

S.H. Srinivasan, Satyam Computers
Ravi Amur, Satyam Computers

Table of Contents

Keynote Address

- Life's Duplicities: Sex, Death, and Valis 1
Bud Mishra

Session I – Performance Issues and Power-Aware

Architectures

Chair: Rajeev Kumar

- Performance Analysis of Blue Gene/L Using Parallel Discrete Event Simulation 2
Ed Upchurch, Paul L. Springer, Maciej Brodowicz, Sharon Brunett, T.D. Gottschalk

- An Efficient Web Cache Replacement Policy 12
A. Radhika Sarma, R. Govindarajan

- Timing Issues of Operating Mode Switch in High Performance Reconfigurable Architectures 23
Rama Sangireddy, Huesung Kim, Arun K. Soman

- Power-Aware Adaptive Issue Queue and Register File 34
Jaume Abella, Antonio González

- FV-MSB: A Scheme for Reducing Transition Activity on Data Buses 44
Dinesh C. Suresh, Jun Yang, Chuanjun Zhang, Banit Agrawal, Walid Najjar

Session II – Parallel/Distributed and Network

Algorithms

Chair: Javed I. Khan

- A Parallel Iterative Improvement Stable Matching Algorithm 55
Enyue Lu, S.Q. Zheng

- Self-Stabilizing Distributed Algorithm for Strong Matching in a System Graph 66
Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, Pradip K. Srimani

- Parallel Data Cube Construction: Algorithms, Theoretical Analysis, and Experimental Evaluation 74
Ruoming Jin, Ge Yang, Gagan Agrawal

Efficient Algorithm for Embedding Hypergraphs in a Cycle	85
<i>Qian-Ping Gu, Yong Wang</i>	

Mapping Hypercube Computations onto Partitioned Optical Passive Star Networks	95
<i>Alessandro Mei, Romeo Rizzi</i>	

Keynote Address

The High Performance Microprocessor in the Year 2013: What Will It Look Like? What It Won't Look Like?	105
<i>Yale Patt</i>	

Session III – Routing in Wireless, Mobile, and Cut-Through Networks

Chair: *Pradip K Srimani*

FROOTS – Fault Handling in Up*/Down* Routed Networks with Multiple Roots	106
<i>Ingebjørg Theiss, Olav Lysne</i>	

Admission Control for DiffServ Based Quality of Service in Cut-Through Networks	118
<i>Sven-Arne Reinemo, Frank Olaf Sem-Jacobsen, Tor Skeie, Olav Lysne</i>	

On Shortest Path Routing Schemes for Wireless Ad Hoc Networks	130
<i>Subhankar Dhar, Michael Q. Rieck, Sukesh Pai</i>	

A Hierarchical Routing Method for Load-Balancing	142
<i>Sangman Bak</i>	

Ring Based Routing Schemes for Load Distribution and Throughput Improvement in Multihop Cellular, Ad hoc, and Mesh Networks	152
<i>Gaurav Bhaya, B.S. Manoj, C. Siva Ram Murthy</i>	

Session IV – Scientific and Engineering Applications

Chair: *Gagan Agrawal*

A High Performance Computing System for Medical Imaging in the Remote Operating Room	162
<i>Yasuhiro Kawasaki, Fumihiko Ino, Yasuharu Mizutani, Noriyuki Fujimoto, Toshihiko Sasama, Yoshinobu Sato, Shinichi Tamura, Kenichi Hagiwara</i>	

Parallel Partitioning Techniques for Logic Minimization Using Redundancy Identification	174
<i>B. Jayaram, A. Manoj Kumar, V. Kamakoti</i>	

Parallel and Distributed Frequent Itemset Mining on Dynamic Datasets	184
<i>Adriano Veloso, Matthew Eric Otey, Srinivasan Parthasarathy, Wagner Meira Jr.</i>	

A Volumetric FFT for BlueGene/L	194
<i>Maria Eleftheriou, José E. Moreira, Blake G. Fitch, Robert S. Germain</i>	

A Nearly Linear-Time General Algorithm for Genome-Wide Bi-allele Haplotype Phasing	204
<i>Will Casey, Bud Mishra</i>	

Keynote Address

Energy Aware Algorithm Design via Probabilistic Computing: From Algorithms and Models to Moore's Law and Novel (Semiconductor) Devices	216
<i>Krishna V. Palem</i>	

Session V – System Support in Overlay Networks, Clusters, and Grid Chair: Subhankar Dhar

Designing SANs to Support Low-Fanout Multicasts	217
<i>Rajendra V. Boppana, Rajesh Boppana, Suresh Chalasani</i>	

POMA: Prioritized Overlay Multicast in Ad Hoc Environments	228
<i>Abhishek Patil, Yunhao Liu, Lionel M. Ni, Li Xiao, A.-H. Esfahanian</i>	

Supporting Mobile Multimedia Services with Intermittently Available Grid Resources	238
<i>Yun Huang, Nalini Venkatasubramanian</i>	

Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks	248
<i>Vinod Tippalapura, Manojkumar Krishnan, Jarek Nieplocha, Gopalakrishnan Santhanaraman, Dhabaleswar Panda</i>	

Session VI – Scheduling and Software Algorithms Chair: Rahul Garg

Scheduling Directed A-Cyclic Task Graphs on Heterogeneous Processors Using Task Duplication	259
<i>Sanjeev Baskiyar, Christopher Dickinson</i>	

XVIII Table of Contents

Double-Loop Feedback-Based Scheduling Approach for Distributed Real-Time Systems	268
<i>Suzhen Lin, G. Manimaran</i>	

Combined Scheduling of Hard and Soft Real-Time Tasks in Multiprocessor Systems	279
<i>B. Duwairi, G. Manimaran</i>	

An Efficient Algorithm to Compute Delay Set in SPMD Programs	290
<i>Manish P. Kurhekare, Rajkishore Barik, Umesh Kumar</i>	

Dynamic Load Balancing for I/O-Intensive Tasks on Heterogeneous Clusters	300
<i>Xiao Qin, Hong Jiang, Yifeng Zhu, David R. Swanson</i>	

Keynote Address

Standards Based High Performance Computing	310
<i>David Scott</i>	

Session VII – Network Design and Performance Issues Chair: Rajendra Boppana

Delay and Jitter Minimization in High Performance Internet Computing	311
<i>Javed I. Khan, Seung S. Yang</i>	

An Efficient Heuristic Search for Optimal Wavelength Requirement in Static WDM Optical Networks	323
<i>Swarup Mandal, Debasish Saha</i>	

Slot Allocation Schemes for Delay Sensitive Traffic Support in Asynchronous Wireless Mesh Networks	333
<i>V. Vidhyashankar, B.S. Manoj, C. Siva Ram Murthy</i>	

Multicriteria Network Design Using Distributed Evolutionary Algorithm	343
<i>Rajeev Kumar</i>	

Session VIII – Grid Applications and Architecture Support Chair: Vipin Chaudhary

GridOS: Operating System Services for Grid Architectures	353
<i>Pradeep Padala, Joseph N. Wilson</i>	

Hierarchical and Declarative Security for Grid Applications	363
<i>Isabelle Attali, Denis Caromel, Arnaud Contes</i>	

A Middleware Substrate for Integrating Services on the Grid	373
<i>Viraj Bhat, Manish Parashar</i>	

Performance Analysis of a Hybrid Overset Multi-block Application on Multiple Architectures	383
<i>M. Jared Djomehri, Rupak Biswas</i>	

Complexity Analysis of a Cache Controller for Speculative Multithreading Chip Multiprocessors	393
<i>Yoshimitsu Yanagawa, Luong Dinh Hung, Chitaka Iwama, Niko Demus Barli, Shuichi Sakai, Hidehiko Tanaka</i>	

Keynote Address

One Chip, One Server: How Do We Exploit Its Power?	405
<i>Per Stenstrom</i>	

Session IX – Performance Evaluation and Analysis

Chair: Krishnaiya Thulasiraman

Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms	406
<i>Sandhya Krishnan, Sriram Krishnamoorthy, Gerald Baumgartner, Daniel Cociorva, Chi-Chung Lam, P. Sadayappan, J. Ramanujam, David E. Bernholdt, Venkatesh Choppella</i>	

Performance Evaluation of Working Set Scheme for Location Management in PCS Networks	418
<i>Pravin Amrut Pawar, S.L. Mehndiratta</i>	

Parallel Performance of the Interpolation Supplemented Lattice Boltzmann Method	428
<i>C. Shyam Sunder, G. Baskar, V. Babu, David Strenski</i>	

Crafting Data Structures: A Study of Reference Locality in Refinement-Based Pathfinding	438
<i>Robert Niewiadomski, José Nelson Amaral, Robert C. Holte</i>	

Improving Performance Analysis Using Resource Management Information	449
<i>Tiago C. Ferreto, César A.F. De Rose</i>	

Session X – Scheduling and Migration

Chair: Baba C. Vemuri

Optimizing Dynamic Dispatches through Type Invariant Region Analysis	459
<i>Mark Leair, Santosh Pande</i>	

Thread Migration/Checkpointing for Type-Unsafe C Programs	469
<i>Hai Jiang, Vipin Chaudhary</i>	
Web Page Characteristics-Based Scheduling	480
<i>Yianxiao Chen, Shikharesh Majumdar</i>	
Controlling Kernel Scheduling from User Space: An Approach to Enhancing Applications' Reactivity to I/O Events.	490
<i>Vincent Danjean, Raymond Namyst</i>	
High-Speed Migration by Anticipative Mobility	500
<i>Luk Stoops, Karsten Verelst, Tom Mens, Theo D'Hondt</i>	
Author Index	511

Life's Duplicities: Sex, Death, and Valis

Bud Mishra

Computer Science & Mathematics (Courant Institute)

New York University

251 Mercer Street, New York, N.Y. 10012

&

Watson School of Biological Sciences

Cold Spring Harbor Laboratory

1 Bungtown Rd., Cold Spring Harbor, NY 11724

mishra@nyu.edu,

<http://cs.nyu.edu/faculty/mishra/>

<http://bioinformatics.cat.nyu.edu/>

Abstract. Is there a unifying principle in biology that can be deciphered from the structures of all the present day genomes? Can one hope to determine the general structure and organization of cellular information as well as elucidate the underlying evolutionary dynamics, by systematically exploring various statistical characteristics of the genomic and proteomic data at many levels? A large-scale software and hardware system, Valis, developed by NYU bioinformatics group, aims to do just that. We analyze word-frequency and domain family size distributions across various genomes, and the distribution of the potential "hot-spots" for segmental duplications in the human genome. We hypothesize and test, by computational analysis, that such a pattern is the result of a generic dominating evolution mechanism, "evolution by duplication", originally suggested by Susumu Ohno. We examine what implications these duplications may have in determining the translocations of male-biased genes from sex chromosomes, genome structure of sex chromosomes, copy-number fluctuations (amplifications of oncogenes and homozygous or homozygous deletions of tumor suppressor genes) in cancer genomes, etc. We examine, through our explorations with Valis, how important a role information technology is likely to play in elucidating biology at all levels.

Performance Analysis of Blue Gene/L Using Parallel Discrete Event Simulation

Ed Upchurch, Paul L. Springer, Maciej Brodowicz, Sharon Brunett, and
T.D. Gottschalk

Center for Advanced Computing Research, California Institute of Technology
1200 E. California Boulevard, Pasadena, CA 91125, etu@cacr.caltech.edu

Abstract. High performance computers currently under construction, such as IBM's Blue Gene/L, consisting of large numbers (64K) of low cost processing elements with relatively small local memories (256MB) connected via relatively low bandwidth (0.375 Bytes/FLOP) low cost interconnection networks promise exceptional cost-performance for some scientific applications. Due to the large number of processing elements and adaptive routing networks in such systems, performance analysis of meaningful application kernels requires innovative methods. This paper describes a method that combines application analysis, tracing and parallel discrete event simulation to provide early performance prediction. Specifically, results of performance analysis of a Lennard-Jones Spatial (LJS) Decomposition molecular dynamics benchmark code for Blue Gene/L are given.

1 Introduction

Caltech's Center for Advanced Computing Research (CACR) is conducting application and simulation analyses of Blue Gene/L[1] in order to establish a range of effectiveness of the architecture in performing important classes of computations and to determine the design sensitivity of the global interconnect network in support of real world ASCI application execution.

Accurate cycle-by-cycle level simulation of BlueGene/L is necessary for detailed machine design [2] but long simulation runtimes make it difficult to predict performance of application kernels for a full 64K node system. Our team is taking a statistical approach using parameterized models of the applications (workloads) and statistical (queuing) models of processing node message traffic derived from traces produced by the computational experiments. All 64K nodes of BG/L are explicitly represented, but the model is not cycle-by-cycle accurate although effects of adaptive routing and network contention are of necessity reliably modeled. For a further increase in simulation performance, SPEEDES [3], an optimistic parallel discrete event simulation (PDES) framework is employed running on a 128-node SGI Origin 2000.

2 Approach

A benchmark code for Lennard-Jones Spatial Decomposition (LJS) molecular dynamics [4], was selected as an example of an important class of numerical problems and a methodology of algorithm analysis, tracing and simulation was used to produce performance estimates for a 64K node Blue Gene/L.

LJS is an example of a fast parallel algorithm for short range molecular dynamics applications simulating Newtonian interactions in large groups of atoms. Such simulations are large in two dimensions: number of atoms and number of time steps. The spatial decomposition case was selected where each processing node keeps track of the positions and movement of the atoms in a 3-D box. Since the simulations model interactions on an atomic scale, the computations carried out in a single time step (iteration) correspond to femto-seconds of real time. Hence, a meaningful simulation of the evolution of the system's state typically requires a large number (thousands) of time steps. Point-to-point MPI messages are exchanged across each of the 6 sides of the box at each time step. The code is written in Fortran and MPI.

And finally a parallel BG/L simulation model is used to estimate LJS performance on BG/L for a variety of mappings to various numbers of BG/L compute nodes. Performance metrics including: time to solution; network utilization and scalability are reported.

3 LJS Benchmark Experiment Configuration

The runtime parameters of the simulation along with their values are listed in Table 1.

Table 1. LJS Experiment Runtime Parameters

Name	Value	Description
Physics		
Dt	0.00442	Timestep size in reduced units
T_0	1.444	Initial temperature in reduced units
ρ	0.8442	Density in reduced units
r_c	2.5	Cutoff distance in reduced units
r_s	2.8	Extended cutoff distance in reduced units
Problem definition and execution control		
n_x, n_y, n_z	50, 50, 50 (per CPU)	Dimensions of domain bounding box (integer units)
$Alat$	$(4/\rho)^{1/3} \approx 1.68$	Linear scaling factor
T	5	Number of simulation timesteps
n_{neigh}	20	Number of timesteps between re-binning
$nbin_x, nbin_y, nbin_z$	0.6 $n_x, 0.6 n_y, 0.6 n_z$	Number of cells per each dimension of the domain

The total number of particles, N , is given as: $N = 4 n_x n_y n_z$,

where n_i are integers (there is a fixed average of four particles per unit cube). The problem is executed on a grid of P processors, such that

$$P = p_x p_y p_z, \text{ with } p_i = n_i/k_i \text{ where } k_i \text{ are integers.}$$

In the benchmarks $k_i = 50$, hence the problem size was 50x50x50 (or 500,000 particles) on a single, 100x100x50 on four, 100x100x100 on eight and 200x200x200 on 64 processors. Such configurations require approximately 200MB of memory per CPU for all LJS data structures well within expected user memory for Blue Gene/L. Note that the cutoff distances are significantly smaller than the linear dimensions of the domain fragment assigned to a single processor ($50 \cdot alat \cong 84$), hence the spatial decomposition algorithm is performing efficiently (time spent in all communication phases is significantly smaller than the total computation time and didn't exceed 15% of the application runtime in the experiments).

LJS initializes its data structures by assigning particle positions on a regular 3-D mesh (thus emulating a crystal lattice) and computing velocity vectors to satisfy the initial temperature requirement. The velocities are otherwise random in magnitude and direction. In the next few time steps of the simulation the particles move from their positions on the grid (the crystal melts). Therefore, to capture the application behavior as close to the average (no imbalances of particle counts between processors), tracing was limited to the first five iterations.

3.1 MPI Communication

LJS uses a small subset of MPI-1 calls for message passing. The collective calls (*Barrier*, *Bcast*, *Allreduce*) are invoked only during the setup phase and when computing the thermodynamic state of the system (typically at the end of execution). The bulk of data is transferred by point-to-point calls (blocking *Send* and non-blocking *Irecv*, which enable overlapping of bi-directional transmissions). For the parameter set listed above, the messages originating from each node are emitted to its six nearest neighbor (in a 3-D grid) nodes only. Due to the use of a periodic Cartesian communicator, particles migrating outside the domain from boundary cells in any dimension, appear in the opposite boundary cell in that dimension.

3.2 LJS Execution Phases

The source code of LJS was augmented with calls injecting markers at the endpoints of the following phases:

- Setup and initialization (procedures: *input*, *setup_general*, *setup_memory*, *setup_comm*, *setup_neigh*, *setup_atom*, *scale_velocity*, *exchange*, *borders*, *neighbor*)
- Iteration of the main loop (*integrate*):
 - Calculation of the new positions of particles
 - Communication: update of the positions of remote particles (*communicate*)
 - Computation of forces (*force_newton*)
 - Reverse communication: propagation of forces (*reverse_comm.*)

- Calculation of particle velocities
- Final thermodynamics evaluation and printout (*thermo, output*)

3.3 Computational Profile

The computational workload was very consistent from iteration to iteration and across the nodes. This is expected due to uniform initial distribution of particles and symmetric neighborhoods of each cell. Table 2 shows counter values collected for the setup, intermediate phases of the fourth time step of the simulation (which is representative for other iterations as well) and finalization phase for different number of processors.

Table 2. Operation Counts for Various Numbers of CPU's

Units: $\times 10^6$	Setup	Compute position	Forward comm.	Compute force	Reverse comm.	Compute velocity	Statistics & output
1 CPU, grid: 50x50x50							
Cycles	5052.20	10.18	5.77	906.53	3.98	24.41	1582.67
Instructions	4429.95	6.25	1.30	646.41	1.63	8.75	1108.98
FPU ops	1770.20	1.50	0.34	311.09	0.34	1.50	543.05
4 CPUs, grid: 100x100x50							
Cycles	5221.11	14.83	10.17	919.68	20.71	28.38	1629.53
Instructions	4511.17	6.25	3.45	642.52	17.94	8.75	1127.78
FPU ops	1778.32	1.50	0.25	309.25	0.40	1.50	539.98
8 CPUs, grid: 100x100x100							
Cycles	5219.71	19.11	12.34	904.65	69.27	33.61	1611.66
Instructions	4559.75	6.25	3.68	642.81	71.84	8.75	1126.83
FPU ops	1778.87	1.50	0.19	309.41	0.38	1.50	540.17
64 CPUs, grid: 200x200x200							
Cycles	8482.76	18.83	21.94	905.02	79.77	33.84	1778.04
Instructions	7700.18	6.25	14.51	642.81	72.79	8.75	1283.72
FPU ops	1782.45	1.50	0.20	309.13	0.36	1.50	540.69

The only significant inconsistencies are variances in event counts for the communication phases. This is understandable, since the message passing is inherently non-deterministic. For example, messages of identical sizes can be split into different number of packets depending on the transient condition of the interconnect network and hence the overhead of message fragmentation and reassembly may not be identical. Note that even though the network traversal time should be excluded from timings in user mode, the actual behavior is strongly implementation dependent; if the MPI library uses busy waiting to poll for incoming messages, this fact will be reflected in counts. The global trend of increasing the overhead with the problem size is, however, sustained. LJS deploys a “leapfrog” integrator, whose operation is expressed as (only position computation shown; velocity calculation is identical in complexity with properly adjusted dt):

$$x_i(t+1) = x_i(t) + v_i(t+1/2) dt, \text{ for dimension } i = 1, 2, 3 \text{ in iteration } t.$$

This is in nearly perfect agreement with the FPU counts: 500,000 particles per CPU with 3 dimensional components yield 1.5 million operations. The compiler takes advantage of the fact that the Power ISA includes a multiply-add instruction;

otherwise the FPU counts would be twice as high. The number of cycles spent in velocity calculation phase is higher than that of position integration, since the previous values of velocity vectors need to be preserved for thermodynamic state computations, while the old position vectors are simply overwritten. The copy operation doesn't use the FPU; hence the additional overhead manifests itself only in increased instruction/cycle counts. Still, by far the most dominant portion of each time step is devoted to the force computation, thus justifying the presence of reverse communication step.

3.4 Communication Profile

Due to symmetry of the problem decomposition and repeatability of parameters passed to MPI calls, only one-iteration behavior on a single processor was analyzed. The results are collected in Table 3 listing destinations and sizes (in bytes) of messages transmitted from node 0 in each communication phase (Fwd = forward, Rev = reverse).

Table 3. Communications Profile

CPUs	Phase	Message	#1	#2	#3	#4	#5	#6
4	Fwd	Dest. node	2	2	1	1	-	-
		Size	496800	372600	1063152	797352	-	-
	Rev	Dest. node	1	1	2	2	-	-
		Size	797352	1063152	372600	496800	-	-
8	Fwd	Dest. node	4	4	2	2	1	1
		Size	480000	360000	513600	385200	549552	412152
	Rev	Dest. node	1	1	2	2	4	4
		Size	412152	549552	385200	513600	360000	480000
64	Fwd	Dest. node	48	16	12	4	3	1
		Size	480000	360000	513600	385200	549552	412152
	Rev	Dest. node	3	1	12	4	48	16
		Size	412152	549552	385200	513600	360000	480000

This scheme is repeated in every time step. Differences across the nodes are relevant only to destination node numbers, but they stay fixed throughout the execution for a given sender node. Note that the number of messages is reduced when running on less than 8 processors. This is because 2^3 CPUs is the smallest configuration where the computational domain can be decomposed into at least two partitions along each dimension. A small inefficiency of LJS may be observed when running on less than 64 processors: the messages within the same communication phase are emitted to repeated destinations. When scaling beyond 64 CPUs, message sizes and number of destinations stay fixed as long as the size of local grid on every processor is preserved.

3.5 Algorithm Scaling

To verify the characteristics of program execution for other problem sizes, LJS was traced with reduced grid size of $n_x = n_y = n_z = 100$ on 64 processors. The

computational workload parameters (fourth iteration only) and message sizes are given in Table 4 along with ratios of message sizes relative to 200³ grid case.

Table 4. Tracing Results for $n_x = n_y = n_z = 100$ on 64 CPU's

Units: $\times 10^6$	Setup	Compute position	Forward comm.	Compute force	Reverse comm.	Compute velocity	Statistics & output
64 CPUs, grid: 100x100x100							
Cycles	4518.99	1.130	4.574	109.366	7.548	3.222	206.038
Instructions	4403.61	0.782	3.009	79.950	6.008	1.095	148.078
FPU ops	224.58	0.188	0.052	38.479	0.092	0.188	67.258
Phase	Message	#1	#2	#3	#4	#5	#6
Fwd	Dest. node	48	16	12	4	3	1
	Size	120000	90000	136800	102600	155952	116952
	Size ratio	1 : 4	1 : 4	1 : 3.75	1 : 3.75	1 : 3.52	1 : 3.52
Rev	Dest. node	3	1	12	4	48	16
	Size	116952	155952	102600	136800	90000	120000
	Size ratio	1 : 3.52	1 : 3.52	1 : 3.75	1 : 3.75	1 : 4	1 : 4

As can be easily seen, the computational workload decreased proportionally to the problem volume (2^3 times, as the problem size in each dimension was halved). The memory allocation for LJS data arrays was 26.5MB per processor, again – roughly 1/8 of that required for 200x200x200 configuration. Message sizes were reduced approximately four times, what agrees well with the assumption of communication volume being proportional to the cell surface area. Note that the ideal ratio of four is observed only for the exchanges along the first dimension. This figure is distorted for transmissions along the second and third dimension due to the fact that the presence of volume characteristic decreases in subsequent data sends.

The spatial decomposition algorithm implemented in LJS behaves consistently over a wide range of grid sizes. The anomalies resulting from the cutoff distance r_s being comparable with the physical dimensions of a sub-grid assigned to a single processor arise for relatively small problem sizes, for which the communication overhead nearly always exceeds the cumulative duration of computations. To investigate such a case, the program was configured to run a 20x20x20 problem (on 64 CPUs this yields a 5x5x5 grid per processor) with artificially increased values of $r_c = 10$ and $r_s = 11.2$. The communication characteristics are given in Table 5 with the “reverse” communication phase omitted for brevity.

Since r_s is longer than the linear dimension of the local sub-domain ($d = 5 \cdot alat \approx 8.4$), the communication must involve not only the immediate neighbors of a processor, but also cells located one more grid “hop” away (because $d < r_s < 2d$). As LJS processes don't communicate with the remote neighbors directly, the data are passed in multiple steps through the immediate neighbors' buffers. Unlike in the typical scenario described in section 3.2, the ratio of communication volume along the third dimension (the last four entries in the table) to that of the first dimension (the first four entries) is significantly larger due to much larger final volume of data accumulated from neighboring cells within the cutoff distance in all three dimensions compared to that for just one dimension. The memory consumption, 10.7MB per processor, also

deviates from the simplistic estimates, most likely due to excessive buffer space required for communication. However, such situations arise rarely in practical short-range problems when executed on machines with sufficient amount of memory per node.

Table 5. Tracing Results for $n_x = n_y = n_z = 20$ on 64 CPU's

Message	#1	#2	#3	#4	#5	#6
Destination node	48	16	48	16	12	4
Size (bytes)	12000	12000	4800	3600	44400	44400
Message	#7	#8	#9	#10	#11	#12
Destination node	12	4	3	1	3	1
Size (bytes)	17760	13320	164280	164280	65712	49272

4 Model Development

Blue Gene/L uses a 3-D torus based network for point-to-point communications between nodes [1][2]. The torus router that exists on each BG/L node is modeled including the 6 injection FIFOs, as well as FIFOs associated with the 6 input and output network links at each node. Each link has associated with it two virtual channels that are used for adaptive routing, and 1 virtual channel used for deterministic routing. The latter is used for deadlock avoidance, and only when congestion prevents a packet from being adaptively routed. Tokens are used for flow control between routers. Virtual cut-through routing is used to minimize latencies.

The fundamental programming construct in SPEEDES is the simulation object. Each such object communicates to other objects by sending and receiving time-stamped messages. Receipt of a message eventually triggers the receiving object to process that message. The simulation time of the received message becomes the simulation time associated with the corresponding object when it executes. If an object executes during a certain time slice and is not rolled back, that event is said to be committed.

Each network node as a unique SPEEDES object and each network packet is modeled as a message in the simulation. This level of granularity is needed because of the complexity involved in the adaptive routing being used by the network. As congestion builds up in one part of the network, traffic patterns change in an attempt to route messages around the congested area.

Simulation messages are small, and only contain information relating to the transfer process, such as the origination node, destination node, and time of origination. When a message is sent from one object to another, it triggers an arbitration process in the receiving object that determines whether the message needs to be sent on, and if so, what route it should take. In parallel with this, the same object is examining its workload queue to determine whether new messages are being generated. Information about congestion on that node is sent back to the originating node to assist in flow control and the adaptive routing algorithm.

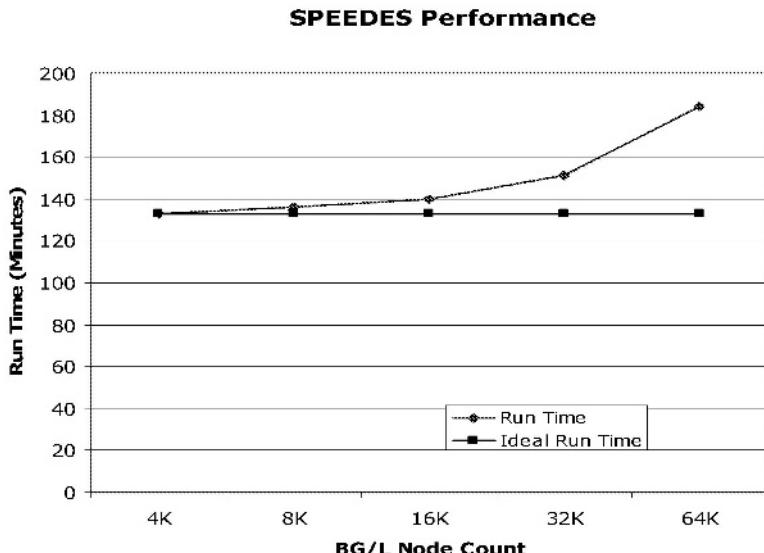


Fig. 1. SPEEDES Scaling Performance

4.1 Random Workload Scaling Experiment

A simple base case experiment was performed to determine the performance, scaling, and use of SPEEDES. This case, while modeling the BG/L nodes and network, did not exercise BG/L message flow control or adaptive routing but instead used simple dimension ordered routing. The experimental workload consisted of a uniform burst of 256 byte packets injected at each BG/L node at simulation start time destined for randomly chosen nodes in the network.

With the large numbers of real messages being sent between physical nodes and a small amount of computation used to process each message, good simulation speedup was not expected. The focus has been on scaling the size of the simulation to 64K nodes. Nevertheless a small series of speedup tests yielded a pleasant surprise. A speedup of 3.9 was measured when going from 4 physical processors to 16 physical processors for simulating 4096 BG/L nodes. Additionally, the simulation scaling performance was within about 10% of ideal, except for the 128 Origin processor case (see Figure 1).

4.2 LJS Molecular Dynamics Experiment

This experiment was driven a statistical workload generated from the message passing pattern from the LJS molecular dynamics application. Full support for adaptive routing and flow control was built into the software. The application is comprised of a number of cycles, with each cycle consisting of a compute stage and a communication stage. During each communication stage, a message is sent from each cell to its six immediate neighboring cells in three dimensional space. Messages

average about 1750 packets in size, with each packet holding 256 bytes. The model maps a single cell to a single BG/L node, and only simulates a single communication stage, because of the repetitive nature of the communication and computation cycles.

The BG/L simulations were run on an SGI Origin 2000 with 128 R12000 processors available, each running at 300 MHz. These are configured as 2 processors per node. With each node containing 1 GB of RAM, it has a total of 64 GB of RAM.

Two cases were modeled. In the first case, ideal placement of cells is assumed, i.e. nearest neighbor cells are located on the physical nearest neighbor BG/L nodes. The second case assumes cells are mapped randomly to BG/L nodes. In the ideal first case, all messages move only one hop, from the originating cell to its nearest neighbor. Under this scenario, no flow control is needed: packets are collected on the receiving node as soon as they arrive, and need not be passed on. Maximum use of the bandwidth, and good scaling were observed. The model sizes used were 4K, 8K, 16K, 32K, and 64K BG/L nodes, with the ratio of one application cell per BG/L node remaining constant. These were run on 8, 16, 32, 64, and 128 Origin 2000 processors, respectively. Because not all messages were exactly the same size, the upper limit of bandwidth utilization one could expect to see was 82%. For each model size, 81% or better was observed.

For the second case, random mapping of application to BG/L nodes, model sizes of 64, 512 and 4096 BG/L nodes, each simulated on 8 nodes of the Origin have been run. Because of the high volume of packets sent, and the multiple hops required for each delivery, one can see the effects of congestion in the network. Again using the figure of a maximum possible bandwidth utilization of 82%, we see 58%, 54% and 49% usage respectively.

Table 6. Fixed vs. Scaled Problem Size

Scaled Size N (atoms)	BG/L Nodes	Compute Time (ms)	Communications Time (ms)
500,000	1	1480	N/A
256,000,000	512	1480	6.3
512,000,000	1024	1480	6.3
1,024,000,000	2048	1480	6.3
2,048,000,000	4096	1480	6.3
4,096,000,000	8192	1480	6.3
8,192,000,000	16384	1480	6.3
16,384,000,000	32768	1480	6.3
32,768,000,000	65536	1480	6.3
Fixed Size N (atoms)	BG/L Nodes	Compute Time (ms)	Communications Time (ms)
500,000	1	1480	N/A
500,000	512	2.89	3.76
500,000	1024	1.45	2.34
500,000	2048	0.72	1.46

For BG/L the workload is specified for an average iteration by 1.48 seconds compute time, 6.3 ms of communications time for the “best” case independent of the number of nodes (for the random case communications time is 26 ms, 50 ms and 100 ms for 64, 512 and 4096 nodes respectively) for sending 3.6Mbytes of data/node. These numbers are for the problem size defined and traced where there are $50^3 \times 4$ atoms/processing node (or 500,000 atoms/node).

Table 6 shows a comparison between cases (both best case mapping to BG/L nodes) where the problem size was kept fixed at 500,000 atoms/processor and a fixed problem size of 500,000 atoms.

5 Conclusions and Future Work

Several observations can be made from the experiments:

- SPEEDES enabled simulation of 64K nodes for full application workloads
- placement of application nodes on the torus to reduce hops can result in significant communications performance increase

Planned future work includes:

- Run LJS on actual BG/L hardware when it becomes available for model validation
- Explore various optimistic time management techniques to determine the effect on simulation performance, especially for unbalanced workloads

Acknowledgments. This work was performed under contract with Lawrence Livermore National Laboratory, Contract No. B520721 (Applications Requirements Machine Model Simulator).

References

- [1] Adiga, N R, et al, “An Overview of the BlueGene/L Supercomputer.” In *Proceedings of SC2002*, November, (2002)
- [2] Heidelberger, P., and Steinmacher-Burow, B., “Overview of the BG/L Torus Network” <http://www.llnl.gov/asci/platforms/bluegene/talks/heidelberger.pdf>
- [3] Steinman, Jeffrey, “SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation.” In *Proceedings of the SCS Western Multiconference on Advances in Parallel and Distributed Simulation* (PADS91), vol 23, 1 (1991), 95–103
- [4] Plimpton, S., “Fast Parallel Algorithms for Short-Range Molecular Dynamics”, *Journal of Computational Physics* 117, 1–19 (1995)

An Efficient Web Cache Replacement Policy

A. Radhika Sarma and R. Govindarajan

Supercomputer Education and Research Center,
Indian Institute of Science,
Bangalore, 560 012, INDIA

radhika@rishi.serc.iisc.ernet.in, govind@serc.iisc.ernet.in

Abstract. Several replacement policies for web caches have been proposed and studied extensively in the literature. Different replacement policies perform better in terms of (i) the number of objects found in the cache (cache hit), (ii) the network traffic avoided by fetching the referenced object from the cache, or (iii) the savings in response time. In this paper, we propose a simple and efficient replacement policy (hereafter known as SE) which improves all three performance measures. Trace-driven simulations were done to evaluate the performance of SE. We compare SE with two widely used and efficient replacement policies, namely Least Recently Used (LRU) and Least Unified Value (LUV) algorithms. Our results show that SE performs at least as well as, if not better than, both these replacement policies. Unlike various other replacement policies proposed in literature, our SE policy does not require parameter tuning or a-priori trace analysis and has an efficient and simple implementation that can be incorporated in any existing proxy server or web server with ease.

1 Introduction

Caching of web objects is done at various levels of hierarchy to reduce both the average response time for a web request and the network traffic [15], [1], [4], [5], [11]. It is done at the browser, proxy and server levels. Caching at the browser and proxy caches are done to reduce the network traffic and the average delay experienced by a user request. At the server end, caching is done essentially to reduce the time taken to fetch the object from the file system.

In order to achieve the full benefits of caches, it is important that a good replacement policy is employed, which ensures that, the frequently accessed objects remain in the cache, and the objects that are least likely to be accessed, are replaced. A good replacement policy for web caches should select an object for replacement taking into account [4] : (i) the response time of the objects, (ii) network traffic generated by the replacement, and (iii) the non-uniformity in size of the objects cached.

The performance measures normally used in the context of web caches are Hit Ratio (HR), Byte Hit Ratio (BHR), and Delay Saving Ratio (DSR) [4]. These performance metrics measure performance of the replacement policies in terms of the number of requested objects present in cache (HR), the number of bytes

saved from retransmission (BHR) and the reduction in the latency of fetching a requested object (DSR). A number of replacement policies have been proposed in the literature to improve one of the three performance measures [16], [15], [12]. However, only a few have tried to improve all the three performance measures; and those that have, have achieved varying degrees of success.

The replacement policies for proxy or server caches, are designed keeping in mind the traffic characteristics encountered at the proxy or at the server, as their performance is dependent on traffic characteristics of WWW. However, some of the replacement policies, e.g., the Least Unified Value (LUV) policy, require fine tuning of parameters depending on the past traffic characteristics to achieve better performance [4]. This often makes implementing these algorithms more complex and require continuous performance tuning.

Time and space complexities are other important aspects of any replacement policy. Time complexity is important as the new object has to be brought into the cache and sent to the client requesting for it as fast as possible. Also, it is important that the replacement policy should also have a low space complexity. The space required to maintain the reference history of the object should not be more than a few bytes per object. Except for a few policies, e.g., [11] and [12] most satisfy this requirement.

In this paper, we propose a simple replacement policy (referred to as SE policy), that optimizes all three performance measures without incurring extra overheads in terms of time and space complexities. This policy takes into account the frequency and recency of access of a particular object, apart from the non-uniformity of the object, to compute the cost. The object with the lowest cost is chosen for replacement. We evaluate the proposed replacement policy on a number of web traces, displaying different traffic characteristics. The performance of the SE policy is compared with LUV and Least Recently Used (LRU) policies in terms of all three performance metrics, viz., HR, BHR and DSR.

Our results show that the performance of SE is comparable to the best of LRU and LUV replacement policies. The policy that we propose has an efficient and simple implementation which can be incorporated in any existing proxy server or web server with ease. Further, SE has a low space complexity ($O(1)$ per object) and a time complexity of $O(\log_2 n)$ (where n is the number of objects present in cache), and does not require any parameter tuning or a-priori trace analysis.

In Section 2 we present the necessary background. In Sections 3 and 4, we describe our replacement policy, SE and its implementation. Section 5 deals with experimental results. Concluding remarks are provided in Section 6.

2 Background and Motivation

2.1 Background

A number of cache replacement policies have been proposed [1,4,5,7,9,11,12,15, 16] in the literature. These replacement policies have been classified into one or

more of the following three categories [14]: (i) traditional policies and their direct extensions [15], (ii) key-based policies [1,15,16], and (iii) cost-based policies [4, 5,7,9,11,12,16].

In this paper we compare the proposed SE replacement policy with the Least Recently Used (LRU) policy and the Least Unified Value (LUV) policy [4]. The LRU policy uses recency of access of objects as a parameter to select an object for eviction. LRU has a time complexity of $O(1)$ and a space complexity of $O(1)$ per object.

LUV is a cost-based replacement policy, which associates a benefit value, or $bValue$, with each object. The $bValue$ of an object indicates the benefit of keeping the object in the cache. In LUV, the $bValue$ associated with each object is calculated using the estimated reference potential $p(i)$ of an object and its weight $w(i)$. The weight of every object is the retrieval cost c_i of the object normalized by its size s_i where c_i can be defined differently for measures of different interest. For estimation of reference potential $p(i)$, each reference of the $object_i$ in the past contributes to $p(i)$ and a weighing function $F(x)$ determines the contribution of a particular reference to the reference potential of the object. The function $F(x)$ is defined as

$$F(x) = (1/2)^{\lambda * x} \quad (0 \leq \lambda \leq 1)$$

where x is the time span from the past reference to the current time. The value of λ determines the amount of emphasis given to recency of reference and frequency of reference. When λ is equal to 0, LUV is reduced to a weighted LFU. As the value of λ increases, emphasis given to the recent references increases. For λ equal to 1, LUV is reduced to weighted LRU.

The effect of λ on the performance LUV depends on the characteristics of user access to the objects and the size of the cache used. Hence, it is important that the value of λ is tuned very carefully according to the user access pattern; otherwise the LUV policy performs rather poorly. For example, in our performance evaluation, we found that for a certain trace set and cache size, for $\lambda = 0.5$, HR is 0.1704. For the same trace set and cache size, for values of λ equal to 10^{-3} and 10^{-9} HR is 0.255 and 0.165 respectively. The LUV policy has a time complexity of $O(\log_2 n)$ and space complexity of $O(1)$ per object in cache, where n is the number of objects in cache.

2.2 Performance Measures

Three widely used performance measures in web caching are Hit Ratio, Byte Hit Ratio and Delay Saving Ratio [4]. They are defined as follows:

$$HR = \sum h_i / \sum t_i$$

$$BHR = \sum b_i * h_i / \sum b_i * t_i$$

$$DSR = \sum d_i * h_i / \sum d_i * t_i$$

where h_i and t_i represent respectively the number of hit references and the total number of references to object i , b_i is the size (in bytes) of object i , and d_i is the delay involved in fetching the object i .

Hit Ratio is the measure used in traditional caching systems. Byte Hit Ratio or BHR measures the bytes saved from retransmission due to a cache hit. Delay Saving Ratio is a measure that takes into account the latency of fetching an object. It represents the reduced latency due to a cache hit over total latency when there is no cache present. In this paper we use all three performance measures to compare our replacement policy with other policies.

2.3 Motivation

Existing replacement policies either (i) do not optimize all three performance measures, e.g., LRU, LFU, Size [15], and LNC-R-W3-U [12], or (ii) they incur a large overhead in time complexity, e.g., LNC-R-W3 replacement policy [12] and Mix replacement policy [9] have a time complexity of $O(n)$, or (iii) incur a large overhead in space complexity e.g., LNC-R-W3 replacement policy [12] has a space complexity of $O(k)$ where the policy takes in to account the last k references to an object, (iv) have parameters which are required to be fine tuned to a given set of traces and cache size (LUV [4]), or (v) require trace analysis to be performed (e.g., LRV replacement policy [11]).

3 A New Replacement Policy

In this section we will describe our SE replacement policy which uses a mix of both recency and frequency to achieve good performance. Like all the cost-based policies, our simple equation (SE) replacement policy associates a value with every object in cache and when required, evicts the object with lowest value. The SE replacement policy evaluates the cost of an object based on : (i) the recency of its last access, (ii) frequency of its access, (iii) the latency of fetching the object, and (iv) the size of the object. More specifically, the $bValue(i)$ of $object_i$ is calculated as

$$bValue(i) = W(i) * H(i)$$

where $W(i)$, weight of $object_i$ is defined as $W(i) = c_i/s_i$ as in [4]. The value we have used for cost of an object is its retrieval latency. $H(i)$ is the reference potential of $object_i$. It determines the likelihood of an object being accessed based on its past references. It is defined as

$$H(i) = RecRef(i) * FreqRef$$

where RecRef and FreqRef represent respectively the recency of the last reference to $object_i$ and the total number of requests serviced, for all users, including the current reference to object i .

4 Implementation Details

In the SE policy, the objects are sorted based on the values $bValues$ associated with them. Anytime an object needs to be evicted from cache, the object with the smallest value $bValue$ is chosen. As the $bValue(i)$ of an object i depends of the frequency and recency of its references, the value associated with the object changes every time the object is referenced. The change in value $V(i)$ of an object causes its position, in the relative ordering of the objects according to their respective values, to change. Hence the $bValue$ of objects are sorted using a heap data structure.

Algorithm 1 SE Algorithm

```

if  $object_i$  is present in cache then
    delete the node corresponding to the object from the heap
    re-compute the value associated with the  $object_i$ ,  $V(i)$ 
    insert a node with the new value in the Heap
else
    fetch the  $object_i$ 
    while ( size of  $object_i$   $\geq$  free space available) do
        find the node with smallest value in the Heap
        find the corresponding object in the cache
        remove the object from cache
        delete the node from the Heap
    end while
    place the  $object_i$  in cache
    insert a new node into the heap with the new value associated with the  $object_i$ 
end if

```

The SE replacement algorithm is described in Algorithm 1. The time complexity of the SE replacement policy is $O(\log_2 n)$. The replacement policy requires normalized weight, last reference time, and number of references for each object to be stored. Hence the its space complexity is $O(1)$. Despite the fact that time and space complexities of LUV is the same as those of SE, the implementation of LUV is much more complex. An efficient implementation of LUV requires the usage of two data structures, a heap and a linked list which are required to be managed in tandem. In comparison the implementation of SE is very simple as it uses a simple heap to maintain the objects in the order required.

5 Results and Observations

In this section, we discuss the results of our trace driven simulations. We have used two proxy cache traces, the *rtp* traces (Research Triangle Park, North Carolina, USA [10]) from the National Lab for Applied Network Research (NLANR)

proxy traces, and one server cache trace, from the the World Cup '98 log [3], available in the public domain.

The characteristics of the traces used are summarized in Table 1. The unique requests processed include only the requests for unique objects which are processed by this cache. The number of unique bytes accessed refers to the number of bytes transferred due to first time accesses to objects.

It has been reported in previous studies of web servers [3], proxy servers [8] and World Cup '98 Characterization [2] that mean and median document transfer sizes are quite small, fewer than 13K and 3K bytes respectively. This means that, large objects are typically accessed very infrequently. Hence, in our experiments, objects larger than 1 MB in size are not cached and requests to these objects are not included in our analysis. Also, like in other web cache replacement policy studies [4], the requests in which no data are transferred; are not processed.

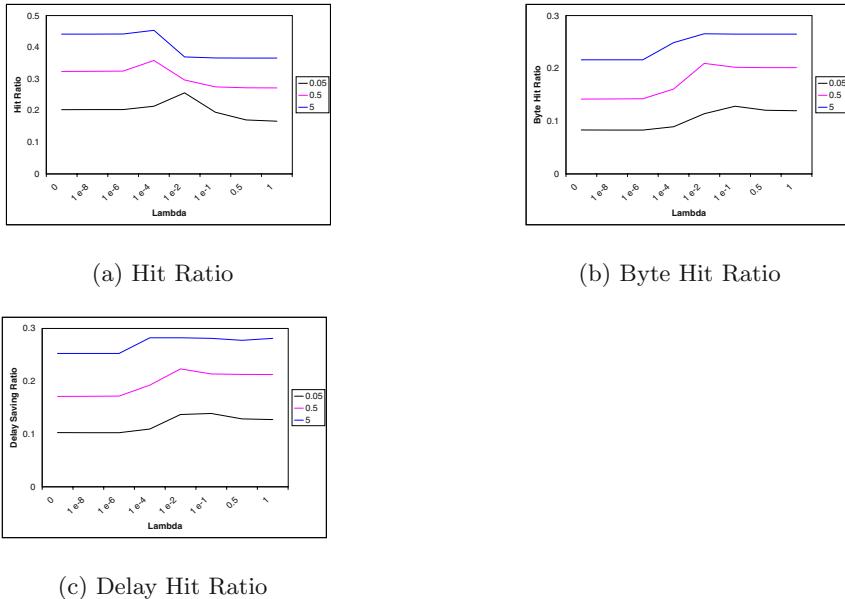
Table 1. Characteristics of Web Proxy and Web Server Traces

Trace	Duration of collection of Traces	Total Requests Processed	Total Unique Requests Processed	Total Mbytes Transferred	Total Unique Mbytes Transferred
		<i>in millions</i>			
NLANR	17/10/2002 - 20/10/2002	4.83	2.37	24133.82	16259.24
World Cup	24/06/1998 03/07/1998	349.23	0.015	864484.53	123.49

We compare our SE replacement policy with LRU and LUV, two widely used web cache replacement policies which achieve very good performance. Apart from LUV, there are replacement policies like GD-Size [6], which also optimize all the three performance measures. GD-Size considers non uniformity in size of the objects but does not take into account the frequency of access of an object. Further its performance was shown to be relatively poorer compared to LUV in [4]. Hence we have not considered it for our comparison study.

Before we proceed to compare the different replacement policies, we study the impact of λ on LUV, and empirically choose the best value for λ . In our experiments, the value of λ was varied from 10^{-9} to 1. In Figure 1, we present the performance of the LUV policy in terms of HR, BHR, and DSR on NLANR trace set for various values of λ and cache sizes. In the following discussion, cache sizes are represented in terms of percentages of the total number of unique bytes accessed.

From Figure 1, we notice that the value of λ for which HR achieves the best value (highest value for the given trace set and cache size) does not necessarily result in the highest value of BHR and DSR and this value of λ varies with cache size and trace set used. We also observe that LUV obtains the best hit ratios for smaller values of λ , for values less than 0.1. This indicates that very little

**Fig. 1.** Impact of Lambda (λ) for NLANR Traces

emphasis is given to the more recent references to obtain the best Hit Rate. However, for the same trace set and cache size, the best values of BHR and DSR are obtained at relatively larger values of λ . This indicates that more emphasis must be given to recent references to obtain a better BHR and DSR.

Due to the above mentioned issue of HR, BHR and DSR achieving their best values (for the same cache size and trace set) at different values of λ and in order to make a fair comparison amongst different replacement policies, we use the same value of λ to obtain HR, BHR and DSR values. We choose the value of λ that gives the best HR value for a given trace set and cache size. However, the value of λ used is different for different trace set and cache sizes. As a consequence, we refer to the LUV policy used in our experiment as *Best-LUV* to indicate that it uses the best value for λ , for a given cache size and trace set, to optimize HR. While we have chosen the value of λ that gives the best HR values for a given trace set and cache size, λ can also be chosen to get the best BHR or DSR values for a given trace set and cache size. Though this will improve BHR or DSR, this improvement will be at the cost of HR as can be seen from the Figure 1.

In Figure 2 we compare the performance (in terms of HR, BHR, and DSR) of LRU, Best-LUV, and SE replacement policies for various cache sizes and for various trace sets. As can be seen from the graphs, the performance of the three algorithms is highly dependent on the characteristics of the traces and cache size. For the World Cup '98 trace set, HR, BSR and DSR are much higher than the corresponding values for NLANR. The reason for this is the high temporal locality of the World Cup '98 traces. As can be seen from Table 1, the number of

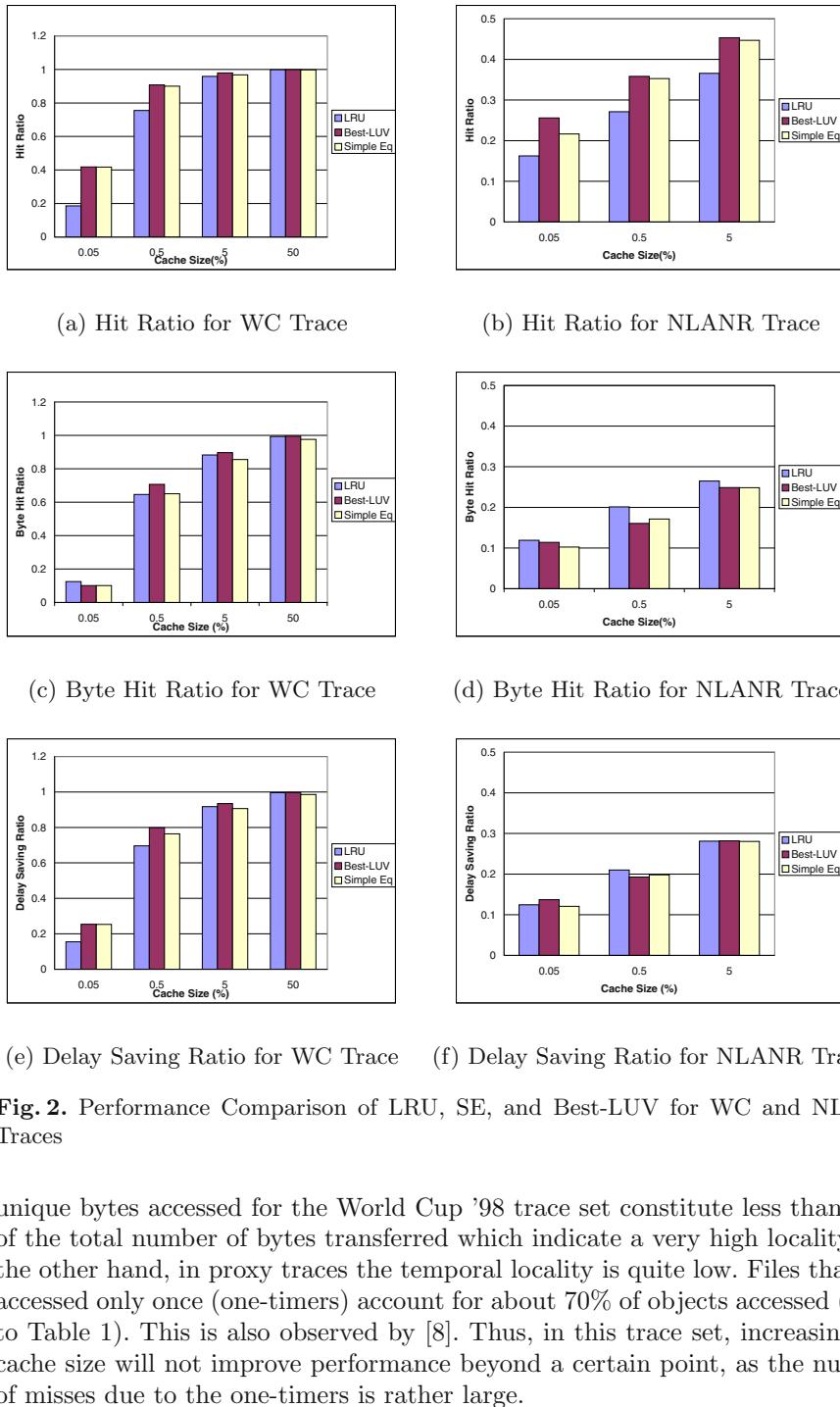


Fig. 2. Performance Comparison of LRU, SE, and Best-LUV for WC and NLANR Traces

unique bytes accessed for the World Cup '98 trace set constitute less than 10% of the total number of bytes transferred which indicate a very high locality. On the other hand, in proxy traces the temporal locality is quite low. Files that are accessed only once (one-timers) account for about 70% of objects accessed (refer to Table 1). This is also observed by [8]. Thus, in this trace set, increasing the cache size will not improve performance beyond a certain point, as the number of misses due to the one-timers is rather large.

We observe that SE achieves nearly the same performance as Best-LUV in terms of all three performance measure for the Word Cup '98 trace set. For this trace, LRU performs worse than LUV or SE for smaller cache sizes (for cache size less than 5%). The only exception is in the case of a very small cache size ($=0.05\%$), where LRU achieves a better BHR than LUV and SE.

For the NLANR proxy traces, no replacement policy performs consistently better than the other for all three performance measures. Best-LUV performs better in terms of HR, but not in terms of BHR or DSR. This is somewhat understandable as in Best-LUV, the value of λ is empirically optimized for HR rather than for BHR or DSR. SE performs better than LRU in terms of HR. The hit ratio achieved by SE is within 3-4% of the hit ratio of Best-LUV.

Table 2. Summary of Performance Comparison of LRU, LUV, and SE

Cache Size	Perf. Metric	Relative Ordering (from worst to best)	
		WC Trace	NLANR Trace
0.05%	HR	LRU < LUV \approx SE	LRU < SE < LUV
0.05%	BHR	LUV \approx SE < LRU	SE < LUV < LRU
0.05%	DSR	LRU < LUV \approx SE	LRU \approx SE < LUV
0.5%	HR	LRU < LUV \approx SE	LRU < LUV \approx SE
0.5%	BHR	LRU \approx SE < LUV	LUV < SE < LRU
0.5%	DSR	LRU < SE < LUV	LUV < SE < LRU
5.0%	HR	LRU \approx SE < LUV	LRU < LUV \approx SE
5.0%	BHR	SE < LRU < LUV	LUV \approx SE < LRU
5.0%	DSR	LRU \approx SE < LUV	LRU \approx SE \approx LUV
50.0%	HR	LRU \approx SE \approx LUV	—
50.0%	BHR	SE < LRU \approx LUV	—
50.0%	DSR	SE < LRU \approx LUV	—

In terms of BHR and DSR, LRU and SE perform better than LUV. Emphasis on recent references seems to increase BHR and DSR values. This is evident from the fact that LRU obtains better BHR and DSR than SE and Best-LUV. A similar result can be seen in LUV, when the value of λ is varied from 10^{-9} to 1.

As mentioned before the performance of the replacement policies is largely dependent on the characteristics of the trace sets used. We have seen that to obtain the best HR, BSR and DSR a combination of recency and frequency have to be used. Though LUV provides us the flexibility to change the weightage assigned to the two parameters, it is not possible to obtain best HR, BHR, and DSR values simultaneously. SE assigns equal weightage to both recency and frequency and performs as well as LUV for traces with high temporal locality. For traces with low temporal locality, except for some cache sizes it performs as well as the best of the other two replacement policies. Further, SE achieves these performance using a much simpler implementation and without requiring any parameter tuning.

Table 2 summarizes the performance comparison of the replacement policies while Table 3 provides a qualitative comparison.

Table 3. Qualitative Comparison of LRU, LUV, and SE

Repl. Policy	Time Compl.	Space Compl.	Nonunif. of objects considered	Work done for accessing an object	Advantages	Disadvantages
LRU	$O(1)$	$O(1)$	No	Minimum	Simple to implement	Does not consider frequency of access
LUV	$O(\log_2 n)$	$O(1)$	Yes	Maximum	Uses complete ref. history	Parameter tuning
SE	$O(\log_2 n)$	$O(1)$	Yes	Between LRU and LUV	Simple to implement	Complete Ref. history not used

6 Conclusions

In this paper we have presented a new web cache replacement policy, SE. This policy uses frequency and recency of references to an object to calculate and associate a benefit value (*b Value*) with the object; an object with the lowest *b Value* is chosen for eviction. SE has an efficient implementation in both space and time complexities. Using trace driven simulation, we have evaluated and compared the performance of SE with two other replacement policies, viz., the Least Unified Value (LUV) and the Least Recently Used (LRU) policies for a set of web traces. Our results indicate that the performance of SE is comparable to the best performance achieved by the other replacement policies in terms of all three performance metrics, namely hit ratio (HR), byte hit ratio (BHR), and delay saving ratio (DSR). SE is simpler to implement than LUV and does not require any parameter tuning.

References

1. M. Abrams, C. R. Standridge, G. Abdulla, S. Williams and E. A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of 4th International WWW Conference '95*.
2. M. Arlitt and T. Jin. Workload characterization of the 1998 World Cup web site. In *Technical Report HPL-1999-35R1 Hewlett Packard '99*.
3. M. Arlitt and C. Williamson. Internet web servers: Workload characterization and performance implications. In *IEEE/ACM Transactions on Networking '97*.
4. H. Bahn, S. H. Noh, S. L. Min, and K. Koh. Using full reference history for efficient document replacement in web caches. In *Proceeding of 2nd USENIX symposium on Internet Technologies and Systems '99*
5. P. Cao and S. Irani. Cost-Aware WWW proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technology and Systems (USITS 97) '97*.

6. L. Cherkasova. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical Report HPL-98-69R1, Hewlett Packard 1998.
7. T. Kelly, Y. M. Chan and S. Jamin. Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value. In *Proceedings of 4th International Web Caching Workshop '99*.
8. A. Mahanti, C. Williamson and D. Eager. Traffic analysis of a web proxy caching hierarchy. In *IEEE Network Magazine: Special Issue on Web Performance '00*.
9. N. Niclaussem, Z. Liu and P. Nain. A new and efficient caching policy for World Wide Web. In *Proceedings of Workshop Internet Server Performance (WISP) '98*.
10. <ftp://ircache.net/Traces>
11. L. Rizzo and L. Vicisano. Replacement policies for a proxy cache . In *IEEE/ACM Transactions on networking '00*.
12. J. Shim, P. Scheuermann and R. Vingralek. Proxy cache design: Algorithms Implementation and Performance. In *IEEE Transactions on Knowledge and Data Engineering '99*.
13. R. Tewari, M. Dahlin, H. M. Vin, J. S. Kay, Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. Technical Report TR98-04, Univ. of Texas at Austin, 1998.
14. J. Wang. A survey of web caching schemes for Internet. In *ACM Communications Review '99*.
15. S. Williams, M. Abrams, C. R. Standridge, G. Abdulla and E. A. Fox. Removal Policies in network caches for World Wide Web documents. In *Proceedings of SIG COMM '96*.
16. R. P. Wooster and M. Abrams. Proxy caching that estimates page load delays. In *Proceedings of 6th International WWW conference '97*.

Timing Issues of Operating Mode Switch in High Performance Reconfigurable Architectures*

Rama Sangireddy, Huesung Kim, and Arun K. Somani

Dependable Computing & Networking Laboratory
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011, USA
{sangired,huesung,arun}@iastate.edu

Abstract. The concept of a reconfigurable coprocessor controlled by the general purpose processor, with the coprocessor acting as a specialized functional unit, has evolved to accelerate applications requiring higher computing power. The idea of Adaptive Balanced Computing (ABC) architecture has evolved, where a module of Reconfigurable Functional Cache (RFC) is configured with a selective core function in the application whenever a higher computing resources are required. Initial results have proved that the ABC architecture provides with speedups ranging from 1.04x to 5.0x depending on the application and the speedups in the range of 2.61x to 27.4x are observed for the core functions. This paper further explores the issues of management of RFC, where the impact of various schemes for configuration of core function into the RFC module is studied. This paper also gives a detailed analysis on the performance of ABC architecture for various configuration schemes, including the study of the effect of the percentage of the core function in an entire application over the management of RFC modules.

1 Introduction

The technology of reconfigurable hardware, to implement a computation intensive function in a single unit whenever required, has evolved to accelerate the execution of selective functions and these reconfigurable chips are used as coprocessors in tandem with the general purpose processor. To accelerate structured and regular computations, such as DSP and multimedia applications, FPGA-like reconfigurable logic that is specialized for these computations has been developed and integrated into on-chip microprocessors [1,2,3,4].

Current processor designs often devote a largest fraction (up to 80%) of on-chip transistors to caches. However, many workloads, like media processor applications, do not fully utilize the large cache resources due to streaming nature and lack of temporal locality for the data. From these observations, an idea of a different kind of computing machine - Adaptive Balanced Computing (ABC)

* The research reported in this paper is partially funded by the grants Carver's Trust and Nicholas Professorship from Iowa State University, and grant No. CCR9900601 from NSF.

architecture, has evolved. ABC uses a dynamic configuration of a part of on-chip cache memory to convert it into a specialized computing unit. A reconfigurable functional cache (RFC) operates as a conventional cache memory or a specialized computing unit [7]. This paper explores two methodologies for the configuration of the RFC module. In the first scheme, the RFC is configured with the core function at the beginning of the multimedia application and this would result in a reduced cache capacity for the entire application. In the second scheme, the RFC is configured with the core function only when the core computation is necessary and the RFC module is released when not required to be used as a normal cache memory by the other computations within the multimedia application. With various cache mapping organizations, we study the overall impact on the performance of selected benchmarks, such as multimedia and DSP applications. The results prove that performance of each scheme varies depending on the structure of the application used.

The rest of the paper is organized as follows. In Section II we present an overview and the preliminary results obtained in building an ABC microprocessor. In Section III we discuss the main goal of this paper by proposing various RFC configuration schemes. Section IV presents a comprehensive discussion on the impact of various parameters, related to the microarchitecture as well as the application, on the overall execution time. In Section V, we discuss the results obtained from applying various RFC configuration schemes. Finally, Section VI concludes the discussion.

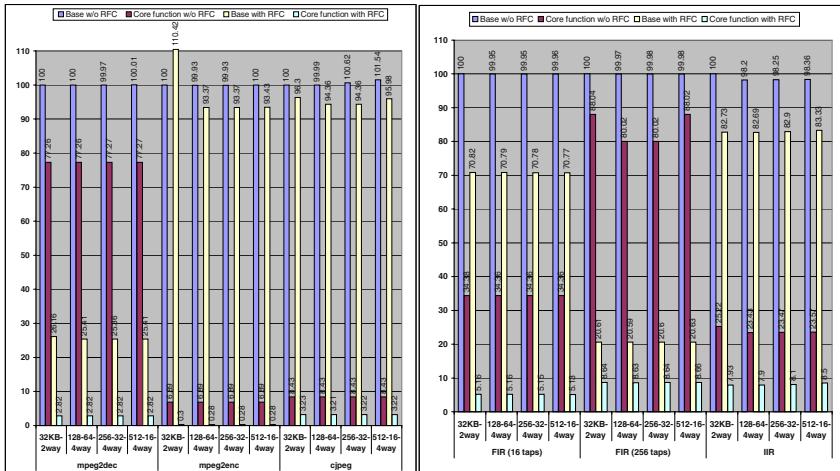


Fig. 1. Normalized execution cycles in base processor without RFC and ABC processor with RFC, with varying cache organizations.

2 Performance of ABC Architecture

The organization of a reconfigurable cache module (RFC) had been extensively discussed in [7]. To analyze the performance of the ABC architecture, the comparison is made between the number of cycles taken to execute each application with varying cache parameters. The performance, in terms of the total number of execution cycles normalized to the execution cycles in the base processor with 32KB 2-way set associative cache for each of the benchmarks, mpeg2dec, mpeg2enc, jpeg, FIR-16taps, FIR-256taps, and IIR, is shown in Figure 1.

The performance improvement can be obtained from the normalized total cycles for the execution of the overall function and also the core function. The specialized computing units configured in the RFCs improve the performance of each core function significantly. The most important factors for speed-up are the reduced number of instructions and the acceleration of computation with a specialized unit. Hence, it is observed that the overall speed-up is largely proportional to the frequency of the core function calls in the entire application. However, the initial simulation results show that the RFC is not a good idea to implement in a low associative cache in certain cases where it dramatically reduces the cache capacity. For example, a performance degradation is observed in the execution of mpeg2encode application in the ABC processor with 32KB 2-way set associative cache. This is due to the fact that in a 2-way set associative cache, when one of the cache modules is used as an RFC, the cache memory capacity is reduced to 50% and also the data cache acts as direct-mapped and hence causes the performance degradation. This motivated us to look further into this issue in the design of the ABC architecture, and hence we set-out to explore various methodologies of the management of RFC between the functional unit and the cache memory modes.

3 RFC Configuration Schemes

In the multimedia applications, it has been observed that the applications like mpeg2decode, mpeg2encode, FIR and other filters are highly iterative and hence multiple instances of the core functions would be occurring during the processing of the entire application. For example, while running mpeg2decode applications it has been observed that the DCT/IDCT function is called 15 times more as compared to while running mpeg2encode application [9]. Further, it is noted that the core functions like DCT/IDCT occupy a varying degree of percentage of entire application.

The other factor that has a predominant effect on the utilization of the RFC for the execution of the core function would be the configuration overhead. It is the time, in number of cycles, taken to load the configuration data into the target hardware. The configuration overhead would be greater as more number of instances of RFC configuration occur. The configuration data to program a cache into a function unit may be either available in an on-chip cache or an off-chip memory. Load time for the configuration data in the latter case

will be larger than in the former case. In the ABC architecture, the time to configure the RFC, from the normal cache memory mode to a functional unit mode, depends on the number of cycles to write words into a cache. On the other hand, an RFC operating as a functional unit can also be partially configured at run-time using write operations to the cache. When a partial reconfiguration occurs, the function unit must wait for the reconfiguration to complete before feeding the input data. Since the computation data (input and output data) and the configuration data (contents of LUTs) for an RFC unit share the global lines for data buses, we cannot perform both the computation and the partial reconfiguration simultaneously. It is possible to perform both the operations simultaneously, if separate data buses for computation and configuration are provided, which would dramatically increase the cost of the microarchitecture design.

Thus it becomes imperative for us to study and devise various RFC management schemes for an efficient utilization of the computing and memory resources to achieve a better performance from the integration of the RFCs into a conventional microprocessor.

3.1 Scheme1: One-Time Configuration of RFC

In the current design [8] of the ABC architecture, the configuration of the core function into the RFC module is performed *a priori*, i.e., at the beginning of the application so that the configuration of the same core function at multiple instances can be avoided. However, due to this method, the computations other than the core function and those computed by the main processor, are denied the usage of all the cache modules.

This scheme would prove to be beneficial when the core function is dominant in its execution over the entire application as in the case of DCT/IDCT function in the mpeg2decode. However, the scheme has some limitations for adoption, as it can be observed that it will have a negative impact when the percentage of core function is significantly small over the entire application as in the case of DCT/IDCT function in mpeg2encode application. Also, the scheme would not be better for adoption, when the configuration overhead, the total number of cycles taken to configure the core function, would prove to be significantly smaller and can be offset by the maximum utilization of the cache memory resources.

3.2 Scheme2: Continual Configuration of RFC

To overcome the limitations of the continuous denial of usage of full cache memory capacity for computations other than the core function, the RFC module can be configured with the core function at every instance of its occurrence and then be released to function as normal cache memory for the benefit of other computations. Thus the RFC module has to follow a pattern of continuous switching of modes, between the normal cache memory mode and the functional unit mode.

As it was originally anticipated and as argued in the case of the first scheme, the proposed continual reconfiguration scheme also has its gains and limitations.

This scheme will have a negative impact when the percentage of core function execution time is significantly large in the entire application as in the case DCT/IDCT function in the mpeg2decode. Besides, the scheme would prove to be limiting when the number of instances of the occurrence of the core function is high as the configuration overhead would be significantly large.

4 Analysis of Execution Time

In this Section, we present our arguments to justify the need of an analytical study of the two RFC configuration schemes described above. To adopt the RFC based ABC architecture for processing various multimedia applications, one of the above schemes has to be incorporated in the design. In view of the gains and limitations of the proposed RFC configuration schemes, and keeping in view of the nature of the multimedia applications where the percentage of core function in the entire application and the number of instances of occurrence of core function vary, we develop a mathematical model for the analysis of the performance in terms of total execution time, to determine which scheme proves to be beneficial under varying circumstances.

The architecture and application parameters, that will have significant impact on the overall execution time, have to be considered for the trade-off analysis among various schemes. The parameters that need to be considered are, the number of cycles required for configuration of RFC (C_p), the number of instances of occurrence of core function in the application (N), the fraction of time the core function is processed over the entire application (P), and the cache blocking factor (ϕ). Note that (1-P) represents the fraction of time for the computations other than the core function. Also, since we design the architecture for the faster execution of applications by accelerating the core function, we assume that there always exists a portion of the core function that can be mapped into the RFC. Hence the assumption that $P \neq 0$, holds true for the entire discussion.

Table 1. Architecture Design Parameters.

C_p	Cycle time for RFC configuration when data is fetched from off-chip memory.
P	Fraction of core function over the entire application.
N	Number of instances of core function over the entire application.
ϕ	Cache blocking factor.
X_A	Execution time of core function in the base processor.
X_B	Execution time of core function in the RFC functional unit.
S	Speed-up of core function = $(\frac{X_A}{X_B})$

The total execution time (X_{org}) of an application in the base processor without RFC can be represented by the expression:

$$X_{org} = X_A \left(\frac{1-P}{P} \right) + X_A = \frac{X_A}{P} \quad (1)$$

Similarly, the total execution time (X_{rfc}) of the application in the RFC integrated ABC processor can be represented by:

$$X_{rfc} = X_A \left(\frac{1-P}{P} \right) \phi + C'_p n_c + X_B \quad (2)$$

The $X_A \left(\frac{1-P}{P} \right)$ accounts for the execution time of the computations other than the core function. When the continual RFC configuration scheme is employed, the cache blocking factor (ϕ) ≈ 1 , i.e., all the cache modules are available for utilization as cache memory during the execution of computations other than the core function. The ϕ is assumed to be approximately equal to 1 instead of being exactly equal to 1 in the continual configuration scheme, in the process of RFC being switched between various modes, the main processor will be experiencing cache misses, which might be hit under normal cache operation. When the one-time RFC configuration scheme is employed, the cache blocking factor (ϕ) > 1 , as one of the cache modules is reserved for the functional unit and hence execution of the computations other than core function would take more time due to the rise in cache miss rate and the reduced cache capacity.

The $C'_p n_c$ accounts for the RFC configuration overhead over the entire application. When the one-time RFC configuration scheme is integrated into the design, the configuration overhead will be significantly smaller, as $n_c = 1$. Also, $C'_p = C_p$ since we assume that during the first time configuration of RFC, all the configuration data would be a miss in the cache and hence need to be fetched from the off-chip memory. However, when the continual RFC configuration scheme is employed, the configuration overhead forms a considerable portion of the total application execution time, as $n_c = N$. However the bright spot in this case is that the simulation has proved that $C'_p \ll C_p$, since the subsequent instances of configuration of RFC with the same function as the preceding instance configuration will not take as many cycles as the first instance of configuration. This is due to the fact that, between the adjacent calls of the core function when the other computations are being executed, a few data blocks in the RFC module get replaced and hence during the subsequent configuration, only the corrupted configuration data is fetched from the off-chip memory.

Table 2. Variation in parameters for various RFC configuration schemes.

	One-time configuration	Continual configuration
Cache blocking	$\phi > 1$	$\phi \approx 1$
Configuration instances	$n_c = 1$	$n_c = N$
Configuration cycles	$C'_p = C_p$	$C'_p \ll C_p$

The summary of the variation of parameters with respect to each of the schemes employed is as shown in Table 2. The parameters, ϕ in the case of

one-time RFC configuration scheme, and C'_p in the case of continual RFC configuration scheme are the two factors that affect the total execution time predominantly. It is not possible to exactly quantify these two factors, as both ϕ and C'_p are entirely dependent on the amount of data cache accesses, the cache organization and the cache miss rate. However, fortunately we can define the upper bounds for each of the parameters with the goal of obtaining a better performance from the RFC based ABC microprocessor.

4.1 Execution Time Analysis with One-Time RFC Configuration

When the design of ABC architecture is incorporated with the one-time RFC re-configuration scheme, the expression for X_{rfc} will be modified, after substituting relevant parameters from Table 2, as given by the following expression:

$$X_{rfc} = X_A \left(\frac{1-P}{P} \right) \phi + C_p + X_B \quad (3)$$

We are aware that, in the worst case, the configuration time (C_p) can be in the order of thousands of cycles while, X_B would be in the order of millions of cycles and hence C_p can be conveniently neglected in the expression. Now, for a gain in the performance, the execution time of application in the RFC based ABC processor should be less than the execution time of application in the base processor. Hence, solving $X_{rfc} < X_{org}$ yields

$$X_A \left(\frac{1-P}{P} \right) \phi + X_B < \frac{X_A}{P} \quad (4)$$

from which we can obtain the upper bound for the cache blocking factor as:

$$\phi < \left[\frac{1 - \left(\frac{P}{S} \right)}{1 - P} \right] \quad (5)$$

From the above expression, it is obvious that the lower bound on ϕ is 1, as $\frac{X_B}{X_A} \leq 1$ under all circumstances, i.e., the speed-up obtained for the core function by computing in the RFC would be at least 1.

The variation of the cache blocking factor with respect to the fraction of the core function (P) and the speed-up of the core function (S) is shown in Figure 2. It can be observed that, for applications with a larger fraction of the core function, the upper bound of the cache blocking factor is large which signifies that the cache blocking factor (ϕ) can be raised up to its upper bound without degradation in the overall performance when the one-time RFC configuration scheme is implemented. Similarly, when the percentage of the core function is small, the upper bound of ϕ is small and hence the value of ϕ needs to be maintained within that tight bound to avoid a degradation in the performance of the ABC processor. Besides, it can be observed that for a fixed portion of the core function, the upper bound of the ϕ remains almost constant irrespective of a large variation in the speed-up of the core function.

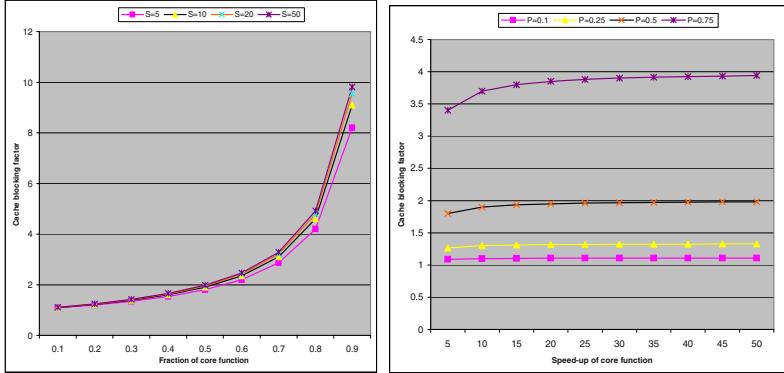


Fig. 2. Variation of cache blocking factor with (a) fraction of core function (b) speed-up in core function.

4.2 Execution Time Analysis with Continual RFC Configuration

When the design of ABC architecture is integrated with the continual RFC re-configuration scheme, the expression for X_{rfc} will be modified, again substituting the relevant parameters from Table 2, as shown:

$$X_{rfc} = X_A \left(\frac{1-P}{P} \right) + C'_p N + X_B \quad (6)$$

As argued in the earlier subsection, for the implementation of continual RFC configuration scheme, it is assumed that the cache blocking factor (ϕ) = 1 for practical purposes, though in the ideal scenario, $\phi \approx 1$. Now, for a gain in the performance, the execution time of application in the RFC based ABC processor should be less than the execution time of application in the base processor. Hence, solving $X_{rfc} < X_{org}$ yields

$$X_A \left(\frac{1-P}{P} \right) + C'_p N + X_B < \frac{X_A}{P} \quad (7)$$

from which the upper bound for the average configuration time (C'_p) can be obtained as:

$$C'_p < \left[\frac{X_A - X_B}{N} \right] \quad (8)$$

The upper bound on the average configuration time (C'_p) signifies that the total configuration overhead in the application should not exceed the difference in the execution time of the core function in the base processor and the RFC. This inference synchronizes with the generalized condition for the RFC based ABC processor to perform better than the base processor.

4.3 Effect of Number of Core Function Instances (N)

From the expression for X_{rfc} , it can be observed that, in general, the execution time $\propto N$, i.e., more the number of instances of the core function in the application, more will be the execution time due to the higher configuration overhead. However, for a considerably smaller number of N, the continual RFC configuration scheme would prove to be beneficial while the effect of a larger N can be offset by employing the one-time RFC configuration scheme. For example, the simulation study shows that the DCT/IDCT function has been called 129600 times in the mpeg2decode application, while it has been called 8,448 times in mpeg2encode application. Hence, while running the mpeg2decode application, the one-time RFC configuration need to be employed while the continual configuration scheme proves to be better while running the mpeg2encode application, as discussed in the results section.

4.4 Effect of Percentage of Core Function (P)

From the expression for X_{rfc} , it can be observed that the execution time $\propto (\frac{1-P}{P})$ and hence, more the percentage of the core function in the application, higher will be the speed-up of the overall application. This phenomenon is typical of the characteristic of Amdahl's Law, where the speed-up of the overall application is proportional to the portion of the application being accelerated. It can be deduced that when the portion of core function is smaller over the entire application, then it will not make a significant impact even when it is accelerated using an RFC. The simulation results, as shown in Section III, proves this principle. From Figure 1, it can be observed that the speed-up of around 4X is obtained in the execution of mpeg2decode application in RFC integrated ABC processor while the maximum speedup obtained in the execution of mpeg2encode application in RFC integrated ABC processor is only 1.07X and it can be seen that the DCT/IDCT computation occupies a portion of 77.27% in the mpeg2decode application, while it occupies a portion of only 6.89% in the mpeg2encode application.

4.5 Effect of Cache Blocking Factor (ϕ)

When all the cache modules are not available during the application execution time, size of the cache reduces along with reduction in the cache associativity, which causes the cache miss rate to increase. Subsequently, the execution time of the application increases. Hence, from the expression for X_{rfc} , it is obvious that the execution time $\propto \phi$. However, when the portion of the computations other than the core function in the overall application is significantly smaller, it will not be appropriate to employ the continual RFC configuration scheme, as the gain obtained due to the availability of full cache capacity for smaller execution time would be offset by the configuration overhead.

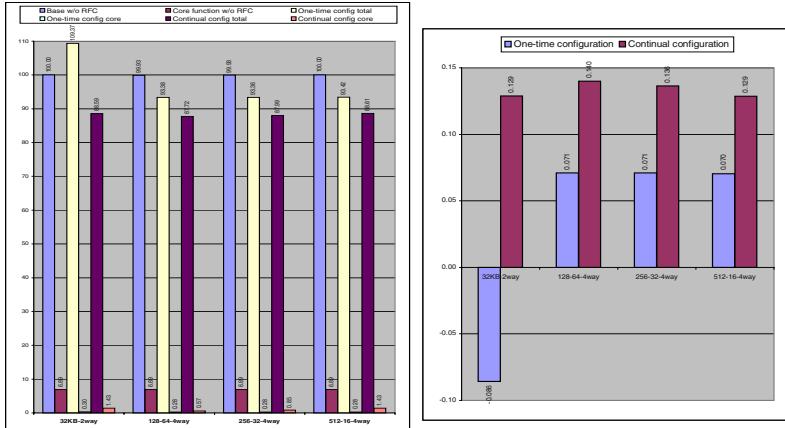


Fig. 3. (a) Normalized execution cycles in the base processor without RFC, and the ABC processor with different RFC configuration schemes (b) Relative performance improvement in two RFC configuration

5 Results and Analysis

From Figure 3(a), it can be observed that the performance of ABC processor with continual RFC configuration scheme is better than the base processor without the RFC and also the ABC processor with one-time RFC configuration scheme, for mpeg2encode application. This is due to the fact that the percentage of the core function over the entire application is significantly smaller and also the number of instances of the core function is not big enough to generate a considerable configuration overhead. Note that the portion of core function indicated in the continual RFC configuration scheme includes both, the RFC configuration overhead and the computation time for the core function. Another interesting observation is that when the core function is called for configuration in RFC N times, and the configuration time is (C_p) cycles for full loading of the RFC module with the configuration data from the memory, the total configuration overhead is found to be only 32% of the expected overhead $N*(C_p)$ cycles.

For the above reasons, the continual RFC configuration scheme performs better even in the 32KB 2-way cache while the one-time RFC configuration scheme results in a performance degradation, as shown in Figure 3(b). Also, it can be observed that there is an improvement in the performance of the continual RFC configuration scheme, with the reduction in the number of blocks in the RFC cache module. As the number of blocks in a cache module increases, the configuration time for RFC module increases and subsequently configuration overhead is significant, thus having a negative impact on the overall performance.

6 Conclusions

Reconfigurable Functional Cache (RFC) accelerates the computations using a specialized computing unit with minimal modification and overhead in area/time domains in the cache and microarchitecture. In continuance of our effort to build an efficient ABC architecture with improved performance, various RFC configuration schemes have been studied. The impact of various architectural parameters and the factors governing the structure of an application over the execution time of an application has been extensively studied. With the help of the study undertaken, the design of ABC microprocessor can be incorporated with the dynamic decision capability, so that appropriate RFC configuration scheme is chosen dynamically for running a particular application.

References

1. J. R. Hauser and J. Wawrzynek, “Garp: a MIPS processor with a reconfigurable coprocessor”, *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1997, pp. 12–21.
2. Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, “CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit”, *Proc. 27th International Symposium on Computer Architecture*, 2000, pp. 225–235.
3. A. DeHon, “DPGA-coupled microprocessors: commodity ICs for the early 21st Century”, *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, 1994, pp. 31–39.
4. R. Razdan and M. D. Smith, “A High-Performance Microarchitecture With Hardware-Programmable Functional Units”, *Proc. 27th Annual International Symposium on Microarchitecture, MICRO-27*, 1994, pp. 172–180.
5. P. Ranganathan, S. Adve, N. P. Jouppi, “Reconfigurable caches and their application to media processing” *Proc. 27th International Symposium on Computer Architecture*, 2000, pp. 214–224.
6. Huesung Kim, A. K. Somani, and A. Tyagi, “A Reconfigurable Multi-function Computing Cache Architecture”, *Proc. FPGA 2000*, February 2000, pp. 85–94.
7. Huesung Kim, A. K. Somani, and A. Tyagi, “A reconfigurable multifunction computing cache architecture”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume: 9, Issue: 4 , August 2001, pp. 509–523.
8. Huesung Kim, “Towards Adaptive Balanced Computing (ABC) Using Reconfigurable Functional Caches (RFCs)”, Ph. D. Dissertation, Dept. of Electrical and Computer Engineering, Iowa State University, July 2001.
9. Chunho Lee, M. Potkonjak and W. H. Mangione-Smith, “MediaBench: a tool for evaluating and synthesizing multimedia and communications systems”, *Proc. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, 1997, pp. 330–335.
10. Texas Instruments, “TMS320C6000 benchmarks”, 2000, available on <http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm>
11. Doug Burger and Todd M. Austin, “The SimpleScalar Tool Set, Version 2.0”, Computer Sciences Department Technical report # 1342, University of Wisconsin-Madison, June 1997.

Power-Aware Adaptive Issue Queue and Register File

Jaume Abella¹ and Antonio González^{1,2}

¹ Computer Architecture Department, U. Politècnica Catalunya, Barcelona (Spain)

{jabella, antonio}@ac.upc.es

² Intel Barcelona Research Center, Intel Labs, U. Politècnica Catalunya, Barcelona (Spain)

Abstract. In this paper, we present a novel technique to reduce dynamic and static power dissipation in the issue queue. The proposed scheme is based on delaying the dispatch of instructions whenever this delay is expected not to degrade performance. The proposed technique outperforms previous schemes in both performance and power savings. It achieves more than 34% dynamic and 21% static power savings in the issue queue at the expense of just 1.77% IPC loss. Significant power savings may be also achieved for the register file.

1 Introduction

Power dissipation has become a critical issue for both high performance and mobile processors. The issue logic and the management of speculative register values are some of the main sources of energy consumption in current superscalar microprocessors [7]. Additionally, these structures are one of the processor hotspots. Thus, reducing both dynamic and static power dissipation in these power hungry structures is critical not only from the energy standpoint but also from the temperature standpoint. Power savings can be achieved turning off some parts of these structures if they are not used. Additional savings can be achieved if some parts are turned off when they would hardly contribute to additional performance.

Literature on power reduction using adaptive schemes is very extensive. Among them, we could point out some schemes to reduce power and complexity [2][3][14].

Karkhanis, Smith and Bose [9] proposed a mechanism for saving energy by means of just-in-time instruction delivery. Their mechanism tries to limit the number of in-flight instructions. Their scheme triggers the resizing mechanism when the program enters a new program phase, which is detected by a change in the IPC or the number of executed branches. This approximation works well for coarse granularity but it may miss to identify some phase transitions correlated with other factors. For fine granularity schemes like ours, significant variations in the issue queue requirements can be observed for the same code section. Karkhanis' approach and ours work at different granularity so both schemes may be combined for higher benefits.

Different approaches have been recently proposed in order to reduce the dynamic power of the issue queue [8][6][11]. Folegnani and González [8] propose an issue queue design where energy consumption is effectively reduced using a dynamic resizing mechanism of the issue queue. Buyuktosunoglu et. al. [6] propose a detailed multiple-banked issue queue implementation for a similar scheme. In [6], the authors propose a simple mechanism to dynamically adapt the issue queue size. Their mechanism to reduce the issue queue is just based on the usage of the issue queue

entries. Ponomarev et. al. [11] propose a mechanism for resizing the issue queue, the rename buffer and the load/store queue also based on the usage of the entries of these structures. The approach presented in this work, as well as the approach presented in [8], are more aggressive than the techniques presented in [6] and [11] in the sense that they can reduce the size of the issue queue even when the entries would have been occupied but it is observed that they hardly contribute to improve performance.

In this work, we propose an adaptive microarchitecture for the issue queue and the reorder buffer that achieves significant dynamic and static power savings in the issue queue and the register file at the expense of a very small performance loss. Our proposal is based on observing how much time the instructions spend in the reorder buffer and the issue queue, and taking resizing decisions based on these observations. We compare this scheme with the approach in [8], and show that the proposed technique provides significant advantages.

The rest of the paper is organized as follows. Section 2 describes the baseline issue queue, register file and reorder buffer designs that have been assumed. Section 3 describes the proposed technique and the mechanism proposed in [8], which is used for comparison purposes. Section 4 evaluates the performance of the proposed approach. Finally, section 5 summarizes the main conclusions of this work.

2 Baseline Microarchitecture

2.1 Issue Queue

This work is based on a multiple-banked issue queue where the instructions are placed in sequential order. As previous works [8][6], no compaction mechanism for the issue queue has been assumed since compaction results in a significant amount of extra energy consumption.

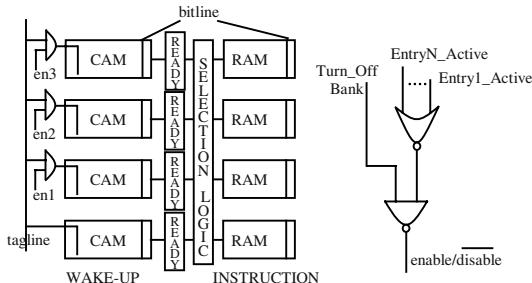


Fig. 1. Multiple-banked issue queue

The assumed issue queue is similar to the one described in [8][6] in which each bank can be turned off independently. Figure 1 shows a block diagram of the issue queue and the logic to turn off a bank. This scheme provides a simple mechanism to turn off at the same time the CAM array where the tags to be compared are stored, and the RAM array where the operands of the instructions are stored. The selection logic is always turned on but its energy consumption is much lower than that of the wakeup logic [10].

In order to avoid the wakeup of empty entries placed in turned on banks, we assume that the wakeup is gated in each individual entry that is empty [8]. This capability has been assumed for all the compared mechanisms including the baseline.

2.2 Register File

Integer and FP register files are identical. Registers are split into banks of 8 entries each. In order to reduce the access time, the bank selection and the decoding of the entry to be accessed may be done in parallel. This implementation of the register file increases its dynamic energy consumption. If the access time of this structure was not critical, a sequential decoding scheme could be considered.

Turning off unused banks can save static power for both schemes and dynamic power for the parallel one. A given bank is turned on as soon as at least one of its registers is assigned to an instruction, and it is turned off when none of its registers is being used. This scheme can be easily implemented adding a bit (*BusyBit*) to each register. This bit is set when a register is assigned to an instruction and is reset when it is freed. The bank *enable/disable* signal is a NOR function of its registers' *BusyBits*.

In order to maximize the number of banks that are turned off, when a free register is requested, the one with the lowest bank identifier is chosen so that the activity in the register file is concentrated on the banks with lower identifiers.

2.3 Reorder Buffer

Since the reorder buffer does not store register values, its contribution to the total energy consumption is very small and thus, reducing its energy consumption is not the objective of this work. The proposed reorder buffer has just one difference with respect to a conventional one: its occupancy can be limited dynamically. This feature is used to control the number of in-flight instructions and thus, to control the pressure on the issue queue and the register file.

3 Adaptive Schemes

This section describes the proposed mechanism and the mechanism used for comparison purposes [8].

3.1 Proposed Mechanism

Underlying Concepts. Superscalar processors try to keep full both the reorder buffer and the issue queue. In general, dispatching instructions as soon as possible is beneficial for performance, but not for power. From a performance standpoint, it is desirable not to delay the issue of any instruction. From a power standpoint, it is desirable that instructions remain in the issue queue for the minimum number of cycles in order to minimize the number of wakeup/selection attempts. Besides, reducing the issue queue occupancy allows for more opportunities to turn off parts of it. Our proposal tries to achieve these objectives by means of various heuristics:

- The first heuristic tries to reduce the time that instructions spend waiting for being issued in the issue queue. If it is observed that instructions wait too long, the reorder buffer size is reduced and thus, the dispatch of new instructions may be delayed. Reducing the number of entries in the reorder buffer reduces the number of instructions in the issue queue and reduces the number of registers in use.
- The second heuristic tries to prevent situations where the limited instruction window size (reorder buffer size) is harming performance. Even if instructions spend too much time in the issue queue, it is not desirable being too aggressive if there are few instructions in the reorder buffer.
- Finally, L2 data cache misses have a very long latency and stall the commit of instructions for many cycles. Thus, in case of an L2 miss it is interesting to increase the instruction window size to allow the processor to process more instructions while the miss is being serviced.

Deciding when instructions spend too long in the issue queue is one of the tricky parts of the mechanism. We are interested in finding out the minimum number of cycles that the instructions require to spend in the issue queue without losing significant IPC. In order to gain some insight, we have experimentally observed the behavior of different programs for short intervals of time. If we just consider the intervals with similar IPC, we observe some trends: a) the minimum time that the instructions spend in the issue queue and the time that they spend in the reorder buffer are correlated, b) this correlation is not linear: the longer the time in the reorder buffer, the longer the time in the issue queue but the ratio between the latter and the former decreases as the time spent in the reorder buffer increases.

Implementation of the Mechanism. The first and second heuristics outlined above are based on measuring the number of cycles that instructions spend in the issue queue and in the reorder buffer. However, an exact computation of these parameters may be quite expensive in hardware (e.g. time stamps for each entry) and consume a non-negligible amount of energy. According to Little's law [13] for queuing systems in which a steady-state distribution exists, the following relation holds:

$$L_q = \lambda W_q. \quad (1)$$

where L_q , λ and W_q stand for the average queue size, the average number of arrivals per time unit and the average time that a customer spends in the queue. In the issue queue and the reorder buffer, the arrival rates (λ) are exactly the same so, instead of counting how many cycles (W_q) every committed instruction spends in the issue queue and the reorder buffer we will count how many instructions (L_q) are in these structures every cycle. Then, all instructions that arrive to the queues but do not commit are also counted. We have observed in our experiments that the effect of considering or not these instructions does not provide significant differences. We have observed that this relation between queue size and waiting time holds.

In order to leverage the relation between the time spent in the issue queue and the time spent in the reorder buffer, the proposed mechanism uses the ratio between both occupancies (IQ occupancy / ROB occupancy) to take resizing decisions.

These thresholds are dynamically adapted depending on the reorder buffer size. Figure 2 details our approach. *ROB_size* stands for the physical size of the reorder buffer (128 instructions in our evaluation), and *ROB_dyn_size* stands for the

```

(1) THRESHOLD_LOW = 1 - ROB_dynamic_size / ROB_size
(1) THRESHOLD_HIGH = THRESHOLD_LOW + 1/8

(2) FRACTION = #instr_in_IQ / #instr_in_ROB

(3) if (FRACTION > THRESHOLD_HIGH)
(3)     ROB_dyn_size = max(ROB_dyn_size-8, 32)
(3) else if (FRACTION < THRESHOLD_LOW)
(3)     ROB_dyn_size = min(ROB_dyn_size+8, ROB_size)

(4) if (L2 miss during the period)
(4)     ROB_dyn_size = min(ROB_dyn_size+8, ROB_size)

(5) if (#cycles_disp_stall > IQ_THRESHOLD_HIGH)
(5)     IQ_dyn_size = min(IQ_dyn_size+8, IQ_size)
(5) else if (#cycles_disp_stall < IQ_THRESHOLD_LOW)
(5)     IQ_dyn_size = max(IQ_dyn_size-8, 8)

```

Fig. 2. Heuristics to resize the reorder buffer and the issue queue

maximum number of allowed instructions in the reorder buffer at a given time (similar definition is applied to *IQ_size* and *IQ_dyn_size*). In order to avoid an extremely small size of the reorder buffer dynamic size, the following constraint is applied: $ROB_size/4 \leq ROB_dyn_size \leq ROB_size$. The thresholds are set according to (1). The fraction of time that instructions spend in the issue queue versus the time that they spend in the reorder buffer is approximated as (2). This parameter is averaged for each interval of time. At the end of each interval, resizing decisions are taken according to the criteria described in (3).

Finally, whenever there is an L2 cache miss, the reorder buffer size is increased if it is below its maximum size, as (4) in figure 2 shows. In theory only data misses should be considered but for the sake of simplicity, we do not distinguish between instruction and data misses since most of them correspond to data. This heuristic avoids losing significant performance for those programs with high data miss rates.

Issue queue occupancy is further controlled by a mechanism that monitors how many cycles the dispatch is stalled because the issue queue is full. As detailed in (5), if stalls are too frequent, the issue queue is augmented (#cycles_disp_stall stands for the cycles that dispatch is stalled because the instructions cannot be placed in the issue queue). If they are very rare, the queue is decreased.

These simple mechanisms achieve a significant issue queue size reduction with very small performance loss. In our experiments, the issue queue thresholds that obtain significant power savings and small performance degradation are the following ones: <16,32> and <16,64>. To simplify the implementation and avoid doing some divisions and multiplications, integer arithmetic is used instead of FP one. In particular, the thresholds are scaled as follows:

```

THRESHOLD_LOW = ROB_size - ROB_dyn_size
THRESHOLD_HIGH = THRESHOLD_LOW + ROB_size/8

```

In our experiments we use a 128 entry reorder buffer, so *THRESHOLD_HIGH* corresponds to *THRESHOLD_LOW* + 16. Thresholds are compared with *FRACTION* so this parameter should also be scaled as follows:

```
FRACTION = ROB_size x #instr_in_IQ / #instr_in_ROB
```

The multiplication in the expression above is trivial to implement since the reorder buffer size is a power of 2. For the division, the dividend has 11 bits and the divisor has 7 bits, assuming an interval of 128 cycles. This requires a rather small hardware. In fact, an iterative divider can be used instead of a parallel one, since delaying the resizing decisions by a few cycles does not have any practical impact.

The energy consumption of the additional hardware is negligible because only three small counters are updated every cycle and the rest of the structures work only once every interval (128 cycles in our experiments).

3.2 The Mechanism Used for Comparison

The proposed mechanism has been compared with the mechanism proposed in [8], which will be referred to as *FoGo* in the rest of the paper. The comparison has been done in two ways: using the parameters that the authors of *FoGo* reported as the best ones and using the same resizing interval as our mechanism (128 cycles). The issue queue has the same structure for both the proposed mechanism and the mechanism used for comparison, but the resizing schemes are different.

FoGo mechanism monitors the performance contribution of the youngest bank of the issue queue (8 instructions in their experiments) and measure how much these entries contribute to the IPC. If the contribution is below a threshold, the issue queue size is reduced (one bank is turned off). In particular, this mechanism counts the number of committed instructions that were issued from the 8 youngest entries in the issue queue. If there are less than N instructions issued from the youngest part during an interval of time, the issue queue size is reduced. Their experiments showed that an interval of 1000 cycles and a threshold of 25 instructions save considerable power in the issue queue with a very small performance loss. Every 5 intervals, the issue queue size is increased by one bank.

For the comparison presented below, we have chosen the configuration with the parameters that they report as the more appropriate ones (*FoGo1000*) and the same parameters but with our interval of 128 cycles with a corresponding threshold of 3 instructions issued from the youngest part (*FoGo128*).

4 Performance Evaluation

This section presents results for the proposed mechanism, and compares them with the technique proposed in [8]. Detailed results can be found in the report [1].

4.1 Experimental Framework

Power and performance results are obtained through Wattch [4], which is an architecture-level power and performance simulator based on SimpleScalar [5]. The model required for multiple-banked structures has been obtained from CACTI 3.0 [12], which is a timing, power and area model for banked memories. Table 1 shows the processor configuration.

Table 1. Processor configuration

Fetch, decode, issue, commit width: 8 instructions		
Branch pred.: Hybrid 2K Gshare, 2K bimodal, 1K selector	BTB: 2048 entries, 4-way	
L1 Icache: 64KB, 2-way, 32 byte line (1 cycle)		
L1 Dcache: 64KB, 4-way, 32 byte line, 4 R/W ports (2 cycles)		
L2 cache: 512KB, 4-way, 64 byte line (10 cycles hit, 50 cycles miss, 2 cycles interchunk)		
Fetch queue: 64 entries	IQ: 80 entries (10 banks with 8 entries each)	ROB: 128 entries
INT register file: 112 (14 banks x 8), 16R+8W ports	FP register file: same as INT	
INT functional units: 6 ALU, 3 mult/div	FP functional units: 4 ALU, 2 mult/div	
Technology: 0.10 μ m		

4.2 Benchmarks

For this study we have used the whole Spec2000 benchmark suite [15] with the *ref* input data set. We have simulated 100 million of instructions for each benchmark after skipping the initialization part. The benchmarks were compiled with the Compaq/Alpha compiler with `-O4 -non_shared` flags.

4.3 Interval Length

In order to choose a suitable interval to resize the structures, we have done some experiments. Figure 3 shows the IPC with respect to the baseline for different interval lengths using 3 integer benchmarks and 3 FP ones. It can be seen that in general, longer intervals improve performance. Figure 3 also shows the reorder buffer occupancy reduction for different interval lengths. It can be observed that shorter intervals achieve higher occupancy reduction. Higher occupancy reduction will translate into better opportunities to save power.

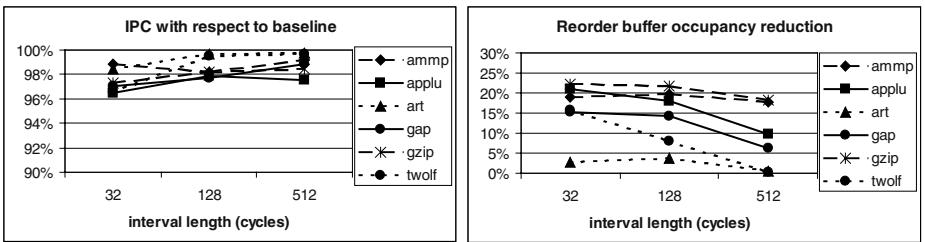


Fig. 3. IPC and reorder buffer occupancy reduction for different interval lengths

Figure 3 shows that a 32-cycle interval hardly reduces the reorder buffer occupancy with respect to a 128-cycle interval whereas it results in slightly higher performance degradation. In addition, the shorter the interval, the higher the energy overhead of resizing the structures. The 512-cycle interval is slightly better in terms of performance but it is not so effective to reduce the reorder buffer occupancy. Thus, the 128-cycle interval achieves the best tradeoff between power and performance.

4.4 Performance and Power Results

The performance evaluation has been done comparing two versions of the proposed technique, two versions of the *FoGo*, *FoGo128* and *FoGo1000* as described above, and a baseline with no adaptive resizing. The two versions of our technique correspond to different threshold values for the *IQ_THRESHOLD_HIGH* (32 or 64). We will refer to them as *IqRob32* and *IqRob64* respectively in the rest of the paper. The baseline architecture has the mechanism that we have assumed for *IqRob* and *FoGo* to avoid the wakeup of empty entries in the issue queue.

Performance. Figure 4 shows the IPC loss for the different mechanisms. *IqRob32* and *IqRob64* have better performance than *FoGo1000* and *FoGo128* respectively for the whole Spec2000. *FoGo* reduces the size of the issue queue when the IPC contribution of the youngest bank is below a fixed threshold. This threshold basically determines the loss of IPC that the mechanism may cause and thus, it has a bigger impact for programs with lower IPC, such as some of the SpecINT2000.

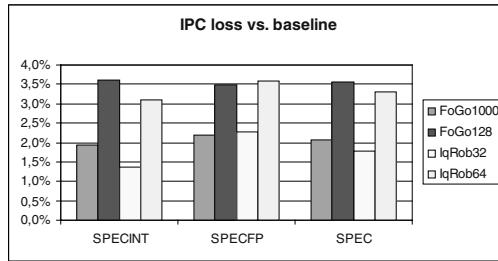


Fig. 4. IPC loss for different techniques

Reorder Buffer. *IqRob* achieves lower reorder buffer occupancy than *FoGo* since the former resizes the reorder buffer in order to stall the dispatch process when it is expected that new instructions will not increase performance. Having fewer instructions in the reorder buffer implies that fewer issue queue entries and registers are used so more power is saved. *IqRob* outperforms *FoGo* especially for integer applications due to their lower ILP.

Table 2 shows the effectiveness of *IqRob* for reducing the reorder buffer size. On average, the maximum reorder buffer size is set to about 70% of its total capacity. About 45% of the entries are occupied and 25% of the entries are enabled but empty. This is mainly due to sections of code where instructions spend few cycles in the issue queue. The *IqRob* mechanism tends to increase the reorder buffer size in these situations because these instructions do not waste much power in the issue queue.

Table 2. Reorder buffer size reduction

	SpecINT	SpecFP	Spec
<i>IqRob32</i>	35.4%	23.5%	29.0%
<i>IqRob64</i>	34.2%	22.4%	27.9%

Issue Queue. Both *FoGo* and *IqRob* resize the issue queue (through different heuristics) but in addition, *IqRob* resizes the reorder buffer, which causes in turn a reduction in the issue queue occupancy as a side effect.

IqRob32 and *IqRob64* achieve higher occupancy reduction in the issue queue than *FoGo1000* and *FoGo128* respectively. This occupancy reduction allows *IqRob64* to turn off 29% of banks for the whole Spec2000. *FoGo128* turns off only about 1% more banks than *IqRob32*, but it loses twice as much performance when compared with the baseline. The effect of turning off these banks is a reduction of the dynamic and static power requirements of the issue queue. Additionally, some extra dynamic power is saved from avoiding the wakeup of empty entries, which is the only source of the power savings for the baseline. Figure 5 shows this effect. *IqRob32* and *IqRob64* outperform *FoGo1000* and *FoGo128* respectively in power savings.

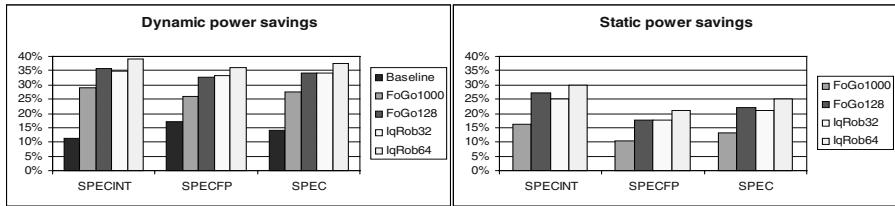


Fig. 5. Issue queue dynamic and static power reduction

Register File. As discussed above, reducing the number of in-flight instructions results in a lower number of registers in use. *IqRob* achieves higher reductions than *FoGo* due to its higher effectiveness at reducing the reorder buffer size. For instance, *IqRob32* reduces the register pressure by 18% and 13% for integer and FP registers respectively. If the unused banks are turned off, 27% dynamic and 31% static power savings are achieved for the integer register file, whereas 14% dynamic and 19% static power savings are achieved for the FP register file.

Dispatched Instructions. The *IqRob* mechanism is not designed to retrain control-flow speculation. However, the stall of instruction dispatching results as a side effect in a reduction of control-flow misspeculated instructions, which results in significant additional power savings. *IqRob32* and *IqRob64* reduce the number of dispatched instructions by 3.3% and 5.1% respectively, whereas *FoGo1000* and *FoGo128* reduce it by 2.5% and 3.7% respectively.

5 Conclusions

We have presented a novel scheme that dynamically limits the number of in-flight instructions in order to save dynamic and static power in the issue queue and the register file. The proposed mechanism is based on monitoring how much time instructions spend in both the issue queue and the reorder buffer and limit their occupancy based on these statistics. The proposed mechanism has been evaluated in terms of dynamic and static power, and performance.

Results have been compared with a previously proposed issue queue resizing technique, and significant gains have been observed in terms of performance and power savings. The proposed technique achieves more than 34% dynamic and 21% static power savings in the issue queue with a performance degradation of only 1.77%. Significant power savings are also achieved for the register file: 27% dynamic and 31% static power savings for the integer registers, and 14% dynamic and 19% static power savings for the FP ones.

Acknowledgements. This work has been supported by CICYT project TIC2001-0995-C02-01, the Ministry of Education, Culture and Sports of Spain, and Intel Corporation. We would like to thank the anonymous reviewers by their comments.

References

1. J. Abella, A. González. Power-Aware Adaptive Instruction Queue and Rename Buffers. Technical Report UPC-DAC-2002-31 at <http://www.ac.upc.es>
2. R.I. Bahar, S. Manne. Power and Energy Reduction Via Pipelining Balancing. ISCA 2001.
3. Y. Bai, R.I. Bahar. A Dynamically Reconfigurable Mixed In-Order/Out-of-Order Issue Queue for Power-Aware Microprocessors. In VLSI 2003.
4. D. Brooks, V. Tiwari, M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In ISCA 2000.
5. D. Burger and T. Austin. The SimpleScalar Tool Set, Version 3.0. Technical report, Computer Sciences Department, University of Wisconsin-Madison, 1999.
6. A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose and P. Cook. A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors. In GLSVLSI 2001.
7. J. Emer. EV8: The post-ultimate alpha. Keynote at PACT 2001.
8. D. Folegnani and A. González. Energy-Effective Issue Logic. In ISCA 2001.
9. T. Karkhanis, J.E. Smith, P. Bose. Saving Energy with Just in Time Instruction Delivery. In ISLPED 2002.
10. S. Palacharla, N.P. Jouppi, J.E. Smith. Complexity-Effective Superscalar Processors. In ISCA 1997.
11. D. Ponomarev, G. Kucuk, K. Ghose. Reducing Power Requirements of Instruction Scheduling Logic Through Dynamic Allocation of Multiple Datapath Resources. In MICRO 2001.
12. P. Shivakumar and N.P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power and Area Model. Research report 2001/2, WRL, Palo Alto, CA (USA), 2001.
13. W.L. Winston. Operations Research Applications and Algorithms. Ed. Duxbury Press. Second edition, 1991.
14. S.H. Yang, M.D. Powell, B. Falsafi, T.N. Vijaykumar. Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay. In HPCA 2002.
15. SPEC2000. www.specbench.org/osg/cpu2000/

FV-MSB: A Scheme for Reducing Transition Activity on Data Buses

Dinesh C. Suresh¹, Jun Yang¹, Chuanjun Zhang², Banit Agrawal¹, and Walid Najjar¹

¹Computer Science and Engineering Department
University of California, Riverside, CA 92521

²Electrical Engineering Department
University of California, Riverside, CA 92521
{dinesh, junyang, chzhang, bagrawal, najjar}@cs.ucr.edu

Abstract. Power consumption becomes an important issue for modern processors. The off-chip buses consume considerable amount of total power [9,7]. One effective way to reduce power is to reduce the overall bus switching activities since they are proportional to the power. Up till now, the most effective technique in reducing the switching activities on the data buses is Frequent Value Encoding (FVE) that exploits abundant frequent value locality on the off-chip data buses. In this paper, we propose a technique that exploits more value locality that was overlooked by the FVE. We found that a significant amount of non-frequent values, not captured by the FVE, share common high-ordered bits. Therefore, we propose to extend the current FVE scheme to take bit-wise frequent values into consideration. On average, our technique reduces 48% switching activity. The average energy saving we achieved is 44.8%, which is 8% better than the FVE

1 Introduction

Power dissipation for modern processors has become more and more important due to reliability concerns, packaging costs and mobility requirements. Among various components, the off-chip buses consume a significant amount of the total power. Stan et.al, have estimated that the power dissipated by the I/O pads of an IC ranges from 10% to 80%with a typical value of 50% for circuits optimized for low power [7]. This is due to high capacitance of the off-chip buses – up to three orders of magnitude higher than the average on-chip interconnect capacitance. It is known that power is roughly proportional to the product of capacitance and circuit switching activity. In other words, when the transition activities of the off-chip buses are increased, the power consumption is also increased proportionally. Hence, one efficient solution to reduce the power on the off-chip buses is to minimize the average transition activities through data encoding.

Both off-chip address buses and data buses are targets for encoding. However, the majority of the research in the past has focused on address buses. This is because the instruction addresses are mostly sequential, creating opportunities for simple and effective encoding. However, encoding for values transferred on data buses is not easy since data streams are less regular than address streams. Currently, encoding schemes that can be applied to data buses without prior knowledge of the applications

include bus-invert encoding [7], working-zone encoding (WZE) [13], adaptive encoding [2] and frequent value encoding [8,1].

The bus-invert encoding transfers a data value either in its original form or its complement form depending on whose hamming distance with the previous bus transmission is smaller. Working-Zone-Encoding (WZE) caches references in each working zone. During subsequent references to the same working zone, the offset with respect to the previous reference is transmitted. Bus Expander [11] and Dynamic Base Register Caching (DBRC) [12] propose compaction techniques to increase the effective bus width. DBRC uses dynamically allocated base registers to cache the high order bits of address values. Bus Expander encodes the high ordered several bits of a data value to effectively increase the bus width. The adaptive encoding scheme is capable of on-line adaptation to the value streams by learning statistics on the fly. As collecting the accurate statistics for the value streams can be very expensive, the proposed adaptive scheme operates bit-wise rather than word-wise. Thus, it loses the correlation among the bits of a single value. The frequent value-encoding (FVE) scheme is by far the most effective way of reducing the transition rate for data buses.

The FVE has exploited the high temporal locality in full-width data value streams successfully. However, we have found that the locality in the partial-width data values is also abundant. This is especially true for the high-ordered bits of the data values such as pointer values and small integer values. In this paper, we propose a new technique that exploits this locality on top of the original FVE. We use a separate small storage to capture the most frequent high-ordered bits of value streams and apply parallel FVE on both full-width and partial-width words. Our scheme achieves 48% reduction in switching activities over the unencoded data and provides an extra 8% energy saving over the original FVE.

In the next section, a brief overview of FVE mechanism is given. Then in Section 3, an in-depth observation of the values transmitted on the data bus is described. Section 4 describes our FV-MSB technique in detail. Evaluation results are illustrated in Section 5 and Section 6 respectively.

2 Previous Work

The frequent value-encoding (FVE) scheme is based on the observation that the data values transmitted on the data buses tend to reoccur frequently [9]. To take advantage of this feature, two identical codebooks are placed at two ends of buses to maintain those frequent values. The codebook is organized as a linear list where each value has a unique index. On each bus transaction, the codebook is looked up to see if the value to be transmitted is frequent or not. If it is frequent, a code is generated indicating the index in the codebook where the value is stored. Thus the receiving end can use this code to locate the frequent value in its own codebook. If a value is non-frequent, it is sent directly onto the bus without encoding. A control signal is needed to distinguish between a non-frequent value and a frequent value only when it can cause confusion at the receiving end.

A code for a frequent value has the flavor of “one-hot-code” meaning that there is only a single-bit “1” present in the code and all the rest bits are “0”s. The position of the bit “1” exactly determines the index of the frequent value in the codebook. For example, a code “1000” means the bus is transmitting the first frequent value in the

four-entry codebooks on both ends of the bus. The advantage of using “one-hot-code” for frequent values is that it can guarantee low transition activity if the frequent values are abundant. The disadvantage is that the codebooks cannot contain more number of frequent values than the number of bus wires.

Changes are made to the frequent value set in the codebook by applying LRU replacement policy on every new value being looked up. Thus the codebooks essentially keep the most recently seen values over a window of ‘n’ values, where ‘n’ is the codebook capacity. The codebooks can be easily implemented in hardware using a Content Addressable Memory (CAM). For a k-bit wide data bus, we use a k-bit wide, k-entry CAM to hold the frequent values. For the rest of this paper, we will refer to the CAM that holds the frequent values as FV CAM.

As used in other techniques [2], a pair of correlator and decorrelator is added to the two ends of buses. They are inverse functions of each other and their purpose is to reduce the correlations between successive values. The schematic block diagram of the frequent value encoding is shown in Fig. 1.

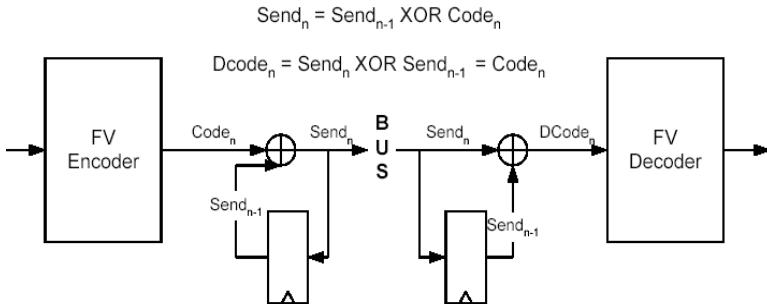


Fig. 1. Frequent Value Encoding Scheme

3 Motivation

Since the FVE scheme has a dynamically changing set of frequent values, it exploits the short-term temporal locality on the data bus effectively. Any improvement to the FVE scheme should focus on capturing more values in the FV CAM .We therefore design such a scheme. The values to be placed on the off-chip data bus depend on the nature of the data set, the program and the micro architectural features of the system. Even for a given application, different phases of the program might exhibit different access patterns. For example, the data access pattern for a normal program and for a linked-list traversal program might differ drastically. Table 1 shows an example of data trace segments for a linked-list traversal and a normal program.

A list might often contain a set of contiguously allocated pointer values and hence the values on the data bus might often differ by just a few bits. If we have multi-threaded execution pattern or if we have instruction values in between data streams, these consecutive values might be separated by other values and hence, passing consecutive pointer values might result in a lot of transitions. Let us look at how such consecutive pointer values are handled by the FVE scheme. Each time a new pointer value is encountered, it is treated as a non-frequent value and hence, the data is sent unencoded. In the example shown in Table 1, the highlighted values in the linked list

trace correspond to pointer values and are always sent unencoded in the FVE scheme. One might observe that, for most of the consecutive pointer values, most of the high ordered bits remain unchanged. Hence, if we can cache the high ordered bits in a separate CAM, many values that were earlier treated as non-frequent by the FVE scheme can be encoded as frequent values.

Table 1. Data trace segments for linked list and a for loop

Linked List trace	Normal program
10005098	00000048
0000001a	00000006
00000012	00000042
100050d8	8048c0f6
00000000	00000047
00000036	00000002
100050f8	00000006
0000001a	00000046
00000012	00000006

Besides pointer values, many non-frequent values might differ from frequent values by small magnitudes. If such non-frequent values were separated from frequent values by other un-related values, we would have a lot of switching activity on the bus. The values 42 and 47 are separated by an unrelated value in the normal program trace shown in Figure2. We find that besides caching frequent values, caching a few upper order bits would reduce the transition activity in the bus significantly.

Based on the above observations, we want to devise a scheme in which both entire data value and its most significant bits (MSB) are kept in CAMs. The FV CAM serves its original purpose while the MSB CAM is to capture the locality in the high-ordered bits as described. We will study how many high-ordered bits are needed and how they interact with the FV CAM. Since we are using more hardware than before, our design goal is to further reduce the transition activity, and thus, the overall power consumption, without adding expensive hardware resources such as extra control signals. Moreover, we design the layout of our codec so that it is fast and consumes power economically. We provide power and delay analysis of the codec and consider the power overhead of the additional hardware when evaluating our technique.

4 FV-MSB Encoding

Fig. 2 gives a high-level picture of our FV-MSB design. The FV-CAM functions as the original FVE scheme and MSB CAM is to capture more localities of the high-ordered m bits for every value being transferred. If a value hits in both CAMs, we give higher priority to the FV CAM since it helps reduce more switching activity (middle AND gate). In other words, in the event of a FV CAM miss, the MSB CAM is useful (AND gate on top). If the value missed in both CAMs, the original binary form is sent over the bus as usual (the AND gate at the bottom). The gates in Fig. 2 represent the logic design of the FV-MSB scheme. In reality, each wire should be

wired in a similar way. We choose to search both CAMs in parallel in order to minimize the delay on the critical path.

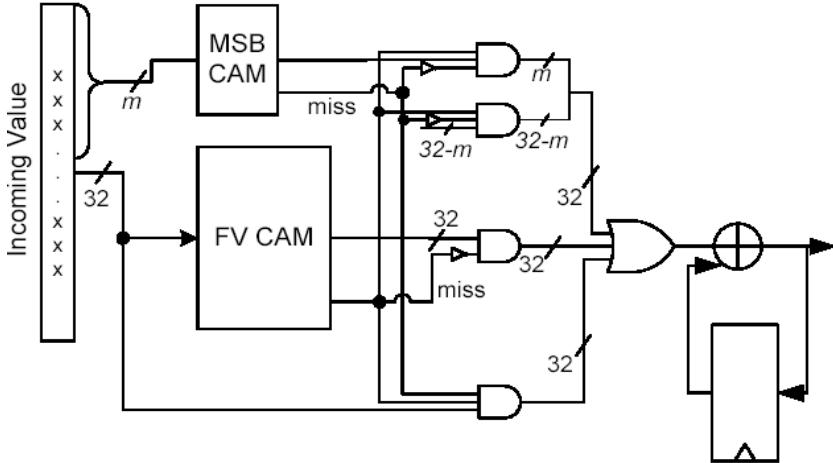


Fig. 2. Frequent Value-MSB encoding scheme

Two difficulties arose while we were designing the FV-MSB scheme. The first is that the MSB CAM might require another control signal just as the original FVE since a non-frequent value may look like a value whose high-ordered m -bits are encoded by the MSB CAM. The second issue is the value of m . If m is small, we would have a higher hit rate in MSB CAM, but each hit does not bring too much reduction in switching. If m is large, the MSB CAM would have lower hit rate, but each hit results in better switching reduction. Next we illustrate these two issues separately.

4.1 Removing the Additional Control Signal

The original FVE requires a control signal to inform the decoder that the transmission on the data bus is a non-frequent value but having a “one-hot code” form. Since the MSB CAM is essentially a smaller scale FV-CAM, a similar control signal would be necessary to guarantee the correctness of the encoding process. This means that we would use two extra off-chip pins and wires to set up special encoding condition. Since off-chip pins and wires are expensive resources and also introduce toggles on the bus, we strive to design the FV-MSB scheme such that the extra control signal is not needed.

Let us first look at those cases that require two control signals. At the receiving side, the decoder should be able to tell the following situations:

1. Is the incoming value a frequent or a non-frequent value? A frequent value may not appear as a pure “one-hot code” since high-ordered m bits might be encoded while the remaining $32-m$ bits contain “1”’s
2. If the incoming value is a frequent value, was it encoded using the MSB or the FV CAM? An incoming value with the “hot” bit being among the high-ordered m bits confuses the decoder in selecting the right CAM to decode.

Let's assume the four combinations of the two control signals represent different encoding meanings as the following. 1) "00" means the value missed both CAMS and the bus transmission is an unencoded value; 2) "01" means the value hit in the MSB CAM; 3) "10" means the value hit in the FV CAM; 4) "11" is left unused.

To remove one signal, we need to combine two cases in the above categories so that we are left with only two cases for which a single signal would suffice. Combining "00" and "01" is impossible due to the reason listed in 1 above. Combining "00" and "10" is also impossible since a non-frequent value with "one-hot code" form cannot be distinguished. Therefore, we can only combine "01" and "10" but at the cost of encoding less number of frequent values that are being hit in the MSB CAM. Here, we choose to give up encoding those values that hit in the MSB but miss in the FV-CAM and their lower $32-m$ bits are "0"s. Those values will be transmitted as unencoded non-frequent values. After this modification, the receiving side will see only the following forms of bus values:

- a) A value of pure one-hot code form.
- b) A value of "one-hot code" in the high-ordered m bits and non-zero in the low-ordered $32-m$ bits.
- c) A value of other forms including non-frequent values having "one-hot code" form and frequent values whose "hot" bit is in the high-ordered m bits.

Cases a) and b) are encoded frequent values and can be indicated by a signal setting at "1", and case c) is for non-frequent values and can be indicated by the signal setting at "0". Thus, we successfully removed the additional control signal.

4.2 Selecting the Size of m

The number of bits (m) per value we choose to store in the MSB CAM is also the number of entries of the CAM. Intuitively, small values of m bring in large number of hits in the MSB CAM. However, large number of hits need not necessarily imply a large reduction in switching activities. For example, if $m=3$, then there are about $3/8$ ($m/2^m$) of the total values hit in the MSB CAM. However, the average number of reduction in switching activities is only 0.5 ($3/2 - 1$). If m is larger, the hit rate will decrease, but the effect of the hit on switching reduction will increase. To find the best value of m , we conducted experiments where we varied m from 8 to 30 with step value of 2 and measured 1) hit ratios in MSB CAM, and 2) overall switching reductions of the FV-MSB. The experiments were carried out with configurations specified in Section 5

Fig. 3 plots the averaged ratio of the MSB CAM hits normalized to the FV CAM hits over the 7 benchmarks we ran. The curve is bell shaped which shows an interesting feature. Intuitively, the hit ratio should be higher when m is smaller. Meanwhile the number of entries in the MSB CAM is fewer which means that the MSB CAM keeps fewer number of high-ordered m bits and hence, captures less locality. These two contradictory conditions cause the curve to rise from 8-entry to 16-entry MSB CAMs. The curve reaches the peak at 16-entry CAM and falls from then on.

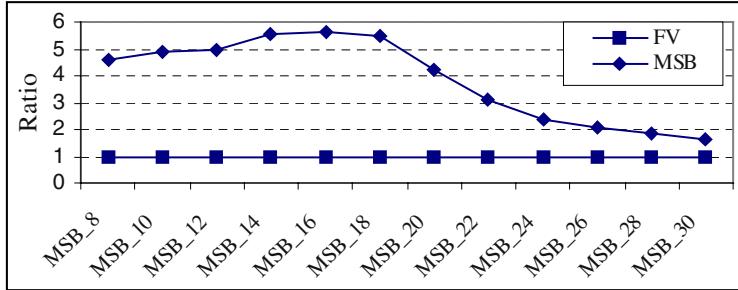


Fig. 3. Number of hits in the MSB table vs. number of hits in the FV table

Fig. 4 shows the percentage of switching reduction in the same experiment. As we expected, having higher hit rate in the MSB CAM does not necessarily result in higher switching reductions. However, a low hit rate will definitely hurt the reductions in the switching activities. Hence, an optimal point that gives the peak reduction is desired. From the graph, we can see that the MSB CAM with 20 entries outperforms all other MSB CAMs. Therefore, we use a 20-entry MSB CAM to hold the high 20-bits of data values in our FV-MSB scheme.

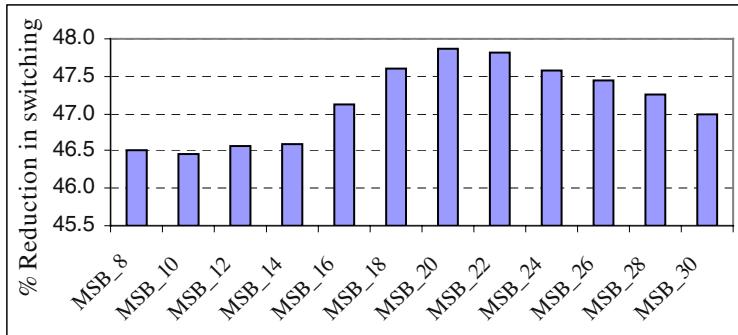


Fig. 4. Percentage of reductions in switching activity for SPEC2000INT

5 Evaluation

In this section, we present the experimental evaluations of FV-MSB and compare the data bus switching reductions with four other schemes. We used execution-driven Simplescalar-3.0 tool set [4] with a configuration that conforms to modern high performance processors. We modified the source code of *sim-outorder.c*, an out-of-order execution simulator to monitor the activities on the off-chip data bus which is between two level caches and the memory. The L1 cache size is 64KB for both data and instructions; the L2 cache is 512KB. We modeled the FV-MSB scheme together

with four other analogous schemes for comparison. The descriptions of two out of four schemes are as follows (the first one is bus-inversion and the fourth one is FVE).

FVE-64 (a 64-entry FV CAM encoding): The FV-MSB scheme uses an MSB CAM in addition to the FV CAM. The total number of bits stored in the CAMs altogether is higher than the normal FVE. One way of comparing the two schemes is to increase the capacity of the CAM in FVE. However, the authentic FVE does not allow the number of entries more than the number of bus wires (in this paper, we assumed 32). A straightforward way of realizing it is to use a CAM whose number of entry is a multiple of 32. Meanwhile, we use the same number of control signals to select among different 32 entries. For example, if we use a 64-entry FV CAM in FVE, we need to use 1 signal to choose between the upper 32 entries and the lower 32 entries of the CAM. In addition, we need a second signal to indicate a non-frequent value having “one-hot code” form. Notice that this method is neither scalable in terms of CAM size nor realistic in terms of the number of control signals it requires. For the purpose of comparison, we pessimistically pick a 64-entry CAM to compare with our FV-MSB (32-bit, 32-entry FV CAM and a 20-bit 20 entry MSB CAM).

FV-Inversion: This scheme combines the traditional bus-invert and FVE together: whenever a value miss occurs in the FV CAM, the bus invert algorithm is applied. In reality, the two encodings can be implemented in parallel to reduce delays just as in our FV-MSB scheme. However, unlike our FV-MSB scheme, this combination also requires two additional control signals, one for bus-invert and the other for FVE.

Fig. 5 shows the percentage reduction in switching activities for SPECINT applications. FVE-64 and FV-Inversion have an additional control signal overhead compared to the FVE-32 and the FV-MSB scheme. Besides using more hardware than the FVE-32 scheme, they can only provide modest reduction in switching activities. On average, the FV-MSB scheme provides 10% more switching reductions compared with the FVE-32 scheme and provides 48% switching reductions compared with unencoded data. The increase in switching reductions is chiefly due to the fact that FV-MSB scheme can track pointer values effectively. Fig. 7 in Section 6 gives the percentage energy reduction for different schemes while running the SPECINT benchmarks.

6 Energy, Delay, and Area Measurement

To determine the power consumption and delay of the coder itself, we created an actual layout of the CAM and other components of the FV-MSB scheme. Fig. 6 (left) is the CAM cell circuit, which is composed of a conventional six-transistor SRAM cell and dynamic XOR comparators. Since CAM search time is critical in our FV-MSB design, we used two separate bit lines: Cbit and Bit, to decrease the capacitance on the Cbit search line. Fig. 6 (right) shows the layout of the CAM cell. We used Cadence [3] layout tools and extracted the circuit from the layout.

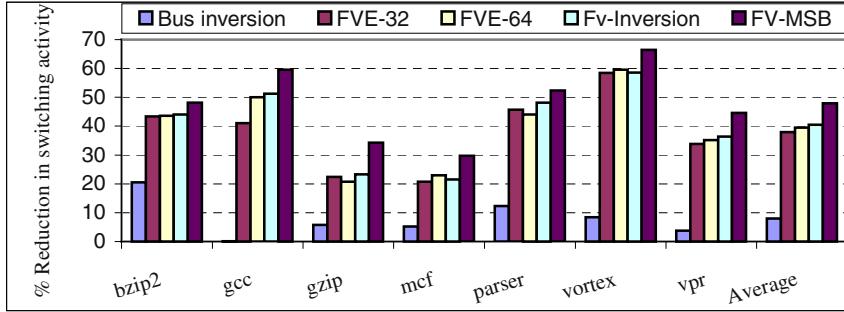


Fig. 5. Percentage reductions in switching activity for SPEC2000INT

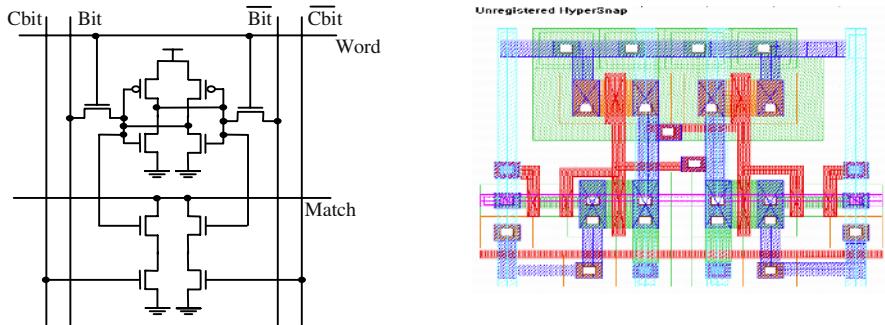


Fig. 6. CAM cell circuit (left) and layout (right)

The technology we used was TSMC 0.18 μ , the most advanced modern CMOS technology available to universities through the MOSIS program [6]. The dimensions of our CAM cell are 5.3 μ m x 5.6 μ m (29.7(μ m)² area). We used Cadence's Spectra to simulate the net list of the extracted circuits. We measured the access time and energy consumption of the encoder from the outputs of the simulation. Table 2 lists the results for original FVE coder in which the comparator and timestamps are necessary constructs. We derived our energy calculations based on these results.

$$E_{\text{Total}} = E_{\text{CAMs}} + E_{\text{timestamp}} + E_{\text{comparator}} + E_{\text{Xor}} + E_{\text{32 and/or}}$$

Table 2. Energy measurement for different FVE components

Component	Energy consumption	time
Comparator	1.27 pJ/access	0.2ns
XOR gate	0.095 pJ/Transition pair	0.1ns
64 entry CAM	28 pJ	0.2ns
timestamps	0.07pJ	0.5ns

Using the above equation, we calculated the value of E_{Total} for FVE-32, FVE-64 and FV-MSB to be 17.38pJ, 34.65pJ, and 36.65pJ respectively. The off-chip bus energy per bus wire is given by the formula:

$$E \propto C \times V^2 \times A,$$

where C is the off-chip bus capacitance, V is the supply voltage and A is the number of bus transitions. Assuming bus capacitance and bus voltage values of 60pF [5] and 3.3 volts respectively, the energy per bus transition is given by:

$$E_{\text{bus-transition}} = 60 \times 10^{-12} \times 3.32 = 600 \text{ pJ}$$

If we have an average of 10 transitions on the 32 bus wires during each cycle, one might notice that the per-cycle-energy dissipation in the FV-MSB codec is less than the energy spent in a single off-chip bus cycle by a factor of 200. Even with a supply voltage of 1.8V, the energy consumed by the FV-MSB codec is nearly 60 times lesser than the per-cycle off-chip bus energy. Hence, we can conclude that the energy consumed by the circuit components of our encoding scheme is quite insignificant when compared with the energy saved through the reductions in switching activities. The total energy dissipated in the bus is the sum of the total energy consumed during the bus transition and the energy consumed by the encoder and decoder (two times the encoder energy). We plotted the results in Fig. 7. Here, we compare three techniques: FV-32, FV-64 and FV-MSB. We can clearly see that FV-MSB saves more energy than the other two by a factor of 8%.

Table 2 also shows the latency for different components of the FV-MSB scheme. The critical path of the FV-MSB scheme is composed of CAM access, updating timestamps, selection AND/OR gates and the XOR gates. Adding delays for each one of them gives a total of 1.1ns delay ($0.2 + 0.5 + 0.1 + 0.3$). Notice that the comparator in the original FVE operates in parallel with the CAM access, therefore not taking the extra critical path delay.

Finally, the FV CAM and the MSB CAM are the major contributors to the area overhead. Since a CAM cell is of area $29.7 (\mu\text{m})^2$, multiplying it by the total number of bits in the FV-MSB yields $42 \times 10^3 \text{ mm}^2$ ($29.7 \times (32 \times 32 + 20 \times 20)$). Thus, our conclusion is that the FV-MSB scheme is an economic design in terms of energy, delay and area overhead.

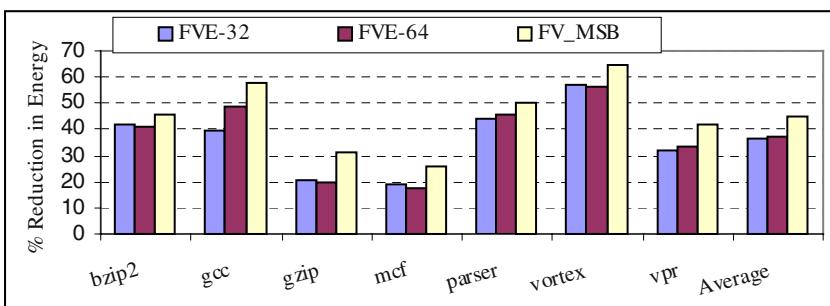


Fig. 7. Percentage of energy reductions for SPEC2000INT

Acknowledgement. This work was supported in part by NSF Award ITR 0083080.

References

- [1] K. Basu, Q. Choudhary, J. Pisharath and M. Kandemir, "Power Protocol: Reducing Power dissipation on off-chip Data Buses", The 35th IEEE/ACM International symposium on Microarchitecture, pages 345–355, 2002.
- [2] L. Benini, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Synthesis of Low-Overhead Interfaces for Power-Efficient Communication Over Wide Buses", ACM/IEEE Design Automation Conference, pages 128–133, 1999.
- [3] Cadence Corporation: <http://www.cadence.com>
- [4] D. Burger, and T. Austin, "The SimpleScalar Tool Set, Version 2.0", Technical Report 1342, Universoty of Wisconsin-Madison, Computer science Department, 1997
- [5] H. Messmer, "The Indispensable PC Hardware Book, Fourth edition, Addison-Wesley, 2002
- [6] The Mosis Service: <http://www.mosis.com/>.
- [7] M.R. Stan and W.P. Burleson, "Bus-invert coding for low-power I/O", IEEE Transactions on very Large Scale Integration (VLSI) systems, pages 49–58, Vol.3, 1995
- [8] J. Yang and R. Gupta, "FV Encoding for Low-Power Data I/O", ACM/IEEE International Symposium on Low Power Electronic Design, pages 84–87, 2001
- [9] Y. Zhang, J. Yang and R. Gupta, "Frequent Value Locality and Value-centric Data Cache Design", The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), pages 150–159, 2000.
- [10] A. Efthymiou and J.D. Garside, "An Adaptive Serial-Parallel CAM Architecture for Low-Power Cache Blocks,", Proceedings of the International Symposium On Low Power Electronics and Design, 2002.
- [11] D. Citron and L. Rudolph, "Creating a wider bus using caching techniques ", Proceedings of the first International symposium on High Performance Computer Architecture, pp 90–99, Jan 1995.
- [12] Matthew Farrens and Arvin Park. "Dynamic base register caching: A technique for reducing address bus width" In Proceedings of 18th Annual International Symposium on Computer Architecture, pages 128–137, May 1991. Toronto, Canada.
- [13] E. Musoll, T. Lang, and J. Cortadella. "Working-zone encoding for reducing the energy in microprocessor address buses". IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 6, 1998

A Parallel Iterative Improvement Stable Matching Algorithm

Enyue Lu and S.Q. Zheng

Department of Computer Science, University of Texas at Dallas
Richardson, TX 75083-0688, USA
`{enyue, sizheng}@utdallas.edu`

Abstract. In this paper, we propose a new approach, parallel iterative improvement (PII), to solving the stable matching problem. This approach treats the stable matching problem as an optimization problem with all possible matchings forming its solution space. Since a stable matching always exists for any stable matching problem instance, finding a stable matching is equivalent to finding a matching with the minimum number (which is always zero) of unstable pairs. A particular PII algorithm is presented to show the effectiveness of this approach by constructing a new matching from an existing matching and using techniques such as randomization and greedy selection to speedup the convergence process. Simulation results show that the PII algorithm has better average performance compared with the classical stable matching algorithms and converges in n iterations with high probability. The proposed algorithm is also useful for some real-time applications with stringent time constraint.

1 Introduction

The stable matching problem (or stable marriage problem) was first introduced by Gale and Shapley [6]. Given n men, n women, and $2n$ ranking lists in which each person ranks all members of the opposite sex in order of preference, a *matching* is a set of n pairs of man and woman with each person in exactly one pair. A matching is *unstable* if there are two persons who are not matched with each other, and each of whom strictly prefers the other to his/her partner in the matching; otherwise, the matching is *stable*. Gale and Shapley showed that every instance of the stable matching problem admits at least one stable matching and such a matching can be computed in $O(n^2)$ iterations. The paper of Gale and Shapley sparked much interest in many aspects and variants of the classical stable matching problem. For a good survey on this subject, refer to [8].

Recently, the solutions to the stable matching problem have been applied to switch scheduling for packet/cell switches. Many scheduling algorithms based on stable matchings have been proposed for both input queued (IQ) switches and combined input and output queued (CIOQ) switches (e.g. [3,15,17,18,20]). It has

been shown that scheduling algorithms based on stable matchings can provide QoS guarantees.

For real-time applications, the algorithm proposed by Gale and Shapley, simply GS algorithm, is not fast enough. Attempting to find stable matching algorithms with low complexity was made by many researchers (e.g. [1,7,10,11, 16,21,22]). Up to date, the best known algorithm for stable matching problem takes $O(\sqrt{n} \cdot \log^3 n)$ time [5]. This parallel algorithm runs on a CRCW PRAM (concurrent-read concurrent-write parallel random access machine) of n^4 processors, which makes it infeasible for applications in packet/cell-switched networks. The parallelizability of the stable matching problem is far from being fully understood. It was suggested that parallel stable matching algorithms cannot be expected to provide high speedup on the average [13,19]. Thus, designing efficient parallel algorithms that perform well for most cases is a challenging endeavor.

In this paper, we propose a new approach, parallel iterative improvement (PII), to solving the stable matching problem. The PII algorithm consists of two alternating phases, INITIATION_PHASE and ITERATION_PHASE. An INITIATION_PHASE is a procedure that randomly generates a matching. An ITERATION_PHASE consists of multiple improvement iterations. We try to speedup the convergence process by exploring parallelism in identifying a subset of unmatched pairs to replace matched pairs for an existing matching so that the number of unstable pairs in newly obtained matching can be reduced. We show that an INITIATION_PHASE and an iteration of an ITERATION_PHASE take $O(\log n)$ time on both completely connected multiprocessor system and array with multiple broadcasting buses, and $O(\log^2 n)$ time on both hypercube and MOT, all assumed having n^2 processor elements (PEs). Simulation results show that the PII algorithm has better average performance compared with the classical stable matching algorithms and converges in n iterations with high probability.

2 Preliminaries

Let $M = \{m_1, m_2, \dots, m_n\}$ and $W = \{w_1, w_2, \dots, w_n\}$ be the sets of n men and n women respectively. Let $mL_i = \{wr_{i,1}, wr_{i,2}, \dots, wr_{i,n}\}$ and $wL_i = \{mr_{i,1}, mr_{i,2}, \dots, mr_{i,n}\}$ be the *ranking lists* for man m_i and woman w_i respectively, where $wr_{i,j}$ (resp. $mr_{i,j}$) is the rank of woman w_j (resp. man m_j) by man m_i (resp. woman w_i). Let A be a *ranking matrix* of size of $n \times n$, where each entry of A is a pair $a_{i,j} = (wr_{i,j}, mr_{j,i})$. We call $wr_{i,j}$ (resp. $mr_{j,i}$) the *left value* (resp. *right value*) of $a_{i,j}$, and denote it by $a_{i,j}^L$ (resp. $a_{i,j}^R$). For convenience, we use $(a_{i,j}^x, a_{i,j}^y)$ to denote the indices (i, j) of pair $a_{i,j}$. Clearly, the ordered list of left values of all pairs in row i of A is man ranking list mL_i and the ordered list of right values of all pairs in column j is woman ranking list wL_j . Example 1 shows the ranking matrix obtained from the given ranking lists.

Example 1. An instance of stable matching problem:

Man ranking lists:	Woman ranking lists:	Ranking matrix:
$mL_1 : \{4, 2, 3, 1\}$;	$wL_1 : \{1, 4, 2, 3\}$;	4, 1 2, 1 3, 4 1, 3
$mL_2 : \{3, 1, 2, 4\}$;	$wL_2 : \{1, 2, 3, 4\}$;	3, 4 1, 2 2, 2 4, 1
$mL_3 : \{2, 4, 1, 3\}$;	$wL_3 : \{4, 2, 3, 1\}$;	2, 2 4, 3 1, 3 3, 4
$mL_4 : \{1, 4, 3, 2\}$.	$wL_4 : \{3, 1, 4, 2\}$.	1, 3 4, 4 3, 1 2, 2

A pair $a_{i,j}$ in A corresponds to a man-woman pair (m_i, w_j) . A matching, denoted as \mathcal{M} , corresponds to n pairs of A with no two pairs in the same row/column. If a pair of A is in \mathcal{M} , it is called a *matching pair* of \mathcal{M} and otherwise a *non-matching pair*. For any matching \mathcal{M} of ranking matrix A , we define the *marked ranking matrix*, $A_{\mathcal{M}}$, as the ranking matrix with all matching pairs marked. Thus for any matching \mathcal{M} , each row i (resp. column j) of $A_{\mathcal{M}}$ has exactly one matching pair, which is denoted as $\mathcal{M}(R_i)$ (resp. $\mathcal{M}(C_j)$). A pair $a_{i,j}$ is an *unstable pair* if $a_{i,j}^L < \mathcal{M}(R_i)^L$ and $a_{i,j}^R < \mathcal{M}(C_j)^R$. By the definition of stable matching, we have:

Property 1. A matching \mathcal{M} is stable if and only if there is no unstable pair in $A_{\mathcal{M}}$.

With respect to $A_{\mathcal{M}}$, we define a set NM_1 of *type-1 new matching pairs* (simply *nm₁-pairs*) as follows. If there is no unstable pair in $A_{\mathcal{M}}$, $NM_1 = \emptyset$. Otherwise, for every row R_i with at least one unstable pair, select the one with the minimum left value among all unstable pairs in row R_i as an *nm₁-generating pair*; for every column C_j with at least one *nm₁*-generating pair, select the one with the minimum right value as an *nm₁-pair*.

Based on NM_1 , we define a set NM_2 of *type-2 new matching pairs* (simply *nm₂-pairs*) by a procedure that first identifies *nm₂*-generating pairs and then identifies *nm₂*-pairs using an *nm₂*-generating graph. For any *nm₁*-pair $a_{i,j}$ in $A_{\mathcal{M}}$, pair $a_{l,k}$ with $l = \mathcal{M}(C_j)^x$ and $k = \mathcal{M}(R_i)^y$ is called the *nm₂-generating pair* corresponding to $a_{i,j}$. We say that *nm₁*-pair $a_{i,j}$ and its corresponding *nm₂*-generating pair $a_{l,k}$ are associated with matching pairs $a_{i,k}$ and $a_{l,j}$. We define an *nm₂-generating graph* $G_{\mathcal{M}}$ as follows: $V(G_{\mathcal{M}}) = \{\text{all } nm_2\text{-generating pairs}\}$, and $E(G_{\mathcal{M}}) = \{e = (u, v) \mid \text{two } nm_2\text{-generating pairs } u \text{ and } v \text{ are associated with a common matching pair}\}$. Since each *nm₂*-generating pair is associated with 2 matching pairs, we have:

Property 2. Given any $A_{\mathcal{M}}$, the degree of *nm₂*-generating graph $G_{\mathcal{M}}$ is at most 2.

By Property 2, each connected component in $G_{\mathcal{M}}$ is a cycle or chain, named *nm₂-generating cycle* or *nm₂-generating chain* (an isolated node is a chain of length 0). If a node in $G_{\mathcal{M}}$ has degree 2, it is called an *internal node* and otherwise an *end node*. Clearly, if an *nm₂*-generating pair $a_{i,j}$ is an internal node in $G_{\mathcal{M}}$, there are two *nm₁*-pairs, one in row i and the other in column j ; if an *nm₂*-generating pair $a_{i,j}$ is an end node in $G_{\mathcal{M}}$, there is at most one *nm₁*-pair in

row i or column j . We call an end node $a_{i,j}$ a *row end* (resp. *column end*) of an nm_2 -generating chain if there is no nm_1 -pair in row i (resp. column j) of $A_{\mathcal{M}}$. An isolated node is both row end and column end.

By the nm_2 -generating graph, we can generate the set NM_2 of nm_2 -pairs as follows. For each nm_2 -generating chain with row end a_{i_1,j_1} and column end a_{i_2,j_2} , we generate an nm_2 -pair a_{i_1,j_2} . No nm_2 -pair is generated from any nm_2 -generating cycle. Hence, there is a one-to-one correspondence between an nm_2 -generating chain and an nm_2 -pair. Let $NM = NM_1 \cup NM_2$. We call NM the set of *new matching pairs* (simply *nm-pairs*). Based on the way that NM is generated, we know that NM_1 and NM_2 are disjoint, and each row/column of $A_{\mathcal{M}}$ contains at most one nm -pair.

A matching pair $a_{i,j}$ in $A_{\mathcal{M}}$ is called a *replaced matching pair* (simply *rm-pair*), if it is in the same row/column of an nm -pair. We denote the set of rm -pairs by RM . Based on the way that RM is constructed, we have:

Lemma 1. *If there is at least one unstable pair in $A_{\mathcal{M}}$, then $\mathcal{M}' = (\mathcal{M} - RM) \cup NM$ is a matching different from \mathcal{M} .*

3 Parallel Iterative Improvement Matching Algorithm

In this section, we present our main result, a *parallel iterative improvement algorithm* (PII algorithm) for a completely connected multiprocessor system, which consists of a set of PEs connected in such a way that there is a direct connection between every pair of PEs. We assume that each PE can communicate with at most one adjacent PE during every communication step. The PII algorithm uses n^2 PEs. To facilitate our discussion, these n^2 PEs are placed as an $n \times n$ array. As input, $PE_{i,j}$, ($1 \leq i, j \leq n$), contains $a_{i,j}$ of ranking matrix A . When the algorithm terminates, a stable matching is found by $PE_{i,j}$ indicating whether pair (m_i, w_j) is in the matching. The key idea of the PII algorithm is to construct a new matching \mathcal{M}' from an existing matching \mathcal{M} in hope that \mathcal{M}' is “closer” to a stable matching than \mathcal{M} .

3.1 Constructing an Initial Matching

To randomly generate an initial matching can be reduced to generate a random permutation, which can be done by a sequential algorithm proposed in [4]. In the following, we show how to implement it in parallel on a completely connected multiprocessor system with N^2 PEs.

Let each PE maintain a pointer. Initially, every $PE_{i,j}$ sets its pointer to point to $PE_{i+1,j}$, ($1 \leq i \leq n-1$), and as a result, there are n disjoint lists. Then, each $PE_{i,i}$ will randomly choose a j ($i \leq j \leq n$) to swap their pointers, i.e. $PE_{i,i}$ points to $PE_{i+1,j}$ and $PE_{i,j}$ points to $PE_{i+1,i}$. Consequently, n new disjointed lists originated from $PE_{1,j}$ are formed. After performing $\log n$ times of pointer jumping [12], each $PE_{1,j}$ finds another end $PE_{n,p(1,j)}$ of its list, where $p(1,j)$ is the column position of the PE pointed by $PE_{1,j}$. Hence, a matching $\{(j, p(1,j)) | 1 \leq j \leq n\}$ is formed. Clearly, this parallel implementation takes $O(\log n)$ time since each list has length of n .

3.2 Construct a New Matching from an Existing Matching

A basic operation of PII algorithm is to construct a new matching $\mathcal{M}' = (\mathcal{M} - RM) \cup NM$ from an existing matching \mathcal{M} if \mathcal{M} is unstable. In the following, we describe 6 steps to carry out this operation.

Step 1: Recognize Unstable Pairs. Every PE with a matching pair in \mathcal{M} broadcasts its column/row position and the left/right value of its matching pair to all PEs in the same row/column. If $PE_{i,j}$'s both values are smaller, set its Boolean variable $u_{i,j} := true$, which indicates that pair $a_{i,j}$ is unstable; otherwise set $u_{i,j} := false$. The broadcasting in rows/columns takes $O(\log n)$ time.

Step 2: Stability Checking. Find if there exists a $PE_{i,j}$ with $u_{i,j} := true$ by binary searching in rows/columns. Since each row/column has n PEs, the searching takes $O(\log n)$ time. If $f_{i,j} := false$ for any $PE_{i,j}$, then the current matching \mathcal{M} is stable, and the algorithm terminates. Otherwise, go to the next step.

Step 3: Find NM_1 . For each row with at least one unstable pair, find the unstable pair with the minimum left value, and mark this pair as an nm_1 -generating pair. For each column with at least one nm_1 -generating pair, find the nm_1 -generating pair with the minimum right value, and mark this pair as an nm_1 -pair. The find-minimum operation in rows/columns takes $O(\log n)$ time.

Step 4: Find nm_2 -Generating Pairs. For each $PE_{i,j}$ containing an nm_1 -pair, mark the pair in $PE_{l,k}$ as a nm_2 -generating pair, where $l = \mathcal{M}(C_j)^x$ and $k = \mathcal{M}(R_i)^y$. Clearly, this step only takes $O(1)$ time.

Step 5: Find NM_2 . This step consists of two major objectives: (1) each nm_2 -generating node that is both row end and column end in $G_{\mathcal{M}}$ recognizes itself as an isolated node, and (2) the row end of each nm_2 -generating chain in $G_{\mathcal{M}}$ finds its column end. Let each $PE_{i,j}$ containing an nm_2 -generating pair maintain two pointers, r -pointer and c -pointer. The r -pointer (resp. c -pointer) of $PE_{i,j}$ points to the PE containing an nm_2 -generating pair in column $\mathcal{M}(R_i)^y$ (resp. row $\mathcal{M}(C_j)^x$) if there is an nm_1 -pair in row i (resp. column j), and otherwise to itself. If both r -pointer and c -pointer of $PE_{i,j}$ point to itself, then it corresponds to an isolated node in $G_{\mathcal{M}}$; if the r -pointer (resp. c -pointer) of $PE_{i,j}$ points to itself but another pointer points to some other PE, then $PE_{i,j}$ contains an nm_2 -generating pair that is the row (resp. column) end of an nm_2 -generating chain; if both r -pointer and c -pointer of $PE_{i,j}$ point to other PEs, its nm_2 -generating pair corresponds to an internal node of $G_{\mathcal{M}}$. Fig. 1 shows an example. By a completely connected multiprocessor with N^2 PEs, objective (1) can be easily achieved in $O(1)$ time and objective (2) can be achieved by performing $\lceil \log n \rceil$ times of pointer jumping [12] since the length of each nm_2 -generating chain is at most n . Once objectives (1) and (2) are accomplished, the nm_2 -pairs can be easily computed in $O(1)$ time.

Step 6: Construct New Matching. Each $PE_{i,j}$ containing an nm -pair marks the matching pair in row i as a replaced pair, and marks itself as a matching pair. This step takes $O(1)$ time.

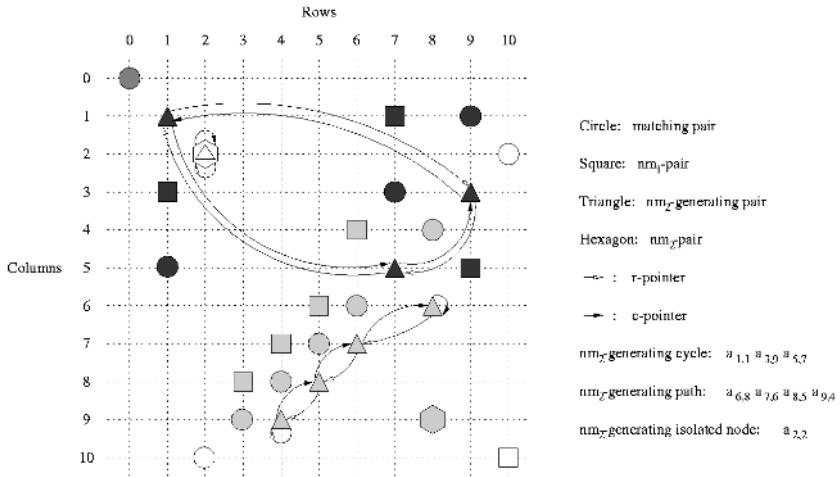


Fig. 1. Finding a new matching \mathcal{M}' from an existing matching \mathcal{M} , where $\mathcal{M} = \{a_{0,0}, a_{1,9}, a_{2,10}, a_{3,7}, a_{4,8}, a_{5,1}, a_{6,6}, a_{7,5}, a_{8,4}, a_{9,3}, a_{10,2}\}$, $NM_1 = \{a_{1,7}, a_{3,1}, a_{4,6}, a_{5,9}, a_{6,5}, a_{7,4}, a_{8,3}, a_{10,10}\}$, $NM_2 = \{a_{2,2}, a_{9,8}\}$, $RM = \{a_{1,9}, a_{2,10}, a_{3,7}, a_{4,8}, a_{5,1}, a_{6,6}, a_{7,5}, a_{8,4}, a_{9,3}, a_{10,2}\}$ and $\mathcal{M}' = (\mathcal{M} - RM) \cup NM = \{a_{0,0}\} \cup NM_1 \cup NM_2$.

3.3 PII Algorithm

Now we are ready to present our PII algorithm. Conceptually, the PII algorithm has two alternating phases: INITIATION_PHASE and ITERATION_PHASE. The INITIATION_PHASE finds an initial matching arbitrarily. The ITERATION_PHASE contains at most $c \cdot n$ iterations, where c is a constant for controlling the number of iterations in the ITERATION_PHASE. Each iteration of ITERATION_PHASE checks whether an existing matching \mathcal{M} is stable. If \mathcal{M} is stable, the algorithm terminates; otherwise, a new matching \mathcal{M}' is constructed. Then, \mathcal{M}' is used as \mathcal{M} for the next iteration. After $c \cdot n$ iterations in each ITERATION_PHASE, the PII algorithm goes back to INITIATION_PHASE to generate a new initial matching randomly and a new ITERATION_PHASE is effected based on this new generated matching. As we analyzed, an INITIATION_PHASE and an iteration of an ITERATION_PHASE takes $O(\log n)$ time on a completely connected multiprocessor system with n^2 PEs.

In an iteration of an ITERATION_PHASE, a new matching $\mathcal{M}' = (\mathcal{M} - RM) \cup NM_1 \cup NM_2$ is constructed from an existing matching \mathcal{M} . It is easy to verify that the pairs in NM_1 were unstable for \mathcal{M} , but become stable for \mathcal{M}' ; the pairs in NM_2 are stable for \mathcal{M}' , regardless whether they were stable for \mathcal{M} ; and the pairs in \mathcal{M} , which were stable for \mathcal{M} , remain to be stable for \mathcal{M}' . Intuitively, the number of unstable pairs for \mathcal{M}' is smaller than the number of unstable pairs for \mathcal{M} . For most cases, it is true. This is the heuristic behind the PII algorithm.

However, new unstable pairs may be generated for \mathcal{M}' . Let the initial matching be \mathcal{M}_0 and the matching generated in the i -th iteration be \mathcal{M}_i . Since the

set of nm_1 -pairs, nm_2 -pairs and rm -pairs with respect to \mathcal{M}_{i-1} is unique, the matching \mathcal{M}_i is constructed uniquely from \mathcal{M}_{i-1} . Hence, if $\mathcal{M}_i \in \{\mathcal{M}_j | j \in \{0, 1, \dots, i-1\}\}$, i.e. the newly generated matching is the same as a previously generated matching, no stable matching can be found. It is possible to include a procedure for detecting this cyclic situation. Such a procedure, however, is too time-consuming. This is why we decided to start a new round after $c \cdot n$ iterations of an INITIATION-PHASE. The random permutation generating algorithm we used generates random matchings with uniform distribution according to [2]. Therefore, by the existence of a stable matching, the PII algorithm can always find one for any instance of stable matching problem.

Our simulation results (see Section 5) indicate that the PII algorithm has better performance compared with GS algorithm. However, we are unable to theoretically exclude the possibility that the cases in which the total number of iterations is very large. In order to enforce a bound for the number of iterations, we propose to run the PII algorithm and parallel GS algorithm simultaneously in a time-sharing fashion. We denote this modified PII algorithm as PII-GS, which terminates once one of the algorithms generates a stable matching. Clearly, the PII-GS algorithm converges to a stable matching with $O(n^2)$ iterations in the worst case.

4 Implementations of PII Algorithm on Parallel Computing Machine Models

In this section, we consider implementing the PII algorithm on three well-known parallel computing systems – hypercube, mesh of trees (MOT) and array with multiple broadcasting buses. Without loss of generality, assume $n = 2^k$. If $2^k < n < 2^{k+1}$, the PII algorithm can be implemented on a 2^{2k} -processor system with a constant slow-down factor. The n^2 PEs in each system are placed as $n \times n$ array, and n PEs in each row/column form a row/column connection (see Fig. 2). We assume that our parallel computing systems operate in a synchronous fashion. Basic $O(1)$ -time parallel operations of a hypercube and a MOT can be found in [14]. For an array with multiple broadcasting buses, we assume that each bus has a controller. A processor can request to communicate with the controller or any other processor on the bus. At any time, more than one processor on a bus may send requests to the bus controller, and the controller selects one request (if any) to grant the bus access arbitrarily. The controller of a bus can broadcast a message to all the processors on the bus. We assume that each processor-to-processor, processor-to-controller and broadcasting operation takes $O(1)$ time.

It is simple to notice that multiple-broadcasting, finding minimum, and pointer jumping are the most time consuming operations in the PII algorithm. The pointer jumping can be carried out by sorting. Let C be a parallel computing machine with n^2 processors, and let $T_B(n)$, $T_M(n)$ and $T_S(n)$ be the time required for multiple-broadcasting, finding minimum and sorting on C , respectively. Then, an INITIATION-PHASE of PII algorithm can be implemented

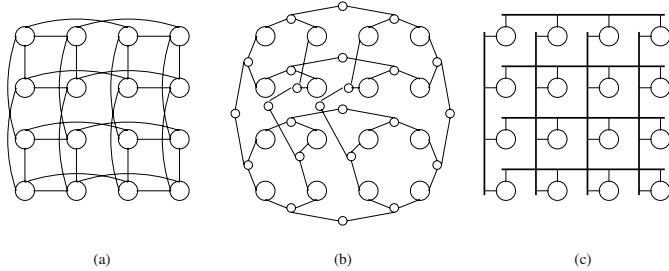


Fig. 2. (a) A 16-processor hypercube (b) A 4×4 mesh of trees (c) A 4×4 array with multiple broadcasting buses

on C in $O(T_S(n) \cdot \log n)$ time, and each iteration of an INITIATION_PHASE of PII algorithm can be implemented on C in $O(\max\{T_B(n), T_M(n), T_S(n) \cdot \log n\})$ time.

For a hypercube and a MOT, the operations of broadcasting and finding-minimum in PII are performed in parallel row-wise or column-wise. Thus, $T_B(n) = T_M(n) = O(\log n)$ for a hypercube and a MOT. For an array with multiple broadcasting buses, $T_B(n) = O(1)$. Finding-minimum operation can be carried out on a bus in $O(\log n)$ time using a binary searching method. For an n^2 -processor hypercube $T_S(n) = O(\log^2 n)$ while $T_S(n) = \Omega(n)$ for a MOT and an array with multiple broadcasting buses since either of their bisection widths is n . If we use sorting to implement pointer jumping operations, both an INITIATION_PHASE and an iteration in an ITERATION_PHASE of PII algorithm require $O(\log^3 n)$ time on a hypercube and $\Omega(n \log n)$ time on a MOT and an array with multiple broadcasting buses. In the following, however, we show that sorting can be avoided on these parallel computing models using special features of PII algorithm.

First, we show how to implement an INITIATION_PHASE without pointer jumping. This can be done by adopting a parallel implementation in [9] of the algorithm of [4]. Let π_i , $(1 \leq i \leq n - 1)$, be the permutation interchanging i and r_i that is chosen randomly from the set $\{i, \dots, n\}$ while leaving other elements of $\{1, 2, \dots, n\}$ fixed. Let π_n be an identity permutation. Initially, we use row i to represent π_i . The computation $\pi = \pi_1 \circ \pi_2 \circ \dots \circ \pi_{n-1} \circ \pi_n$ is organized in a complete binary tree of height $\log n$. For example, for $n = 8$, $\pi = ((\pi_1 \circ \pi_2) \circ (\pi_3 \circ \pi_4)) \circ ((\pi_5 \circ \pi_6) \circ (\pi_7 \circ \pi_8))$. Hence, all that remains is to consider the composition of two permutations. Given a permutation π' , let $D(\pi') = \{i | 1 \leq i \leq n, \text{ and } \pi'(i) \neq i\}$. The algorithm of [9] associates $|D(\pi')|$ processors to π' . In our implementation, we mimic the operations of one processor in [9] using a set of processors and their connections. More specifically, we associate each row/column i to π_i at the beginning. Let $\pi_{(i,j)} = \pi_i \circ \pi_{i+1} \circ \dots \circ \pi_j = \pi_{(i,(j-i+1)/2)} \circ \pi_{((j-i+1)/2+1,j)}$, where $j-i+1$ is an integer of power of 2. Note that $|D(\pi_{(i,j)})| \leq j - i + 1$. Thus, we can use row i through row j and column i through column j to perform the operations assigned to the processors for

computing $\pi_{(i,j)}$ in the algorithm of [9]. The communication paths for computing the composition of permutations at the same level of the binary computation tree are disjoint, because they use disjoint sets of row and column connections. Since the height of the binary computation tree is $O(\log n)$, an INITIATION_PHASE of PII algorithm takes $O(\log^2 n)$ time on a hypercube and a MOT. If an array with multiple broadcasting buses is used, an INITIATION_PHASE of PII algorithm takes $O(\log n)$ time.

We now show how to implement an iteration of ITERATION_PHASE without using pointer jumping. Since each row/column contains at most one nm_2 -generating pair, each pointer jumping of Step 5 can be decomposed into disjoint parallel 1-to-1 row communications followed by disjoint parallel 1-to-1 column communications without conflicts. Thus, every pointer jumping can be implemented in $O(\log^2 n)$ time on a hypercube and a MOT, and in $O(\log n)$ time on an array with multiple broadcasting buses. We also note that simulating an $n \times n$ MOT by an $n/2 \times n/2$ MOT (which has $3n^2/4 - n < n^2$ processors) results in a constant slowdown factor. To summarize, we show the improvement of the time complexity of PII algorithm on three parallel computing systems in Table 1.

Table 1. Time complexity for implementations of PII algorithm on three parallel computing machine models

Machine models	INITIATION_PHASE		An iteration in ITERATION_PHASE	
	with sorting	without sorting	with sorting	without sorting
Hypercube	$O(\log^3 n)$	$O(\log^2 n)$	$O(\log^3 n)$	$O(\log^2 n)$
MOT	$O(n \log n)$	$O(\log^2 n)$	$O(n \log n)$	$O(\log^2 n)$
Array with Buses	$O(n \log n)$	$O(\log n)$	$O(n \log n)$	$O(\log n)$

5 Simulation Results

We have simulated PII, PII-GS, and parallel GS algorithms for different sizes $n \in \{10, 20, \dots, 100\}$ of stable matching, with 10000 runs each. The ranking lists and initial matchings are generated by random permutation algorithm [4]. Each ITERATION_PHASE contains n iterations. The performance comparisons are based on the average number of parallel iterations for each algorithm to generate a stable matching and the frequency for each algorithm to converge in n iterations. From the simulation, we notice that PII and PII-GS algorithms significantly outperform GS algorithm. Fig. 3 shows that PII and PII-GS algorithms converge in n iterations with very high probabilities, while the probability for GS algorithm to converge with the same number of iterations decreases quickly as the sizes of problem increase.

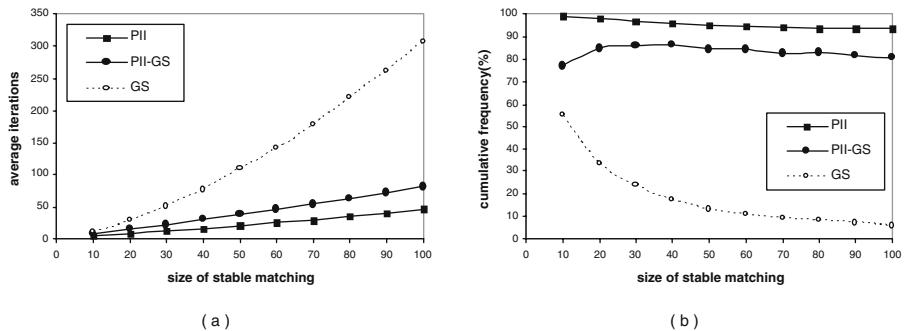


Fig. 3. Performance Comparisons (a) The average number of iterations for algorithms to find a stable matching (b) The frequencies for algorithms to find a stable matching within n iterations

6 Concluding Remarks

In this paper, we proposed a new approach, parallel iterative improvement, to solving the stable matching problem. The PII algorithm requires n^2 PEs, among which n PEs are required to perform arithmetic operations (for random number generation) and the other PEs can be simple comparators. The classical GS algorithm and most existing stable matching algorithms can only find the man-optimal or woman-optimal stable matching. By [8], the man (resp. woman)-optimal stable matching is women (resp. men)-pessimal, i.e. every man/woman gets the best partner while every woman/man gets the worst partner over all stable matchings. However, due to randomness, the PII algorithm constructs a stable matching that is contained in the set of stable matchings. Therefore, this algorithm will generate the stable matching with more fairness.

In some applications, such as real-time packet/cell scheduling for a switch, stable matching is desirable, but may not be found quickly within tight time constraint. Thus, finding a “near-stable” matching by relaxing solution quality to satisfy time constraint is more important for such applications. Most of existing parallel stable matching algorithms cannot guarantee a matching with a small number of unstable pairs within a given time interval. Interrupting the computation of such an algorithm does not result in any matching. However, the PII algorithm can be stopped at any time. By maintaining the matching with the minimum number of unstable pairs found so far, a matching that is close to a stable matching can be computed quickly.

References

1. Abeledo, H., Rothblum, U.G.: “Paths to marriage stability”, *Discrete Applied Mathematics*, 63 (1995) 1–12

2. Anderson, R: "Parallel algorithms for generating random permutations on a shared memory machine", *Proc. of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, (1990) 95–102
3. Chuang, S.T., Goel, A. , McKeown, N., Prabhakar, B.: "Matching output queuing with a combined input/output-queued switch", *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 6 (1999) 1030–1039
4. Durstenfeld, R.: "Random permutation (Algorithm 235)", *Communication of ACM*, Vol. 7, No. 7 (1964) 420
5. Feder, T., Megiddo, N., Plotkin, S.: "A sublinear parallel algorithm for stable matching", *Theoretical Computer Science*, Vol. 233 (2000) 297–308
6. Gale, D., Shapley, L.S.: "College admissions and the stability of marriage", *American Mathematical Monthly*, Vol. 69 (1962) 9–15
7. Gusfield, D.: "Three fast algorithms for four problems in stable marriage", *SIAM Journal on Computing*, Vol. 16, No. 1 (1987) 111–128
8. Gusfield, D., Irving, R.W.: *The Stable Marriage Problem Structure and Algorithms*, MIT Press (1989)
9. Hagerup, T.: "Fast parallel generation of random permutations", *Proc. of the 18th Annual International Colloquium on Automata, Languages and Programming*, (1991) 405–416
10. Hattori,T., Yamasaki,T., Kumano, M.: "New fast iteration algorithm for the solution of generalized stable marriage problem", *Proc. of IEEE International Conference on Systems, Man, and Cybernetics*, Vol. 6 (1999) 1051 -1056
11. Hull, M.E.C.: "A parallel view of stable marriages", *Information Processing Letters*, Vol. 18, No. 1 (1984) 63–66
12. Jaja, J.: *An Introduction to Parallel Algorithms*, Addison-Wesley (1992)
13. Kapur, D., Krishnamoorthy, M.S.: "Worst-case choice for the stable marriage problem", *Information Processing Letters*, Vol. 21 (1985) 27–30
14. Leighton, F. T.: *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes*, Morgan Kaufmann Publishers (1992)
15. McKeown, N.: "Scheduling algorithms for input-buffered cell switches", Ph.D. Thesis, University of California at Berkeley (1995)
16. McVitie, D.G., Wilson, L.B.: "The stable marriage problem", *Communication of the ACM*, Vol. 14, No. 7 (1971) 486–490
17. Nong, G., Hamdi, M.: "On the provision of quality-of-service guarantees for input queued switches", *IEEE Communications Magazine*, Vol. 38, No. 12 (2000) 62–69
18. Prabhakar, B., McKeown, N.: "On the speedup required for combined input- and output-queued switching", *Automatica*, Vol. 35, No. 12 (1999) 1909–1920
19. Quinn, M.J.: "A note on two parallel algorithms to solve the stable marriage problem", *BIT*, Vol. 25 (1985) 473–476
20. Stoica, I., Zhang, H.: "Exact emulation of an output queuing switch by a combined input output queuing switch", *Proc. of the 6th IEEE/IFIP IWQoS'98*, Napa Valley, CA, (1998) 218–224
21. Subramanian, A.: "A new approach to stable matching problems", *SIAM Journal on Computing*, Vol. 23, No. 4 (1994) 671–700
22. Tseng, S.S., Lee, R.C.T.: "A parallel algorithm to solve the stable marriage algorithm", *BIT*, Vol. 24 (1984) 308–316

Self-Stabilizing Distributed Algorithm for Strong Matching in a System Graph

Wayne Goddard, Stephen T. Hedetniemi, David P. Jacobs, and
Pradip K. Srimani

Department of Computer Science, Clemson University, Clemson, SC 29634–0974

Abstract. We present a new self-stabilizing algorithm for finding a maximal strong matching in an arbitrary distributed network. The algorithm is capable of working with multiple types of demons (schedulers) as is the most recent algorithm in [1,2]. The concepts behind the algorithm, using IDs in the network, promise to have applications for other graph theoretic primitives.

1 Introduction

The traditional approach to building fault-tolerant, distributed systems uses *fault masking*. It is *pessimistic* in the sense that it assumes a worst case scenario and protects the system against such an eventuality. Validity is guaranteed in the presence of faulty processes, which necessitates restrictions on the number of faults and on the fault model. But fault masking is not free; it requires additional hardware or software, and it considerably increases the cost of the system. This additional cost may not be an economic option, especially when most faults are transient in nature and a temporary unavailability of a system service is acceptable for a short period of time. The paradigm of *Self-stabilization* can cope with the transient faults in a very elegant and cost effective way to design fault tolerant protocols for networked computer systems. Self-stabilization is an *optimistic* way of looking at system fault tolerance, because it provides a built-in safeguard against transient failures that might corrupt the data in a distributed system. Although the concept was introduced by Dijkstra in 1974 [3], and Lamport [4] showed its relevance to fault tolerance in distributed systems in 1983, serious work only began in the late nineteen-eighties. A good survey of self-stabilizing algorithms can be found in [5]. Herman's bibliography [6] also provides a fairly comprehensive listing of most papers in this field. Because of the size and nature of many ad hoc and geographically distributed systems, communication links are unreliable. The system must therefore be able to adjust when faults occur. But 100% fault tolerance is not warranted. The promise of self-stabilization, as opposed to fault masking, is to recover from failure in a reasonable amount of time and without intervention by any external agency. Since the faults are transient (eventual repair is assumed), it is no longer necessary to assume a bound on the number of failures.

A fundamental idea of self-stabilizing algorithms is that the distributed system may be started from an arbitrary global state. After a finite amount of time

the system reaches a correct global state, called a *legitimate* or *stable* state. An algorithm is self-stabilizing if (i) for any initial illegitimate state it reaches a legitimate state after a finite number of node moves, and (ii) for any legitimate state and for any move allowed by that state, the next state is a legitimate state. A self-stabilizing system does not guarantee that the system is able to operate properly when a node continuously injects faults in the system (Byzantine fault) or when communication errors occur so frequently that the new legitimate state cannot be reached. While the system services are unavailable when the self-stabilizing system is in an illegitimate state, the repair of a self-stabilizing system is simple; once the offending equipment is removed or repaired the system provides its service after a reasonable time.

A distributed system can be modeled with an undirected graph $G = (V, E)$, where V is a set of n nodes and E is a set of m edges. If $i \in V$, then $N(i)$, its *open neighborhood*, denotes the set of nodes to which i is adjacent, and $N[i] = N(i) \cup \{i\}$ denotes its *closed neighborhood*. Every node $j \in N(i)$ is called a *neighbor* of node i . Throughout this paper we assume G is connected and $n > 1$. In a self-stabilizing algorithm, a node changes its local state by making a *move* (a change of local state). The algorithm is a set of rules of the form “**if** $p(i)$ **then** M ”, where $p(i)$ is a predicate and M is a move. A node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume a serial model in which no two nodes move simultaneously. A central daemon selects, among all privileged nodes, the next node to move. If two or more nodes are privileged, one cannot predict which node will move next. Multiple protocols exist [7,8,9] that provide such a scheduler. Our proposed algorithm can easily be combined with any of these protocols to work under different schedulers as well.

There has been a recent spurt of activities in designing self-stabilizing distributed algorithms for different graph theoretic problems in the context of ad hoc networks [10,11]. In this paper we present the first self-stabilizing algorithm for finding the maximal the strong (induced) matching in an arbitrary graph. Given an undirected graph $G = (V, E)$, a **matching** is defined to be a subset M of edges $(i, j) \in E$, where $i, j \in V$ such that for all nodes $i \in V$ at most one edge of M is incident on i . If M is a matching, and the edge $(i, j) \in M$, we say that the nodes i and j are saturated by the edge (i, j) . A matching M is acalled **strong** if every saturated node is adjacent to only one sturated node [12]. These strong or induced matchings in graphs have, in addition to theoretical importance, many practical significance, especially when the graph denotes the communication network [13].

2 Self-Stabilizing Maximal Strong Matching Algorithm

Our algorithm requires that every node has a unique ID. We will sometimes use i interchangeably to denote a node and the node's ID. We assume there is a total ordering on the IDs.

In the algorithm, each node i maintains only a pointer $P(i)$: the value of $P(i)$ is either another node, or one of two special values: **Open** or **Unav**(for “unavailable”).

Definition 1. A node i is called **available** iff $P(i) \neq \text{Unav}$.

We use a total ordering \preceq on the set of undirected edges (or more precisely on the set of pairs of nodes) called the *lexicographic order* (imposed by the total ordering of the unique node IDs). If edges $e = \{u, v\}$ and $f = \{x, y\}$ do not have a node in common, then the one incident with the smallest node is the smaller; if they share a node, then they are ranked by the other end.

Definition 2. If $e = \{u, v\}$ and $f = \{x, y\}$, then $e \prec f$ if and only if either $\min(e) < \min(f)$ or $\min(e) = \min(f)$ and $\max(e) \leq \max(f)$. If $e \preceq f$ and $e \neq f$ then we will write $e \prec f$.

Remark 1. For any node j , when the value of the variable $P(j)$ is an adjacent node (i.e., $P(j) \notin \{\text{Open}, \text{Unav}\}$), the pair $\{j, P(j)\}$ denotes an edge incident on node j .

Definition 3. For any node i , we define a special edge $A(i)$ as

$$A(i) = \min\{ \{j, P(j)\} : j \in N(i) \wedge P(j) \notin \{\text{Open}, \text{Unav}\} \},$$

Remark 2.

- The comparison of edges in Definition 3 is done using the ordering \prec given in Definition 2.
- $A(i)$ is the smallest edge incident to a neighbor of i , as shown by the pointers of its neighbors.
- For any node i in an arbitrary global state, $A(i)$ may not exist. $A(i)$ is unique (when it exists) and is computable by node i using local information.

Definition 4. The minimum neighbor of any node i , that is available, is defined as

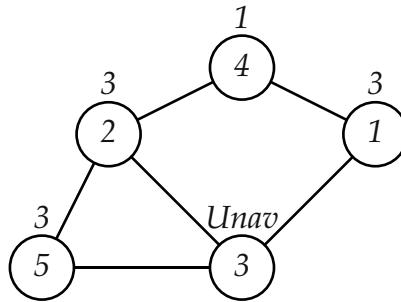
$$B(i) = \min\{ j : j \in N(i) \wedge P(j) \neq \text{Unav} \}$$

Remark 3. For any node i in an arbitrary global state, $B(i)$ may not exist.

For example, consider the graph shown in Figure 1 (where the values inside the nodes give the nodes’ IDs and the values outside the value of P). For the node 2, $A(2) = \{1, 4\}$ and $B(2) = 4$.

Definition 5. We define the **consistent** value $Q(i)$ for a node i as follows:

$$Q(i) = \begin{cases} \text{Unav} & \text{iff } A(i) \text{ exists and either } B(i) \text{ does not exist or } A(i) \prec \{i, B(i)\} \\ B(i) & \text{iff } B(i) \text{ exists and either } A(i) \text{ does not exist or } \{i, B(i)\} \preceq A(i) \\ \text{Open} & \text{iff neither } A(i) \text{ nor } B(i) \text{ exists} \end{cases}$$

**Fig. 1.** An example

Note that the value $Q(i)$ can be computed by the node i (i.e., it uses only local information).

Definition 6. *A node i in any given state is consistent iff $P(i) = Q(i)$; that is, for a node i to be consistent its $P(i)$ value must be unavailable if there is an edge incident with a neighbor that is smaller than any that it could be in; or must point to the smallest available neighbor if there is one; otherwise it must be open.*

The self-stabilizing algorithm SM for maximal strong matching in a given graph consists of one rule as shown in Figure 2. Thus, a node is **privileged** if $P(i) \neq Q(i)$. If it executes, then it sets $P(i) = Q(i)$.

```

if ( $P(i) \neq Q(i)$ )
  then set  $P(i) = Q(i)$ 
  
```

Fig. 2. Algorithm SM: Maximal Strong Matching Algorithm

3 Correctness at Convergence

Assume that the algorithm SM terminates in finite time; that is, the algorithm reaches a global state where all nodes are consistent, i.e., no node is privileged.

Definition 7. *A global state is called legitimate iff no node is privileged.*

Lemma 1. *In a legitimate state, for any node i , if $P(i) = j$ then $P(j) = i$.*

Proof. We claim that if $P(j) = \text{Unav}$ or $P(j) < i$, then i is privileged. For, if $P(j) = \text{Unav}$ then $B(i) \neq j$ so that $P(i) \neq Q(i)$; while if $P(j) < i$ then since $A(i) \preceq \{j, P(j)\}$ it follows that $Q(i) = \text{Unav}$ or $Q(i) \leq P(j)$, so that $P(i) \neq Q(i)$. Further, we claim that if $P(j) = \text{Open}$ or $P(j) > i$, then j is privileged. For, since $B(j) \leq i$, it follows that $Q(j) = \text{Unav}$ or $Q(j) \leq i$.

Remark 4. In a legitimate state, the set of edges $\{\{i, j\}; i, j \in V \wedge P(i) = j \wedge P(j) = i\}$ defines a matching \mathfrak{M} in the graph.

Lemma 2. *The set of edges in the matching \mathfrak{M} induces a collection of isolated edges, i.e., the matching \mathfrak{M} is strong.*

Proof. Suppose adjacent nodes i and j are both in the matching, but the edge between them is not part of the matching. That is, edges $e = \{i, P(i)\}$ and $f = \{j, P(j)\}$ are disjoint. Without loss of generality, $e \prec f$. Then j is privileged, a contradiction.

Lemma 3. *The strong matching \mathfrak{M} is maximal.*

Proof. Suppose an edge $e = (i, j)$ can be added to \mathfrak{M} and it still be a strong matching. Then neither i nor j has a neighbor in \mathfrak{M} . So neither $A(i)$ nor $A(j)$ exists. This means that since i and j are consistent, neither is unavailable; but then $B(i)$ and $B(j)$ exist, and hence both nodes are inconsistent, a contradiction.

4 Convergence or Termination in Finite Time

In this section we show that the algorithm SM terminates in a finite number of moves when it starts from an arbitrary initial state. The algorithm terminates when no node is privileged.

Definition 8. *Let X be a totally ordered set. Consider a sequence \mathfrak{S} of subsets $S(t)$ of X , $t = 1, 2, \dots$, such that each subset is obtained from the previous one by either the addition or deletion of one element. We say such a sequence is **downward** if for all elements i , if i is added at time t and then deleted at time t' , then some element smaller than i is added between time t and t' . That is, if $i \in S(\tau)$ for $t \leq \tau < t'$ but $i \notin S(t-1), S(t')$, then there exists $j < i$ and τ with $t < \tau < t' - 1$ such that $S(\tau) = S(\tau-1) \cup \{j\}$.*

Lemma 4. *If X is finite, then a downward sequence \mathfrak{S} on X is finite.*

Proof. We prove by induction on $|X|$. Consider the minimum element of X ; call it 1. It might be in or out of $S(1)$. But once added, it cannot be deleted. So define \mathfrak{S}' as the subsequence up to 1's addition, if it exists, and \mathfrak{S}'' as the subsequence after 1's addition. (If 1 is never added then set $\mathfrak{S}' = \emptyset$ and $\mathfrak{S}'' = \mathfrak{S}$.) Then ignore the transition where 1 is deleted, if this occurs, and restrict both subsequences to the set $X - \{1\}$. The result is two downward sequences on the set $X - \{1\}$. Hence, if $M(m)$ denotes the maximum length of a downward sequence for a set of cardinality m , it follows that $M(m) \leq 2 + 2M(m-1)$.

Definition 9. For any node k , in a given global state, we define the set of edges

$$S_k = \{ \{j, P(j)\} : j \geq k \wedge P(j) < k \}.$$

Remark 5. Note that S_k exists for a node k only when $P(k) \notin \{\text{Open}, \text{Unav}\}$. Also, $0 \leq |S_k| \leq n - 1$.

Definition 10. We say that a node is **static** if it does not execute (irrespective of it is privileged or not). We say a node is **stable** if does not execute at all until termination of the algorithm.

Remark 6. During the interval from the time the algorithm starts from an arbitrary state to the time it converges, a node may be static over a period of time but may not be stable during that period.

Lemma 5. Assume the nodes less than node k , for a given k , are static over a period of time. Then S_k can change only a finite number of times during that period.

Proof. Suppose at some stage $P(i)$ becomes j , with $i \geq k > j$. Since the nodes less than k are stable, it follows that $B(i) = j$ throughout. So for i to change again, it must happen that $A(i)$ changes to a value smaller than $\{i, j\}$.

That is, if at some stage $\{i, j\}$ is added to S_k , then before it is deleted some smaller edge must be added to S_k . (Note that if at the start $P(i) = j'$ with $j' < k$, then we assume the change in S_k occurs in two steps: add $\{i, j\}$ and then delete $\{i, j'\}$.) Thus, the sequence of S_k is a downward sequence. By Lemma 4, S_k can change only a finite number of times.

Lemma 6. Assume the nodes less than k , for a given k , are static over a period of time.

1. If node k is at some stage consistent and available, and later declares itself not available, then in between there was an addition to S_k .
2. If k is available throughout, and v is some neighbor of k that is at some stage consistent and available, and later declares itself not available, then in between there was an addition to S_k .

Proof. 1. If node k declares itself not available, a better edge must have been created in its neighborhood since node k was consistent. Hence there was an addition to S_k .
2. If k is available, then at the moment of declaring itself not available the node v must see an edge smaller than $\{v, k\}$, which did not exist when it was consistent. Hence there must have been an addition to S_k .

Lemma 7. Assume the nodes less than k are stable. Then k moves a finite number of times.

Proof. There cannot be an infinite cycle of node k doing just the following types of moves: Unav to Open; Open to a value and/or decrease. So consider each time that k declares itself unavailable, increases value or changes from a value to being open. The latter two can only occur if some neighbor declares itself not available. Hence each such move involves either k or one of its neighbors declaring itself unavailable. By Lemmas 5 and 6, this can occur only a finite number of times. Hence k can be privileged only a finite number of times.

Theorem 1. *Starting from an arbitrary initial state, the algorithm terminates in finite amount of time.*

Proof. We use induction and Lemma 7; it follows that nodes $\{1, \dots, k\}$ can be privileged (or make a move) only a finite number of times. Hence the algorithm terminates.

5 Complexity Analysis

We note that in our application, the downward sequences have the additional property that if i is deleted because of j 's addition, then i cannot be re-added until j is deleted. It can readily be shown that such sequences have length $O(|X|^2)$. However, this provides no improvement in the bound on the running time of the overall algorithm—which is $O(n^n)$ since no state can repeat.

The algorithm we present does indeed have exponential running time. Consider the following example. Take any graph G and add three new nodes x_3, x_2, x_1 such that x_3 is adjacent to all nodes, while x_1 and x_2 are adjacent only to each other and to x_3 , and such that these three nodes have the smallest IDs and $x_3 > x_2 > x_1$. Call the resulting graph G' .

Assume all nodes start as Unav. Then assume the demon proceeds as follows:

- (1) fire G until it stabilises;
- (2) fire x_1 (so goes to Open) and then x_3 (so points to x_1);
- (3) fire all nodes in G (so go to Unav);
- (4) fire x_2 (so points to x_1) and then x_3 (so goes to Unav);
- (5) fire G until it stabilises.

If $M(G)$ denotes the maximum number of steps on graph G assuming all nodes start as unavailable, then it follows that $M(G') \geq 2M(G)+4$. By repeating this construction, it follows that running time can be at least $2^{n/3}$.

Acknowledgment. This work was supported by NSF grant # ANI-0218495 and Srimani's work was partly supported also by NSF grant # ANI-0073409.

References

1. M Gradinariu and S Tixeuil. Self-stabilizing vertex coloration of arbitrary graphs. In *4th International Conference On Principles Of DIistributed Systems, OPODIS'2000*, pages 55–70. Studia Informatica Universalis, 2000.

2. S Dolev and JL Welch. Crash resilient communication in dynamic networks. *IEEE Transactions on Computers*, 46:14–26, 1997.
3. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.
4. L. Lamport. Solved problems, unsolved problems, and non-problems in concurrency. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, 1984.
5. M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.
6. T. Herman. A comprehensive bibliograph on self-stabilization, a working paper. *Chicago J. Theoretical Comput. Sci.*, <http://www.cs.uiowa.edu/ftp/selfstab/bibliography>.
7. G Antoniou and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-Par'99 Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.
8. J Beauquier, AK Datta, M Gradinariu, and F Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer LNCS:1914*, pages 223–237, 2000.
9. M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer LNCS:1693*, pages 254–268, 1999.
10. J. R. S. Blair and F. Manne. Efficient self-stabilizing algorithms for tree networks. In *Proceedings of ICDCS-2003*, Rhode Island, 2003.
11. S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Fault tolerant distributed coloring algorithms that stabilize in linear time. In *Proceedings of the IPDPS-2002 Workshop on Advances in Parallel and Distributed Computational Models*, pages 1–5, 2002.
12. K. Cameron. Induced matchings. *Discrete Applied Mathematics*, 24:97–102, 1989.
13. M. C. Golumbic and M. Lewenstein. New results in induced matchings. *Discrete Applied Mathematics*, 101(1-3):157–165, 2000.

Parallel Data Cube Construction: Algorithms, Theoretical Analysis, and Experimental Evaluation*

Ruoming Jin, Ge Yang, and Gagan Agrawal

Department of Computer and Information Sciences

Ohio State University, Columbus OH 43210

{jinr,yangg,agrawal}@cis.ohio-state.edu

Abstract. Data cube construction is a commonly used operation in data warehouses. Because of the volume of data that is stored and analyzed in a data warehouse and the amount of computation involved in data cube construction, it is natural to consider parallel machines for this operation. This paper presents two new algorithms for parallel data cube construction, along with their theoretical analysis and experimental evaluation. Our work is based upon a new data-structure, called the aggregation tree, which results in minimally bounded memory requirements. An aggregation tree is parameterized by the ordering of dimensions. We prove that the same ordering of the dimensions minimizes both the computational and communication requirements, for both the algorithms. We also describe a method for partitioning the initial array, which again minimizes the communication volume for both the algorithms. Experimental results further validate the theoretical results.

1 Introduction

Analysis on large datasets is increasingly guiding business decisions. Retail chains, insurance companies, and telecommunication companies are some of the examples of organizations that have created very large datasets for their decision support systems. A system storing and managing such datasets is typically referred to as a data warehouse and the analysis performed is referred to as On Line Analytical Processing (OLAP).

Computing multiple related group-bys and aggregates is one of the core operations in OLAP applications. Jim Gray has proposed the *cube* operator, which computes group-by aggregations over all possible subsets of the specified dimensions [5]. When datasets are stored as (possibly sparse) arrays, data cube construction involves computing aggregates for all values across all possible subsets of dimensions. If the original (or *initial*) dataset is an n -dimensional array, the data cube includes C_m^n m -dimensional arrays, for $0 \leq m \leq n$. Developing sequential algorithms for constructing data cubes is a well-studied problem [6,9,7].

Data cube construction is a compute and data intensive problem. Therefore, it is natural to use parallel computers for data cube construction. There is only a limited body of work on parallel data cube construction [2,3,4].

* This work was supported by NSF grant ACR-9982087, NSF CAREER award ACR-9733520, and NSF grant ACR-0130437.

This paper develops and analyzes efficient algorithms for parallel data cube construction. We have developed two algorithms, Level One Parallel and All Levels Parallel, for this problem. Our work is based upon a new data-structure called the *aggregation tree*, which ensures maximal cache and memory reuse in data cube construction. The aggregation tree is parameterized by the ordering of dimensions. If the original array is n -dimensional, there are $n!$ instantiations of the aggregation tree. We show that the same ordering of the dimensions ensures that each array is computed from its minimal parent, as well as minimizes the communication volume. This result applies to both the algorithms we have developed. The communication volume is further dependent upon the partitioning of the original array between the processors. We have developed an algorithm for partitioning the array. We show that our approach minimizes the interprocessor communication volume. Again, this result applies to both the algorithms we have developed.

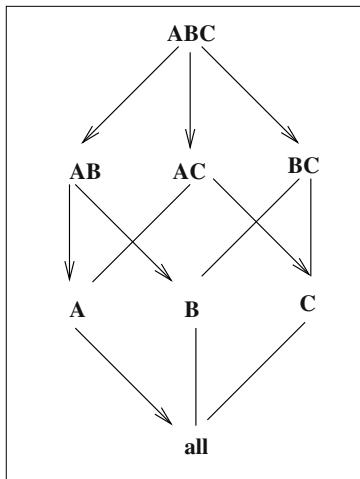


Fig. 1. Lattice for data cube construction. Edges with arrows show the minimal spanning tree when $|A| \leq |B| \leq |C|$

2 Data Cube Construction

Organizations often find it convenient to express facts as elements of a (possibly sparse) multidimensional array. For example, a retail chain may store sales information using a three-dimensional dataset, with item, branch, and time being the three dimensions. An element of the array depicts the quantity of the particular item sold, at the particular branch, and during the particular time-period.

In data warehouses, typical queries can be viewed as *group-by* operations on a multidimensional dataset. To provide fast response to the users, a data warehouse computes

aggregated values for all combination of values. If the original dataset is n dimensional, this implies computing and storing nC_m m-dimensional arrays, for $0 \leq m \leq n$.

For simplicity, assume that the original dataset is three-dimensional. Let the three dimensions be A , B , and C . The sizes along these dimensions are $|A|$, $|B|$, $|C|$, respectively. Without loss of generality, we assume that $|A| \leq |B| \leq |C|$. We denote the original array by ABC. Then, data cube construction involves computing arrays AB, BC, AC, A, B, C, and a scalar value *all*. As an example, the array AB has the size $|A| \times |B|$.

We now discuss the major issues in efficient algorithm design. We use the above example to further illustrate these issues.

Cache and Memory Reuse: Consider the computation of AB, AC, and BC. These three arrays need to be computed from the initial array ABC. When the array ABC is disk-resident, performance is significantly improved if each portion of the array is read only once. After reading a portion or chunk of the array, corresponding portions of AB, AC, and BC can be updated simultaneously. Even if the array ABC is in main memory, better cache reuse is facilitated by updating portions of AB, AC, and BC simultaneously. The same issue applies at later stages in data cube construction, e.g., in computing A and B from AB.

Using minimal parents: In our example, the arrays AB, BC, and AC need to be computed from ABC, by aggregating values along the dimensions C, A, and B, respectively. However, the array A can be computed from either AB or AC, by aggregating along dimensions B or C. Because $|B| \leq |C|$, it requires less computation to compute A from AB. Therefore, AB is referred to as the *minimal parent* of A.

A lattice can be used to denote the options available for computing each array within the cube. This lattice is shown in Figure 1. A data cube construction algorithm chooses a spanning tree of the lattice shown in the figure. The overall computation involved in the construction of the cube is minimized if each array is constructed from the *minimal parent*. Thus, the selection of a *minimal spanning tree* with minimal parents for each node is one of the important considerations in the design of a sequential (or parallel) data cube construction algorithm.

Memory Management: In data cube construction, not only the input datasets are large, but the output produced can be large also. Consider the data cube construction using the minimal spanning tree shown in Figure 1. Sufficient main memory may not be available to hold the arrays AB, AC, BC, A, B, and C at all times. If a portion of the array AB is written to the disk, it may have to be read again for computing A and B. However, if a portion of the array BC is written back, it may not have to be read again.

Communication Volume: Consider the computation of AB, AC, and BC from ABC. Suppose we assume that the dataset will be partitioned along a single dimension. Then, the communication volume required when the dataset is partitioned along the dimensions A, B, or C is $|B| \times |C|$, $|A| \times |C|$, and $|A| \times |B|$, respectively. If $|A| \leq |B| \leq |C|$, then the minimal communication volume is achieved by partitioning along the dimension C.

High communication volume can easily limit parallel performance. It is important to minimize communication volume for the entire data cube construction process, possibly by considering partitioning along multiple dimensions.

3 Spanning Trees for Cube Construction

This section introduces a data-structure that we refer to as the *aggregation tree*.

To introduce the aggregation tree, we initially review *prefix tree*, which is a well-known data-structure [1].

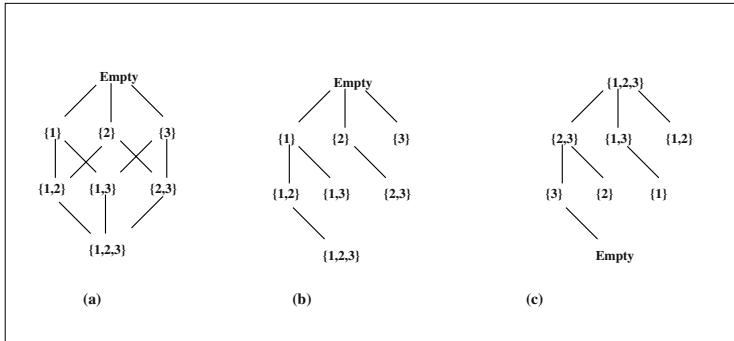


Fig. 2. Prefix Lattice (a), Prefix Tree (b), and Aggregation Tree (c) for $n = 3$

Consider a set $X = \{1, 2, \dots, n\}$. Let $\rho(X)$ be the power set of X .

Definition 1. $L(n)$ is a lattice (V, E) such that:

- The set of nodes V is identical to the power set $\rho(X)$.
- The set of edges E denote the immediate superset relationship between elements of the power set, i.e, if $r \in \rho(X)$ and $s \in \rho(X)$, $r = s \cup \{i\}$, and $i \notin s$, then $(r, s) \in E$.

The lattice $L(n)$ is also referred to as the *prefix lattice*. The lattice we had shown earlier in Figure 1 is a complement of the prefix lattice, and is referred to as the *data cube* lattice.

A prefix tree $P(n)$ is a spanning tree of the prefix lattice $L(n)$. It is defined as follows:

Definition 2. Given a set $X = \{1, 2, \dots, n\}$, a prefix tree $P(n)$ is defined as follows:

- (a) ϕ is the root of the tree.
- (b) The set of nodes of the tree is identical to the power set $\rho(X)$.
- (c) A node $\{x_1, x_2, \dots, x_m\}$, where $m \leq n$, and $1 \leq x_1 < x_2 < \dots < x_m \leq n$, has $n - m$ children. These children, ordered from left to the right are, $\{x_1, x_2, \dots, x_m\} \cup \{x_m + 1\}, \dots, \{x_1, x_2, \dots, x_m\} \cup \{n\}$.

Given a prefix tree $P(n)$, the corresponding aggregation tree $A(n)$ is constructed by complementing every node in $P(n)$ with respect to the set X . Formally,

Definition 3. Given a set $X = \{1, 2, \dots, n\}$ and the prefix tree $P(n)$ as defined earlier, an aggregation tree $A(n)$ is defined as follows:

- (a) If r is a node in $P(n)$, then there is a node r' in $A(n)$, such that $r' = X - r$.
- (b) If a node r has a child s in $P(n)$, then the node r' in $A(n)$ has a child s' .

Figure 2 shows the prefix lattice, prefix tree and the aggregation tree for $n = 3$.

```

Construct_Cube( $D_1, D_2, \dots, D_n$ )
{
    Evaluate( $\{D_1, D_2, \dots, D_n\}$ )
}

Evaluate( $l$ )
{
    Compute all children of  $l$ 
    For-each children  $r$  from right to left
        If  $r$  has no children
            Write-back to the disk
        Else Evaluate( $r$ )
        Write-back  $l$  to the disk
}

```

Fig. 3. Sequential Data Cube Construction Using the Aggregation Tree

Since an aggregation tree is a spanning tree of the data cube lattice, it can be used for data cube construction. We next present an algorithm that uses the aggregation tree and has minimally bounded memory requirements. Figure 3 shows this sequential algorithm.

An important result is as follows.

Theorem 1. Consider an original n dimensional array D_1, D_2, \dots, D_n where the size of the dimension D_i is $|D_i|$. The total memory requirement for holding the results in data cube construction using the algorithm in Figure 3 are bounded by

$$\sum_{i=1}^n \left(\prod_{j=1, j \neq i}^n |D_j| \right)$$

4 Parallel Data Cube Construction Using the Aggregation Tree

In this section, we present two parallel algorithms for data cube construction using the aggregation tree. The first algorithm is referred to as the *level one parallel* algorithm and the second algorithm is referred to as the *all levels parallel* algorithm.

4.1 Level One Parallel Algorithm

Consider again a n -dimensional initial array from which the data cube will be constructed. Suppose we will be using a distributed memory parallel machine with 2^p processors. We partition the dimension D_i along 2^{k_i} processors, such that $\sum_{i=1}^n k_i = p$. Each processor is given a unique label $\{l_1, l_2, \dots, l_n\}$ such that $0 \leq l_i \leq 2^{k_i} - 1$. Since $\sum_{i=1}^n k_i = p$, it is easy to verify that there are 2^p unique labels. A processor with the label l_i is given the l_i^{th} portion along the dimension D_i .

A processor with the label $l_i = 0$ is considered one of the *lead* processors along the dimension D_i . There are $2^p/2^{k_i}$ lead processors along the dimension D_i . The significance of a lead processor is as follows. If we aggregate along a dimension, then the results are stored in the lead processors along that dimension.

Parallel algorithm for data cube construction using the aggregation tree is presented in Figure 4.

```

Construct_Cube( $D_1, D_2, \dots, D_n$ )
{
    Evaluate( $\{D_1, D_2, \dots, D_n\}$ ) on each processor
}

Evaluate( $l$ )
{
    Locally aggregate all children of  $l$ 
    Forall children  $r$  from right to left
        Let  $r' = X - r = \{D_{i1}, \dots, D_{im}\}$ 
        If the processor is the lead processor along  $D_{i1}, \dots, D_{im}$ 
            Communicate with other processors to finalize portion of  $r$ 
        If  $r$  has no children
            Write-back the portion to the disk
        Else Evaluate( $r$ )
    Write-back  $l$  to the disk
}

```

Fig. 4. Level One Parallel Algorithm for Data Cube Construction

An important question is, “*what metric(s) we use to evaluate the parallel algorithm?*”. The dominant computation is at the first level, and it is fully parallelized by the algorithm. Our earlier experimental work [8] has shown that communication volume is a critical factor in the performance of parallel data cube construction on distributed memory parallel machines. Therefore, we focus on communication volume as a major metric in analyzing the performance of a parallel data cube construction algorithm.

Lemma 1. Consider a node $r = \{y_1, y_2, \dots, y_k\}$ and its child $s = \{y_1, y_2, \dots, y_k, m\}$ in the prefix tree, where $1 \leq y_1 < y_2 < \dots < y_k < m \leq n$. Then, the communication volume in computing the corresponding node s' in the aggregation tree from the node r' is given by

$$\left(\prod_{i=1, i \neq y_1, y_2, \dots, y_k, m}^n |D_i| \right) \times (2^{k_m} - 1)$$

Theorem 2. *The total communication volume for data cube construction is given by*

$$\left(\prod_{i=1}^n |D_i| \right) \times \left(\sum_{i=1}^n \frac{2^{k_i} - 1}{|D_i|} \times \left(\prod_{j=1}^{i-1} \left(1 + \frac{1}{|D_j|} \right) \right) \right)$$

4.2 All Levels Parallel Algorithm

In the algorithm we described above, the parallelism is limited. As we go down the tree, the computation of children and the process of writing back results take place only on the lead processors, and other processors do not do anything after sending out their portions to their corresponding lead processors. In this subsection we present another algorithm, referred to as the All Levels Parallel Algorithm. This algorithm further increases the parallelism by partitioning the children in the aggregation tree.

There are no lead processors in this algorithm. Each child is further partitioned into several sections. Each processor keeps an *owned section* of each child and needs to communicate with other processors in order to finalize its owned section. We call the process *finalization* when a processor receives corresponding sections from other processors and aggregates the incoming sections to its owned section. The owned section can be determined by the label of the processor. Moreover, for a d -dimensional child in the aggregation tree, we associate with it a vector $\{k_i\}$ of length d to denote along which dimensions the child is partitioned.

We make the same assumptions and use the same terminology as used for presenting the Level One Parallel Algorithm. The algorithm is presented in Figure 5.

5 Optimality Properties and Partitioning

In this section, we focus on two issues, and establish important results that apply to both of the algorithms. The first issue is about ordering of dimensions. As we had stated earlier, an aggregation tree is parameterized with the ordering of dimensions. In computing data cube starting from an n dimensional array, $n!$ instantiations of the aggregation tree are possible. In this section, we show important results, which is that the same ordering of dimensions minimizes both the communication volume and the computation cost, for both the algorithms. The latter also means that all nodes in the data cube lattice are computed from minimal parents.

The second issue is partitioning of the initial dataset. We present an algorithm that determines how different dimensions are partitioned. Then, we state results establishing that this algorithm minimizes communication volume for both of the parallel algorithms.

Theorem 3. *Using aggregation tree ensures that all arrays are computed from their minimal parents iff $|D_1| \geq |D_2| \geq \dots \geq |D_n|$.*

```

Construct_Cube( $D_1, D_2, \dots, D_n$ )
{
    Evaluate( $\{D_1, D_2, \dots, D_n\}$ ) on each processor
}

Evaluate( $l$ )
{
    Locally aggregate all children of  $l$ 
    Forall children  $r = \{D_{i_1}, \dots, D_{i_m}\}$  from right to left on each processor
        Let  $r' = X - r = \{D_{j_1}, \dots, D_{j_{n-m}}\}$ 
        If  $\sum_{t=1}^{n-m} 2^{k_{j_t}} > 1$ 
            Further partition  $r$  into  $\sum_{t=1}^{n-m} 2^{k_{j_t}}$  sections
            Communicate with other corresponding processors to finalize the owned section  $r_0$  of  $r$ 
            If  $r$  has no children
                Write-back the owned section  $r_0$  to the disk
            Else Evaluate( $r_0$ )
            Else If  $r$  has no children
                Write-back  $r$  to the disk
            Else Evaluate( $r$ )
        Write-back  $l$  to the disk
}

```

Fig. 5. All Levels Parallel Data Cube Construction

Theorem 4. Among all instantiations of the aggregation tree, minimal communication volume using the Level One Parallel Algorithm is achieved by the instantiation where $|D_1| \geq |D_2| \geq \dots \geq |D_n|$.

Theorem 5. Among all instantiations of the aggregation tree, minimal communication volume using the All Levels Parallel Algorithm is achieved by the instantiation where $|D_1| \geq |D_2| \geq \dots \geq |D_n|$.

The next issue we focus on is partitioning of the original dataset between the processors. The expression for communication volume we derived for the Level One Parallel Algorithm is dependent on the partitioning of the original array between the processors, i.e., the values of k_i , $i = 1, \dots, n$. Given 2^p processors and an original array with n dimensions, there are a total of ${}^{n+p}C_n$ distinct ways of partitioning the array between processors. In general, it is not feasible to evaluate the communication costs associated with each of these partitions. We have developed an $O(p)$ time algorithm for choosing the values of k_i , $i = 1, \dots, n$, $\sum_{i=1}^n k_i = p$, to minimize the total communication volume.

We have proved that this method minimizes the interprocessor communication volume for the Level One Parallel Algorithm. Moreover, even though there is no closed form expression for the total communication volume with the All Levels Parallel Algorithm, the same method still minimizes the communication volume.

Recall that the expression for communication volume we derived is

$$\left(\prod_{i=1}^n |D_i| \right) \times \left(\sum_{i=1}^n \frac{2^{k_i} - 1}{|D_i|} \times \left(\prod_{j=1}^{i-1} \left(1 + \frac{1}{|D_j|} \right) \right) \right)$$

This can be restated as

$$\left(\prod_{i=1}^n |D_i| \right) \times \left(\sum_{i=1}^n \frac{2^{k_i}}{|D_i|} \times \left(\prod_{j=1}^{i-1} \left(1 + \frac{1}{|D_j|} \right) \right) \right) - \sum_{i=1}^n \frac{1}{|D_i|} \times \left(\prod_{j=1}^{i-1} \left(1 + \frac{1}{|D_j|} \right) \right)$$

Our goal is to choose the values of k_i for a set of given values of $|D_i|$, $i = 1, \dots, n$. Therefore, we state the communication volume as

$$c_0 \times \left(\sum_{i=1}^n 2^{k_i} \times X_i \right) - d_0$$

where,

$$X_i = \frac{1}{|D_i|} \times \left(\prod_{j=1}^{i-1} \left(1 + \frac{1}{|D_j|} \right) \right)$$

and the values of c_0 and d_0 do not impact the choices of k_i .

The algorithm is presented in Figure 6. Initially, k_i , for all values of i , are initialized to 0. In each iteration of the algorithm, we find the X_i with the minimal value, increment the corresponding k_i by 1, and replace X_i with $2 \times X_i$.

```

Partition( $n, p, X_1, X_2, \dots, X_n$ )
{
    Initialize  $k_1 = k_2 = \dots = k_n = 0$ 
    While ( $p > 0$ ) {
        Let  $X_i = \min(X_1, X_2, \dots, X_n)$ 
         $k_i = k_i + 1$ 
         $X_i = 2 \times X_i$ 
         $p = p - 1$ 
    }
}

```

Fig. 6. Partitioning Different Dimensions to Minimize Communication Volume

Theorem 6. *Partitioning done using the algorithm in Figure 6 minimizes the interprocessor communication volume, for both Level One Parallel Algorithm and the All Levels Parallel Algorithm.*

6 Experimental Results

Because of limited space, we only give a brief summary of the experimental results. We carried out a series of experiments to 1) evaluate the parallelism obtained from both the algorithms, and 2) validate if our theoretical results on optimal partitions correspond to actual performance. Our results show that 1) Choosing data distribution to minimize communication volume made a substantial difference in the performance in most of the cases. This difference became more significant with increasing sparsity level in the dataset. 2) Both the algorithms give good speedups. Using parallelism at all levels gave better performance, even though it increases the total communication volume.

7 Conclusions

In this paper, we have addressed a number of algorithmic and theoretic results for sequential and parallel data cube construction.

For sequential data cube construction, we have developed a data-structure called aggregation tree. If the data cube is constructed using a right-to-left depth-first traversal of the tree, the total memory requirements are minimally bounded. As compared to the existing work in this area, our approach achieves a memory bound without requiring frequent writing back to the disks. We have presented a number of results for parallel data cube construction. First, we have presented an aggregation tree based algorithm for parallel data cube construction. Again, we have shown that memory requirements are minimally bounded. We have also developed a closed form expression for total communication volume in data cube construction. We have shown that the same ordering of dimensions minimizes both the communication volume as well as computation. Finally, we have presented an algorithm with $O(p)$ time complexity for optimally partitioning the input array on 2^p processors, with the goal of minimizing the communication requirements.

References

1. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.
2. Frank Dehne, Todd Eavis, Susanne Hambrusch, and Andrew Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases: An International Journal (Special Issue on Parallel and Distributed Data Mining)*, to appear, 2002.
3. Sanjay Goil and Alok Choudhary. High performance OLAP and data mining on parallel computers. Technical Report CPDC-TR-97-05, Center for Parallel and Distributed Computing, Northwestern University, December 1997.
4. Sanjay Goil and Alok Choudhary. PARSIMONY: An infrastructure for parallel multidimensional analysis and data mining. *Journal of Parallel and Distributed Computing*, 61(3):285–321, March 2001.
5. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregational Operator for Generalizing Group-Bys, Cross-Tabs, and Sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, 1995.

6. S.Agrawal, R. Agrawal, P. M.Deshpande, A. Gupta, J.F.Naughton, R. Ramakrishnan, and S.Sarawagi. On the computation of multidimensional aggregates. In *Proc 1996 Int. Conf. Very Large Data Bases*, pages 506–521, Bombay, India, September 1996.
7. Yin Jenny Tam. Datacube: Its implementation and application in olap mining. Master's thesis, Simon Fraser University, September 1998.
8. Ge Yang, Ruoming Jin, and Gagan Agrawal. Implementing data cube construction using a cluster middleware: Algorithms, implementation experience and performance evaluation. In *The 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, Berlin, Germany, May 2002.
9. Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. An array based algorithm for simultaneous multidimensional aggregates. In *Prceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–170. ACM Press, June 1997.

Efficient Algorithm for Embedding Hypergraphs in a Cycle

Qian-Ping Gu and Yong Wang

School of Computing Science
Simon Fraser University
Burnaby B.C. Canada V5A 1S6
{qgu, ywangb}@cs.sfu.ca

Abstract. The problem of *Minimum Congestion Hypergraph Embedding in a Cycle (MCHEC)* is to embed the hyperedges of a hypergraph as paths in a cycle such that the maximum congestion (the maximum number of paths that use any single link in the cycle) is minimized. This problem has many applications, including minimizing communication congestions in computer networks and parallel computations. The MCHEC problem is NP-hard. We give a 1.8-approximation algorithm for the problem. This improves the previous 2-approximation results. The algorithm has the optimal $O(mn)$ time for the hypergraph with m hyperedges and n nodes.

Keywords: Hypergraph embedding in a cycle, approximation algorithms, communication on rings, link congestion minimization.

1 Introduction

The *Minimum Congestion Hypergraph Embedding in a Cycle (MCHEC)* problem [3] is to embed the hyperedges of a hypergraph as paths in a cycle on the same number of nodes such that the maximum congestion is minimized, where the congestion of a link in the cycle is the number of paths that use the link. The MCHEC problem has applications in network communication, parallel computation, electronic design automation, and so on [2,5,6,7,9]. A communication application on a network can be defined by a set of routing requests and each request is defined by a subset of network nodes to be connected in the request. For one-to-one routing (unicast), each request involves two nodes, the source and destination. For more complicated communication applications, such as multi-cast, a request may involve more than two nodes of the network. In general, a communication application on a network can be further modeled as a hypergraph on the same node set of the network with each routing request in the application represented by a hyperedge in the hypergraph. To realize a communication application, a routing algorithm is required to set-up a virtual or dedicated routing path in the network to connect the nodes in each hyperedge. One of the most important issues in designing routing algorithms is to minimize the congestion on the links in the network. An algorithm for the MCHEC problem realizes the

communication application represented by the hypergraph on the ring with the maximum congestion over all links minimized. Similarly, in parallel computation, the processors of a parallel computer can be represented by the nodes of a hypergraph and a group of processors which are required to communicate with one another can be represented by a hyperedge. A solution to the MCHEC problem gives a routing for the communications among the processors in each group on the ring connected parallel computers such that the maximum congestion over all links is minimized.

If each hyperedge of the hypergraph contains exactly two nodes then the MCHEC problem becomes *Minimum Congestion Graph Embedding in a Cycle* problem and the optimal solution of the problem can be solved in polynomial time [2]. The algorithm used to solve the graph embedding in a cycle problem is based on the Okamura-Seymour planar multicommodity flow theorem [10], in which each connection request consists of a source node and a destination node, so it can not be extended to solve the general hypergraph (each hyperedge may contain more than two nodes) embedding in a cycle problem. For general hypergraphs, Ganley and Cohoon [3] proved that the MCHEC problem is NP-hard and gave a 3-approximation algorithm for the problem. They also gave an algorithm that determines whether an instance of the MCHEC problem has a solution with maximum congestion at most l in $O((mn)^{l+1})$ time for hypergraphs with m hyperedges and n nodes. The $O((mn)^{l+1})$ result implies that the MCHEC problem can be solved in polynomial time for subproblems with constant maximum congestions. For subproblems with non-constant maximum congestions, better approximation algorithms have been proposed using various approaches [1,4,8]. Carpenter et al. [1] gave a simple algorithm based on *clockwise embedding* for the MCHEC problem. In clockwise embedding, the nodes in the cycle are clockwise numbered by $0, 1, \dots, n - 1$ and the nodes in each hyperedge are connected by a path in the cycle from the smallest numbered node to the largest numbered node in the clockwise direction. Gonzalez [4] presented two algorithms for the MCHEC problem. The first algorithm transforms the MCHEC problem to an Integer Linear Programming (ILP) problem. Approximate solutions for the ILP problem are obtained by rounding-off the solutions of the corresponding linear programming problem in polynomial time. The second algorithm transforms the MCHEC problem to the graph embedding in a cycle problem. Lee and Ho [8] proposed an algorithm based on *longest adjacent path removing*. All the algorithms in [1,4,8] have the approximation ratio two.

In this paper, we give a 1.8-approximation algorithm for the MCHEC problem. Our algorithm starts from the clockwise embedding. Then the algorithm tries to re-embed some hyperedges to reduce the maximum congestion. Let L be the maximum congestion in the clockwise embedding and L^* be the maximum congestion in the optimal embedding. Our algorithm tries to re-embed k hyperedges to get an embedding with maximum congestion $L-k$ for k as large as possible. The approximation ratio of the algorithm is $(L-k)/L^*$. Since $\lceil L/2 \rceil \leq L^*$ [1], the approximation ratio of our algorithm is at most $(L-k)/\lceil L/2 \rceil$. This gives a good approximation ratio when k is large. When k is small, we prove a new

lower bound on L^* . Roughly speaking, the approximation ratio of the algorithm increases from 1 to 1.8 as k decreases from $L/2$ to $L/10$, the approximation ratio decreases from 1.8 to 1.5 as k decreases from $L/10$ to 0, and the ratio is 1.8 in the worst case. Our algorithm has the optimal $O(mn)$ time for the hypergraph with m hyperedges and n nodes.

The rest of the paper is organized as follows. In Section 2, we give the preliminaries of the paper and review the clockwise embedding. Section 3 gives our algorithm and the analysis of the algorithm. The final section concludes the paper.

2 Preliminaries

A *cycle* C of n nodes is an undirected graph with node set $\{i \mid 0 \leq i \leq n - 1\}$. There is a link between nodes i and j if $i = j \pm 1$, where and in what follows, the arithmetic involving nodes is performed implicitly using modulo n operation. A *hypergraph* $H(V, E_H)$ of n nodes and m hyperedges is a hypergraph with node set $V = \{i \mid 0 \leq i \leq n - 1\}$ and hyperedge set $E_H = \{e_1, \dots, e_m\}$, where each hyperedge e_i is a subset of V with two or more nodes. For $1 \leq i \leq m$, a *connecting path* (or *c-path*) p_i in C for hyperedge e_i is a path in C such that all nodes in e_i are connected by p_i . An embedding of hypergraph $H(V, E_H)$ in cycle C is a set of c-paths in C such that each hyperedge of $H(V, E_H)$ has exactly one c-path in the set. Given an embedding of a hypergraph, the congestion of each link in C is the number of c-paths that contain the link. The MCHEC problem is that given a hypergraph and a cycle on the same node set, find an embedding of the hypergraph such that the maximum congestion (i.e. the maximum number of c-paths using any single link in the cycle) over all links in the cycle is minimized.

We assume that the nodes of C are clockwise numbered by $0, 1, \dots, n - 1$ and link $(i, i + 1)$ in C is numbered i (see Fig.1 (a)). Carpenter et al. [1] gave a simple 2-approximation algorithm called clockwise embedding: The c-path for each hyperedge e_i in the given hypergraph connects all the nodes of e_i in C from the smallest numbered node to the largest numbered node in the clockwise direction. Fig.1 (a) gives the clockwise embedding of the hypergraph with edges $e_1 = \{0, 1, 8\}$, $e_2 = \{1, 4, 7\}$, $e_3 = \{2, 3, 4\}$, and $e_4 = \{3, 5, 7\}$.

A *segment* of cycle C is a connected subgraph of C . Cycle C is cut into two segments by removing any two links x and y . We call the two links a $\text{cut}(x, y)$ of C . A hyperedge is *separated* by a cut if there is at least a node of the hyperedge in each of the two segments obtained from the cut. Obviously, the c-path in any embedding for a separated hyperedge must contain at least one of the two links in the cut. Let $N(x, y)$ be the number of hyperedges separated by links x and y . Then $\lceil N(x, y)/2 \rceil \leq L^*$. For the clockwise embedding, define $l(i)$ to be the congestion of link i in C and $L = \max\{l(i) \mid 0 \leq i \leq n - 1\}$ be the maximum congestion. Notice that $l(n - 1) = 0$. Assume that link s has the maximum congestion L . Then $N(n - 1, s) = L$. In any embedding, the c-path for a separated hyperedge by $\text{cut}(s, n - 1)$ must contain at least one of the links

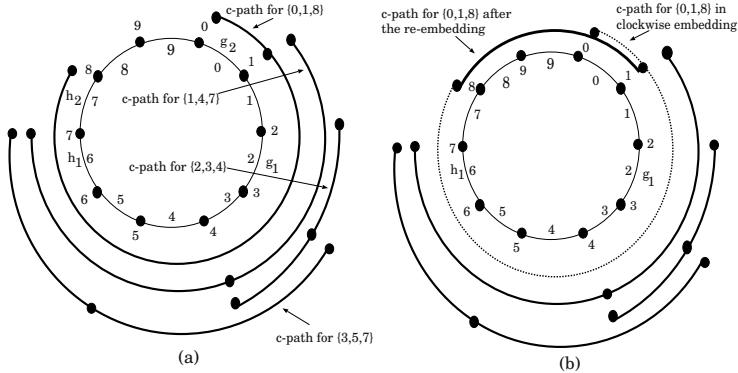


Fig. 1. (a) The cycle and clockwise embedding. (b) The c-paths for the candidate w.r.t. $k = 1$ in clockwise embedding and after the re-embedding.

$n - 1$ and s . From this, we have $\lceil L/2 \rceil \leq L^*$ and the approximation ratio of the clockwise embedding is $L/L^* \leq L/\lceil L/2 \rceil \leq 2$ [1].

In what follows, we define *segment* $\langle i, j \rangle$ of C to be the segment of C that includes the nodes from i to j in the clockwise direction.

3 Embedding Algorithm

In this section, we describe and analyze our algorithm for the MCHEC problem. We first introduce terminology for this purpose. Recall that $l(i)$ is the congestion of link i and L is the maximum congestion in the clockwise embedding, and L^* is the maximum congestion in the optimal embedding. For integer k with $1 \leq k \leq \lfloor L/2 \rfloor$, let g_k be the smallest link with $l(g_k) \geq L - 2k + 1$. Similarly, let h_k be the largest link with $l(h_k) \geq L - 2k + 1$. Notice that $0 \leq g_k \leq h_k < n - 1$. In Fig. 1 (a), $L = 4$, $g_k = 0$ and $h_k = 7$ for $k = 2$. For $k = 1$, $g_k = 2$ and $h_k = 6$. We call a hyperedge a *re-embedding candidate with respect to k* (or *candidate w.r.t. k*) if the hyperedge has a node in segment $\langle 0, g_k \rangle$, has a node in segment $\langle h_k + 1, n - 1 \rangle$, and has no node in segment $\langle g_k + 1, h_k \rangle$. In Fig. 1 (a), hyperedge $\{0, 1, 8\}$ is a candidate w.r.t. $k = 1$. For each candidate w.r.t. k , we can embed it in such a way that the c-path does not contain any link i with $g_k \leq i \leq h_k$ (see Fig. 1 (b)). From the definitions of g_k , h_k , and the candidate w.r.t. k , we can get $g_{k+1} \leq g_k$, $h_{k+1} \geq h_k$, and a candidate w.r.t. $k + 1$ is also a candidate w.r.t. k (the inverse may not be true). Let x_k be the number of candidates w.r.t. k . Then $x_{k+1} \leq x_k$.

The outline of our algorithm is as follows: We start with the clockwise embedding. Then we try to re-embed the candidates w.r.t. k such that the c-path for each candidate does not contain any link i with $g_k \leq i \leq h_k$. The re-embedding process starts from $k = \lfloor L/2 \rfloor$. If $x_k \geq k$ then we re-embed k or $k + 1$ candidates and the algorithm terminates. Otherwise, k is decreased by one and the re-embedding process is repeated. As shown later, the algorithm outlined above

has approximation ratio 1.8 except for a few special cases of constant L . Although the $O((mn)^{L^*+1})$ time algorithm in [3] can be used to find the optimal embedding for constant L , the time complexity could be too high in practice. We give a subroutine to handle those special cases. The subroutine is efficient and guarantees the 1.8 approximation ratio for the special cases. Our algorithm and subroutine are given in Fig. 2.

Procedure R_EMBEDDING

Input: A hypergraph on the same node set of the cycle.

Output: An embedding of the hypergraph in the cycle.

begin

1. Perform the clockwise embedding for the hypergraph.

 Let L denote the maximum congestion in the clockwise embedding.

2. Find links g_k and h_k , and compute x_k for $k = 1, 2, \dots, [L/2]$.

x_k is defined to be 0 for $k = [L/2] + 1$ and $k = 0$.

3. $k := [L/2]$.

while $k \geq 1$ **do**

if ($x_k \geq k$) **then** goto step 4 **else** $k := k - 1$.

4. **If** $x_k \geq k + 1$ **and** $x_{k+1} \geq 1$ **then**

 re-embed $k + 1$ candidates w.r.t. k including

 at least one candidate w.r.t. $k + 1$

else

if (($L = 2$ **or** $L = 4$) **and** $k = 0$) **or** ($L = 12$ **and** $k = 1$) **then**

 call Subroutine Special_Cases

else re-embed k candidates w.r.t. k .

end.

Subroutine Special_Cases

Input The clockwise embedding of the hypergraph.

Output An embedding of the hypergraph in the cycle.

begin

/* Let s be a link with the maximum congestion L in the clockwise embedding,

E be the set of hyperedges whose c-paths contain link s in the clockwise embedding, and p_i be the c-path for $e_i \in E$ that does not contain link s . */

If ($L = 2$ **or** $L = 4$) **and** $k = 0$ **then**

for every $e_i \in E$ **do**

if re-embedding e_i by p_i reduces L by one **then**

 re-embed e_i by p_i and **return**

else /* $L = 12$ **and** $k = 1$ */

for every pair $e_i, e_j \in E$ **do**

if re-embedding e_i, e_j by p_i, p_j reduces L by two **then**

 re-embed e_i by p_i and e_j by p_j and **return**

 re-embed one candidate w.r.t. $k = 1$.

Return

end.

Fig. 2. The embedding algorithm for the MCHEC problem.

Algorithm R_Embedding terminates with k taking one of $\{0, 1, \dots, \lfloor L/2 \rfloor\}$. Let L_k be the maximum congestion of the embedding by the algorithm.

Lemma 1. $L_k = L - k$ or $L_k = L - k - 1$.

Proof: When Algorithm R_Embedding terminates without calling Subroutine Special_Cases, either k candidates w.r.t. k are re-embedded, or $k + 1$ candidates w.r.t. k , including at least one candidate w.r.t. $k + 1$, are re-embedded. After a candidate w.r.t. k is re-embedded, the congestion of each link i with $g_k \leq i \leq h_k$ is decreased by one and the congestion of each link i with $i < g_k$ or $i > h_k$ is increased by at most one.

Assume that k candidates w.r.t. k are re-embedded. Since for link i with $g_k \leq i \leq h_k$, $l(i) \leq L$, and for i with $i < g_k$ or $i > h_k$, $l(i) \leq L - 2k$, we can have an embedding of maximum congestion $L - k$.

Assume that $k + 1$ candidates w.r.t. k are re-embedded. For each link i with $g_k \leq i \leq h_k$, the congestion of i after the re-embedding becomes $l(i) - (k + 1) \leq L - k - 1$. For each link i with $i < g_{k+1}$ or $i > h_{k+1}$, the congestion of i after the re-embedding is at most $l(i) + (k + 1) \leq L - 2(k + 1) + (k + 1) = L - k - 1$. Since the $k + 1$ candidates re-embedded include at least one candidate w.r.t. $k + 1$, the congestion of link i with $g_{k+1} \leq i < g_k$ or $h_{k+1} \geq i > h_k$ is bounded by $l(i) - 1 + k \leq L - 2k - 1 + k = L - k - 1$.

Assume that Subroutine Special_Cases is executed. For $k = 0$, $L_k = L - 1$ or $L_k = L$. For $k = 1$, $L_k = L - 2$ or $L - 1$. \square

From the above lemma, the approximation ratio of our algorithm can be obtained from $(L - k)/L^*$ or $(L - k - 1)/L^*$. For large k , we can use $\lceil L/2 \rceil$ as the lower bound on L^* and get the approximation ratio $(L - k)/\lceil L/2 \rceil$ or $(L - k - 1)/\lceil L/2 \rceil$ for our algorithm. This suggests that when the algorithm terminates with a large k , we have a good approximation ratio. However, if the algorithm terminates with a small k , e.g., $k = 0$ then the approximation ratio given by $L_k/\lceil L/2 \rceil$ is 2, no better than the clockwise embedding. In what follows, we prove that if the algorithm terminates with a small k then a lower bound better than $\lceil L/2 \rceil$ on L^* can be found. By this better lower bound, we can get the 1.8 approximation ratio of our algorithm. The lower bound $\lceil L/2 \rceil$ is obtained from the cut of two links in the cycle. The new lower bound derived below involves three links $n - 1$, g_k , and h_k .

To derive the lower bound, we need some new notation. For arbitrary links g and h in the cycle with $0 \leq g < h < n - 1$, we define

- W : the set of the hyperedges such that each hyperedge has a node in segment $\langle 0, g \rangle$, a node in segment $\langle g + 1, h \rangle$, and a node in segment $\langle h + 1, n - 1 \rangle$;
- X : the set of the hyperedges such that each hyperedge has a node in segment $\langle 0, g \rangle$, has NO node in segment $\langle g + 1, h \rangle$, and has a node in segment $\langle h + 1, n - 1 \rangle$;
- Y : the set of the hyperedges such that each hyperedge has a node in segment $\langle 0, g \rangle$, a node in segment $\langle g + 1, h \rangle$, and has NO node in segment $\langle h + 1, n - 1 \rangle$; and
- Z : the set of the hyperedges such that each hyperedge has NO node in segment $\langle 0, g \rangle$, has a node in segment $\langle g + 1, h \rangle$, and a node in segment $\langle h + 1, n - 1 \rangle$.

In Fig. 1 (a), let $g = 2$ and $h = 6$. Then hyperedge $\{1, 4, 7\}$ is in W , hyperedge $\{0, 1, 8\}$ is in X , hyperedge $\{2, 3, 4\}$ is in Y , and hyperedge $\{3, 5, 7\}$ is in Z .

The intuition for proving the new lower bound on L^* is as follows: When Algorithm R_EMBEDDING terminates with a small k , the number of c-paths in clockwise embedding that use link g_k (and h_k) is close to L . A hyperedge whose c-path in clockwise embedding uses link g_k (resp. h_k) belongs to one of the sets W, X , or Y (resp. W, X or Z). A key observation is that a c-path in any embedding for a hyperedge in W must contain at least two of the three links $n - 1$, g_k , and h_k . This implies that the lower bound on L^* is at least $2|W|/3$. Also, a hyperedge in X is separated by links $n - 1$ and g_k (or h_k). Similarly, a hyperedge in Y (resp. Z) is separated by links g_k and h_k (or $n - 1$) (resp. by links h_k and $n - 1$ (or g_k)). Based on the above observations, a better lower bound than $\lceil L/2 \rceil$ on L^* can be obtained for small k . Especially we have the following result.

Lemma 2. *For any links g and h with $0 \leq g < h < n - 1$,*

$$L^* \geq \frac{2}{3}|W| + \frac{1}{3}(|X| + |Y| + |Z|).$$

Proof: Let I denote an arbitrary embedding, and $l_I(i)$ denote congestion on link i in embedding I . Then

$$L^* \geq \min_I \{\max\{l_I(g), l_I(h), l_I(n - 1)\}\}.$$

Let $T = l_I(g) + l_I(h) + l_I(n - 1)$. Notice that in any embedding I , a c-path for any hyperedge in W must contain at least two of the links g , h , and $n - 1$. Therefore, each hyperedge in W contributes at least 2 to T in any embedding I . Similarly, a c-path for any hyperedge in X , Y , or Z must contain at least one of the links g , h , and $n - 1$ in any embedding I , so each hyperedge in X , Y , or Z contributes at least 1 to T .

Based on above analysis, it is concluded that in any embedding I ,

$$T \geq 2|W| + |X| + |Y| + |Z|.$$

Then in any embedding I ,

$$\max\{l_I(g), l_I(h), l_I(n - 1)\} \geq \frac{2}{3}|W| + \frac{1}{3}(|X| + |Y| + |Z|).$$

So

$$L^* \geq \frac{2}{3}|W| + \frac{1}{3}(|X| + |Y| + |Z|).$$

□

Notice that there are hypergraphs with L^* tight to the lower bound in Lemma 2. Please refer to [11] for an example of such hypergraphs.

Now, we use the lower bound given in Lemma 2 to derive the approximation ratio for our algorithm for small k .

Theorem 1. *The approximation ratio of Algorithm R_EMBEDDING is bounded above by 1.8.*

Proof: Assume that Algorithm R_EMBEDDING terminates with the maximum congestion L_k . From Lemma 2,

$$L^* \geq \frac{2}{3}|W| + \frac{1}{3}(|X| + |Y| + |Z|)$$

for any links g and h with $0 \leq g < h < n - 1$. Since $|Y| = l(g) - |W| - |X|$ and $|Z| = l(h) - |W| - |X|$, we have

$$L^* \geq \frac{2}{3}|W| + \frac{1}{3}(|X| + |Y| + |Z|) = \frac{1}{3}(l(g) + l(h) - |X|). \quad (1)$$

Notice that for links g_k and h_k , if $g_k = h_k$ then g_k is the unique link with the maximum congestion L . From the definition of x_k , we have $x_k = L$ for $g_k = h_k$. The rest of the proof is divided into two cases.

Case 1: $L_k = L - k - 1$.

Taking $g = g_{k+1}$ and $h = h_{k+1}$ in inequality (1), we have $|X| = x_{k+1}$. Since the algorithm terminates with maximum congestion L_k , we have $x_{k+1} \leq k$. This implies $g_{k+1} < h_{k+1}$. Since $l(g_{k+1}) \geq L - 2(k+1) + 1$ and $l(h_{k+1}) \geq L - 2(k+1) + 1$,

$$\begin{aligned} L^* &\geq \frac{1}{3}\{l(g_{k+1}) + l(h_{k+1}) - x_{k+1}\} \\ &\geq \frac{1}{3}\{2(L - 2(k + 1) + 1) - k\} = \frac{1}{3}(2L - 5k - 2). \end{aligned}$$

From this, an upper bound on the approximation ratio of the algorithm is

$$\frac{L_k}{L^*} \leq \frac{3(L - k - 1)}{2L - 5k - 2}.$$

Since $\lceil L/2 \rceil$ is also a lower bound on L^* ,

$$\frac{L_k}{L^*} \leq \frac{L - k - 1}{\lceil L/2 \rceil}.$$

Therefore, the approximation ratio of the algorithm is bounded by

$$\min\left\{\frac{L - k - 1}{\lceil L/2 \rceil}, \frac{3(L - k - 1)}{2L - 5k - 2}\right\}.$$

Function $(L - k - 1)/\lceil L/2 \rceil$ is decreasing in k and function $3(L - k - 1)/(2L - 5k - 2)$ is increasing in k . For $k \geq \lfloor L/10 \rfloor$, $(L - k - 1)/\lceil L/2 \rceil \leq 1.8$, and for $k \leq \lfloor L/10 \rfloor - 1$, $3(L - k - 1)/(2L - 5k - 2) \leq 1.8$.

Case 2: $L_k = L - k$.

In this case, either $x_k = k$ or $x_{k+1} = 0$. Assume that $x_k = k$. Taking $g = g_k$ and $h = h_k$ in inequality (1), then $|X| = x_k = k$ and $g_k < h_k$. Since $l(g_k) \geq L - 2k + 1$ and $l(h_k) \geq L - 2k + 1$,

$$L^* \geq \frac{1}{3}\{l(g_k) + l(h_k) - x_k\} \geq \frac{1}{3}\{2(L - 2k + 1) - k\} = \frac{1}{3}(2L - 5k + 2).$$

From this, $L^* \geq \lceil L/2 \rceil$, and $L_k = L - k$, the approximation ratio of the algorithm is bounded by

$$\min\left\{\frac{L-k}{\lceil L/2 \rceil}, \frac{3(L-k)}{2L-5k+2}\right\}.$$

For $k \geq \lceil L/10 \rceil$, $(L-k)/\lceil L/2 \rceil \leq 1.8$, and for $k \leq \lceil L/10 \rceil - 1$, $3(L-k)/(2L-5k+2) \leq 1.8$.

Assume that $x_{k+1} = 0$. Then

$$L^* \geq \frac{1}{3}\{l(g_{k+1}) + l(h_{k+1}) - x_{k+1}\} \geq \frac{1}{3}\{2(L-2(k+1)+1)\} = \frac{1}{3}(2L-4k-2).$$

The approximation ratio of the algorithm is bounded by

$$\min\left\{\frac{L-k}{\lceil L/2 \rceil}, \frac{3(L-k)}{2L-4k-2}\right\}.$$

For $k \geq \lceil L/10 \rceil$, $(L-k)/\lceil L/2 \rceil \leq 1.8$, and for $k \leq \lceil L/10 \rceil - 1$, $3(L-k)/(2L-4k-2) \leq 1.8$, except for the following cases:

- $\langle 1 \rangle L = 2$ and $k = 0$,
- $\langle 2 \rangle L = 4$ and $k = 0$, and
- $\langle 3 \rangle L = 12$ and $k = 1$.

It is not difficult to show that $L_k/L^* \leq 1.8$ for $\langle 1 \rangle$, $\langle 2 \rangle$, and $\langle 3 \rangle$. Due to the space limit, we omit the details of the proof, which can be found in [11]. \square

Step 1 of Algorithm R.Embedding takes $O(mn)$ time for the hypergraph with m hyperedges and n nodes. Since $L = O(m)$, Steps 2 and 4 can be done in $O(mn)$ time and Step 3 can be done in $O(m)$ time. Subroutine Special_Cases takes $O(n)$ time. So, the time complexity of Algorithm R.Embedding is $O(mn)$ that is optimal since the total number of links in the c-paths for any embedding for the hypergraph can be $\Omega(mn)$.

4 Concluding Remarks

A 1.8-approximation algorithm is given for the MCHEC problem. This improves the previous 2-approximation results. The improvement is based on the following observations: If some hyperedges can be re-embedded such that the maximum congestion of the clockwise embedding can be reduced then we have a better embedding. Otherwise, a better lower bound on L^* can be obtained to guarantee the approximation ratio. Whether the 1.8-approximation ratio can be improved further is open. A possible approach is to find a better lower bound on L^* by looking at four links of the cycle (the lower bound of this paper is obtained by considering three links). Finding better approximation algorithms for specific classes of hypergraphs may be worth further investigation as well.

Acknowledgment. We thank the anonymous reviewers for their constructive comments. This work was partially supported by the NSERC research grant (RGPIN250304), President Research Grant and Endowed Research Fellowship of Simon Fraser University.

References

1. T. Carpenter, S. Cosares, J.L. Ganley, and I. Saniee. A simple approximation algorithm for two problems in circuit design. *IEEE Trans. on Computers*, 47(11):1310–1312, 1998.
2. A. Frank, T. Nishizeki, N. Saito, H. Suzuki, and E. Tardos. Algorithms for routing around a rectangle. *Discrete Applied Mathematics*, 40:363–378, 1992.
3. J.L. Ganley and J.P. Cohoon. Minimum-congestion hypergraph embedding on a cycle. *IEEE Trans. on Computers*, 46(5):600–602, 1997.
4. T. Gonzalez. Improved approximation algorithms for embedding hyperedges in a cycle. *Information Processing Letters*, 67:267–271, 1998.
5. T. Gonzalez and S.L. Lee. A 1.6 approximation algorithm for routing multiterminal nets. *SIAM J. on Computing*, 16:669–704, 1987.
6. T. Gonzalez and S.L. Lee. A linear time algorithm for optimal routing around a rectangle. *Journal of ACM*, 35(4):810–832, 1988.
7. A.S. LaPaugh. A polynomial time algorithm for optimal routing around a rectangle. In *Proc. of the 21st Symposium on Foundations of Computer Science (FOCS80)*, pages 282–293, 1980.
8. S.L. Lee and H.J. Ho. Algorithms and complexity for weighted hypergraph embedding in a cycle. In *Proc. of the 1st International Symposium on Cyber World (CW2002)*, pages 70–75, 2002.
9. M. Sarrafzadeh and F.P. Preparata. A bottom-up layout technique based on two-rectangle routing. *Integration: The VLSI Journal*, 5:231–246, 1987.
10. H. Okamura and P.D. Seymour. Multicommodity flows in planar graphs. *Journal of Combinatorial Theory, Series B*, 31:75–81, 1981.
11. Q.P. Gu and Y. Wang. Efficient algorithm for embedding hypergraphs in a cycle. *Technical Report 2003-03, School of Computing Science, Simon Fraser University*, 2003.

Mapping Hypercube Computations onto Partitioned Optical Passive Star Networks

Alessandro Mei and Romeo Rizzi

Department of Computer Science, University of Rome “La Sapienza”
Department of Information and Communication Technology, University of Trento

Abstract. This paper shows that an $n = 2^k$ processor Partitioned Optical Passive Stars (POPS) network with g groups and d processors per group can simulate either a mono-directional move of an n processor hypercube or a bi-directional move of an $n/2$ processor hypercube using one slot when $d = 1$ and $\lceil d/g \rceil$ slots when $d > 1$. Moreover, as a direct application of the simulation, it is shown how a POPS(d, g) network, $n = dg$ and $d \leq g$, can compute the prefix sums of n data values in $\log_2 n + O(1)$ slots, faster than the best previously known *ad-hoc* algorithm for this problem.

1 Introduction

The Partitioned Optical Passive Star (POPS) network [2,5,4,8] is a SIMD interconnection network that uses a fast optical network composed of multiple Optical Passive Star (OPS) couplers. A $d \times d$ OPS coupler is an all-optical passive device which is capable of receiving an optical signal from one of its d sources and broadcast it to all of its d destinations. Being a passive all-optical technology it benefits from a number of characteristics such as no opto-electronic conversion, high noise immunity, and low latency. The number of processors of the network is denoted by n , and each processor has a distinct index in $\{0, \dots, n - 1\}$. The n processors are partitioned into $g = n/d$ groups in such a way that processor i belongs to group $group(i) := \lfloor i/d \rfloor$. It is assumed that d divides n , consequently, each group consists of d processors. For each pair of groups $a, b \in \{0, \dots, g - 1\}$, a coupler $c(b, a)$ is introduced which has all the processors of group a as sources and all the processors of group b as destinations. The number of couplers used is g^2 . Such an architecture will be denoted by POPS(d, g).

The POPS network model has been used to develop a number of non trivial algorithms. Several common communication patterns are realized in [4]. Simulation algorithms for the mesh and hypercube interconnection networks can be found in [11]. Algorithms for data sum, prefix sum, consecutive sum, adjacent sum, and several data movement operations are also described in [11] and [3]. An algorithm for matrix multiplication is provided in [10]. An optimal solution to the off-line permutation routing problem can be found in [7]. Moreover, [1] shows that POPS networks can be modeled by directed and complete stack graphs with loops, and uses this formalization to obtain optimal embeddings of rings and de Bruijn graphs into POPS networks.

The hypercube is one of the most flexible and powerful parallel architectures [6]. Hypercubes have been shown to support an extremely large and rich class of algorithms during the last two decades. Moving this class toward efficient implementation on POPS networks is thus important to demonstrate the flexibility and power of this kind of optical interconnection networks. Sahni, in [11], shows that a bi-directional move of an $n = 2^k$ processor hypercube can be simulated in $2\lceil d/g \rceil$ slots on a $\text{POPS}(d, g)$ network, where $n = dg$. In this paper, we show that a $\text{POPS}(d, g)$ network, $n = dg$, can simulate a mono-directional move of an n processor hypercube and a bi-directional move of an $n/2$ processor hypercube in $\lceil d/g \rceil$ slots. Note that this is far from being trivial. To demonstrate the strength of these simulation results, it is shown that tackling the prefix sums problem on the POPS network by simply applying our simulation to the well-known hypercube algorithm [9] significantly improves on the literature. This far, the best *ad-hoc* algorithm for computing the prefix sums of n data values was presented by Sahni in [11]. Sahni, after showing that his hypercube simulation yields a $2 \log n$ slots prefix sums algorithm on a $\text{POPS}(\sqrt{n}, \sqrt{n})$ network, directly attacks the problem with an *ad-hoc* solution improving the time complexity to $(3/2) \log n + O(1)$ slots on the same network. By using our novel hypercube simulation, the prefix sums problem can be solved in $\log n + O(1)$ slots on the same $\text{POPS}(\sqrt{n}, \sqrt{n})$ network, which improves even on the best *ad-hoc* algorithm and we can show to be optimal.

2 The Hypercube and the POPS Network

Let $\mathbb{N}_n \triangleq \{0, \dots, n - 1\}$ denote the set of the first n natural numbers. Assuming $n = 2^k$ is a power of two, $x \in \mathbb{N}_n$, and $v \in \mathbb{N}_k$, let $\text{bit}_v(x)$ be the v -th bit of the binary representation of x . Moreover, let $\oplus_v(x)$ be the element of \mathbb{N}_n whose binary representation differs from the binary representation of x in the v -th bit only. Finally, with \bar{x} we will refer to the element of \mathbb{N}_n such that each bit in the binary representation of \bar{x} is the complement of the corresponding bit in the binary representation of x .

A k -dimensional hypercube is composed of $n = 2^k$ synchronous processors, each with a distinct index in \mathbb{N}_n . During a so called *mono-directional move* along dimension v and direction b , where $v \in \mathbb{N}_k$ and $b \in \{0, 1\}$, each processor p such that $\text{bit}_v(p) = 1 - b$ sends a packet to processor $\oplus_v(p)$. All packets are sent in parallel, and such a move is thus performed in one computational step. Two mono-directional moves along dimension v and directions 0 and 1, performed in parallel, compose a bi-directional move along dimension v . In a hypercube, each step of the computation consists of the following two sub-steps: first, each processor does some local computations, second, the hypercube performs either a mono-directional or a bi-directional move. Given a hypercube H , a set of r distinct dimensions v_1, \dots, v_r and r bits b_1, \dots, b_r define a *sub-hypercube* S composed of all the processors p of H such that $\text{bit}_{v_i}(p) = b_i$, for $i = 1, \dots, r$. Clearly, S consists of $n/2^r$ processors.

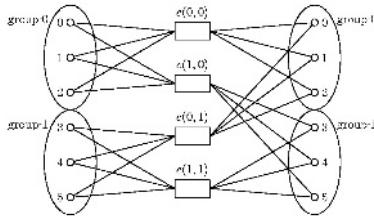


Fig. 1. A POPS(3, 2).

A *POPS* network of n synchronous processors is shown in Figure 1. For all $p \in \mathbb{N}_n$, processor p has g transmitters which are connected to couplers $c(t, \text{group}(p))$, for all $t \in \mathbb{N}_g$. Similarly, processor p has g receivers connected to couplers $c(\text{group}(p), s)$, for all $s \in \mathbb{N}_g$. During a step of the computation, each processor in parallel: (1) Performs some local computations; (2) sends a packet to a subset of its transmitters; (3) receives a packet from one of its receivers. In order to avoid conflicts, there shouldn't be any pair of processors sending a packet to the same coupler. The time needed to perform such a step is referred to as a *slot*.

3 Preliminaries

One of the advantages of a $\text{POPS}(d, g)$ network is that its diameter is 1. A packet can be sent from processor p to processor q , $p \neq q$, in one slot by using coupler $c(\text{group}(q), \text{group}(p))$. Under proper hypothesis, multiple packets can also be delivered to destination in a single slot. We will say that m packets, each with a different destination, are arranged according to a *fair distribution* in a $\text{POPS}(d, g)$ network if no two packets are stored in the same processor, and no two packets with the same destination group are stored in the same group. In this case, we will also say that the packets are *fairly distributed*. It is straightforward to see that a fairly distributed set of packets is routable in one slot. Indeed, no conflict occurs on any coupler.

Fact 1 *In a $\text{POPS}(d, g)$ network, a fairly distributed set of m packets can be routed to destination in one slot.*

In a $\text{POPS}(d, g)$ network, the permutation routing problem consists of a set of n packets m_0, \dots, m_{n-1} . Packet m_p is stored in the local memory of processor p , for all $p \in \mathbb{N}_n$, and has a desired destination $\pi(p)$. The problem is to route the packets to their destinations in as few slots as possible. Of course, the n packets are almost never fairly distributed. However, when π is known in advance, this problem has been optimally solved in [7].

Theorem 1 (Theorem 2 of [7]). *A $\text{POPS}(d, g)$ network can route any permutation π among the $n = dg$ processors using one slot when $d = 1$ and $2\lceil d/g \rceil$ slots when $d > 1$.*

4 Mapping a Hypercube onto a POPS Network

4.1 A Hypercube Coloring

Definition 1. A g -coloring of an $n = 2^k$ processor hypercube is a mapping $X : \mathbb{N}_{2^k} \mapsto \mathbb{N}_g$. The lower half of X is the g -coloring X' of an $n/2 = 2^{k-1}$ processor hypercube such that $X'(p) = X(p)$, $p \in \mathbb{N}_{2^{k-1}}$, while the upper half of X is the g -coloring X'' such that $X''(p) = X(p + 2^{k-1})$, $p \in \mathbb{N}_{2^{k-1}}$.

We will uniquely represent a g -coloring X of an $n = 2^k$ processor hypercube with a matrix over \mathbb{N}_g . When $k = 0$, X is represented by the following 1×1 matrix:

$$[X(0)].$$

When k is odd, X is represented by a $2^{(k-1)/2} \times 2^{(k+1)/2}$ matrix:

$$[X' \ X''],$$

where X' and X'' are the $2^{(k-1)/2} \times 2^{(k-1)/2}$ matrices representing the lower and upper halves of X , respectively. Finally, when k is even, $k \geq 2$, X is represented by a $2^{k/2} \times 2^{k/2}$ matrix:

$$\begin{bmatrix} X'' \\ X' \end{bmatrix},$$

where X' and X'' are the $2^{k/2-1} \times 2^{k/2}$ matrices representing the lower and upper halves of X , respectively. In the following, we will often identify a coloring X with its matrix, and vice-versa, as if they were the same mathematical object.

Definition 2. A g -coloring X of an n processor hypercube is balanced if $|X^{-1}(c)| = n/g$ for all $c \in \mathbb{N}_g$.

Definition 3. Let X be a coloring of a $n = 2^k$ processor hypercube, $v \in \mathbb{N}_k$, and $p, q \in \mathbb{N}_n$, $p \neq q$. We say that p and q generate a conflict in dimension v if $X(p) = X(q)$ and $X(\oplus_v(p)) = X(\oplus_v(q))$.

Definition 4. A coloring X of an $n = 2^k$ processor hypercube is conflict-free in a single direction if for all $v \in \mathbb{N}_k$, and for all $p, q \in \mathbb{N}_n$ with $p \neq q$ and $\text{bit}_v(p) = \text{bit}_v(q)$, p and q do not generate a conflict in dimension v .

In order to check whether a coloring X is conflict-free in a single direction, we can restrict to check, for all v , all p and q such that $\text{bit}_v(p) = \text{bit}_v(q) = 0$. Indeed, if there exist p and q generating a conflict in dimension v and such that $\text{bit}_v(p) = \text{bit}_v(q) = 1$, then $\oplus_v(p)$ and $\oplus_v(q)$ generate a conflict in dimension v and are such that $\text{bit}_v(\oplus_v(p)) = \text{bit}_v(\oplus_v(q)) = 0$.

Definition 5. A coloring X of an $n = 2^k$ processor hypercube is conflict-free in both directions if for all $v \in \mathbb{N}_k$, and for all $p, q \in \mathbb{N}_n$ with $p \neq q$, p and q do not generate a conflict in dimension v .

In this section, we are interested in finding balanced g -colorings for 2^k processor hypercubes with conflict-free properties. In particular, how can we construct a conflict-free in a single direction (both directions) balanced g -coloring for a 2^k processor hypercube with minimal number of colors g ? Our motivation is that balanced and conflict-free g -colorings will be used as tools to efficiently map hypercube computations on the POPS(d, g) network.

We start by restricting our interest to colorings with a structure. Later, it will be clearer that such a structure implies a number of useful properties.

Definition 6. *Given a coloring X of a 2^k processor hypercube, the reversal of X is the coloring $[X]^R$ such that $[X]^R(p) = X(\bar{p})$ for all $p \in \mathbb{N}_{2^k}$. Note that the matrix representation of $[X]^R$ is obtained by flipping the matrix representing X both horizontally and vertically.*

Definition 7. *A coloring X is symmetric if $X = [X]^R$.*

The notion of symmetry for g -colorings will be useful in the following. Indeed, to obtain a g -coloring of a 2^k processor hypercube which is conflict-free in both directions, simply consider the lower half of a g -coloring of a 2^{k+1} processor hypercube conflict-free in a single direction. This fact, in a conveniently more general form, is stated in Lemma 1 here below. The generalization comes with the following observation.

Observation 1 *The following properties of a coloring are invariant under shuffling the dimensions of the hypercube: being balanced, symmetric, conflict-free (both versions).*

Lemma 1. *Let X be a symmetric coloring of an $n = 2^k$ processor hypercube H . Let Y be the restriction of X to an $n/2$ processor sub-hypercube $H' \subset H$. If X is conflict-free in a single direction, then Y is conflict-free in both directions. Moreover, X is balanced iff Y is balanced.*

Our goal is to define a class A_{2h} , $h \in \mathbb{N}^+$, of symmetric balanced 2^h -colorings of 2^{2h} processor hypercubes which are conflict free in a single direction. By Lemma 1, we will extract from such a class another class B_{2h-1} , $h \in \mathbb{N}^+$, of balanced 2^h -colorings of 2^{2h-1} processor hypercubes which are conflict free in both directions. We start from defining a few building blocks we will be using in the following, and then present these classes by means of a recursive construction.

Definition 8. *Given a g -coloring B of an n processor hypercube, we define the $2g$ -colorings B^E , B^O , and $B + g$ of an n processor hypercube such that:*

$$B^E(p) = \begin{cases} B(p) + g & \text{if } \text{parity}(p) = 0; \\ B(p) & \text{otherwise;} \end{cases} \quad (1)$$

$$B^O(p) = \begin{cases} B(p) + g & \text{if } \text{parity}(p) = 1; \\ B(p) & \text{otherwise;} \end{cases} \quad (2)$$

$$(B + g)(p) = B(p) + g; \quad (3)$$

where $\text{parity}(p)$ denotes the parity of the number of ones in the binary representation of p .

Lemma 2. *If a g -coloring B is conflict-free in a single direction (in both directions), then $[B]^R$, B^E , B^O , and $B + g$ are conflict-free in a single direction (in both directions).*

As the base of our construction, we start from A_2 .

$$A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad (4)$$

which is a symmetric balanced 2-coloring. Inspired by Lemma 1, from A_2 we can extract B_1 , by using the following general construction which allows to build the class of colorings B_{2h-1} from the class of colorings A_{2h} .

Given a symmetric balanced 2^h -coloring $A_{2h} : \mathbb{N}_{2^{2h}} \mapsto \mathbb{N}_{2^h}$, let $B_{2h-1} : \mathbb{N}_{2^{2h-1}} \mapsto \mathbb{N}_{2^h}$ be the lower half of A . The matrix representing B_{2h-1} can thus be built from the matrix representing A_{2h} by horizontally cutting A_{2h} in two halves of equal size and taking the lowest one. Since A_{2h} is symmetric, the upper half is equal to $[B_{2h-1}]^R$, that is,

$$A_{2h} = \begin{bmatrix} [B_{2h-1}]^R \\ B_{2h-1} \end{bmatrix}. \quad (5)$$

And both B_{2h-1} and $[B_{2h-1}]^R$ are balanced. Following the above general construction, B_1 is represented by a 1×2 matrix:

$$B_1 = [0 \ 1]. \quad (6)$$

Starting from B_{2h-1} , we propose to build A_{2h+2} in the following way:

$$A_{2h+2} = \begin{bmatrix} B_{2h-1}^O & [B_{2h-1}]^R \\ [B_{2h-1}^E]^R & B_{2h-1} + 2^h \\ [B_{2h-1}]^R + 2^h & B_{2h-1}^E \\ B_{2h-1} & [B_{2h-1}^O]^R \end{bmatrix}. \quad (7)$$

The block structure displayed in Equation 7 implies that A_{2h+2} is symmetric irrespectively of the actual values of the entries in the building block B_{2h-1} . Hence, A_{2h} is a symmetric 2^h -coloring of a 2^{2h} hypercube for all $h \in \mathbb{N}^+$, where \mathbb{N}^+ denotes the positive naturals.

Next theorem shows that a central property of the class of colorings A_{2h} , $h \in \mathbb{N}^+$, as defined above. Before coming to the main theorem, the following lemma introduces a preliminary result on colorings A_{2h} , $h \in \mathbb{N}^+$.

Lemma 3. *For all $h \in \mathbb{N}^+$, A_{2h} assigns even colors to even entries and odd colors to odd entries.*

Theorem 2. For all $h \in \mathbb{N}^+$, A_{2h} is symmetric, balanced, and conflict-free in a single direction.

Finally, it is also proved that B_{2h-1} , $h \in \mathbb{N}^+$, is a class of balanced and conflict-free in both directions 2^h -colorings of 2^{2h-1} processor hypercubes.

Theorem 3. For all $h \in \mathbb{N}^+$, B_{2h-1} is balanced and conflict-free in both directions.

4.2 Hypercube Simulation

We are interested in simulating an $n = 2^k$ processor hypercube on a POPS(d, g) network of the same number of processors. The idea is to map each processor of the hypercube into each processor of the POPS(d, g) network by a 1:1 mapping $\mu : \mathbb{N}_n \mapsto \mathbb{N}_n$. The key point is to choose a mapping μ such that the communication patterns required to simulate the hypercube moves can be efficiently performed on the POPS(d, g) network.

Previously introduced g -colorings are important tools to find “good” mappings. Given a g -coloring A of an $n = 2^k$ processor hypercube H , let μ_A be a $1 : 1$ mapping $\mu : \mathbb{N}_n \mapsto \mathbb{N}_n$ from the processor indexes in H to the processor indexes in a POPS(d, g) network P . We ask μ_A to assign indexes in such a way that if processor p in the hypercube is colored with color $c = A(p)$, then processor $\mu(p)$ belongs to the c -th group of P . Since A is balanced, there are many straightforward ways to build a mapping μ_A with such a property. However, for our purposes, all such mappings are equivalent, and we can think of μ_A as unique. We will say that H is mapped into P according to A meaning that mapping μ_A is used.

Conflict-less properties of g -colorings in Definitions 4 and 5 translate into conflict-less communication patterns on the POPS(d, g) network when simulating hypercube moves. This result is formally described and proved in the following lemma.

Lemma 4. Given a g -coloring A of an $n = 2^k$ processor hypercube H , balanced and conflict-free in a single direction (both directions), then a POPS(d, g) network, $n = dg$, can simulate every mono-directional (bi-directional) move of H in one slot.

Proof. Assume that H is mapped into the POPS(d, g) network according to A , and that a mono-directional move along dimension v on the hypercube, $v = 0, \dots, k-1$, is to be simulated on the POPS(d, g) network. Say, without loss of generality, that processor $\mu(p)$ has to send a message to processor $\mu(\oplus_v(p))$, for all p such that $\text{bit}_v(p) = 0$. Our claim is that no conflict occurs on any passive star of the POPS(d, g) network, and thus that the above communication can be completed in one slot. Indeed, take two distinct processors $\mu(p)$ and $\mu(q)$ such that $\text{bit}_v(p) = \text{bit}_v(q) = 0$. Processor $\mu(p)$ has a message for processor $\mu(\oplus_v(p))$, while processor $\mu(q)$ has a message for processor $\mu(\oplus_v(q))$. These communications on the POPS(d, g) network generate a conflict on a passive star if and only

if $\text{group}(\mu(p)) = \text{group}(\mu(q))$ and $\text{group}(\mu(\oplus_v(p))) = \text{group}(\mu(\oplus_v(q)))$. Stated equivalently, this happens if and only if $A(p) = A(q)$ and $A(\oplus_v(p)) = A(\oplus_v(q))$, that is, if and only if p and q generate a conflict on dimension v . However, this is impossible since A is conflict-free in a single direction. Consequently, no conflict occurs and the mono-directional move can be simulated in one slot.

The case when A is conflict-free in both directions can be dealt with in a similar way.

Theorem 4. *An $n = 2^k$ processor POPS(d, g) network such that $d < g$ can simulate every bidirectional move of an n processor hypercube using one slot.*

The above theorem is essentially tight, meaning that even a slight reduction in the number of passive stars of the network leads to the following impossibility result.

Proposition 1. *At least two slots are required to simulate any bidirectional move of an n processor hypercube on an $n = 2^k$ processor POPS(d, g) network such that $d = g = \sqrt{n}$.*

Finally, we can prove our main simulation theorems.

Theorem 5. *An $n = 2^k$ processor POPS(d, g) network can simulate every mono-directional move of an n processor hypercube using $\lceil d/g \rceil$ slots.*

Theorem 6. *An $n = 2^k$ processor POPS(d, g) can simulate every bidirectional move of an $n/2$ processor hypercube using $\lceil d/g \rceil$ slots.*

5 Data Sum and Prefix Sums

Assume each processor of a POPS(d, g) network, $dg = n = 2^k$, stores a data value. The *data sum* problem consists in summing up all the n data values and leaving the sum in processor 0.

In the literature on POPS networks, this problem is first attacked by Gravenstreter and Melhem [4] as a particular instance of the more general *global reduction* operation. The algorithm is optimal as using $\log_2 n$ slots, but works only when $d \leq \sqrt{2n}$. Later, in [11], Sahni develops a hypercube simulation algorithm for the POPS network such that every bi-directional move of an n processor hypercube can be simulated using $2\lceil d/g \rceil$ slots on the POPS(d, g) network, $dg = n$. By means of this simulation result, a POPS(d, g) network can simulate the well-known $\log_2 n$ mono-directional move hypercube algorithm for computing the data sum [6,9], which directly yields a data sum algorithm for the POPS(d, g) network using $2\lceil d/g \rceil \log_2 n$ slots. Motivated by the non-optimality of such a simulation approach, Sahni designs an *ad-hoc* algorithm for the same problem using only $\lceil d/g \rceil \log_2 n$ slots [11].

It is interesting to note that our hypercube simulation immediately yields exactly the same optimal result as the one obtained by Sahni's custom algorithm. Indeed, the $\log_2 n$ mono-directional move hypercube algorithm to compute the data sum can be simulated by using our Theorem 5, and the following corollary directly follows.

Corollary 1 ([11]). *An $n = 2^k$ processor POPS(d, g) network can sum n data values using $\lceil d/g \rceil \log_2 n$ slots.*

Recently, Datta and Soundaralakshmi [3] showed that, when $d > g$, hypercube simulation is not an efficient way to solve this problem. Indeed, [3] provides an improved $\lceil d/g \rceil + 2 \log_2 g - 1$ slots algorithm to compute the data sum in POPS(d, g) networks with large group size d .

Another fundamental problem is computing the *prefix sums*. Given n data values x_0, \dots, x_{n-1} , the problem consists in computing the prefix sums $s_i = \sum_{j=0}^i x_j$, $i = 0, \dots, n-1$. Sahni [11] assumes that x_i is input (and s_i is output) at processor i , and describes how a POPS(d, g) network, $n = dg$, can perform this task in $3 + \log_2 n + \log_2 d$ slots when $1 < d \leq g$ and $2d/g(1 + \log_2 g) + \log_2 d + 1$ slots when $d > g$, by using a custom algorithm. The design of a custom algorithm allows to improve on the direct application of the simulation algorithm in [11] to the well-known $\log_2 n$ bi-directional move hypercube algorithm, which yields a prefix sums algorithm running in $2\lceil d/g \rceil \log_2 n$ slots. By using our improved simulation results, it is possible to show how a POPS network can perform the same task in $\lceil d/g \rceil(1 + \log_2 n)$ slots, significantly improving even on the best *ad-hoc* result.

Theorem 7. *Given $n = 2^k$ data values stored one per processor in a POPS(d, g) network, $n = dg$, their prefix sums can be computed using $\log_2 n$ slots when $d < g$, and $\lceil d/g \rceil(1 + \log_2 n)$ slots when $d \geq g$.*

The above theorem assumes that the data values are input to the POPS(d, g) network according to a particular mapping $\mu_{C_a^b}$. In some cases, for example when the prefix sum operation is used as a sub-routine of a more complex algorithm, this cannot be assumed. Thanks to Theorem 1, it is easy to show that, by paying $2\lceil d/g \rceil$ extra slots, the same result extends to any 1:1 input mapping.

Corollary 2. *Given $n = 2^k$ data values stored one per processor according to any mapping in a POPS(d, g) network, $n = dg$, their prefix sums can be computed using $\lceil d/g \rceil(3 + \log_2 n)$ slots.*

Note that the recently proposed prefix sums algorithm for POPS(d, g) with large group size [3], running in $2\lceil d/g \rceil + 4 \log g + 6$ slots, uses Sahni's prefix sums algorithm as a subroutine. Consequently, by using our results, also the running time of this algorithm is improved for all sizes d and g .

Prefix sum sub-routines often appear in many algorithms for solving more complex problems. An example is the *concentrate operation*. In a *concentrate operation* we are given a subset $D \subseteq \mathbb{N}_n$ of *selected* processors in a POPS(d, g) network, $n = dg$, each containing a data value. The problem is to concentrate all data values in the first $|D|$ processors of the network, without changing their order in the network. Formally, each data value x , stored in processor $p \in D$, is to be routed to processor p' , where $p' = |\{q \in D : q < p\}|$. In [11], it is shown how this problem can be solved in $5 + 3 \log g$ slots in a POPS(g, g) network, the computational bottleneck being a prefix sums operation. Thanks to our Theorem 2, the above complexity can be reduced to $5 + 2 \log g$ slots, and can be similarly reduced for all values of d and g .

6 Conclusion

In this paper we propose a novel simulation algorithm for hypercubes onto the POPS network. This simulation improves over the literature and is shown to be essentially optimal for $d \leq g$. Moreover, our results imply a novel algorithm for solving the fundamental prefix sums problem on the POPS, improving even on the best *ad-hoc* algorithm known so far. This proves the strength of the simulation we propose, which can be used to move the extremely rich and complex class of algorithms for the hypercube toward efficient implementation on the partitioned optical passive star network.

References

1. P. Berthomé and A. Ferreira. Improved embeddings in pops networks through stack-graph models. In *Proceedings of the Third International Workshop on Massively Parallel Processing Using Optical Interconnections*, 1996.
2. D. Chiarulli, S. Levitan, R. G. Melhem, J. Teza, and G. Gravenstreter. Multi-processor interconnection networks using partitioned optical passive star (pops) topologies and distributed control. In *Proceedings First International Workshop on Massively Parallel Processing Using Optical Interconnections*, 1994.
3. A. Datta and S. Soundaralakshmi. Basic operations on a partitioned optical passive stars network with large group size. In *Proceedings of ICCS*, volume 2329 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.
4. G. Gravenstreter and R. G. Melhem. Realizing common communication patterns in partitioned optical passive stars networks. *IEEE Transactions on Computers*, 47(9), September 1998.
5. G. Gravenstreter, R. G. Melhem, D. Chiarulli, S. Levitan, and J. Teza. The partitioned optical passive star (pops) topology. In *Proceedings Ninth International Parallel Processing Symposium*, 1995.
6. F. T. Leighton. *Introduction to parallel algorithms and architectures: arrays, trees, hypercubes*. Morgan Kaufmann, San Mateo, CA, 1992.
7. A. Mei and R. Rizzi. Routing permutations in partitioned optical passive stars networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002)*, Apr. 2002.
8. R. G. Melhem, G. Gravenstreter, D. Chiarulli, and S. Levitan. *The Communication Capabilities of Partitioned Optical Passive Star Networks*, pages 77–98. Kluwer Academic Publishers, 1998.
9. S. Ranka and S. Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition*. Springer Verlag, New York, NY, 1990.
10. S. Sahni. Matrix multiplication and data routing using a partitioned optical passive stars network. *IEEE Transactions on Parallel and Distributed Systems*, 11(7), July 2000.
11. S. Sahni. The partitioned optical passive stars network: Simulations and fundamental operations. *IEEE Transactions on Parallel and Distributed Systems*, 11(7), July 2000.

The High Performance Microprocessor in the Year 2013: What Will It Look Like? What It Won't Look Like?

Yale Patt

University of Texas at Austin

Abstract. Moore's Law promises more than one billion transistors on a chip processing at more than 10 GHz. Is there anything left to do in the microarchitecture to make use of all this capability, or should we fold our tents and go home? I have given talks with this theme for the past fifteen years. The theme remains the same, brought on by the naysayers who proclaim we have reached the end of our rope. But the details continue to change. Progress in process technology requires updating what is available on the chip. And microarchitecture ingenuity provides new avenues to exploit the increase in on-chip resources. Yesterday's news is today's old hat, but there is still plenty we can do. In this talk, I will spend a little time discussing the rational of the naysayers, and then move on to what I think we will see in the microprocessor of the year 2013, including the block-structured ISA, stronger use of SSMT, greater use of microcode, dedicated infrequently used functional units, and most importantly, a stronger coupling with the compiler and algorithm technologies. If time permits, I will discuss some things I do not think we will see on the chip, like qbits and machines that think.

FROOTS – Fault Handling in Up*/Down* Routed Networks with Multiple Roots

Ingebjørg Theiss and Olav Lysne

Simula Research Laboratory, Norway
`{theiss,olav.lysne}@simula.no`

Abstract. Faults in irregular networks have previously been handled through a global or local network reconfiguration, which leaves the network unavailable or disconnected until the reconfiguration is finished. We present FRoots, which leaves the network available and connected by using redundant paths. Only packets residing in the network at the fault time might become unrouteable, and FRoots can be designed to offer an alternative path for such packets. To accommodate sequential, multiple faults, reconfiguration of the obsolete paths is done without a severe downgrade of network throughput. FRoots is based on a load balancing version of Up*/Down* routing, and uses a modest number of virtual channels to achieve redundancy properties. However, FRoots utilizes all channels in the fault-free case, thus experiencing no performance loss compared to its predecessor routing algorithm.

1 Introduction

As an interconnection network grows, the chance of some link or routing node failing increases dramatically. In a large network, fault handling becomes a necessity to prevent global network failure as single faults occur.

By interconnects, we mean high speed networks used to connect e.g. processors in multicomputers, or as in InfiniBand[1], between processors, memory, I/O etc. For a survey of classical interconnection networks we refer to [2].

In recent years, Networks of Workstations (NOW) have been introduced as alternatives to custom-made multicomputers. NOWs allow great wiring flexibility, and irregular structures become important to easily accommodate network changes. AutoNet[3], Myrinet[4], and InfiniBand[1] are examples of cut-through interconnects for irregular NOWs.

The most prominent irregular network routing algorithm in interconnects is the Up*/Down* routing [3], which has inspired several modifications. MRoots [5] uses virtual channels to allow multiple Up*/Down* graphs, of which the roots are spread to reduce bottlenecks. Up*/Down* has also been refined by imposing fewer turn restrictions, e.g. [6]. A few other recent approaches are layered shortest path routing (LASH) [7], and transition ordered shortest path routing [8].

Faults in irregular networks can be handled in several ways, such as having redundant components, globally or locally redundant paths, and global or local reconfiguration. For redundant components, backup hardware handles faults.

Redundant paths rely on having two or more separate paths between all sources and destinations. For global reconfiguration the entire network is reconfigured with a new routing algorithm, while in local reconfiguration, only parts of the network are reconfigured and the routing tables of the remaining parts are reused.

This paper proposes FRoots, a load balancing Up*/Down* routing, offering a fault handling method for irregular networks which combines redundant paths with reconfiguration without a significant loss of performance.

The paper is organized as follows: Section 2 describes some reconfiguration approaches for irregular networks. Section 3 describes some notions and the predecessor routing algorithms to FRoots, which in turn is described in section 4. Simulation results are presented in section 5, before we conclude in section 6.

2 Related Work

In cut-through regular networks, the implicit knowledge of the network structure can be exploited. Several fault-tolerant routing algorithms have been suggested, one example is the Planar-Adaptive routing [9], many others exist.

Examples of fault handling in irregular networks have so far, to the best of our knowledge, used reconfiguration to provide all source/destination pairs with new legal routes. LORE [10] is local, and a pure transient fault tolerance approach, rerouting packets headed through the fault, only expanding the routing tables of a few selected nodes. Others, such as variants of static reconfiguration (e.g. [3]), Partial Progressive Reconfiguration [11], the Single and Double Schemes [12], and Skyline [13] rely on new routing tables to handle faults.

In static reconfiguration, usage of the old and new routing algorithm is separated in time to avoid dependencies between old and new packets. Injection of new packets is stopped until the entire network is emptied and reconfigured. Independent of the routing algorithm used, the network can be emptied by discarding all current packets (Flush), or allow packets to proceed while still routable (Drain).

A dynamic reconfiguration process attempts to keep the network available at all times, we shall focus on the Double Scheme, which uses virtual channels to separate packets in space rather than in time. The Double Scheme alternate on which channel set to drain, denying access to the draining set while access to the other channel set is allowed. The Double Scheme is independent of the routing function, although it needs enough virtual channels to remain deadlock free. In the presence of faults, the Double Scheme cannot guarantee routes between all source/destination pairs during reconfiguration.

FRoots handles faults by having redundant paths, that is, if the network remains physically connected, there is a legal route for every source/destination pair in the presence of a singular fault. Parts of the network resources are drained and then reconfigured in order to prepare for the next network changes, be it a fault or insertion of a new node or link. FRoots combine the low number of unrouteable packets of Drain and the network availability of the Double Scheme.

3 Preliminaries

In the following discussion, we will assume that all links are full duplex links, and that networks are physically connected both before and after changes occur. It will also be convenient to clearly define some notions, and to introduce a few terms, before both Up*/Down* and MRoots are described.

layer - a virtual network, containing an exact representation of the original network, using the same nodes, but instead of links, it has virtual channels.

For our use, a virtual channel exists in exactly one layer.

route - a path in the network, starting at a node, continuing through a connected sequence of alternating links and nodes, ending in a node.

legal route - a route used by a packet routed by the routing function.

bypass traffic - the portion of traffic across a node which neither originates nor ends at the node.

Leaf node - a node having only outgoing “Up” links.

coherence, coherent - a network is coherent if its routing function is connected, that is, there is a legal route from every source to every destination.

3.1 Up*/Down* Routing

Up*/Down* routing [3] is a well known and popular routing algorithm which can be physically adaptive or deterministic, in both cases it can also choose to be virtually adaptive, that is, a packet can change virtual channels as it travels through the network. Up*/Down* can be distributed or source routed, minimal or non-minimal, and there are several ways to calculate the routing tables. The core of Up*/Down* routing is to assign directions to each link; we assume one direction is “Up”, and the opposite direction is “Down”. To avoid deadlock, transitions from “Down” to “Up” links are forbidden, and no cycles containing only “Up” links or only “Down” links are allowed.

To ensure coherence, a root node is chosen, and all nodes must be able to reach the root following only “Up” links. It is common to choose a root, and then find a spanning tree of the network. All links in the spanning tree are assigned their “Up” direction toward the root. The choice of the root can be made completely at random, according to ID, or using a set of heuristics to decide on the “best” root (e.g. [6]). The spanning tree can be found in several ways, e.g. a breadth first search (BFS) or a depth first search (DFS) [6].

When assigning directions to the links not found in the spanning tree, care must be taken to avoid creating cycles of “Up” links or “Down” links. The original method [3] was to assign the “Up” direction toward the node closest to the root, in case of a tie, toward the node with the lower ID. The resulting network is a cycle free directed graph.

3.2 Up*/Down* with Multiple Roots – MRoots

MRoots [5] uses virtual channels and a number of Up*/Down* graphs to reduce bottleneck problems inherent in the original Up*/Down* routing. It is a layered

routing algorithm, and separate Up*/Down* routing tables are assigned to each layer. All forms of Up*/Down* are legal, but to balance load, the Up*/Down* root of a layer is selected to be as far from all other previously selected roots of other layers as possible. MRoots can be physically adaptive if its various Up*/Down* routing tables are physically adaptive. A source must resolve the forwarding layer when inserting a packet, and the packet remains in this layer until the destination drains it.

4 Fault Tolerance and Reconfiguration with FRoots

In case of a network fault, the goals are to keep the network available, coherent, and deadlock free before, during, and after the fault happens. By this we mean that the routing tables must be designed to ensure delivery of packets from any source to any destination, without creating circular waits among any packets, regardless of the packets being routed by the old or new routing tables.

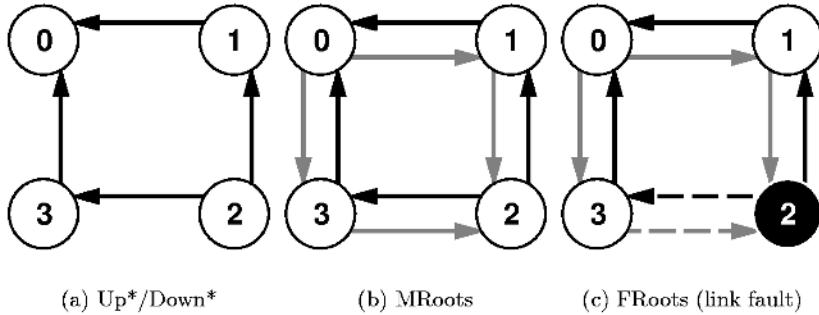
FRoots goes far in delivering these goals. Given enough layers and a physically connected network, the network is available, coherent, and deadlock free at all times. If a fault occurs, only a few packets may disappear or become unrouteable. Obviously, those packets corrupted by the fault must be discarded, and for node faults, also those packets headed for the fault. The only unroutable packets are packets residing in the network at fault time, and packets inserted into the network by a source from the fault occurred and until the source registers the fault. The key idea is to ensure that when a source registers a fault, it can find a layer where all its packets can reach their destinations. How FRoots achieves these properties is presented thoroughly in the next sections.

4.1 One Fault Tolerance

Both links or nodes can fail, the latter will also render links attached to it useless. Even if such a fault doesn't physically disconnect the network, it may still make some destinations unreachable for some sources, and the network is no longer coherent. Consider the nodes and black arrows of Fig. 1a as an Up*/Down* graph where node 0 is the root. There are two paths from every source to every destination, but not two routes: packets from 1 to 3 have to pass through node 0, as do packets from 3 to 1.

Now, if we use MRoots, and choose the black arrows of Fig. 1b as layer 0 having node 0 as root, and the gray arrows as layer 1 having node 2 as root, there are at least two routes not sharing any physical links or nodes, from any source to any destination. But for arbitrary networks, finding roots giving BFS spanning trees which guarantee redundancy for all source/destination pairs is difficult. Therefore, FRoots uses the general idea of MRoots, but discard the approach used for choosing roots and creating BFS spanning trees.

The problem of having redundancy for all source/destination pairs is equivalent to the problem of ensuring coherence regardless of what fault occurs. Let us begin with node faults. Obviously, nodes with no bypass traffic can be removed

**Fig. 1.** Networks with different routing algorithms

without problems, no more packets are coming from the fault, and packets to the fault have to be discarded anyway. Leaf nodes are guaranteed to have no such bypass traffic, since a “Down” to “Up” transition is illegal.

In FRoots, the Up*/Down* graphs assigned to each layer must be designed to ensure that every node is a leaf in at least one layer. The purpose of the Up*/Down* graphs has shifted: MRoots spreads the bottleneck of the root, while FRoots spreads the leaves.

Table 1. Algorithm to calculate FRoots’ graphs

- Create a copy C of the network topology.
- Create a set $notleaf$ initially consisting of all nodes in C .
- While $notleaf$ is nonempty
 - Find a previously unused layer ℓ in the network
 - Choose a node γ in $notleaf$ as ℓ ’s guaranteed leaf.
 - Remove γ from C .
 - Create an empty set $leafcandidates$.
 - Move γ from $notleaf$ to $leafcandidates$.
 - Find the articulation points of C .
 - For each node δ in $notleaf$ where δ is not an immediate neighbor of any node in $leafcandidates$ and δ is not an articulation point of C
 - Remove δ from C .
 - Find the new articulation points of C .
 - Create an Up*/Down* graph of C as normally, except ignoring dangling links (links connected to the nodes of $leafcandidates$)
 - The dangling links’ directions are set to have the Up-direction from the nodes of $leafcandidates$ to the nodes in C .
 - Add all nodes of $leafcandidates$ back to C .
 - Copy the link directions of C into layer ℓ .

FRoots uses the algorithm in Table 1 to iterate over the network in order to make all nodes a leaf in some layer, but doesn’t attempt to find the mini-

mum layers needed. The idea is to choose a non-leaf node for a layer, guarantee that this node is a leaf in this layer, and then sequentially add other nodes that can be a leaf in the same layer without disrupting the coherence of that layer. Articulation points are those nodes in a network which cannot be removed without disconnecting the network, and all articulation points of a network can be found in $O(|nodes| + |links|)$ [14]. At first glance, the for-loop iterates $|nodes|$ times worst case, and so does the while-loop. However, the while loop is obviously determined by the number of layers needed, and the for-loop will, on average, loop $|nodes|/|layers|$ times, since it only looks at nodes which haven't yet been assigned a leaf. The test performed in each iteration by the for-loop can, with bookkeeping, be done in $O(1)$. The body of the for-loop will run a total of $|nodes|$ times, yielding $O(|nodes| * (|links| + |nodes|))$ or, since $|links| > |nodes|$, $O(|links| * |nodes|)$ for the entire algorithm.

The algorithm in Table 1 produces a consistent Up*/Down* graph, since the Up*/Down* graph of the connected graph C is consistent, and adding a leaf node to an Up*/Down* graph can always be done without compromising the consistency. Obviously, the new leaf node has legal routes to all other nodes, because it can use the Up-link to its neighbor, and from there reach all other nodes. The other nodes can reach the new leaf node through the opposite routes. Furthermore, inserting a leaf node cannot close any cycles of Up-links, since this would require the leaf node to have an incoming Up-link, which it per definition cannot have. If the network has more layers than FRoots needs, it is possible to utilize these layers to improve the number of safe layers of each node.

Definition 1 *A safe layer of a node is a layer in which the node is a leaf node.*

When a node fault is registered by a source node, the source can exclusively use the fault's safe layers for new packets, thereby guaranteeing packet arrival. A source can stop sending new packets to a failed node, the network will always discard such packets.

For link faults, two nodes are attached to the faulty link, and the safe layers of one of these are chosen to be the coherent layers of the link fault. Which node to choose is arbitrary, we shall use the lower ID node. In Fig. 1c, the FRoots routing tables were originally identical to the MRoots routing tables of Fig. 1b, but now we assume that the link between 2 and 3 has gone down. The lower ID node is node 2, and the safe layer for this link fault is the layer 0 (black arrows).

By restricting traffic to the safe layers, we ensure that there are no legal routes using the lower ID node as a bypass node, since it is a leaf. Therefore, no legal routes go through the lower ID node, over the faulty link, and to/beyond the higher ID node. Similarly, no legal routes go from/through the higher ID node, over the faulty link, and beyond the lower ID node.

However, there can be legal routes using the faulty link if their source or destination is the lower ID node. These routes are of course unavailable, and the sources of the unavailable routes need to calculate new legal routes to the destinations of the unavailable routes. The new legal routes must be based on the Up*/Down* graphs of the safe layers, to avoid deadlock. As an example, consider the legal route from node 2, via link 2,3 to node 3. This route is now unavailable,

but 2 still has a legal route to node 3 via nodes 1 and 0. To summarize, the lower ID node must calculate new legal routes to all destinations it previously reached through the faulty link, and all sources which previously reached the lower ID node through the faulty link must calculate a new legal route to the lower ID. In the following lemma, we show that such legal routes must exist.

Lemma 1 *Given an Up*/Down* graph which remains connected in the presence of one link fault, if the failing link is attached to a leaf node, there must be a legal route from the leaf to all other nodes, and from all other nodes to the leaf, not using the failing link.*

Proof. Since the graph is still connected, the leaf must have a neighbor reachable through an Up-link. Since Up*/Down* is coherent, all nodes have legal routes to this neighbor. These routes cannot use the failed link¹, since the leaf has no bypass traffic. The neighbor can use the opposite legal routes to reach all nodes.

Routes to the leaf can be found by extending legal routes ending at the neighbor with one hop. These extensions must yield legal routes, because the extensions follow a Down-link, which is always legal to use. The leaf can use the opposite routes to reach all destinations. ■

These changes to the routing tables can be used as soon as they are calculated. Since packets using these changes are also using the same Up*/Down* graph as those using unchanged routes, no deadlocks can occur.

4.2 Reconfiguration and Multiple Faults

Guaranteeing redundant legal routes between any source/destination pair does not accommodate multiple faults, nor does it support intended network changes. To do so, we view multiple faults or changes as a series of one-faults or one-changes, separated by at least the time it takes to do a reconfiguration of the network. When the fault or change occur, the algorithm in Fig. 2 reconfigures the network in order to restore redundant one-fault tolerance, and to again allow using layers that are not safe.

In short, the algorithm drains and reconfigures unsafe layers by using one static draining algorithm per unsafe layer, and leave the safe layers intact, except for possible expansions of the routing tables. Safe layers are always coherent, and need neither draining nor reconfiguration.

It is difficult to guarantee one-fault tolerance after a fault or insertion, for two reasons: the obvious physical requirements of high connectivity, and the limited number of layers available, possibly preventing all nodes from being a leaf.

5 Simulation Results

The first simulations try to answer how many layers are typically needed for various network sizes. We have tested for several networks having 16, 32, 256,

¹ Non-minimal legal routes from the leaf to the neighbor might use the failed link, but obviously the leaf can always reach the neighbor through the direct link.

Table 2. Reconfiguring algorithm to reestablish one-fault tolerance

- For any change in the network, we identify some safe layers (the remaining are the *unsafe layers*) as such:
 - For node and link faults, the safe layers are as earlier explained.
 - For node insertion, use the safe layers of the lower ID neighboring node, and for the safe layers, assign the “Up” directions of the new links away from the new node. The new node is a leaf in the safe layers, but all neighbor lose any status as a leaf and must later find new safe layers.
 - For link insertion, use the safe layers of the lower ID node, and for the safe layers, assign the “Up” direction of the new link away from the lower ID node. The lower ID node is still a leaf in the safe layers, but the higher ID node loses any status as a leaf and must later find new safe layers.
 - In insertion cases, coherence is still guaranteed and deadlocks cannot be introduced in the safe layers, since the link direction assignment in the safe layers are according to an Up*/Down* graph. Safe layer routing tables only need to be enhanced to utilize the new resources, and can be used as soon as they are calculated.
- A message describing the change is broadcasted by the neighbors.
- As source nodes receive the broadcast, insertion into unsafe layers is stopped.
- In time, the unsafe layers become drained and are reconfigured, preferably in such a way that all nodes again have at least one safe layer. As soon as all unsafe layers are reconfigured, global reconfiguration is done.

1024, and 16384 nodes. To ensure a high network connectivity, each network either two, three or four times as many links as nodes. For the network sizes 16 through 1024, we have generated 10 000 topologies, but for 16384, only 500. Topologies are generated completely at random, with the exception that these topologies are physically connected after one fault, and that the node degree is a minimum of 4 and a maximum of 12.

Table 3. Layers needed for various network sizes

Nodes - Links	16 - 2	16 - 3	16 - 4	32 - 2	32 - 3	32 - 4	256 - 4	1024 - 4	16K - 4
Maximum	6	7	8	6	7	8	8	8	8
Average	4.2	5.1	6.2	4.3	5.2	6.1	6.6	7.0	7.2

Table 3 shows the number of layers needed to guarantee one-fault tolerance in all networks tested. Since the selection of the guaranteed leaf is random for each layer, our algorithm does not necessarily find the minimum number of layers needed. Still, the table show that the number of layers needed is modest, and scales very well, in fact, the maximum of 8 doesn’t change at all from 16 nodes/64 links to 16384 nodes/65536 links. 8 layers is feasible to implement in hardware, in InfiniBand[1], 15 virtual channels are available for data (although these are intended for QoS purposes, not for fault tolerance). An interesting observation is

that the number of average layers needed increase more when the links to nodes ratio increases than when the number of nodes increases. Note that after one fault has happened, there is no guarantee that every node still has a safe layer to ensure two-fault tolerance, although most nodes will have one. Increasing the number of layers beyond these maximum figures will alleviate this problem.

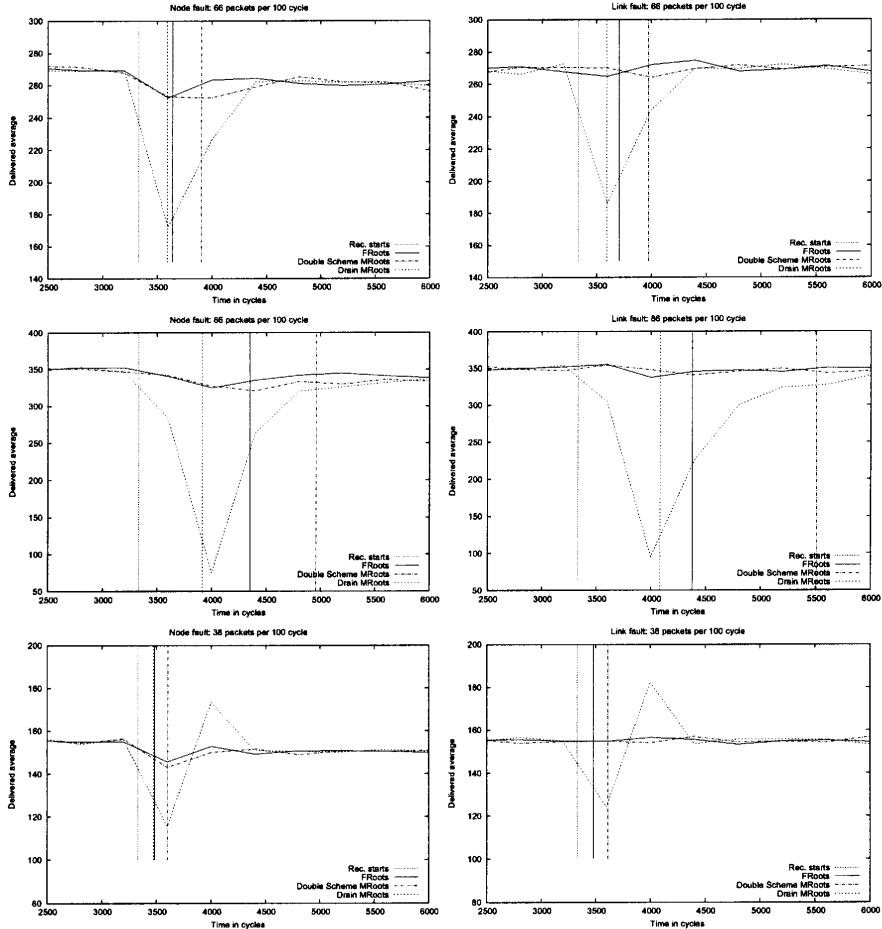


Fig. 2. Top to bottom: throughput with medium, heavy and light traffic

To get an idea on how FRoots perform, we have chosen to show results for two global reconfiguration methods using MRoots as the basic routing algorithm; the dynamic Double Scheme and static Drain. The reason for this choice is natural, FRoots is very similar to MRoots and can certainly be implemented on the same technologies. To the best of our knowledge, there are no existing reconfiguration methods which keep the network coherent and available at all times.

The Drain for MRoots is straight forward, as soon as sources register a change, they stop injection of packets until reconfiguration is done. The Double Scheme for MRoots is based on the enhanced Double Scheme [12], but since transitions between layers is forbidden the effect is slightly different. In the first phase of the Double Scheme for MRoots, sources allow injection of packets in layers 0 through m , and drain and reconfigures layers $m+1$ through n . The second phase begins when layers $m+1$ through n have been reconfigured; sources drain and reconfigures layers 0 through m and allow injection into layers $m+1$ through n . When reconfiguration of layers 0 through m is done, the network has finished reconfiguration. Even if a coherent layer set exists, the Double Scheme cannot choose it to guarantee coherence through the first phase.

The drainage algorithm is the same as in [12], a notification from leaf nodes toward the root node(s) propagates as Up-links are drained, then the root node(s) propagate notifications toward the leaves as Down-links are drained. This algorithm is done concurrently for each layer in FRoots, sequentially for the two phases in the Double Scheme, and once for the entire network in Drain.

Topology acquisition is not implemented, but could be done as in AutoNet [3]. All other necessary information is passed through control flits, which have their own virtual channels with higher priority than data channels.

The performance simulator is event-driven, and cut-through switched on flit-level. Packet size is 32 flits, the flit size is not set, however, it must be large enough to hold all routing information needed, or all information in a control message. Switches are first come first served full crossbars, and it takes one cycle to move an already routed and connected flit across the switch. Routing takes 9 cycles, and setting up the connection takes an additional two cycles. Link speed is the same as the connected switch speed. Receive buffer size is 32 flits (virtual cut-through). We have used 20 topologies with 32 nodes and 128 links, and use 16 virtual channels for data per link. Both MRoots and FRoots utilize all virtual channels. Networks are random, but a spanning tree is built first to ensure connectivity, by first creating node 0, then for $n = 1, \dots, 32$, node n is created and connected to a randomly selected node m , $m = 0, \dots, n - 1$. Then, the remaining links are added. Traffic is random, and uniformly distributed. Only faults where node 0 is the node fault, or link 0, 1 is the link fault, are simulated.

For MRoots, the first layer is assigned the “best” root, meaning the root with the smallest average minimum distance to all other nodes. The other roots are selected as far away from all previous roots as possible. This makes node 0 an important and fragile node for MRoots; the way random trees are generated, node 0 is likely to have a high degree and thus a higher chance of being the “best” root. To compensate, FRoots uses node 0 as the root of the first layer.

The top two plots of Fig. 2 show the throughput under medium traffic load (send queues overflow only in Drain) over time for the three methods. The first vertical bar is the time of the fault, when the reconfiguration starts, the other vertical bars show when the global reconfiguration is done for each method. It is clear that the Drain has a severe downgrade of the throughput in the time after the fault and until it has been reconfigured, since no new packets are

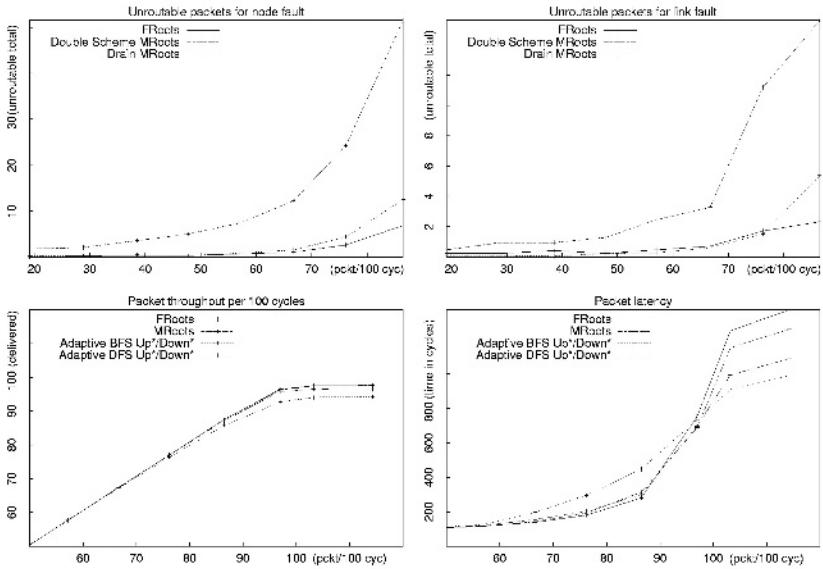


Fig. 3. Top: Unroutable packets, Bottom: Throughput and latency comparison

entered until reconfiguration completes. Also, Drain finishes fast. The Double Scheme and FRoots have similar performance, although Double Scheme dips below FRoots when it's second phase starts. FRoots finishes faster than Double Scheme, which is not very surprising since it doesn't need to drain all layers. The reason for Drain being much faster than both FRoots and MRoots, is that in the former, the draining packets have all resources alone, while for the two latter, new packets keep coming and competes with the packets to be drained. Figure 2 also show plots for heavy (send queues overflow in all methods) and light (send queues do not overflow) traffic loads.

The main contribution of FRoots is guaranteed delivery of packets inserted after a fault is known, thus it is interesting to see how many packets become unroutable. Fig. 3 (top) shows unroutable packets for all methods as traffic load increases. FRoots clearly has less unroutable packets than the Double Scheme. Drain should perform equally to FRoots, since only packets residing in the network at the time of the fault become unroutable. The variations seen in the plots stems from the differences of the FRoots and the MRoots routing algorithm. However, FRoots can still deliver 'unroutable' packets; as the packets arrive at a neighbor, this neighbor inserts them into the safe layers and start their routing "from scratch", these packets behave as new packets from this neighbor.

Finally, since FRoots was designed for fault tolerance, not for high throughput, it is interesting to form an idea of how it performs against other routing algorithms. In Fig.3 we have compared FRoots with MRoots, BFS Up*/Down*, and DFS Up*/Down*, all methods are physically adaptive, and the Up*/Down* methods also have virtual adaptivity. Encouragingly, in our examples, FRoots

shows equal throughput and slightly better latency until saturation compared to DFS Up*/Down* and MRoots, while outperforming BFS Up*/Down*. However, extensive experiments are needed to draw further conclusions on the efficiency of the routing algorithm of FRoots.

6 Conclusion

FRoots is a hybrid fault tolerance method, which uses redundant paths to accommodate one fault, and dynamic global reconfiguration to prepare for future faults and changes. It is based on technologies with virtual channels, and need only a modest number of such channels to fulfill its guarantees. To our knowledge, FRoots is the only method which keeps the network available, coherent, and deadlock free at all times.

FRoots is able to combine the network availability of Double Scheme with the low number of unrouteable packets of Drain. It has a reconfiguration time span closer to Drain than to the Double Scheme, its underlying routing algorithm shows no loss of performance to its predecessor MRoots, and remains a very simple load balanced routing algorithm based on Up*/Down*.

References

1. InfiniBand Trade Association, “InfiniBand Architecture Specification,” .
2. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*, IEEE, 1997.
3. M. D. Schröder et al., “Autonet: a High-Speed, Self-Configuring Local Area Network Using Point-to-point Links,” SRC Research Report 59, DEC, 1990.
4. N. J. Boden et al, “Myrinet—A Gigabit-per-Second Local-Area Network,” IEEE MICRO, 1995.
5. O. Lysne and T. Skeie, “Load Balancing of Irregular System Area Networks through Multiple Roots,” in *Int. Conf. on Communications in Computing*, 2001.
6. J. C. Sancho and A. Robles, “Improving the Up*/Down* Routing Scheme for Networks of Workstations,” in *Proc. of Euro-Par*, 2000, LNCS, pp. 882–889.
7. T. Skeie, O. Lysne, and I. Theiss, “Layered Shortest Path (LASH) Routing in Irregular System Area Networks,” in *Proc. of IPDPS*. 2002, IEEE.
8. J. C. Sancho et al, “Effective Methodology for Deadlock-Free Minimal Routing in InfiniBand Networks,” in *Proc. of ICPP*. 2002, pp. 409–418, IEEE.
9. A. A. Chien and J. H. Kim, “Planar-adaptive routing: Low-cost adaptive networks for multiprocessors,” *Journal of the ACM*, vol. 42, no. 1, pp. 91–123, 1995.
10. I. Theiss and O. Lysne, “LORE - Local Reconfiguration for Fault Management in Arbitrary Networks,” Report 2003-02, Simula Research Laboratory, Norway, 2003.
11. R. Casado et al, “Performance evaluation of dynamic reconfiguration in high-speed local area networks,” in *Proc. of the 6th Int. Symp. on High-Performance Computer Architecture*, 2000.
12. R. Pang, T. Pinkston, and J. Duato, “The Double Scheme: Deadlock-Free Dynamic Reconfiguration of Cut-Through Networks,” in *Proc. of ICPP*. 2000, IEEE.
13. O. Lysne and J. Duato, “Fast Dynamic Reconfiguration in Irregular Networks,” in *Proc. of ICPP*. 2000, pp. 449–458, IEEE.
14. M. A. Weiss, *Data Structures and Algorithm Analysis*, Benjamin/Cummings, 1992.

Admission Control for DiffServ Based Quality of Service in Cut-Through Networks

Sven-Arne Reinemo, Frank Olaf Sem-Jacobsen, Tor Skeie, and Olav Lysne

Simula Research Laboratory

P.O.Box 134, N-1325 Lysaker, Norway

svenar@simula.no, frankose@ifi.uio.no, {tskeie,olavly}@simula.no

Abstract. Previous work on Quality of Service in Cut-through networks shows that resource reservation mechanisms are only effective below the saturation point. Admission control in these networks will therefore need to keep network utilization below the saturation point, while still utilising the network resources to the maximum extent possible. In this paper we propose and evaluate three admission control schemes. Two of these use a centralised bandwidth broker, while the third is a distributed measurement based approach. We combine these admission control schemes with a DiffServ based QoS scheme for virtual cut-through networks to achieve class based bandwidth and latency guarantees. Our simulations show that the measurement based approach favoured in the Internet communities performs poorly in cut-trough networks. Furthermore it demonstrates that detailed knowledge on link utilization improves performance significantly.

1 Introduction

Internet has today evolved into a global infrastructure supporting applications such as streaming media, E-commerce, network storage, etc. Each of these applications must handle an ever increasing volume of data demanded as predictable transfers. In this context the provision of Quality of Service (QoS) is becoming an important issue. In order to keep pace with computer evolution and the increased burden imposed on data servers, application processing, etc. created by the popularity of the Internet, we have in recent years seen several new technologies proposed for System and Local Area Networking (SAN/LAN) [3,14,7,4,24]. Common for this body of technologies is that they rely on point-to-point links interconnected by off-the-shelf switches that support some kind of back-pressure mechanism. Besides, most of the referred technologies also adhere to the cut-through or wormhole switching principles - only Gigabit Ethernet is using the store-and-forward technique. For a survey of some relevant networking principles we refer to [6].

IETF has for several years provided the Internet community with QoS concepts and mechanisms. The best known ones are Integrated Services (IntServ) [8], Resource Reservation Protocol (RSVP) [13], and Differentiated Services (Diff-Serv) [5]. IntServ together with RSVP define a concept based on per flow reservations (signalling) and admission control to be present end-to-end. DiffServ,

however, takes another approach assuming no explicit reservation mechanism in the interior network elements. QoS is here realized by giving data packets differentiated treatment relative to the QoS header code information. In the underlying network technologies QoS has to a less extent been emphasised - the key metrics here have mainly been mean throughput and latency. To provide QoS end-to-end, possibly over heterogeneous technologies this means that the lower layers should also have support for predictable transfer including the ability to interoperate with a higher level IETF concept. This issue is being challenged by emerging SAN/LAN standards, such as InfiniBandTM [4] and Gigabit Ethernet [24] providing various QoS mechanisms.

Recently we have also seen several research contributions to this field. Jaspernite et. al. [9,10] and Skeie et. al [15] discuss different aspects of taking control of the latency through switched Ethernet relative to the IEEE 802.1p standard aiming for traffic priorities. Another body of work is tailored to the InfiniBandTM architecture (IBA) [12, 1, 2]. In [12] Pelissier gives an introduction to the set of QoS mechanisms offered by IBA and the support for DiffServ over IBA. In this approach the presence of admission control is assumed. Alfaro et. al build on this scheme and present a strategy for computing the arbitration tables of IBA networks, moreover a methodology for weighting of virtual layers referring to the dual arbitrator defined by IBA [2]. The concept is evaluated through simulations assuming that only bandwidth sensitive traffic requests QoS. In [1] Alfaro et. al also include time sensitive traffic, besides calculating the worst case latency through various types of switching architectures.

DiffServ is foreseen to be the most prominent concept for providing QoS in the future Internet [17, 11]. DiffServ makes a distinction between boundary nodes and core nodes with respect to support of QoS features. Following the DiffServ philosophy no core switch should hold status information about passing-through traffic. Neither should there be any explicit signalling on a per flow basis to these components. This means that within the DiffServ framework any admission control or policing functionality would have to be implemented by boundary nodes or handled by a dedicated bandwidth broker. The core switches are assumed to perform traffic discrimination only based on service class, which is decided by a QoS tag included in the packet header - all packets carrying the same QoS tag will get equal treatment. From that viewpoint DiffServ is apparently a relative service model having difficulties giving absolute guarantees.

None of the previous debated contributions comply with the DiffServ model. In [12] Pelissier, however, discusses interoperation between DiffServ and IBA on a traffic class and service level basis, but refer to RSVP with respect to admission control. The strategy proposed by Alfaro et. al has to recompute the IBA dual arbitrator every time that a new connection is honoured [1,2]. And such a scheme is not associative with DiffServ. Neither is the admission control scheme presented in [29] by Yum et. al, which use hop by hop bandwidth reservations and requires recomputations of the weighted round robin scheduler at every hop towards the destination. In [25] Reinemo et. al. studied the provision of QoS in cut-through networks by adhering to the DiffServ model. The problem was

approached without any explicit admission control mechanism, as a pure relative model. Empirically they examined the sensitivity of different QoS properties under various load and traffic mixture conditions, hereunder assessing the effect of back-pressure.

In this paper we endeavour to achieve class based QoS in cut-through networks by use of admission control. More specifically, we extend the concept described in [25] with admission control. However, still in compliance with the Diff-Serv paradigm where service classes, as aggregated flows, are the target for QoS. Three different admission control mechanisms are proposed and carefully evaluated through extensive simulations. Two of the schemes assume pre-knowledge of the network's performance behaviour without admission control, and are furthermore implemented as a centralised bandwidth broker. The third scheme is based on endpoint/egress admission control and relies on measurements to assess the load situation, inspired by Internet QoS provisioning. To the best of the authors' knowledge no detailed admission control methods have been proposed for cut-through networks before.

The rest of this paper is organised as follows. In section 2 we give a description of our QoS architecture and routing algorithm, in section 3 our three admission control mechanisms are described, and in section 4 our simulation scenario is described. In section 5 we discuss our performance results, and finally in section 6 we give some concluding remarks.

2 QoS Architecture

The architecture used in our simulations is inspired by IBA link layer technology [4] and is a flit based virtual cut-through (VCT) switch. The overall design is based on the canonical router architecture described in [6].

In VCT the routing decision is made as soon as the header of the packet is received and if the necessary resources are available the rest of the packet is forwarded directly to the destination link [23]. If the necessary resources are busy the packet is buffered in the switch. In addition we use flow control on all links so all data is organised as flow control digits (flits) at the lowest level.

The switch core consists of a crossbar where each link and VL has dedicated access to the crossbar. Each link supports one or more virtual lanes (VL), where each VL has its own buffer resources which consist of an input buffer large enough to hold a packet and an output buffer large enough to hold two flits to increase performance. Output link arbitration is done in a round robin fashion.

To achieve QoS our switch architecture support QoS mechanisms like the ones found in the IBA architecture. IBA supports three mechanisms for QoS which are mapping of service level (SL) to VL, weighting of VLs and prioritising VLs as either low priority (LP) or high priority (HP). A more detailed description of these QoS aspects can be found in [25].

The routing used is a newly introduced routing algorithm called *Layered shortest path routing* (LASH) [16]. LASH is a minimal deterministic routing algorithm for irregular networks which only relies on the support of virtual layers.

There is no need for any other functionality in the switch, so LASH fits well with our simple approach to QoS. An in-depth description of LASH is found in [16].

3 Admission Control

In this section we propose three different admission control (AC) mechanisms that we carefully evaluate in section 5.

3.1 Calibrated Load Based Admission Control

The *Calibrated Load* (CL) approach is a simple scheme relying on the fact that a BB knows the total rate of traffic entering the network. Our AC parameter is the amount of traffic which can be injected into the network while still keeping the load below saturation. As the rate of traffic entering the network reaches the CL parameter no more traffic will be admitted. In most cases the CL parameter must be decided by measurements on the network in question to find the saturation point - our CL is deduced from measurements performed in [25].

To keep HP and LP traffic separated we use two different CL parameters, one for the total HP traffic and one for the total LP traffic. For HP traffic this can be expressed as follows

$$\sum_{i=0}^n L_{HP,i} + P_{HP} < CL_{HP} . \quad (1)$$

Here CL_{HP} is the calibrated load for outgoing HP traffic, $L_{HP,n}$ is the HP load in node n and P_{HP} is the peak rate for the requesting flow. Moreover, the flow is admitted if the total HP load $\sum_{i=0}^n L_{HP,i}$ plus the requested increase P_{HP} is below the calibrated load CL_{HP} . LP traffic can be expressed similarly just substituting HP values with LP values. The strength of this scheme is that it is simple, its weakness is that it is inaccurate since it does not take into account the distribution of flows in the network. And from that viewpoint it is less suitable for handling hot spots.

3.2 Link by Link Based Admission Control

Our second scheme is the Link-by-Link (LL) approach. Here the BB knows the load on every link in the network and will consult the availability of bandwidth on every link between source and destination before accepting or rejecting a flow. Compared to the CL approach, this solution assumes both topological and routing information about the network.

For the AC decision we adopt the *simple sum* approach as presented in [28]. This algorithm states that a flow may be admitted if its peak rate p plus the peak rate of the already admitted flows s is less than the link bandwidth bw . Thus the requested flow will be admitted if the following inequality is true [28]

$$p + s < bw \quad (2)$$

we view p as the increase in peak rate for the flow and s as the sum of the admitted peak rates. As for the *CL* method we deduce the effective bandwidth from the measurements obtained in [25]. Since we are dealing with service levels where each SL have different bandwidth requirements it is natural to introduce some sort of differentiation into equation (2). We achieve this by dividing the link bandwidth into portions relative to the traffic load of the SLs, and include only the bandwidth available to a specific service level bw_{SL} in the equation as follows

$$p + s_{SL} < bw_{SL} \quad (3)$$

where

$$bw_{SL} = bw_{link} * \frac{load_{SL}}{load_{total}} \quad (4)$$

and s_{SL} is the sum of the admitted peak rates for service level SL and bw_{link} is the effective link bandwidth.

3.3 Egress Based Admission Control

Our third scheme is the Egress Based (EB) approach. This is a fully distributed AC scheme where the egress nodes are responsible for conducting the provisions. Basically, we here adopt the Internet AC concept presented by Cetikaya and Knightly in [26]. This method does not assume any pre-knowledge of the network behaviour as was the case with our previous solutions. Also different from the previous approaches is the use of a delay bound as the primary AC parameter. For clarity we give a brief outline of the algorithm below, a more detailed description can be found in [27].

The method is entirely measurement based and relies on that the sending nodes timestamp all packets enabling the egress nodes to calculate two types of measurements. First, by dividing time into timeslots of length τ and counting the number of arriving packets, the egress nodes can deduce the arrival rate of packets in a specific timeslot. By computing the maximum arrival rate for increasingly longer time intervals we get a peak rate arrival envelope $R(t)$, where $t = 0, \dots, T$ timeslots, as described in [26]. Second, by comparing the originating timestamp relative to the arrival time, the egress node can calculate the transfer time of a packet. Having this information available the egress node can furthermore derive the time needed by the infrastructure to service k following packets; i.e. a consecutive stream of packets where the next packet in the service class enters the infrastructure before the previous packet has departed the egress node. By doing this for larger and larger k sequences of packets within a measuring interval of length $T\tau$ and subsequently inverting this function we achieve the service envelope $S(t)$, giving the amount of packets processed by the network in a given time interval t . Now repeating this M times, the mean $\bar{R}(t)$ and the variance $\sigma^2(t)$ of $R(t)$, and the mean $\bar{S}(t)$ and variance $\Psi^2(t)$ of $S(t)$ may be calculated. If a flow request has a peak rate P and a delay bound D it may be accepted if the peak rate P plus the measured arrival rate $R(t)$ is less than the service rate allowing for the delay $D, S(t + D)$.

The EB scheme derives its knowledge of the network from measurements of the traffic passing through the egress nodes. It is therefore difficult for the egress nodes to have a complete picture of the load in the network, moreover the packet latency is used to infer the network load utilising the fact that an increased network load will cause increase in latency as well. From that viewpoint it seems difficult to give a service class bandwidth guarantees since it has no concrete knowledge of the network load. The algorithm will admit as much traffic as it can without breaking the delay bound. The key instrument of the scheme is the given delay bound for the different flows, and the efficiency of the algorithm is linked to its ability to limit the service levels to operate within the delay bounds.

3.4 Target for Admission Control

The main findings for the work in [25] are that (*i*) throughput differentiation can be achieved by weighting of VLs and by classifying the VLs as either low or high priority, (*ii*) the balance between VL weighting and VL load is not crucial when the network is operating below the saturation level. In general this sets the target for the AC, since as long as we can ensure that the load of the various service classes is below saturation level we can also guarantee that each of these classes get the bandwidth they request. The target for admission control is thus the point where the amount of accepted traffic is starting to become less than traffic offered. The effective bandwidth at this point will be used as a steering vehicle by the CL and LL methods.

Another main finding in [25] is that though the latency characteristics below saturation were fairly good, significant jitter was observed. This problem we challenge by proposing the EB method, where a given delay bound is the requested quality of service. Since this concept is continuously monitoring the end-to-end latency characteristics for all pair of nodes one should possibly expect that delay guarantees could be given.

4 Simulation Scenario

For all simulations we have used a flit level simulator developed in house at Simula Research Laboratory. In the simulation results that follow, all traffic is modelled by a normal approximation of the Poisson distribution. We have performed simulations on a network with 32 switches, where each switch is connected to 5 end nodes and the maximum number of links per switch is 10 in addition to the end nodes. We have randomly generated 16 irregular topologies and we have run measurements on these topologies at increasing load. We use LASH [16] as routing algorithm and random pairs as traffic pattern. In the random pairs scheme each source sends only to one destination and no destination receives from more than one source. The link speed is one flit per cycle, the flit size is one byte and the packet size is 32 bytes for all packets.

The five different end nodes send traffic on one of five different service levels. One service level for each node (Table 1), SL 1 and 2 are considered to be of

the expedited forwarding (EF) class in DiffServ terminology. And SL 3 and 4 are considered to be of the assured forwarding (AF) class. SL 5 is considered as best effort (BE) traffic and from that viewpoint is not a subject of AC.

For the CL and LL schemes all simulations were run with an ACT deduced from our measurements in [25]. In the first part of the simulation the send rate is steadily increased by adding more and more flows until admission is denied by the AC scheme. When this happens the current rate is not changed, but the node will continue to try to go beyond the ACT for a fixed number of times before it gives up. For the EB scheme the send rate is increased in the same way, but the AC decision is primarily based on measured latency as described in section 3.3.

Table 1. Services levels

SL	DS ¹	Load %	BW ²	Pri	SL	DS	Load %	BW	Pri	SL	DS	Load %	BW	Pri
1	EF	10	4	high	3	AF	20	8	low	5	AF	30	1	low
2	EF	15	6	high	4	AF	25	10	low					

5 Performance Results

5.1 Throughput

Recall that the target for the AC is to make sure that the network operates below saturation at all times, since below this point we can guarantee that all SLs get the bandwidth they request. The relative requests for each SL are as shown in Table 1. Figure 1(a) shows what happens in a network without AC when it enters saturation. We are no longer able to give all service classes the bandwidth they request and the HP classes preempt LP bandwidth, i.e. the bandwidth differentiation is no longer according to the percentages in Table 1. In the CL scheme, we see from figure 1(b) that we are successful in keeping the accepted load below the saturation point, even as the offered load goes beyond this point. The bandwidth differentiation does not fail as is the case in figure 1(a), but it suffers slightly as we reaches *high* load. As the load increases the differentiation between SLs in the same class is diminished. Thus, the CL scheme is able to keep the load below saturation. However, it appears that it is too coarse to achieve good bandwidth differentiation between SLs of the same class since it makes its AC decisions based on the total load for a class and not for each SL.

Moving on to the LL scheme (figure 1(c)) we see several improvements compared to the CL scheme. First, we get a sharper bandwidth cut-off with much less hesitation than for CL. Second, we achieve a differentiation relative to the requests, meaning that we can give bandwidth guarantees. Third, we are able to

¹ The DiffServ equivalent service class.

² The maximum number of flits allowed to transmit when scheduled.

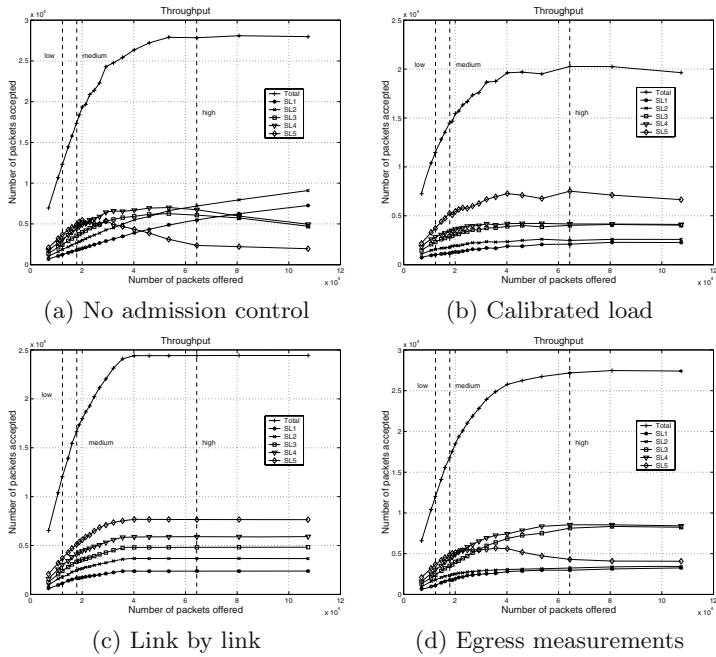


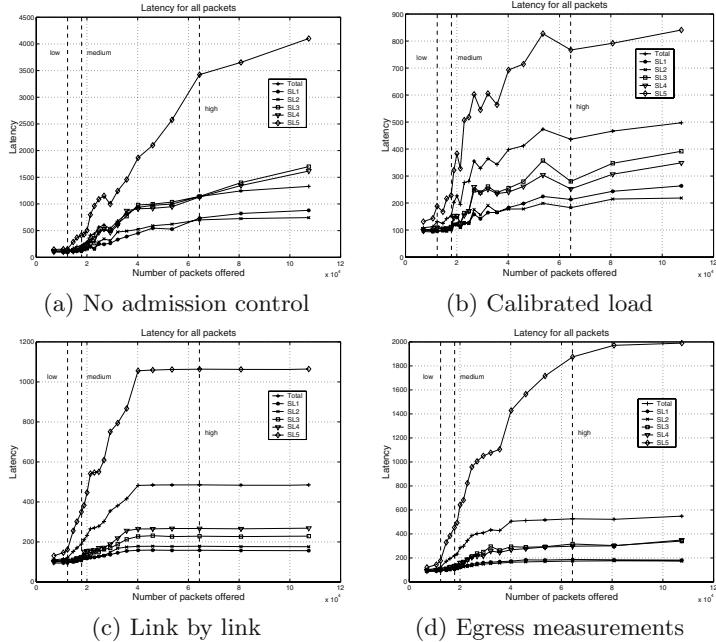
Fig. 1. Throughput

utilise the network resource better as we get closer to the saturation point. This improvement is probably due to the fact that the LL scheme knows the load of every link in the network and is able to make the AC decision based on the load along the actual source/destination path.

Finally, we have the EB method. It is apparent from figure 1(d) that this method is unable to give bandwidth guarantees, as well as increasing the load beyond the saturation point and admitting too much traffic. Now as the load increases beyond saturation the best effort traffic (SL 5) is reduced as it must make way for traffic on the other SLs. The issue here is that delay is the most significant AC parameter in this scheme and bandwidth requirements have more or less been ignored.

5.2 Latency

Let us now turn our attention to the latency results. Figure 2(a) shows the average latency for increasing load values without admission control. Comparing it with the CL results in figure 2(b) shows that the CL scheme is quite close when we look at the same load values. The average latency for all packets is 436 for CL at the high mark which is a 6% increase compared to the scheme without AC at a corresponding load. A problem with the CL scheme is that the latency values are unstable as the load increases since the estimate of current throughput is to coarse to target the exact rejection point. The LL scheme overcomes this

**Fig. 2.** Average latency

problem as shown in figure 2(c). As soon as the LL scheme starts rejecting flows the latency stabilises. The latency values for high load is 485 which is slightly above the CL latency at this particular point. The LL method also gives a more linear increase in latency as we approach the rejection point for new flows. The LL scheme is the better of the two as it gives lower latency to SL 1-4 and higher latency to the best effort traffic in SL 5. Even if it has a higher average latency for all packets (compared to CL) it performs better since the increase in the average is caused by the best effort traffic in SL 5.

The EB scheme uses measured latency as its primary AC parameter. The results are presented in figure 2(d). This scheme produces average results which fall between the CL and LL scheme. Furthermore, it is capable of giving the same latency to SLs fairly independently of weight such as SL 1 and 2, but it is unable to satisfy the delay bound of 100 for SL 1 and 2, and 250 for SL 3 and 4. The achieved latency at the high mark is 187 for SL 1 and 316 for SL 3. So even if using a measurement based method we are unable to give hard delay guarantees. It seems very difficult to give delay bounds in combination with good throughput in cut-through networks. To remedy this problem one possible could reduce the throughput by hardening the AC, or we could turn to other means such as modifying the flow control to better handle delay bound traffic.

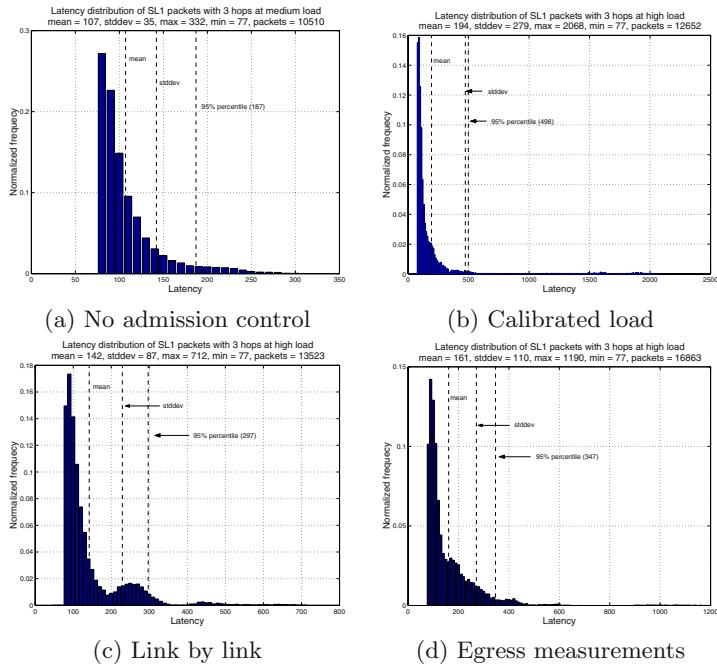


Fig. 3. Latency distribution for packets with 3 hops

5.3 Jitter

Finally let's turn our attention to jitter. Figure 3 shows the latency distributions for packets with a path length of 3 hops. This was the most frequently occurring path length in our simulations. The mean, standard deviation and 95 % percentile are marked with a dashed line in the figures. The distance between the mean mark and the standard deviation mark reflects the standard deviation.

Figure 3(a) shows the latency distribution achieved without AC at the load marked as medium in figure 1(a). Figure 3(b) shows the distribution for the CL scheme at the load marked as high in figure 1(b). Note that the load at this point is about 20% above that without admission control. We see that the CL scheme has quite poor jitter characteristics, which is reasonable if we recall the latency curve we saw in figure 2(b). The mean is 194, the standard deviation 279 and the 95% percentile is 498. Thus 95% of the packets have a latency of 498 or below. Even if the mean value is not too bad, the large standard deviation and the 95% percentile shows that jitter is clearly very high. The LL scheme has better jitter characteristics. From figure 3(c) we see that the histogram has a shorter tail compared to figure 3(b). We have a mean of 142, standard deviation of 87 and a 95% percentile of 297. Which reduces the jitter potential and gives us almost a 40% reduction of packet latency for 95% of the packets. In addition the load for LL at this point is about 7% above the CL load. Thus, better results are achieved at a higher load. For EB scheme in figure 3(d) we see that it is

unable to improve on the results from the LL scheme. With a 95% percentile of 357 it has a 30% improvement over the CL. Note that this is achieved at a load 25% above the CL load. Still, even a measurements based delay bound scheme is unable to give good jitter characteristics in cut-through networks.

6 Conclusion

In this paper we propose and evaluate three different admission control schemes for virtual cut-through networks. Each one suitable for use in combination with a DiffServ based QoS scheme to deliver soft real-time guarantees. Two of the schemes assume pre-knowledge of the network's performance behaviour without admission control, and are both implemented with bandwidth broker. The third method is based on endpoint/egress admission control and relies on measurements to assess the load situation.

Our main findings are as follows. First, bandwidth guarantees for aggregated flows are achievable with the use of the Link-by-Link scheme. While the Calibrated Load and Egress Based methods are unable to achieve such good guarantees. Second, latency and jitter properties are hard to achieve regardless of the method used. This is due to the nature of cut-networks and the way flow control affects latency. Strict admission control can be used to improve latency, but at the cost of lower throughput. To achieve a combination of high throughput and low latency modifications to the flow control may be considered.

References

1. F. J. Alfaro, J. L. Sanchez, J. Duato, and C. R. Das. A strategy to compute the InfiniBand arbitration tables. In *Proceedings of International Parallel and Distributed Processing Symposium*, April 2002.
2. F. J. Alfaro, J. L. Sanchez, and J. Duato. A strategy to manage time sensitive traffic in InfiniBand. In *Proceedings of Workshop on Communication Architecture for Clusters (CAC)*, April 2002.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet – a gigabit-per-second LAN. IEEE MICRO, 1995.
4. InfiniBand Trade Association. Infiniband architecture specification.
5. Differentiated Services. RFC 2475.
6. J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks an engineering approach*, IEEE Computer Society, 1997.
7. R. W. Horst. Tnet: A reliable SAN. *IEEE Micro*, 15(1):37–45, 1995.
8. Integrated Services. RFC 1633.
9. J. Jasernite, and P. Neumann. Switched Ethernet for Factory Communication. In *Proceedings of 8th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'01)*, 205–212, October 2001.
10. J. Jasernite, P. Neumann, M. Theiss, and K. Watson. Deterministic real-time communication with switched Ethernet. In *Proceedings of 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.
11. K. Kilkki. Differentiated services for the Internet. *Macmillian Tech. Publishing*, 1999.

12. J. Pelissier. Providing quality of service over InfiniBandTM architecture fabrics. In *Proceedings of Hot Interconnects X*, 2000.
13. ReSource ReserVation Protocol. RFC 2205.
14. M. D. Schroder et.al., "Autonet: a high-speed, self-configuring local area network using point-to-point links," SRC Research Report 59, Digital Equipment Corporation, 1990.
15. T. Skeie, J. Johannessen, and Ø. Holmeide. The road to an end-to-end deterministic Ethernet. In *Proceedings of 4th IEEE International Workshop on Factory Communication Systems (WFCS'02)*, August 2002.
16. T. Skeie, O. Lysne, and I. Theiss. Layered shortest path (LASH) routing in irregular system area networks. In *Proceedings of Communication Architecture for Clusters*, 2002.
17. X. Xiao and L. M. Ni. Internet QoS: A Big Picture In *IEEE Network Magazine*, 8–19, March/April 1999.
18. J. S. Yang and C. T. King, "Turn-restricted adaptive routing in irregular wormhole-routed networks," in *Proceedings of the 11th International Symposium on High Performance Computing (HPCS97)*, July 1997.
19. A. A. Chien and J. H. Kim, "Approaches to Quality of Service in High-Performance Networks," in *Lecture Notes in Computer Science*, vol. 1417, 1998.
20. J. Duato and S. Yalamanchili and B. Caminero and D. S. Love and F. J. Quiles, "MMR: A High-Performance Multimedia Router - Architecture and Design Trade-Offs," in *HPCA*, pages 300-309, 1999.
21. B. Caminero, C. Carrion, F. J. Quiles, J. Duato and S. Yalamanchili, "A Solution for Handling Hybrid Traffic in Clustered Environments: The MultiMedia Router MMR," in *Proceedings of IPDPS-03*, April 2003.
22. M. Gerla and B. Kannan and B. Kwan and E. Leonardi and F. Neri and P. Palnati and S. Walton, "Quality of Service Support in High-Speed, Wormhole Routing Networks," in *International Conference on Network Protocols (ICNP'96)*, 1996.
23. P. Kermani and L. Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," in *Computer Networks*, no. 4, vol. 3, 1979.
24. R. Seifert, *Gigabit Ethernet*, Addison Wesley Pub Co., 1998.
25. S. A. Reinemo and T. Skeie and O. Lysne, "Applying the DiffServ Model in Cut-through Networks," in *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2003.
26. C. Cetikaya and E. W. Knightly, "Egress admission control," in *INFOCOM (3)*, pages 1471-1480, 2000.
27. J. Schlembach and A. Skoe and P. Yuan and E. Knightly, "Design and Implementation of Scalable Admission Control," in *QoS-IP*, pages 1-15, 2001.
28. S. Jamin and S. J. Shenker and P. B. Danzig, "Comparison of Measurement-Based Admission Control Algorithms for Controlled-Load Service," in *INFOCOM (3)*, pages 973-980, 1997.
29. K. H. Yum, E. J. Kim, C. R. Das, M. Yousif, J. Duato, "Integrated Admission and Congestion Control for QoS Support in Clusters," in *Proceedings of IEEE International Conference on Cluster Computing*, pages 325-332, September 2002.

On Shortest Path Routing Schemes for Wireless Ad Hoc Networks

Subhankar Dhar¹, Michael Q. Rieck², and Sukesh Pai³

¹ San José State University, San José, CA 95192 USA, dhar_s@cob.sjsu.edu

² Drake University, Des Moines, Iowa 50311 USA, michael.rieck@drake.edu

³ Microsoft Corporation, Mountain View, CA 94043 USA, sukeshp@microsoft.com

Abstract. In this paper, we propose two new distributed algorithms for producing sets of nodes that can be used to form backbones of an ad hoc wireless network. Our focus is on producing small sets that are d -hop connected and d -hop dominating and have a desirable ‘shortest path property’. These algorithms produce sets that are considerably smaller than those produced by an algorithm previously introduced by the authors. One of these two new algorithms has constant-time complexity.

1 Introduction

Wireless ad hoc networks have become an area of active research in recent years with the advent of various mobile wireless communication devices. A central question here is how to orchestrate communications across such a network, where it is assumed that messages might need to be sent between any two nodes of the network and the nodes of the network are also expected to move and go in and out of contact with one another. In the absence of any fixed-location message-routing hardware, it is difficult to see how to best manage such a network. Connected dominating sets and d -hop dominating sets have received attention in recent years as infrastructures for facilitating routing in such networks.

Throughout this paper, G will denote the underlying graph of an ad hoc wireless network, and it will be assumed to be connected. The nodes of G are the nodes of the network. Communication links between two nodes are assumed to be bidirectional and are represented by edges in G . In [6], the authors presented an algorithm to construct a d -hop connected d -hop dominating set of vertices in G , used as a virtual backbone for routing in the network. Here d is a fixed integer greater than one. “ d -hop connected” means that given any two nodes u and v in the set, there is a sequence of nodes in the set, beginning with u and ending with v , such that the graph-theoretic distance (hop count) between consecutive nodes in the sequence never exceeds d . “ d -hop dominating” (also called “ d -dominating”) simply means that each node in the network is within d hops of a node in the set. The set in [6] also has a rather special and useful property, as follows:

Definition 1. A set of nodes S has the d -shortest path property if, for any two nodes u and v in the graph, there exists a shortest path (*i.e.* a path whose

graph-theoretic length is as small as possible) connecting u and v such that the set of nodes on this path that are also in S , together with u and v , form a d -hop connected set.

Such a set can be used to facilitate shortest path routing through the network in a straightforward manner, as will now be explained. In [7, Subsection 5.1], Jie Wu and Hailan Li describe how to use their connected dominating set to route messages. The idea is that each of the nodes in this backbone maintains global routing tables, and whenever an average node needs to send a message across the network, and is uncertain how to best route this message, it queries its neighboring backbone nodes. Each node replies by indicating the cost (number of hops) of routing the message through that particular backbone node. The node sending the message then chooses to send the message along the path requiring the least number of hops. While the message will thus follow a shortest possible (*i.e.* fewest hops) backbone path, there is no guarantee that this path will be a shortest path for the network taken as a whole.

In [6], the algorithm of Wu and Li was adapted so as to create d -hop connected d -hop dominating sets, and this was compared with our own algorithm for constructing such sets. Analogous to the above, a d -hop connected d -hop dominating set can facilitate routing, whereby an average node uncertain how to route a particular message is able to query the backbone nodes within d -hops of itself in order to find a suitable route. However, the Wu-Li set is again unable to guarantee that the path taken will be a shortest possible path through the network. Our set does guarantee this property, and this in fact is the reason for the definition of the “shortest path property”.

In this paper, we propose two new algorithms. Our focus is on producing smaller sets than those in [6], while maintaining the same essential features: d -hop connected, d -hop dominating, d -shortest path property.

2 Related Work

B. Liang and Z. J. Haas [5] used a distributed greedy algorithm to produce a small d -hop dominating set. To do so, they reduced the problem to a special case of the Set Covering Problem. In fact, our approach is quite similar in this regard. We also consider a certain Set Covering Problem, although a different one. We too consider finding an approximate solution via the distributed greedy algorithm. However, each node in our method only needs to maintain an awareness of its local $(d + 1)$ -hop neighborhood, as opposed to the $2d$ -hop neighborhood required in [5]. Moreover, the resulting set is not only d -hop dominating, but is also d -hop connected and has the d -shortest path property.

J. Wu and H. Li proposed a basic distributed algorithm [7] for constructing a connected dominating set in a connected graph of radius at least two. However, it does not however have the d -shortest path property. Our research has been concerned with d -hop connected d -hop dominating sets, and was largely motivated by considering extensions/alterations of the Wu-Li method. One such

alteration is quite simple, and just involves applying the Wu-Li method directly to the d -closure G_d of the original graph G . This graph is simply the graph obtained from G by adding edges between two non-adjacent nodes that are within d hops of each other in G . Note that a set of vertices is d -hop dominating (in G) if and only if it is a dominating set in G_d , and it is d -hop connected (in G) if and only if it is connected in G_d .

While applying the Wu-Li method to G_d rather than G , has the advantage of greatly reducing the size of the backbone, this method does not assure shortest path routing. By contrast, the $dCDS$ algorithm [6], which we have chosen to rename d -SPR-I to achieve a greater coherence in the naming of our algorithms, produces a somewhat larger backbone, but one which guarantees shortest path routing. In order to accomplish this, all of the shortest possible paths of length $d + 1$ are discovered and the IDs of the nodes along such paths are considered. The algorithm uses $d + 1$ rounds of broadcasting messages to neighbors in order for each node to learn about all of the shortest paths to all of the nodes that are at a distance $d + 1$ from it. The method guarantees that between any two nodes separated by a distance $d + 1$, there will be at least one d -hop dominating node along a shortest path connecting these. The details are discussed in [6].

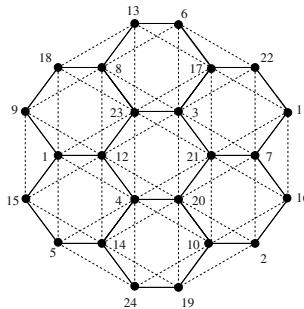


Fig. 1. Example graph representing an ad hoc network

3 An Example

The example that will be considered involves the graph in Fig. 1. The nodes here together with the solid edges constitute the graph G . By adding to this the dashed edges, the graph G_2 is obtained. For the reader familiar with the Wu-Li algorithm, it is straightforward, though somewhat laborious to check that the set resulting from the Wu-Li algorithm for this example is $\{ 1, 3, 4, 7, 8, 9, 10, 12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24 \}$.

Next, we consider applying the d -SPR-I algorithm to the graph G , using $d = 2$. The resulting set in this case consists of all the nodes whose ID is greater than or equal to 9. Seeing this requires a details inspection of all the pairs of nodes x and y that are a distance 3 apart in G . For example, consider nodes 20

and 23. Along shortest paths connecting these we encounter the nodes 3, 4, 12 and 21. Since 21 is the largest of these, it is “elected” to be in the backbone set. Likewise, between 15 and 18, the only nodes are 1 and 9. So 9 is also elected to join the set. The backbone set produced in this way is { 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24 }. Again, the reader should consult [6] for the details of the *d-SPR-I* algorithm.

By means of this example, we will now provide a preliminary sketch of the two algorithms to be described in detail later. The reader may wish to recheck the details in this example after reading Section 5. The *2-SPR-C* algorithm involves a consideration of the same pairs of nodes as in *2-SPR-I*, and the nodes that lie in between along shortest paths. So for example, consider the pair of nodes {1, 14}. The nodes that lie between 1 and 14 along a shortest path are 4, 5, 12 and 15. We say that these nodes “cover” the pair {1, 14}. The *2-SPR-I* algorithm would elect 15 to be a backbone node, since 15 is larger than 4, 5 and 12. The *2-SPR-C* algorithm however makes its decision primarily based on “covering number”. One checks that node 15 only covers three pairs, namely {1, 14}, {5, 9}, {5, 12}. Likewise, node 5 has covering number three, but nodes 4 and 12 have covering number twelve. So 4 and 12 have higher “priority” than 5 and 15. To break the covering number tie between nodes 4 and 12, ID is used as in *2-SPR-I*. Thus node 12 is elected to be in the backbone, guaranteeing at least one backbone node between nodes 1 and 14. A careful check reveals that the backbone set produced by *2-SPR-C* is { 1, 3, 4, 7, 8, 10, 12, 14, 17, 20, 21, 23 }.

When *2-SPR-G* is applied to this graph, the set produced is {1, 3, 7, 8, 10, 12, 14, 17, 20 }. The interested reader is encouraged to construct the bipartite graph H described in subsection 3.2. The set A consists of the nodes of the graph G . The set B consists of pairs of nodes that are a distance three apart in G , such as {1, 14}. A node from A is connected to a pair from B if the node covers the pair, as for example, 15 covers {1, 14}. The greedy algorithm is then applied to produce a small subset of A that cover, namely the one specified above.

4 Framework for Shortest Path Routing

4.1 d-SPR Sets

Recall the following standard terminology from graph theory: A node u in G is said to have *eccentricity* $e(u)$ if G has a node v such that $\delta(u, v) = e(u)$, and for all nodes w in G , $\delta(u, w) \leq e(u)$. The *radius* of G , $r(G)$, is the minimum of the eccentricities of its nodes. At the heart of our methodology is the following non-standard definition:

Definition 2. A set S of nodes will be called a *d-SPR set* if given any pair of nodes u and v of G such that $\delta(u, v) = d + 1$, there exists a $w \in S$ with $\delta(u, w) + \delta(w, v) = d + 1$ and $w \neq u, w \neq v$. Here *d-SPR* stands for ‘*d*-shortest path routing’.

In the next subsection, we will begin to consider the problem of finding small *d-SPR* sets. First, let us establish certain properties associated with such a set.

Theorem 1. Assume that the connected graph G has radius at least $d+1$. Then any d -SPR set has the following properties:

1. It is d -hop dominating
2. It is d -hop connected
3. It has the d -shortest path property

Conversely, a set of nodes with the d -shortest path property is a d -SPR set.

Proof. Fix a d -SPR set S . Consider a node x in G . Since G has a radius of at least $d+1$, there exists a node $y \in V$ at a distance $d+1$ from x . By definition of d -SPR, there exists a node w in S such that $\delta(x, w) + \delta(w, y) = d+1$, $w \neq x, w \neq y$, and so w is within a distance d of x . Hence S is d -hop dominating.

To show the d -shortest path property, fix any two nodes u and v . Let p be a shortest path in G from u to v . Let u_j denote the node arrived at after taking j steps along this path. ($j = 0, 1, 2, \dots, m$, where $m = \delta(u, v)$). If $m \leq d$, then there is nothing to prove. Suppose $m \geq d$. Consider the path between u_0 and u_{d+1} . Since they are $d+1$ distance apart, there exists a $w \in d$ -SPR such that there is a path p of length $d+1$ between u and u_{d+1} . Now create another shortest path in G from u to v by replacing the original subpath from u to u_{d+1} with the path p . Now w lies in this new path from u to v . Repeat the procedure for the path from w to v . Continuing in this way, a suitable path will eventually be produced.

The proof of 2 follows from 3. To prove 3, let u and v both belong to S . So there exists a shortest path connecting them such that the set of nodes on this path and also in S , together with u and v , form a d -hop connected set. Conversely, let a set of nodes S have the d -shortest path property. Consider any pair of nodes u and v such that $\delta(u, v) = d+1$. Then there exists a shortest path connecting u and v as described in the definition of the d -shortest path property. Some node w along this path, and strictly between u and v must be an element of S . Clearly, $\delta(u, w) + \delta(w, v) = d+1$. Hence, S is a d -SPR set.

4.2 Formulation of the d -SPR Problem

Although the sets produced by the d -SPR-I algorithm have nice properties like the d -shortest path property, they are quite large in size compared to the sets produced by the algorithm of Wu and Li. This led the authors to search for new techniques for producing smaller sets having the d -shortest path property. As a result, it was observed that the problem of finding minimal d -SPR sets reduces to the well-known Set Covering Problem, which can be phrased in terms of a bipartite graph as follows: Consider a bipartite graph, whose nodes are separated into two independent sets: A and B . Assume that every node in B is adjacent to some node in A . Then the problem is to find a smallest possible subset C of A such that every node in B is adjacent to some node in C .

The problem of finding a minimal dominating set in an arbitrary graph can be cast in terms of the Set Covering Problem, as is well-known, and discussed for example in [3] and [5]. We apply the Set Covering Problem to our d -SPR problem, by constructing a bipartite graph H as follows: Let A be the set of all

nodes in G . Let B be the set of all pairs of nodes in G that are separated by a distance $d + 1$. For every pair of nodes $\{x, y\}$ in B , let $A_{\{x,y\}} = \{w \mid \delta(x, w) + \delta(y, w) = d + 1, w \neq x, w \neq y\}$. Clearly $A_{\{x,y\}} \subset A$. For every v in $A_{\{x,y\}}$, put an edge between v and $\{x, y\}$ in H .

From the definition, it is clear that a set $C \subset A$ is a d -SPR set if and only if C covers all of the elements of B in H . The problem of finding a minimal such C is likely to be NP-hard, since the class of all possible bipartite graphs that could result from the above construction appears to be quite varied. Rather than focusing on this problem, we instead seek methods for producing reasonably small d -SPR sets, ones that can be computed in reasonable time. The following terminology will prove useful.

Definition 3. The *covering number* of each node w in A is the number of $\{x, y\}$ pairs in B that have an edge incident on it in H . Also, we say that w *covers* the pair $\{x, y\}$.

4.3 Local Views

In anticipation of our discussion in the next section of practical distributed algorithms for producing small d -SPR sets, the following notion will be needed:

Definition 4. The *d -local view* of a node v consists of all the d -hop neighbors of v , together with all edges between these, except for the edges that connect two nodes at a distance d from v .

By means of $d + 1$ rounds of broadcasting messages to neighbors, the nodes of the network can obtain all of the link-state information necessary for each to produce and maintain an internal representation of its own $d + 1$ -local view. This will also allow each node to realize the vertices of the bipartite graph H that are within two hops of it, as we will now see.

Lemma 1. For any nodes x, y, u, v in G such that $\delta(x, u) + \delta(y, u) \leq d + 1$ and $\delta(x, v) + \delta(y, v) \leq d + 1$, it follows that $\delta(u, v) \leq d + 1$.

Proof. We have

$$\delta(x, u) + \delta(u, y) + \delta(y, v) + \delta(v, x) \leq 2(d + 1)$$

By the triangle inequality, we obtain

$$\delta(u, v) \leq \delta(u, x) + \delta(x, v) \text{ and } \delta(u, v) \leq \delta(u, y) + \delta(y, v)$$

Adding the last two equations yields

$$2\delta(u, v) \leq \delta(u, x) + \delta(x, v) + \delta(u, y) + \delta(y, v)$$

Therefore,

$$\delta(u, v) \leq d + 1.$$

Theorem 2. *For any node x, y, v in G satisfying $\delta(x, v) + \delta(y, v) = d + 1$, $\delta(x, y)$ can be computed based solely on the $(d + 1)$ -local view of v . Moreover, all the shortest paths connecting x to y lie inside the $(d + 1)$ -view of v .*

Proof. The first claim follows immediately from the second. To prove the second claim, consider any shortest path p between x and y . Let w be any node on p . Then we have $\delta(x, w) + \delta(y, w) \leq d + 1$ and $\delta(x, v) + \delta(y, v) \leq d + 1$. It follows from Lemma 1 that $\delta(w, v) \leq d + 1$. So all the nodes along shortest paths connecting x to y lie inside the $(d + 1)$ -local view of v .

From the definition of $(d+1)$ -local view, the edges along such paths also appear in v' s $(d + 1)$ -local view, except possibly those which connect two nodes at a distance $d+1$ from v . However, such edges are impossible. To see this, assume such an edge exists and connects two nodes u and w , with u closer to x and w closer to y . Assume that this is part of a shortest path p connecting x and y . Then $\delta(v, x) + \delta(x, u) \geq \delta(v, u) = d + 1$ and $\delta(v, y) + \delta(y, w) \geq \delta(v, w) = d + 1$. But $\delta(v, x) + \delta(v, y) = d + 1$ and $\text{length}(p) = \delta(x, u) + \delta(u, w) + \delta(w, y) = \delta(x, u) + 1 + \delta(w, y) \leq d + 1$. This quickly leads to a contradiction.

We obtain the following immediate corollary, which just reinterprets a special case of Theorem 2 in a different context.

Corollary 1. *Given $u, v \in A$ (i.e. nodes in G), and a pair $\{x, y\} \in B$ such that u and v are both adjacent to $\{x, y\}$ in H , the four nodes u, v, x, y , as nodes of G , are within a distance $d + 1$ of each other.*

5 Some New Algorithms

5.1 d-SPR-I Based on Covering Number as Priority (d-SPR-C)

Our goal now is to design a variant of d -SPR-I algorithm that will produce a set whose size is as close to optimal as possible, but without sacrificing the “constant-time property” discussed in [6]. By this we mean that the time required to produce the set is unaffected by the size of the network as long as the node degrees are bounded. Even if the degrees are allowed to increase without bound too, this property is still highly desirable, because information used in the selection process is not required to propagate beyond a constant number of hops.

We begin by considering a slightly different approach to the implementation of the d -SPR-I algorithm presented in [6]. In the language of the bipartite graph H introduced in the previous section, d -SPR-I works by arranging for each pair $\{x, y\}$ in B to “elect” the node in A that covers it and that has the highest ID among all such nodes. In the original implementation, the nodes x and y discover which node they together should elect, and so informed this node that it now belongs to the d -SPR set being generated. An alternative implementation begins with $d + 1$ rounds of message passing, so that each node is able to learn about its own $(d + 1)$ -local view. Corollary 1 now implies that each node is in

a position to decide for itself whether or not it should join the $d\text{-}SPR$ set. It decides to do so if it discovers a pair $\{x, y\}$ (in B) that it covers for which it has the highest priority among all the nodes that cover $\{x, y\}$.

$d\text{-}SPR\text{-}C$ can now be understood as a variation of this approach. It requires an additional $d + 1$ rounds of message passing. After the first $d + 1$ rounds, each node is aware of its own $(d+1)$ -local view, and is able to compute its own covering number. The subsequent $d + 1$ rounds of message passing are used to allow each node to transmit its covering number to each of its $(d+1)$ -hop neighbors. In the $d\text{-}SPR\text{-}I$ algorithm, given a pair $\{x, y\}$ (from B), the node that covers $\{x, y\}$ with the highest ID is selected for inclusion into the $d\text{-}SPR$ set. Here, instead the node having the highest *priority* is selected. The priority a node is defined to be the ordered pair of numbers (covering number, ID), lexicographically ordered. This is analogous to the approach taken in [3, Subsection 2.1].

5.2 The Greedy d-SPR Algorithm (d-SPR-G)

At the present time, the closest we have been able to get to a minimal $d\text{-}SPR$ set is via the greedy algorithm, which we implement in a distributed manner. Here each node begins in an “undecided state”, maintains information about its $(d + 1)$ -local view, and executes the following steps. From Corollary 1 we know that a node v can “see” all the node pairs $\{x, y\}$ it covers, and also the rest of the nodes that cover such pairs. The fact that the distributed algorithm below terminates and that the set of “selected” nodes is a $d\text{-}SPR$ set follows immediately from the discussion of similar distributed greedy algorithms, as found in [3] and [5]. All of these, including ours, are in essence just distributed implementations of the greedy algorithm for the Set Covering Problem.

Step 1: Each node v gathers information about its $(d + 1)$ -local view. This requires $d + 1$ rounds of message passing. Let B_v denote all the node pairs $\{x, y\}$ covered by v . Let C_v be the set of all nodes that cover some node pair in B_v . (So $v \in C_v$.)

Step 2: v computes its current covering number $|B_v|$ and sends this information, along with its status (undecided, selected or not selected), in a message to each node in C_v . (Note that the first time this step is executed, all nodes are undecided, and the last time a node executes this step, it will be in one of the two decided states.)

Step 3: If v has entered one of the two decided states (selected or not selected), then it essentially terminates its participation in this algorithm, except to help route messages between other nodes. Otherwise, if it is still undecided, then

Step 4: v waits until it receives messages as in Step 2 from each node in C_v . For each such node u that has become decided, v removes u from C_v , and if u has become selected, v also removes any pairs from B_v that u covers. Accordingly, v recomputes its covering number as necessary.

Step 5: If v ’s covering number is now zero, then v enters the “not selected” state, and loops back to Step 2. Otherwise....

Step 6: v checks to see if its own priority is the highest among all the nodes of C_v . If it is, then v enters the “selected” state. In either case, it loops back to Step 2.

5.3 Bound on the d-SPR Set Size

We are able to obtain an upper bound of the set produced by the d -SPR-G algorithm.

Theorem 3. *For any a in A , let B_a be the set of pairs that a covers. For any b in B , let A_b be the set of nodes that cover b . Also, let α denote the minimum value of $|A_b|$, and let β denote the maximum value of $|B_a|$. Let C be the subset of A produced by d -SPR-G. For positive integers j , let $H(j) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{j}$. Then*

$$|C| \leq \left(\frac{|A|}{\alpha}\right)(1 + \ln(\alpha \frac{|B|}{|A|})H(\beta)).$$

Proof. Let O denote the optimal (minimal) d -SPR set. From [2] and [4], we have the well-known theorem of Johnson and Lovász: $|C| \leq H(\beta)|O|$. We now argue similar to Theorem 1.2.2 in [1]. Select a subset X of A randomly by including each a in A with a fixed probability p (independent events). The expected size of X is therefore $E(X) = |A|p$. Let $Y \subset B$ consist of all b not covered by X . For b in B , $Pr(b \in Y) \leq (1-p)^\alpha$, since each of at least α elements of A cannot be in X if b is in Y . It follows that: $E(|X| + |Y|) \leq |A|p + |B|(1-p)^\alpha \leq |A|p + |B|e^{-\alpha p}$. Taking p to be $(\frac{1}{\alpha}) \ln(\alpha \frac{|B|}{|A|})$ here yields the bound $E(|X| + |Y|) \leq (\frac{|A|}{\alpha})(1 + \ln(\alpha \frac{|B|}{|A|}))$. Now, there must be some subset X for which $|X| + |Y|$ is less than or equal to the expected value. But this X can clearly be extended to a covering set (d -SPR set) of size less than or equal to $|X| + |Y|$. Therefore, there exists a d -SPR set of size less than or equal to $(\frac{|A|}{\alpha})(1 + \ln(\alpha \frac{|B|}{|A|}))$. So this is an upper bound on $|O|$, and the desired result now follows.

6 Performance Evaluation of the Algorithms

6.1 Methodology

We implemented the d -SPR-I, d -SPR-C and d -SPR-G algorithms, and compared these with the Wu-Li algorithm applied to G_d . The implementation was run on a single machine while simulating the distributed nature of the algorithms. Each node gathers the information it needs from its neighboring nodes and declares its results. The above mentioned algorithms produced d -hop connected d -hop dominating sets. We compared the size of the dominating sets generated by each of these algorithms. Note that all these algorithms, except d -SPR-G, are constant-time.

For each experiment, a random disk graph was generated and measurements were taken on it. A disk graph is a graph in which a node is connected to all other nodes within a prescribed geometric radius. This radius can be regarded

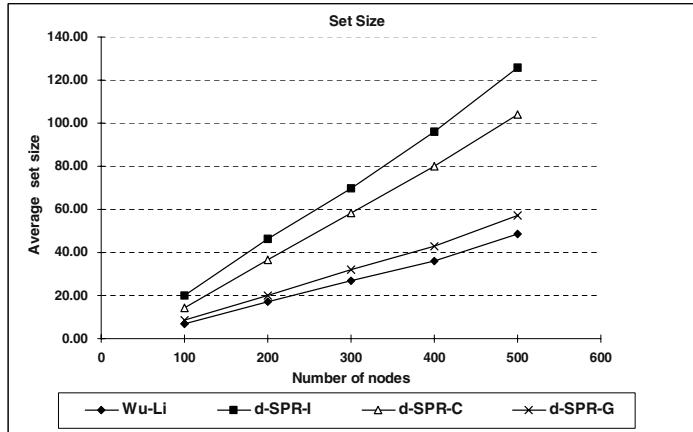


Fig. 2. Set size for $d = 5$

as the coverage radius of a wireless link in the ad-hoc network. A random disk graph with n nodes was created by selecting random points in a 900 by 900 pixel 2-D region. We ran the experiments on graphs with varying number of nodes to compare different algorithms for producing d -hop connected d -hop dominating sets.

6.2 Results

Overall, the d -SPR-G algorithm performed quite well compared to others in terms of set size for sets having the d -shortest path property. All of our methods produce sets larger than that produced by the Wu-Li algorithm though. This is reasonable since our algorithms need to add more nodes into the set to ensure the d -shortest path property. Fig. 2 shows the average size of the sets produced by each algorithm. We computed the sets consisting of nodes 100, 200, 300, 400 and 500 for $d = 5$ and transmission radius 100. Similar results were obtained using $d = 6$ instead. The sets produced by the d -SPR-G are only slightly bigger than the sets produced by the Wu-Li algorithm. The d -SPR-C is also a significant improvement over the d -SPR-I algorithm from the set size perspective.

However, d -SPR-C requires $2(d + 1)$ rounds of message passing whereas d -SPR-I requires only $d + 1$ rounds. The Wu-Li algorithm requires $2d$ rounds of message passing and is expensive in comparison to d -SPR-I. Of course, d -SPR-G involves the greatest number of overhead messages. Unlike the other methods, each node broadcasts a number of messages that is not simply a (linear) function of d , but instead depends on the size of the network. The time complexity for the distributed greedy algorithm in the case of d -SPR-G appears to be the same as in [5], which essentially only differs from d -SPR-G in that a somewhat different covering problem is addressed, and which [3] indicates is polynomial in the size of the network.

From Fig. 3, it is evident that d -SPR-G is substantially more expensive than d -SPR-I. As indicated already, this is to be expected since d -SPR-G involved multiple iterations of the basic $(d+1)$ -local view information flooding, produced by $d+1$ rounds of message passing on the part of each node. After each iteration, some nodes will drop out of active participation in the algorithm, having become either “selected” or “not selected”, but even these may need to continue to participate in forwarding messages for a while. Further iterations of $(d+1)$ -local view flooding are required until every node has becoming either “selected” or “not selected”. The cost ratio between d -SPR-G and d -SPR-I is bounded above by the number of such iterations.

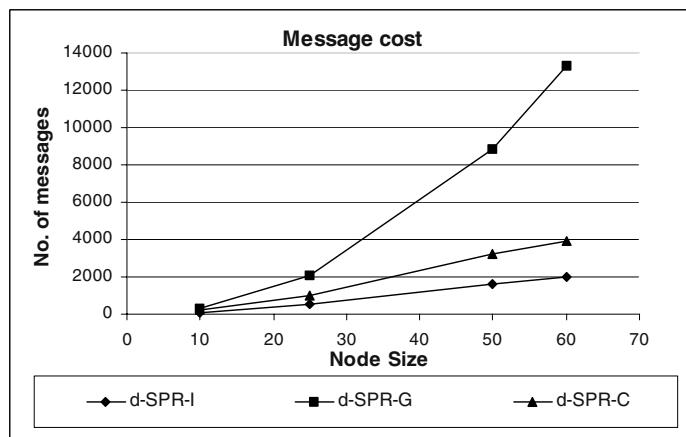


Fig. 3. Message cost for $d = 3$

In order to further reduce the size of the set produced by our algorithms, we are currently exploring some distributed greedy refinements of these proposed algorithms as well as some distributed probabilistic algorithms.

Acknowledgements. We wish to acknowledge the assistance of Eun Jik Kim for various support activities in connection with our investigations.

References

1. N. Alon and J. Spencer. *The Probabilistic Method*, John Wiley & Sons, 2000.
2. D. Johnson. Approximation Algorithms for Combinatorial Problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
3. L. Jia, R. Rajaraman, T. Suel. An Efficient Distributed Algorithm for Constructing Small Dominating Sets. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pp 33–42, August 2001.

4. L. Lovász. On the Ratio of Optimal Integral and Fractional Covers. *Discrete Mathematics*, 13:383–390, 1975.
5. B. Liang and Z.J. Haas. Virtual Backbone Generation and Maintenance in Ad Hoc Network Mobility Management. *Proc. 19th Ann. Joint Conf. IEEE Computer and Comm. Soc. INFOCOM*, Vol. 3, pp. 1293–1302, 2000.
6. M. Q. Rieck, S. Pai, S. Dhar, Distributed Routing Algorithms for Wireless Ad Hoc Networks Using d-hop Connected d-hop Dominating Sets. *Proceedings of the 6th Intl Conference on High Performance Computing in Asia Pacific*, Vol 2, pp 443–450, 2003.
7. J. Wu and H. Li. A Dominating-Set-Based Routing Scheme in Ad Hoc Wireless Networks. *Special issue on Wireless Networks in the Telecommunication Systems Journal*, Vol. 3, 2001, 63–84.

A Hierarchical Routing Method for Load-Balancing

Sangman Bak

3G Core Network Development Team
KTF
KTF Bldg 18th fl. 1321-11 SeoCho-Dong, SeoCho-Gu
Seoul, Korea
smbak@ktf.com

Abstract. The purpose of routing protocols in a computer network is to maximize network throughput. Shortest-path routing protocols have the disadvantage of causing bottlenecks due to their single-path routing. The shortest path between a source and a destination may become highly congested even when many other paths have low utilization. In this paper, we propose a routing scheme for hierarchically structured computer networks. The proposed hierarchical routing algorithm balances traffic load via unequal paths over the whole network; therefore, it removes bottlenecks and increases network throughput. For each data message to be sent from a source s to a destination d , the proposed routing protocol chooses each of three intermediate nodes from a selected set of network nodes, and routes the data message along a path from s through the three intermediate nodes to d . This increases the effective bandwidth between each pair of nodes.

1 Introduction

In any wide-area store-and-forward computer network, such as the Internet, routing protocols are essential. They are responsible for finding an efficient path between any pair of source and destination nodes in the network, and routing data messages along this path. The path must be chosen so that message delay, message loss and other undesirable events are minimized.

In the past, most of the works in network routing have been focused on Shortest-path routing protocols. Unfortunately, these Shortest-path routing protocols suffer performance degradation because all data messages are routed via the same shortest path to the destination until the routing tables have been updated. The problem with these routing protocols is that there are no mechanisms for altering the routing other than updating the routing tables. Routing table updates occur too slowly to respond to significant traffic fluctuations. Furthermore, increasing the frequency of routing table updates leads to unstable network behavior and an increase in the network load due to routing state messages [7, 19].

Note that the shortest path may be highly congested, even when many other paths to the destination are not congested. This congestion may trigger the loss of valuable messages due to buffer overflow at some node [29]. Moreover, using the same path to the destination limits the maximum throughput possible between the source and the

destination to be at most the minimum capacity of any link along the shortest path from the source to the destination.

Maximizing network throughput is an important goal in the design of routing algorithms and networks. A result in network flow theory, known as the max-flow min-cut theorem [8], shows that distributing the traffic load over the available paths between a source and a destination in the network, instead of using only one path of the minimum cost, increases the maximum possible throughput up to the capacity of the minimum cut separating these two nodes.

For example, let's consider Figure 1. The number beside each link represents its capacity. Suppose that node **a** (source) wants to send the data messages to node **f** (destination). Suppose that we use the hop count in order to calculate the cost of a path in the network. Then the effective bandwidth (i.e. the capacity of the minimum cut) between node **a** and node **f** is 30, while the effective bandwidth of the shortest path (**a-h-g-f**) from node **a** to node **f** is 5.

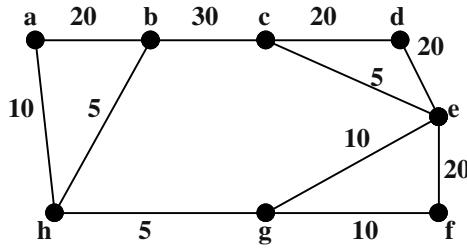


Fig. 1. Network topology

So far, several multiple-path routing (unequal-path routing) techniques have been proposed to increase the effective bandwidth between each pair of nodes and to attempt thereby to improve performance [2, 9, 19, 24, 26, 27, 28, 29]. These routing protocols improve performance by routing data messages via multiple paths to the destination. They maintain a stable routing table and meanwhile provide alternate paths to distribute data traffic when it is accumulating in some nodes of the network. When congestion and network failures do occur, instead of initiating route updating, the source temporarily forwards the traffic along the alternate paths and passes around the congested areas or failed areas. If the changes are persistent, the routing tables will be updated eventually when the next route updating time is due. Some multiple-path routing techniques are Shortest Path First with Emergency Exits [28] based on link-state routing [17, 18], Multiple Disjoint Paths [24] based on distance-vector routing [12, 16, 23], and Dynamic Multi-path Routing [2] based on source routing [10]. But these techniques require considerable processing overhead, need non-negligible storage space, or significantly increase the complexity of the routing algorithms. Several randomized multiple-path routing schemes [6, 21, 25] have been proposed for regular network topologies of parallel computers, such as mesh, torus, and butterfly, but these schemes are not suitable for the Internet, which has an irregular network topology.

In a recent paper [4], we proposed a randomized multi-path (unequal path) routing scheme, which is called Load-Balanced Routing (LBR), to address the bottlenecks associated with Shortest-path routing protocols. The routing protocol was formulated for an IP (Internet Protocol) network with an irregular topology. We showed that the randomized scheme increased throughput in the network with a flat network addressing structure. However, the nodes in the Internet are organized into a hierarchical structure.

There have been several proposals for hierarchical routing [1, 5, 20, 22], which vary in the way in which the nodes are organized and the routing schemes are used. With very few exceptions, existing routing protocols for hierarchically structured Internet have been based on single-path routing protocols.

In this paper we extend LBR to a hierarchical routing scheme based on bounded randomization that improves the network throughput by distributing the traffic load over all useful and unequal paths to the destination in the network. In consequence, network throughput is increased and message loss is reduced.

The rest of this paper is organized as follows. Sections 2 and 3 present the distance-vector routing protocol and the load-balanced routing protocol, respectively. In Sections 4 and 5, we present the assumed two-level hierarchical network model and a hierarchical routing protocol, respectively, which the new hierarchical routing algorithm is based on. Section 6 presents the overview of the hierarchical load-balanced routing protocol, which we call the Hierarchical Load-Balanced Routing (HLBR). In Section 7, we present future work and draw conclusion.

2 Distance-Vector Routing Protocol

HLBR is based on a distance-vector routing protocol [22]. In a distance-vector routing algorithm, each node maintains a routing table and a distance vector, which contain, respectively, a preferred neighbor for the shortest path to each destination in the network and the distance of the path to the destination. Each node has incomplete knowledge of the network topology and knows only its neighboring nodes. From these neighbors, the node chooses the closest neighbor to each destination. At a stable state of the protocol, the path made by consecutive preferred neighbors for a given destination is the shortest path to the destination. Each node periodically sends its distance vector to each of its neighbors to inform it of any distance changes to any destinations. By using Dijkstra's shortest path algorithm [9], the node determines which neighbor is the closest to each destination by comparing the distance vectors of its neighbors.

3 Load-Balanced Routing

In this section, we sketch how LBR protocol routes data messages to the destination. LBR was designed for a network with a flat network structure [3]. Each node should forward data messages to its neighbors so that the cost of the path traversed by each data message is as small as possible, while at the same time attempting to distribute

these data messages evenly throughout the network to avoid congestion and increase network throughput. LBR is based on the distance-vector routing algorithm [23].

3.1 Simple Load-Balanced Routing Protocol

In this subsection, we sketch Simple Load-Balanced Routing protocol (SLBR).

Here is **the algorithm of SLBR**:

1. For each data packet to be sent from a source node s to a destination node d , an intermediate node e is randomly chosen among all the network nodes.
2. It routes the packet via the shortest-distance (or least-cost) path from s to e .
3. Then, it routes the packet via the shortest-distance (or least-cost) path from e to d .

This technique distributes the traffic load over many more the paths between a source and a destination in the network than a single-path routing scheme and increases the effective bandwidth up to the capacity of the minimum cut separating these two nodes, which is the upper bound on the available bandwidth between these two nodes [8].

3.2 Load-Balanced Routing Protocol

The protocol described in Subsection 2.1 has a shortcoming for pairs of nodes of short distance. It is possible that a data message is routed to the destination via a very long path, much longer than a shortest path from the source to the destination.

For example, in Figure 1, suppose that node **a** wants to send a data message to node **b** and chooses, at random, node **f** as the intermediate node. Then the proposed algorithm sends the data message to node **f** via the shortest path (**a-h-g-f**) and then sends it to node **b** via the shortest path (**f-e-c-b**). Although there is the path of length 1 between node **a** and node **b**, the SLBR may use a path of length 6. Clearly, routing paths that are excessively long will waste network resources.

To remedy the problem, we introduce a parameter k , in order to exclude nodes from being candidates for an intermediate node that are too far away from the destination node. The set of candidates is restricted to all the nodes whose distance from the destination is at most k .

Choosing an appropriate value for k is crucial for the performance of the algorithm. Choosing too small a value may exclude many nodes that are far away from the destination from being candidates for an intermediate node, but it will increase the likelihood of a bottleneck. On the other hand, choosing too large a value may waste network resources by routing packets via excessively long paths, but it will increase the effective bandwidth up to the capacity of the minimum cut separating each pair of nodes. To reach a compromise between these two extremes, the parameter k may be chosen to be the average of the distance to each node reachable to the destination (LBR-BR1-D) [4]:

$$\bullet k = \frac{1}{n} \sum_{i=1}^n dist(d, d_i)$$

Formula1. d is the destination node and d_i is a node in the network.

This value is a constant for the destination d , since each link is considered to have a cost of 1. This value, however, has shortcomings. It limits the effective bandwidth between each pair of the nodes in the network to less than the capacity of the minimum cut separating the pair. The static value of k is too strong a restriction for a pair of nodes with a long path length and too weak a restriction for a pair of nodes with a short path length.

To remedy this problem of the static value for the parameter k , we may choose the value of the parameter k more intelligent to be fair to all the pairs of network nodes and consider a dynamic choice of parameter k (LBR-BR2-D):

$$\bullet k = \text{dist}(s, d) * \frac{\text{MAX}(\text{dist}(d, d_i)) - 1}{\text{MAX}(\text{dist}(d, d_i))}$$

Formula2. d_i is a node in the network, s is the source node and d is the destination node.

This value of the parameter k dynamically changes according to the length of the shortest path from the source node s to the destination node d .

4 Two-Level Hierarchical Network Model

In this section, we describe a two-level hierarchical network model, which we assume for the proposed routing protocol. The nodes of a network are clustered into m regions. Each region has at most n nodes. All the nodes that are within the same region are called *local nodes*. Nodes that have links to nodes in other regions are called *gateways*. Two regions are *neighbors* when there is one or more links between nodes in the two regions. In the network, each region is connected in the following sense. For any two nodes p and q in region i , there is a sequence of nodes $p.0, \dots, p.r$ in region i such that p is $p.0$, q is $p.r$, and for every k , $0 \leq k < r$, $p.k$ and $p.(k+1)$ are neighbors in the network. Each node in this network can be uniquely identified by two identifiers i and j . Identifier i indicates the region to which the node belongs. Identifier j indicates the node in region i .

In the proposed protocol, every node keeps its routing table. The routing table of each gateway consists of the following two small tables named *prs* and *rgn*:

- 1) *prs* is a local routing table with n entries, where n is the maximum number of nodes in the local region. Each entry j represents the cost of the path to the local node j and a preferred neighbor node along the path.
- 2) *rgn* is a global routing table with m entries, in which m is the maximum number of regions in the network. Each entry contains the cost of the path to a destination region and a preferred neighbor node along the path to the destination region.

Each local node keeps only its local routing table (*prs*).

5 Hierarchical Routing Protocol

In this section, we present in brief a hierarchical routing protocol, on which the proposed routing protocol is based.

In a network, when a data message is to be sent, the two identifiers (dr, dp) of its destination node are attached to the message before the message is sent. When a data (dr, dp) message arrives at a node, the node uses its routing table and the index (dr, dp), which identifies the message destination, to determine the preferred neighbor to which the message is forwarded.

In hierarchical routing, at least one node in each region is called a gateway of the region. The way a node $p[i, j]$ routes a data(dr, dp) message depends on whether $p[i, j]$ is a gateway of its own region i . When a data(dr, dp) message is received by node $p[i, j]$, the index (dr, dp) of the message destination is compared with the index (i, j) of receiving node. If $p[i, j]$ is not a gateway of region i and $dr \neq i$, then $p[i, j]$ forwards the message towards a gateway of region i , using its prs. If $p[i, j]$ is not a gateway of region i and $dr = i$, then the node $p[i, j]$ forwards the message towards its ultimate destination using its prs. If $p[i, j]$ is a gateway of region i and $dr \neq i$, then $p[i, j]$ forwards the message towards a gateway of region dr , using its rgn. If $p[i, j]$ is a gateway of region i and $dr = i$, then the node $p[i, j]$ forwards the message towards its ultimate destination using its prs.

6 Overview of Hierarchical Load-Balanced Routing Protocol

In this section, we sketch how our method, called Hierarchical Load-Balanced Routing protocol (HLBR) works. The HLBR protocol is a combination of the distance-vector routing algorithm, LBR and hierarchical routing protocol given in the previous sections.

6.1 Simple Hierarchical Load-Balanced Routing Protocol

In this subsection, we first present a Simple Hierarchical Load-Balanced Routing protocol (SHLBR).

Here is the algorithm of SHLBR:

1. For each data message to be sent from a source s to a destination d , SHLBR chooses randomly three intermediate nodes. The 1st intermediate node e_1 is chosen from the local nodes in the source region, the 2nd intermediate node e_2 is chosen from the gateways in the entire network and the 3rd intermediate node e_3 is chosen from the local nodes in the destination region.
2. It routes the data message along the shortest path from s to e_1 and then routes the data message along the shortest path from e_1 to the gateway in the source region.

3. It routes the data message via the shortest path from the gateway in the source region to e2 and then via the shortest path from e2 to the gateway in the destination region.
4. Finally it routes the data message via the shortest path from a gateway in the destination region to e3 and then via the shortest path from e3 to d.

As an example, consider Figure 2. Suppose that node a3 wants to send the data messages to node c3. For each data message to be sent from a3 to c3, SHLBR chooses randomly three intermediate nodes.

The 1st intermediate node (say a5) is chosen from the local nodes in source region A, the 2nd intermediate node (say b4) is chosen from the gateways in the entire network and the 3rd intermediate node (say c5) is chosen from the local nodes in the destination region. SHLBR routes the data message along the shortest path (a3-a5) from a3 to a5 and routes the data message along the shortest path (a5-a6) from a5 to the gateway a6 in the source region A. Then, it routes the data message via the shortest path (a6-b4) from a6 to b4 and then via the shortest path (b4-c1) from b4 to the gateway c1 in destination region C. Finally it routes the data message via the shortest path (c1-c5) from c1 to c5 and then via the shortest path (c5-c4-c3) from c5 to c3.

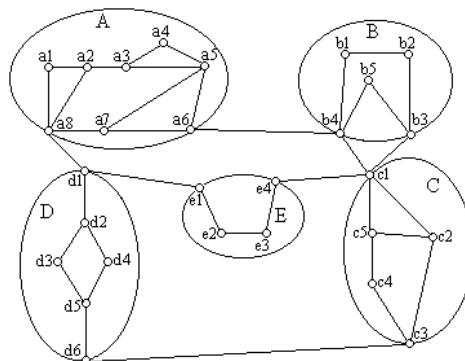


Fig. 2. Network topology

6.2 Bounding the Randomization of the Routing Path

The protocol of the previous subsection, however, has a weakness. It is possible that a data message is routed to the destination via a very long path, much longer than the shortest path between the source and the destination.

For example, in Figure 2, suppose that node a4 in the region A wants to send a data message to the node b1 in the region B and it randomly chooses nodes a1, e4 and b2 as the intermediate nodes. SHLBR routes the data message along the shortest path (a4-a3-a2-a1) from a4 to a1 and routes the data message along the shortest path (a1-a8) from a1 to the gateway a8 in the source region A. Then, it routes the data message via the shortest path (a8-d1-e1-e4) from a8 to e4 and then via the shortest path (e4-c1-

b3) from e4 to the gateway b3 in destination region B. Finally it routes the data message via the shortest path (b3-b2) from b3 to b2 and then via the shortest path (b2-b1) from b2 to b1. Although there is the path of length 4 between node a5 and node b1, SHLBR may use the path of length 11. Clearly, routing paths that are excessively long will waste network resources.

To remedy this problem, we introduce three parameters $k1$, $k2$ and $k3$, in order to exclude the nodes which are too far away from the gateway in the source region from being candidates for the 1st intermediate node, from the gateway in the destination region from being candidates for the 2nd intermediate node and from destination node from being candidates for the 3rd intermediate node, respectively. The set of candidates for the 1st intermediate node is restricted to all those nodes whose distance to the gateway in the source region is at most $k1$. The set of candidates for the 2nd intermediate node is restricted to all those gateways whose distance to the gateway in the destination region is at most $k2$. The set of candidates for the 3rd intermediate node is restricted to all those nodes whose distance to the destination node in the destination region is at most $k3$.

The values chosen for these $k1$, $k2$ and $k3$ affect delay, the traveled path length, load balancing and network throughput. Choosing small values will increase the likelihood of bottleneck. On the other hand, choosing large values will waste network resources by routing messages via excessively long paths. To reach a compromise between these two extremes, we may choose $k1$, $k2$ and $k3$ to be the average of the distance to each node in the source region reachable from the source node, the average of the distance to each gateway in the entire network reachable from the gateway in the source region, and the average of the distance to each node in the destination region reachable from the gateway in the destination region, respectively.

7 Future Work and Conclusion

In this paper, we introduced three parameters $k1$, $k2$ and $k3$, in order to exclude the nodes which are too far away from the gateway in the source region from being candidates for the 1st intermediate node, from the gateway in the destination region from being candidates for the 2nd intermediate node and from destination node from being candidates for the 3rd intermediate node, respectively. One possible direction is to perform simulations with different topologies, different source locations, and different traffic generation distributions, to determine the appropriate values of parameters $k1$, $k2$ and $k3$ for improving network performance.

We presented a hierarchical load-balanced routing protocol to distribute the data traffic via bounded randomization over all available paths to a destination in the network for load balancing. The proposed protocol will alleviate congestion and increase the effective bandwidth between each pair of nodes in the network. To accomplish this, each message needs to carry very little extra information: 3 intermediate nodes (e1, e2 and e3) and 3 routing status bits (bps, br, bpd). Moreover each of the 3 intermediate nodes is used at a different time during routing. They may share one space in the message. The required overhead is a space for one node and 3 bits for 3 routing status.

References

1. C. Alaettinoğlu and A. U. Shankar, The Viewserver Hierarchy for Interdomain Routing: Protocols and Evaluation, IEEE Journal on Selected Areas in Communications, Vol. 13, No. 8, pp. 1396–1410, Oct. 1995.
2. S. Bahk and, M. E. Zarki, Dynamic Multi-path Routing and How it Compares with other Dynamic Routing Algorithms for High Speed Wide Area Networks, Proceedings of the 1992 ACM SIGCOMM Conference, Vol. 22, Oct. 1992.
3. S. Bak and J. A. Cobb, Randomized Distance-Vector Routing Protocol, Proceedings of ACM Symposium on Applied Computing, pp. 78–84, San Antonio, Texas, Feb. 1999.
4. S. Bak, Load-Balanced Routing via Bounded Randomization based on Destination Node, Proceedings of the 13th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2001), Anaheim, California, USA, Aug. 2001.
5. L. Breslau and D. Estrin., Design of Inter-Administrative Domain Routing Protocols, Proc. ACM SIGCOMM '90, pp. 231-241, Philadelphia, Pennsylvania, Sep. 1990.
6. R. Cole, B.M. Maggs, F. Meyer auf der Heide, M. Mitzenmacher, A.W. Richa, K. Schroeder, R.K. Sitaraman, and B. Voecking, Randomized Protocols for low-congestion circuit routing in multistage interconnection networks, Proceedings of the 29th Annual ACM Symposium on the Theory of Computing, pp. 378–388, May 1998.
7. D. P. Bertsekas, Dynamic behavior of shortest path routing algorithms for communication networks, IEEE Trans. Automatic Control, AC-27, pp.60–74, 1982.
8. D. P. Bertsekas, Linear Network Optimization: Algorithms and Codes, The MIT Press, 1991.
9. J. A. Cobb and M. G. Gouda, Balanced Routing, IEEE Proceedings of the International Conference on Network Protocols, 1997.
10. E. W. Dijkstra, A Note on Two Problems in Connection with Graphs, Numerische Mathematik, Vol. 1, pp. 269–271, 1959.
11. R. C. Dixon, D. A. Pitt, Addressing, Bridging, and Source Routing (LAN interconnection), IEEE Network, Vol. 2, No. 1, Jan.1988.
12. J. J. Garcia-Luna-Aceves, A Minimum-Hop Routing Algorithm Based on Distributed Information, Computer Networks and ISDN Systems, Vol. 16, pp. 367-382, May 1989.
13. M. Gouda, Protocol verification made simple: a tutorial, Computer Networks and ISDN Systems, Vol. 25, pp. 969–980, 1993.
14. M. Gouda, The Elements of Network Protocol Design, A Wiley-Interscience Publication, John Wiley & Sons, Inc., 1998.
15. C. Huitema, Routing in the Internet, Prentice Hall, 1995.
16. G. Malkin, RIP Version 2, Internet Request for Comments 1723, Nov. 1994, Available from <http://www.ietf.cnri.reston.va.us>.
17. J.M. McQuillan, Ira Richer and E.C. Rosen, The New Routing Algorithm for the ARPANET, IEEE Trans. on Communications, Vol. COM-28, N0. 5, pp. 711–719, May 1980.
18. J. Moy, Ospf Version 2, Internet Request for Comments 1583, Mar. 1994. Available from <http://www.ietf.cnri.reston.va.us>.
19. S. Murthy and J.J. Garcia-Luna-Aceves, Congestion-Oriented Shortest Multipath Routing, Proc. IEEE INFOCOM 96, San Francisco, California, Mar. 1996.
20. S. Murthy and J.J. Garcia-Luna-Aceves, Loop-Free Internet Routing using Hierarchical Routing Trees, Proc. IEEE INFOCOM 97, Vol. 1, pp. 101–108, Apr. 1997.
21. T. Nesson and S. L. Johnsson, ROMM Routing on Mesh and Torus Networks, Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures, July 1995.
22. C. V. Ramamoorthy and W. Tsai, An adaptive hierarchical routing algorithm, Proc. IEEE COMPSAC 83, pp. 93–104, Chicago, 1983.

23. Segall and M. Sidi, A Failsafe Distributed Protocol for Minimum Delay Routing, IEEE Trans. on Commun., COM-29(5), May 1981.
24. D. Sidhu, R. Nair and S. Abdallah, Finding Disjoint Paths in Networks, Proceedings of the 1991 ACM SIGCOMM Conference, 1991.
25. L. G. Valiant, A Scheme for Fast Parallel Communication, SIAM Journal on Computing, Vol. 11, No. 2, May 1982.
26. S. Vutukury and J.J. Garcia-Luna-Aceves, SMART: A Scalable Multipath Architecture for Intra-domain QoS Provisioning, The proceedings of QoS-IP 2001, Rome, Italy, Jan. 24–26, 2001.
27. S. Vutukury and J.J. Garcia-Luna-Aceves, MDVA: A Distance-Vector Multipath Routing Protocol, Proceedings of IEEE Infocom2001, Apr. 22–26, 2001.
28. Z. Wang and J. Crowcroft, Shortest Path First with Emergency Exits, Proceedings of the 1990 ACM SIGCOMM Conference, 1990.
29. W.T. Zaumen and J.J. Garcia-Luna-Aceves, Loop-Free Multipath Routing Using Generalized Diffusing Computations, Proc. IEEE INFOCOM 98, San Francisco, California, Mar. 29, 1998.

Ring Based Routing Schemes for Load Distribution and Throughput Improvement in Multihop Cellular, Ad hoc, and Mesh Networks*

Gaurav Bhaya, B.S. Manoj, and C. Siva Ram Murthy

Department of Computer Science and Engineering, IIT Madras, INDIA.

gbhaya@dcs.iitm.ernet.in, bsmanoj@cs.iitm.ernet.in, murthy@iitm.ernet.in

Abstract. We propose novel schemes for distributing the load and improving the bandwidth utilization in Wireless networks. In a multihop wireless network with uniform node density and uniform traffic density, the shortest path routing seems to be an attractive alternative to any nonlinear path. However, in the shortest path routing schemes mobile nodes closer to the *Center* of the network lie on more shortest paths than nodes farther away and hence become points of contention, thus relaying more traffic and exhausting their limited battery power. This also leads to reduction in throughput in the region of increased contention. In this paper, we propose Ring Based Routing Schemes to distribute the load, close to the *Center* of the network, over the entire network. We also present theoretical analysis for load distribution and hop count of various schemes. In our simulations using GloMoSim we obtained a better load distribution over the network and an increased throughput of up to 25% to 30% as compared to shortest path routing at the cost of 4% to 8% increase in average hop length.

1 Introduction

With the increasing number of users, the wireless domain is always in a dearth for two main resources: bandwidth and battery power. Many attempts have been made to make the optimal use of the available bandwidth. The use of Multihop Cellular Networks [1,2] creates new issues in cellular networks, as tracing the route to the destination, and the nodes needed to relay the data for other nodes. These issues are naturally present in the Ad hoc networks and Mesh networks as well.

In order to ensure participation and availability of relaying nodes general incentive based pricing schemes are proposed in [3] and [4]. The idea behind these schemes is to motivate nodes relay traffic generated by other nodes either through monetary gains or through mutual benefit. Though pricing scheme attempts to solve the problem of participation [4,5] it does not address the problem of contention for some nodes that lie on many shortest paths, which once again

* This work was supported by Infosys Technologies Ltd., Bangalore, India and Department of Science and Technology, New Delhi, India.

leads to the problem of unfairness in terms of opportunity to relay. In a system where mobility is high this problem may not be very significant because mobility may cause mobile nodes in a more strategic location for relaying data to move to less favorable positions and vice-versa, thus on an average each node will relay the same amount of data.

Overloading of the mobile nodes closer to the *Center* of the network still remains an issue as the nodes lie on many shortest paths, more so in a network which is static or in which mobility rates are low. Some attempts have been made to resolve this problem by using schemes that dynamically determine the load on various nodes in the system and thus route new data through less loaded nodes. But, these schemes may have side effects such as, unbounded increase in path length, increased jitter, and toggling of load between heavily and lightly loaded regions.

Hence, our aim is to propose a scheme with the following objectives. Traffic relayed by the nodes in the network is not dependent on their position. Though this cannot be fully avoided, we would like to minimize the effect. The number of updates must be kept to the minimum. The number of hops taken to reach the destination must be bounded. Oscillation of load in the system must be avoided. Bandwidth utilization must be improved. With these objectives, we propose load distribution schemes in which the nodes follow a set of rules so that the load gets distributed over the network. Under a condition of uniform data generation rate the load on various nodes of the network is predictable.

The organization of the rest of this paper is as follows. Section 2 presents the load distribution schemes with the scenarios of their study. In Sections 3 and 4 we present a theoretical analysis for load distribution schemes and the average hop count, respectively. In Section 5 we discuss simulation results for the Multihop Cellular Network (MCN) architectures, Ad hoc wireless networks and the Wireless Mesh networks and study the performance. Finally, in Section 6 we summarize our work and present our conclusions.

2 Our Work

In this section we discuss three schemes for load distribution under three scenarios stated below.

MCN: The MCN was designed to provide spatial reuse of bandwidth. The MCN architectures proposed in [1,2] use single data channel of transmission range R/k (k is a constant) and a control channel of transmission range R .

Ad hoc Wireless Networks: Ad hoc wireless networks are highly dynamic and self organized and do not depend on infrastructure. These networks are generally set up on the fly, in situations such as disaster relief, military operations, and search and rescue operations. These networks are highly resource constrained and use multihop radio relaying mechanism to deliver packets to the receivers which are not reachable directly.

Wireless Mesh Networks: Wireless Mesh networks are a special category of Ad hoc wireless networks in which the multihop radio relaying mechanism is

used but with all nodes fixed, with some nodes acting as gateway to the wired Internet. Traffic is routed using multiple hops to any of the gateways which is connected to the Internet.

Definition 1: $Hops(a,b)$ is the number of hops along the shortest path from node a to node b .

Definition 2: The *Center node* or *Center* of a network, C is defined as the node for which,

$$\max_{\forall x} (Hops(C, x)) \leq \min_{\forall y} (\max_{\forall z} (Hops(y, z)))$$

Here we assume that no partitions exist in the network. In such cases, however, we may consider them as independent networks.

Definition 3: Each mobile node in the network belongs to a *Ring* denoted by $Ring_i(r_i, r_{i+1})$. A *Ring* is an imaginary division of the network into concentric rings about the *Center* of the network. The *thickness* of a *Ring* is given by $r_{i+1} - r_i$. A node belonging to *Ring* i lies at a *distance* in (r_i, r_{i+1}) from the *Center* of the network. We define the term *distance* as per convenience in the three kinds of wireless networks that we discuss.

2.1 Determination of *Center* and *Rings*

We now discuss some techniques that can be used for determining the *Center* and the division of *Rings* for the three multihop networks stated earlier.

MCNs: The base station which lies in the geographic center of the cell can serve as the ideal *Center* of the cell. Division into *Rings* is also straightforward in this case as the nodes update the base station over the control channel periodically, about their present neighbors. The base station can use the received power over the control channel or the GPS information to determine the approximate distance of the node from itself and thus divide the nodes into various *Rings*. The choice of number of *Rings* is left to the base station. Thus, the term *distance* in **Definition 3** in this case represents the geographical distance from the base. Algorithmic complexity does not increase in this case. No change is required at the mobile nodes.

Ad hoc Wireless Networks: We assume that the Fisheye State Routing [6] protocol used in this network conveys the global topology to the nodes in the network. Using this global topology information each node builds the connectivity matrix for the network and using Warshall's algorithm the *Center* can be determined in $O(n^3)$ time where n is the number of nodes. Tie can be resolved in favor of lesser *node id* or *IP address*. The *Rings* to which the nodes belong can be determined based on the hop distance (*Distance* in **Definition 3**) from the *Center*. No overhead in network traffic is caused due to the above.

Wireless Mesh Networks: Wireless Mesh Network being a kind of Ad hoc network the determination of *Rings* remains the same as in an Ad hoc wireless network.

2.2 Routing Schemes

The three schemes proposed for load distribution and throughput improvement are as follows:

1. Preferred Inner Ring Routing Scheme (PIRS).
2. Preferred Outer Ring Routing Scheme (PORS).
3. Preferred Destination Ring Routing Scheme (PDRS).

A small variation of the third is called the *Preferred Source Ring Routing Scheme (PSRS)*. Dijkstra's shortest path algorithm when applied to network to determine the path from source to the destination, with all edges of weight one, yields the shortest path between the source and the destination in terms of hop count. This is synchronous with our definition of *Hops*.

Preferred Inner Ring Routing Scheme -PIRS (Preferred Outer Ring Routing Scheme - PORS): *The traffic generated by Ring i for Ring j must not go beyond the Rings enclosed by Rings i and j. Further, the packet must be preferably routed through in the inner (outer) of the two Rings. In other words, for the nodes belonging to the same Ring the packet must be transmitted in the same Ring. For nodes belonging to different Rings the packet must go across the Rings in radial direction¹ and all angular transmissions must take place in the inner (outer) of the two Rings (of the source or the destination).*

As stated earlier, tracing route in this scheme only requires changing the weights of the edges in the adjacency matrix of the graph before running Dijkstra's algorithm as in Figure 1 and hence the overall complexity remains the same.

Preferred Destination Ring Routing Scheme (PDRS): The two schemes presented earlier make a trade off between increase in hop count and ability to reduce load closer to the center of the network. Hence, PDRS is the hybrid of the two schemes proposed earlier. The rule followed by PDRS (PSRS) is: *The traffic generated by Ring i for Ring j must not go beyond these Rings and, the Rings enclosed by these Rings. Further, the packet must be preferably transmitted in the Ring of destination (source) node. In other words, for the nodes belonging to the same Ring the packet must be transmitted in the same Ring. For nodes belonging to different Rings the packet must go across the Rings in radial direction and all angular transmissions must take place in the Ring of the destination (source) node.*

In different kinds of networks, some of the schemes may work better than others. The schemes essentially make a trade off between the increase in hop count and the ability to move the load away from the *Center* in a more uniform network. In case uniform distribution of nodes in network, the *PORS* scheme is expected to perform better as it does the maximum to move the traffic away from the *Center* to the periphery. But the hop count increase due to *PORS* may also cause increased delay, which may not be tolerable to many applications. The *PSRS* and the *PDRS* which appear very much similar may give varied

¹ To reduce the number of hops we may not go along the radial direction, but the condition forces the use of minimum number of nodes for transmission in the intermediate Rings.

Scheme Edge between	PIRS	PORS	PDRS	PSRS
Nodes in different Ring	MAX	MAX	MAX	MAX
Nodes in destination Ring	MAX I	I MAX	I	MAX
Nodes in source Ring	I MAX	MAX I	MAX	I
Nodes in other Ring	MAX	MAX	MAX	MAX

Fig. 1. Edge weight to be used in Dijkstra's algorithm for finding route. MAX represents some large number larger than maximum number of hops possible but less than unconnected. Some entries are divided into 2. The upper represents the case of destination *Ring* larger than the source *Ring* and the lower for destination *Ring* smaller than the source *Ring*.

performance in case of Wireless Mesh Networks where the destination may not be decided at random. Hence, *PDRS* may load only a few *Rings* containing gateways which is not the case with *PSRS*.

3 Theoretical Analysis: Load Distribution

In this section we present the load distribution for circular network of radius R units which has uniform node distribution. For brevity, we define the range of a *Ring* as $[a, b]$ where a and b correspond to the minimum and the maximum distance from the *Center* to any node belonging to this *Ring*. We consider a continuous model rather than a discrete model for the ease of evaluation. In our analysis we assume that all nodes generate data at the same rate to random destinations in the network (and the same cell in case of MCN).

3.1 Using Shortest Path

Figure 2 represents the load distribution Vs the distance from the *Center* in case of the shortest path routing scheme. The figure clearly highlights that a node closer to the base station relays a much larger traffic than a node farther away from the base. Also the base station gets saturated much earlier than any other node.

3.2 Preferred Outer Ring Routing Scheme

We define r as the mean distance of *Ring* i from the *Center* and α as the thickness of *Ring* i . Thus,

$$r = \frac{r_i + r_{i+1}}{2} \text{ and } \alpha = r_{i+1} - r_i \quad (1)$$

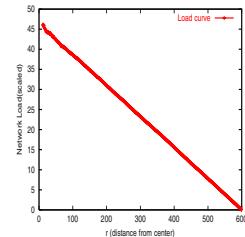


Fig. 2. Load distribution with shortest path routing.

We can derive the total load on individual nodes as:

$$\propto \frac{((r+\frac{\alpha}{2})^2 + (r-\frac{\alpha}{2})^2)}{\pi^2 R^4} \times (\pi r + \alpha - \frac{R^2}{r} - \frac{(r+\frac{\alpha}{2})^2}{r}) \quad (2)$$

PORS pushes most of the load from the center to the periphery of the network.

3.3 Preferred Inner Ring Routing Scheme

We perform an analysis similar to PORS in order to obtain, the load per node in *Ring i* proportional to:

$$\left(\frac{\pi r}{2} + \frac{\alpha}{2} + \frac{(r - \frac{\alpha}{2})^2}{2r} \right) \quad (3)$$

In this case the load close to the *Center* of the cell is pushed to the middle *Rings* of the network.

3.4 Preferred Destination (Source) Ring Routing Scheme

Once again a similar analysis leads us to: the load per node at a distance r from the base station is proportional to:

$$\frac{1}{\pi R^2} \left(\frac{\pi r}{2} + \alpha \right) + \frac{(r - \frac{\alpha}{2})^2 (\pi R^2 - \pi(r + \frac{\alpha}{2})^2)}{\pi^2 R^4 r} \quad (4)$$

4 Theoretical Analysis: Hop Count

Increase in the *hop count* bears a direct impact on the transmission delay associated with the data. Also with the increase in the *hop count* the overall load on the network increases. Hence, it may be important to study the increase in *hop count*. We shall once again consider the continuous model rather than the discrete one and hence analyze *Average Path Length* which represents the *average hop count* for the continuous model.

Shortest Path Scheme:

In the shortest path approach the average path length is the straight-line distance between the source and the destination. We use polar co-ordinates to represent the source and the destination as (r_1, θ_1) and (r_2, θ_2) , respectively. Since we are not interested in the absolute values of θ_1 and θ_2 , we define θ as,

$$\theta = |\theta_1 - \theta_2|, \quad 0 \leq \theta \leq \pi, \quad \text{and if } \theta > \pi \text{ then } \theta = 2\pi - \theta. \quad (5)$$

The average distance between source and destination is given by,

$$\frac{\int_0^R \int_0^R \int_0^\pi \sqrt{(r_1^2 + r_2^2 - 2r_1 r_2 \cos \theta)} dr_1 dr_2 d\theta}{\int_0^R \int_0^R \int_0^\pi dr_1 dr_2 d\theta} \approx 0.726R \quad (6)$$

Preferred Outer Ring Routing Scheme:

Path Length is given by, $\max(r_1, r_2)\theta + |r_1 - r_2|$.
Hence *Average Path Length* is given by,

$$\frac{\int_0^R \int_0^R \int_0^\pi (\max(r_1, r_2)\theta + |r_1 - r_2|) dr_1 dr_2 d\theta}{\int_0^R \int_0^R \int_0^\pi dr_1 dr_2 d\theta} = \frac{(\pi + 1)R}{3} \quad (7)$$

Preferred Inner Ring Routing Scheme:

Path Length is given by, $\min(r_1, r_2)\theta + |r_1 - r_2|$.
Hence *Average Path Length* is given by,

$$\frac{\int_0^R \int_0^R \int_0^\pi (\min(r_1, r_2)\theta + |r_1 - r_2|) dr_1 dr_2 d\theta}{\int_0^R \int_0^R \int_0^\pi dr_1 dr_2 d\theta} = \frac{(\frac{\pi}{2} + 1)R}{3} \quad (8)$$

Preferred Destination (Source) Ring Routing Scheme:

Path Length is given by, $r_2\theta + |r_1 - r_2|$.
Hence *Average Path Length* is given by,

$$\frac{\int_0^R \int_0^R \int_0^\pi (r_2\theta + |r_1 - r_2|) dr_1 dr_2 d\theta}{\int_0^R \int_0^R \int_0^\pi dr_1 dr_2 d\theta} = R \left(\frac{1}{6} + \frac{\pi}{4} \right) \quad (9)$$

5 Simulation Results

In order to study the performance of the load balancing schemes we simulated the system using GlomoSim [7]. The region of simulation is a hexagonal cell (cell radius = 500m) with the base station at the *Center* in case of the MCN architecture. In case of the Wireless Ad hoc and Mesh networks we used a circular region (of radius 500m) for simulation. All simulation results are averaged over 16 seed values. The MAC protocol used in all of the above cases is IEEE 802.11 [8]. We used total available system bandwidth as 6Mbps with a bandwidth of 1 Mbps allocated to the control interface. Also, the backbone routing protocol in case of wireless Ad hoc and Mesh network based simulation was Fisheye Routing protocol.

In case of wireless Ad hoc and Mesh networks two interfaces are provided, one for the data transmission and the other for topology updates. In case of the Mesh networks, the gateways are decided at random at the start of the simulation. In case of Mesh and Ad hoc networks the *Rings* are designated as follows. All nodes reachable from the *Center* in one hop are termed as *Ring 1*. All nodes not designated and reachable by nodes of *Ring 1* in one hop are termed as *Ring 2* and so on.

Cellular Networks: The node density and mobility are varied from 130 nodes/cell to 580 nodes/cell and a speed of 0 m/s to 10m/s, respectively. Figure 3 shows that the differences in the average number of hops for packet transmission is around 4% in case of *PIRS* and around 8% in case of *PDRS* and *PORS*. The x-axis represents the various schemes listed in Table 1. For a similar range

Table 1. Ring Thickness details for Figures 3 and 4 (for x-axis)

Instance	R1	R2	R3	R4	R5
1	100	100	100	100	100
2	110	105	100	95	90
3	120	110	100	90	80
4	130	115	100	85	70
5	140	120	100	80	60
6	60	80	100	120	140
7	70	85	100	115	130
8	80	90	100	110	120

Table 2. Ring Thickness for Figure 5 (x-axis)

Instance	Scheme	Ring Thickness (m)
1	PDRS	100 100 100 100 100
2	PDRS	60 80 100 120 140
3	PDRS	70 85 100 115 130
4	PDRS	80 90 100 110 120
5	PIRS	100 100 100 100 100
6	PIRS	60 80 100 120 140
7	PIRS	70 85 100 115 130
8	PIRS	80 90 100 110 120
9	PORS	100 100 100 100 100
10	PORS	60 80 100 120 140
11	PORS	70 85 100 115 130
12	PORS	80 90 100 110 120
13	Shortest Path	100 100 100 100 100

on x-axis, Figure 4 shows that throughput improvement in terms of packet delivery ratio is 13% in case of *PIRS* and 25% in case of *PDRS* and *PORS*. The results in case of varying mobility, presented in Figure 5 for the schemes in Table 2 show a throughput improvement of up to 33%. Figure 6 presents the load distribution obtained Vs the distance from the *Center* of the network. Table 3 shows that throughput obtained when all the data sent by the nodes in a cell is not directed to the destinations in the same cell with a probability of 0.5 is also significantly better than the shortest path scheme.

Ad hoc Wireless Networks: Figure 7 shows that the throughput in terms of packet delivery ratio of the *PORS* is significantly better than the shortest path routing though the gain reduces as the paths converge at high transmission range.

Wireless Mesh Networks: In case of the Wireless Mesh networks, three gateways are randomly placed over the entire network for the purpose of simulation. We use three heuristics in order to balance the load on the gateways: **Least Loaded Gateway (LLG)** or the gateway with least load, **Nearest Ring Gateway (NRG)** or the gateway in numerically closest *Ring*, or **Hybrid Approach Gateway (HAG)** or a probabilistic version of the previous two.

Figure 8 shows that the throughput for *PORS* is higher in case of lower transmission power while the shortest path scheme dominates in case of lower transmission power in the case of NRG. This can be attributed to the fact that the reuse factor plays an important role in case of the shortest path routing because of the overloading of some areas in the network; whereas in case of the *PORS* this effect is redundant as the reduction in hop count dominates.

Reasons for Throughput Improvement: Normally, an increase in hop length results in decrease in throughput which contradicts our simulation results. In this section we discuss the reasons behind the performance improvement. In the presence of high load the region near the *Center* lies on more shortest path and hence any packet loss at the *Center* causes more nodes to get affected than the packet loss in any other region of the cell. Also, the queue length at the nodes

near the *Center* grows faster than the nodes near the periphery causing loss of packets due to overflow of queues. Further, the number of collisions also increases as the number of nodes contending for the channel increases. Though the number of nodes in the system is the same the number of nodes in the hot area which have packets for transmission is larger. The number of collisions observed from simulations were found to be less by 4% in case of *PORS* as compared to the shortest path.

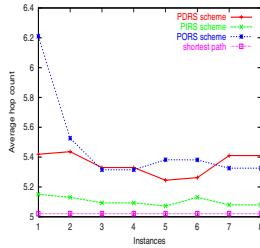


Fig. 3. Average hop count for nodes=130, transmission range=150m, no mobility. For instances refer to Table 1.

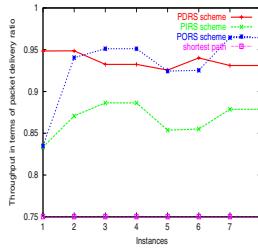


Fig. 4. Throughput comparison for nodes=130, transmission range=150m, no mobility.

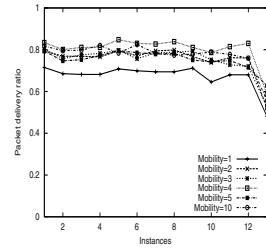


Fig. 5. Throughput comparison for nodes=130, transmission range=200m, mobility variable.

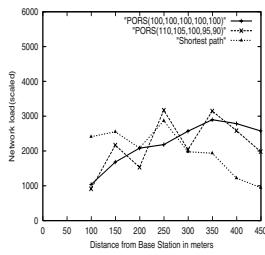


Fig. 6. Load distribution Vs distance for nodes=130, transmission range=150m, no mobility.

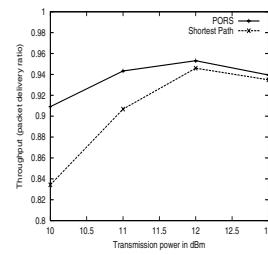


Fig. 7. Ad hoc Networks: Throughput comparison for nodes=104, transmission range=variable, no mobility.

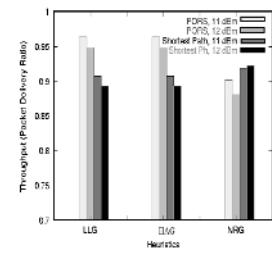


Fig. 8. Throughput for Mesh Network, Nodes=128, Mobility=none.

6 Summary and Conclusions

In this paper we consider the problem of hot spot creation near the base station in Multihop Cellular Networks (MCNs), and the region close to the *Center* of

Table 3. Throughput in terms of Packet Delivery Ratio for no mobility, transmission range=150m, Node Density=104 nodes/cell, locality=0.5

Scheme	Rings (meters)	Throughput
PDRS	100,100,100,100,100	0.542202993577513
PDRS	120,110,100,90,80	0.499960073464825
PIRS	100,100,100,100,100	0.564197530864198
PIRS	120,110,100,90,80	0.531323129892137
PORS	100,100,100,100,100	0.530966077092629
PORS	120,110,100,90,80	0.494235347617901
Shortest Path	100,100,100,100,100	0.486180165731088

the network in the case of wireless Ad hoc and Mesh networks. This occurs even in the case of uniform data generation rate and uniform node distribution. We proposed *Ring* based routing schemes to distribute the load over the entire network from hot areas in a cell. Our methods attempt to distribute the load over the entire network. By doing this we achieve fairness in terms of number of packets relayed by every node. We obtained a throughput improvement of 25% to 30% and a more balanced load distribution over the entire network.

References

- Y. D. Lin and Y. C. Hsu, "Multihop Cellular: A New Architecture for Wireless Communications", in *Proc. IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.
- R. Ananthapadmanabha, B. S. Manoj, C. Siva Ram Murthy, "Multihop Cellular Networks: The Architecture and Routing Protocols", in *Proc. IEEE PIMRC 2001*, San Diego, USA, October 2001.
- Christo Frank. D, B. S. Manoj, and C. Siva Ram Murthy, "Throughput Enhanced Wireless in Local Loop (TWILL) — The Architecture, Protocols, and Pricing Schemes" in *Proc. IEEE LCN 2002*, November 2002.
- E. Fratkin, V. Vijayaraghavan, Y. Liu, D. Gutierrez, TM Li, and M. Baker, "Participation Incentives for Ad hoc Networks", Technical Report, Department of Computer Science, Stanford University, 2001.
- Daniel Barreto, Yan Liu, Jinhui Pan, and Frank Wang, "Reputation-Based Participation Enforcement for Ad hoc Networks", Technical Report, Department of Computer Science, Stanford University, 2002.
- G. Pei, M. Gerla and T. W. Chen, "Fisheye State Routing in Mobile Ad Hoc Networks", in *Proc. 2000 ICDCS Workshops*, Taipei, Taiwan, April 2000.
- X. Zeng, R. Bagrodia, and M. Gerla, "Glomosim: A Library for Parallel Simulation of Large- scale Wireless Networks", in *Proc. PADS-98*, Banff, Canada, May 1998.
- IEEE Standards Board, "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification", in *The Institute of Electrical and Electronics Engineers Inc.*, 1997.

A High Performance Computing System for Medical Imaging in the Remote Operating Room

Yasuhiro Kawasaki¹, Fumihiko Ino¹, Yasuharu Mizutani¹, Noriyuki Fujimoto¹, Toshihiko Sasama², Yoshinobu Sato², Shinichi Tamura², and Kenichi Hagihara¹

¹ Graduate School of Information Science and Technology, Osaka University

1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

² Graduate School of Medicine, Osaka University

2-2 Yamadaoka, Suita, Osaka 565-0871, Japan

Abstract. This paper presents a novel cluster system, named MI-Cluster, for the purpose of developing a testbed for the cluster-assisted surgery. Our system provides a framework for utilizing high performance computing resources from the remote operating room. One current application is an accurate simulator for the range of motion (ROM) adjustment in total hip replacement (THR) surgery. To perform high-quality imaging during surgery, we have parallelized this compute-intensive ROM simulator on a cluster of PCs with 128 processors. Acceleration techniques such as dynamic load-balancing and data compression have been incorporated into our system. The system also provides a remote access service with a secure execution environment. We applied the system to an actual THR surgery performed at Osaka University Hospital and confirmed that the MI-Cluster system realizes intraoperative ROM simulation without degrading the accuracy of the simulation.

1 Introduction

With the rapid advances in information technology, three-dimensional (3-D) medical imaging is increasing its role in surgery. For example, in total hip replacement (THR) surgery, range of motion (ROM) simulators assist the surgeon in selecting and aligning the optimal components of artificial joint. However, since recent X-ray computed tomography (CT) scans produce high-resolution 3-D images, such imaging requires a large amount of computation. Therefore, high performance computing (HPC) approaches are necessary for intraoperative medical imaging, which requires real-time processing.

One emerging platform in HPC is the cluster systems, which provide HPC resources at a lower cost compared to supercomputer systems. Since cluster systems offer us high-speed processing by multiprocessors as well as large-scale data processing by distributed memory, developing medical imaging systems on clusters allows us to realize *the cluster-assisted surgery* with both real-time and high-resolution medical imaging at a low cost. Furthermore, cluster systems have the flexibility to extend its performance for the increasing computational cost of medical imaging.

To realize this novel type of surgery, cluster-enabled systems must have several additional facilities compared to single systems. These facilities can be classified into

two groups whether they arise from preoperative or intraoperative assists. For preoperative assists, we require (a) a secure execution environment to keep patients' privacy. Moreover, (b) data distribution is necessary for data-intensive applications. For intraoperative assists, in addition to facilities (a) and (b), (c) parallel processing is necessary for real-time processing of compute-intensive applications.

In this work, to realize the cluster-assisted surgery and share this novel surgery with several hospitals, we have developed a testbed, named MI-Cluster, composed of 64 off-the-shelf symmetric multiprocessor (SMP) PCs. MI-Cluster provides its HPC resources for remote surgeons via the Internet and currently solves problems (a)-(c) on the following medical applications: (1) an accurate simulation for the ROM adjustment [1,2] in THR surgery, (2) a rigid/non-rigid registration [3] for surgical planning, and (3) a real-time volume rendering [4] for the visualization of invisible parts inside patients.

The rest of this paper is organized as follows. Section 2 introduces some related work concerning the use of HPC during surgery. Section 3 gives the design and implementation of MI-Cluster while Section 4 describes a ROM simulator as an example of medical applications. Section 5 presents some experimental results obtained at Osaka University Hospital. At last, Section 6 concludes this paper.

2 Related Work

Some recent work [5,6,7] have incorporated HPC into the operating room by using shared-memory multiprocessor systems.

In [7], Warfield et al. have developed an intraoperative image segmentation algorithm for image-guided surgery. Using a Sun Ultra HPC 6000 with 20 processors, they classified the brain from intraoperative scanned volumetric data and showed surface rendering of the classified brain with the transparently rendered skin surface. Their parallel implementation classified a $256 \times 256 \times 60$ voxel image in 20.7 seconds, which is rapid enough to realize intraoperative assists, while sequential implementation takes more than 2 minutes. They also have developed a non-rigid registration algorithm [6] that simulates the biomechanical properties of the brain and its deformations during surgery. Their implementation on the Ultra HPC 6000 accelerates this simulation into less than 10 seconds. Although the load imbalance issue is unaddressed, their HPC approach has successfully assisted the surgeon during surgery.

In [5], Rohlfing et al. also have presented a parallel implementation of a fast non-rigid registration algorithm for intraoperative imaging such as a brain deformation during cranial surgery. On a SGI Origin 3800 with 128 processors, their implementation achieved a speedup of at least a factor of 50 and reduced its execution time from hours to minutes.

Thus, some researchers have realized intraoperative medical imaging on shared-memory multiprocessors. However, to the best of our knowledge, there has been no work on developing a framework to provide HPC resources for remote surgeons during surgery.

3 MI-Cluster: A Medical Imaging System for Remote Parallel Processing

This section describes the design and implementation of MI-Cluster.

3.1 Design Issues

To share a cluster system with several hospitals, the system must have the following functionalities.

- *Connection to the Internet:* This connection is necessary for the system to provide its HPC resources for any hospital at any time, such as ubiquitous computing. Ubiquitous computing is a key factor for unexpected surgeries, which includes a significant portion of all surgeries.
- *Automation framework for parallel execution:* This framework encapsulates the sequence of the procedures required for remote parallel processing, so that users are allowed to submit their jobs in the same way as they do in sequential systems. Thus, the sequence has to be simplified in a mission-critical system.
- *Secure execution environment:* Secure execution is necessary for protecting patients' data transmitted over the Internet, because the system has to protect patients' privacy from a viewpoint of clinical ethics. Furthermore, note that the system participates in critical missions, so it has to prevent any unauthorized access to avoid operation hijacking.
- *High-speed data transmission:* High-speed data transmission is necessary for real-time remote processing, since medical applications handle high-resolution 3-D images. For example, a $512 \times 512 \times 512$ voxel image in 16bit color becomes 256MB in size, and transmitting this raw (and large) image can cancel out the time benefits of HPC.

In the following, we describe our MI-Cluster implemented to satisfy the above issues.

3.2 Hardware/Software Architecture of MI-Cluster

Fig. 1(a) illustrates an overview of the MI-Cluster system, including a client PC in the operating room at Osaka University Hospital, located approximately five kilometers distance from MI-Cluster. To make the cluster secure, the system consists of two components: one is a cluster that executes medical applications in parallel and the other is a gateway PC that connects the cluster with the Internet. The gateway also serves as a network firewall that protects the cluster from unauthorized access and provides the cluster's HPC resources for remote client PCs in operating rooms.

The cluster has 64 SMP nodes and each node is a self-made PC composed of off-the-shelf parts such as Pentium III 1GHz processors, 2GB main memory, and 40GB hard disk drives. It also connects to two networks, since the packets for node management such as NIS and NFS can drop the performance of parallelized medical applications due to network congestion. One is the Myrinet network [8] for parallel processing and the other is the Ethernet network for node management, yielding full-duplex bandwidth of 2Gb/s and 100Mb/s, respectively.

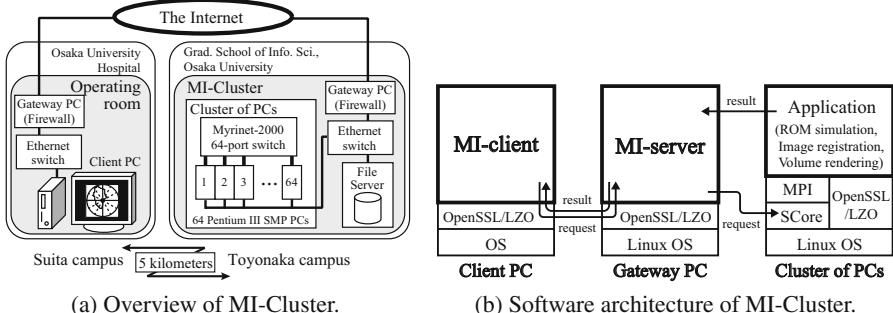


Fig. 1. Hardware and software overview of MI-Cluster.

Fig. 1(b) presents the software architecture of MI-Cluster. Each node in MI-Cluster uses the Linux OS and the SCORE cluster system software [9]. SCORE provides an environment for parallel programming and achieves fault tolerance based on checkpointing and multi-user environment. It also provides the MPICH-SCORE library [10], a fast implementation of the Message Passing Interface (MPI) standard [11]. Medical applications for remote parallel processing are implemented using the C++ language and MPI routines and executed in parallel on SCORE. MI-client and MI-server works as a framework for secure remote parallel processing with high-speed data transmission (described in Section 3.3).

3.3 Framework for Remote Parallel Processing

The framework for remote parallel processing in MI-Cluster has the following facilities:

- *Secure execution*: To protect patients' data transmitted over the Internet, it authenticates users and encrypts any transmitted data by the OpenSSL public key cryptography library [12], which avoids tapping and spoofing.
- *Data Compression*: To provide high-speed data transmission, the system compresses input/output data by the LZO real-time lossless data compression library [13].
- *Data management*: To provide high-speed data transmission, the system avoids any retransmission by registering the data transmitted before.
- *Progress report*: To inform remote surgeons the progress of parallel processing, the system reports the progress by every half second and shows the expected completion time of the submitted job.

The above facilities are realized by MI-client and MI-server based on the client-server paradigm. They encapsulate the sequence of the procedures required for remote parallel processing, so that users are allowed to submit their jobs in the same way as they do in sequential systems.

Fig. 2 shows the sequence of the encapsulated procedures, which consists of four major stages as follows:

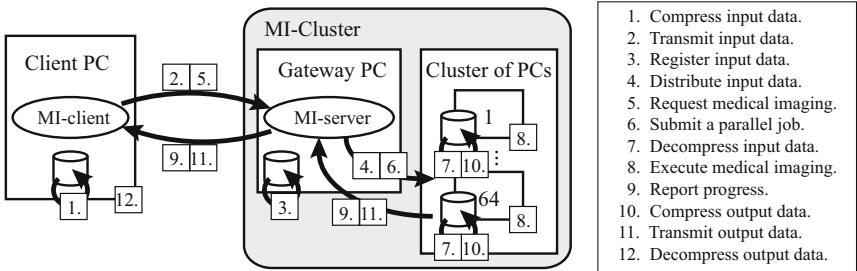


Fig. 2. Sequence of procedures for remote parallel medical imaging.

Stage 1. Input-data transmission: The client (1.) compresses input data and (2.) sends it to the gateway. The gateway (3.) registers the received data and (4.) distributes it to the nodes in the cluster.

Stage 2. Parallel execution: The client (5.) requests the gateway to execute a parallel job over the Internet secured by OpenSSL. When the gateway receives a request from the client, it (6.) submits the requested job to SCore. Each node (7.) decompresses their input data.

Stage 3. Progress report: Each node (8.) executes the submitted job in parallel and (9.) periodically reports the progress of the parallel processing to the client via the gateway.

Stage 4. Output-data transmission: Each node (10.) compresses output data and (11.) transmits it to the client via the gateway. The client (12.) decompresses the received output data.

4 Parallelizing Medical Application

This section describes how we have parallelized a ROM simulator.

4.1 Range of Motion (ROM) Simulation

ROM simulation, which computes the safe ROM, aims at assisting the surgeon in selecting and aligning the optimal components of artificial joint: the cup, head, neck, and stem as illustrated in Fig. 3(a). This assistance is important for both the surgeon and patient, since either inappropriate or malpositioned components increase the risk of clinical problems such as dislocation, wear, and loosening [1,2].

Existing ROM simulators [1,2] are useful in developing preoperative surgical plans, however, the preoperative plans may be required to be changed by unpredicted conditions being revealed during surgery. For example, if the bone tissue around the preoperatively planned position is known to be fragile, the surgeon has to realign the components in a stable position as well as to reselect the optimal components. Therefore, an intraoperative simulator is required for replanning the plans.

Before describing the details of ROM simulation, we briefly show a representation of hip joint motion described in [2] (see Fig. 3(b)). Given the pelvis coordinate system (pelvis-CS) and the femur coordinate system (femur-CS), the hip joint motion is

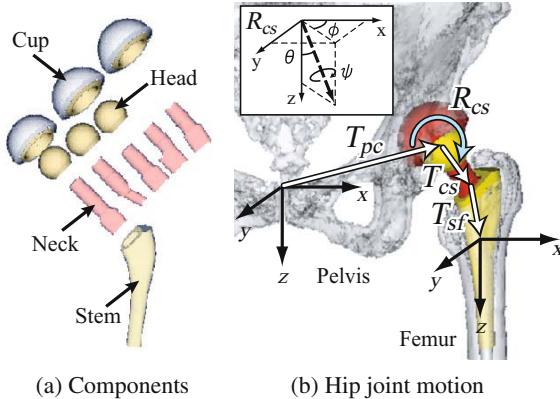


Fig. 3. Components of artificial joint and representation of hip joint motion.

represented by the transformation from the pelvis-CS to the femur-CS, M_{pf} , given by: $M_{pf} = T_{pc}T_{cs}R_{cs}T_{sf}$, where T_{pc} is a 4×4 transformation matrix representing the orientation of the cup in the pelvis-CS, T_{cs} is a fixed transformation matrix determined by the selected head and neck components, R_{cs} is a variable transformation matrix constrained to the rotational motion, and T_{sf} is a transformation matrix representing the reverse orientation of the stem in the femur-CS. Both T_{pc} and T_{sf} are determined by one of the two methodologies as follows. For preoperative assists, the surgeon determines them by experience and visualization. For intraoperative assists, optical 3-D position sensors give measured T_{pc} and T_{sf} from implanted components.

Given T_{pc} , T_{cs} , and T_{sf} , the safe ROM is defined as a set of rotation transformation matrix R_{cs} such that R_{cs} avoids any implant-implant, bone-implant, and bone-bone impingements. Since R_{cs} is defined as a 3-D rotation matrix, ROM simulation is a compute-intensive application. Therefore, a real-time simulator is necessary for patients to reduce their physical load such as a blood transfusion. Furthermore, to obtain precise ROMs, an intraoperative simulation based on measured T_{pc} and T_{sf} is required.

In the following, we represent R_{cs} by the Euler angles, (ϕ, θ, ψ) ($0 \leq \phi < 360^\circ$, $0 \leq \theta < 180^\circ$, $-180^\circ \leq \psi < 180^\circ$), where ϕ , θ , and ψ represent yaw, pitch, and roll angles, respectively (see Fig. 3(b)).

4.2 Parallel Implementation

The calculation in ROM simulation has two features as follows: F1, coarse-grain parallelism and F2, nondeterministic time-complexity. F1 enables us to independently investigate each rotation (ϕ, θ, ψ) whether it causes impingements. However, if we statically assign the equal number of rotations to processors, we fail to derive a reasonable speedup, since F2 causes load imbalance. Besides, it is not easy to predict the workload for each rotation. Therefore, we employ the master/slave (M/S) paradigm to realize dynamic load-balancing (see Fig. 4). In our ROM simulator, the master node manages the safe ROM and assigns tasks to the slave nodes. Here, a task corresponds to the collision

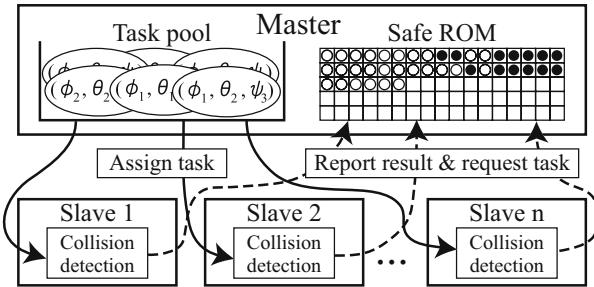


Fig. 4. Master/slave dynamic load balancing for ROM simulation.

detection for a set of rotations and the grain size of each task corresponds to the number of rotations in the set. Task assignments are done serially from the idle slave nodes.

We have implemented our ROM simulator using the C++ language and the MPICH-SCore library. For each of collision detection, we used the V-COLLIDE [14,15] library, which rapidly detects precise collisions at $O(n + m)$ time, where n is the number of polygonal objects and m is the number of polygonal objects very close to each other. Since V-COLLIDE detects all of the collided polygons, we made a simple modification on this library to return as soon as it detects one collided polygon.

5 Experimental Results

In this section, we present some experimental results obtained at Osaka University Hospital. The results consist of (1) the speedup of our ROM simulator with different task grain sizes and (2) the turnaround time of remote parallel processing with different compression algorithms.

5.1 Practical Use in Total Hip Replacement Surgery

We applied our ROM simulator to a THR surgery performed at Osaka University Hospital. Fig. 5 shows the schema for this surgery assisted by both preoperative and intraoperative ROM simulations.

Before the surgery, we constructed pelvis and femur polygon data from the patient's preoperative CT image by using the marching cubes algorithm [16]. Pelvis and femur data consist of 116,270 and 30,821 polygons (5.6MB and 1.5MB in size), respectively. Then, we compressed these data and transmitted them to MI-Cluster. After this, we performed two types of simulation. One is the preoperative ROM simulation and the other is the preoperative limb length simulation to minimize a limb length discrepancy between left and right legs. We repeated these preoperative simulations ten times to select and align the optimal components of artificial joint, in order to develop a preoperative surgical plan.

During the surgery, the surgeon first removed diseased bone tissue and temporarily implanted the components of artificial joint according to the preoperative plan. After

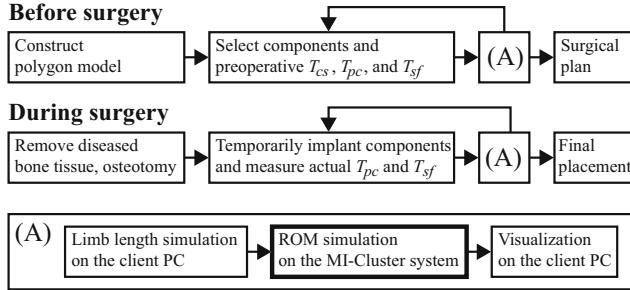


Fig. 5. Schema for total hip replacement surgery.

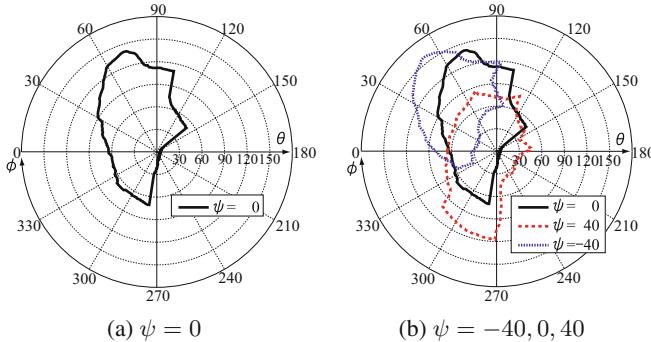


Fig. 6. ROM simulation result.

this, we measured T_{pc} and T_{sf} (200B in size) by an optical sensor and transmitted them to MI-Cluster. We then performed intraoperative limb length and ROM simulations to confirm whether the actual placed components can cause serious clinical problems. We repeated these intraoperative simulations three times to decide the final position where the components are implanted.

Fig. 6 shows the obtained safe ROM, where $\psi = -40, 0, \text{ and } 40$. On a sequential system we have to reduce the number of rotations to achieve real-time processing. For example, we vary only ϕ and θ at $\psi = 0$ and obtain a safe ROM as shown in Fig. 6(a). On the other hand, MI-Cluster enables us to vary all of three angles with keeping both the real-time processing and the accuracy of the simulation, and thereby gives us a 3-D safe ROM as shown in Fig. 6(b).

5.2 Speedup of Parallel ROM Simulator

Before the surgery, we performed a preliminary experiment to measure the speedup of our parallel ROM simulator with different task grain sizes. Note that the results in this section concern only the parallelized part while Section 5.3 presents the entire turnaround time, including the execution time for data transmission.

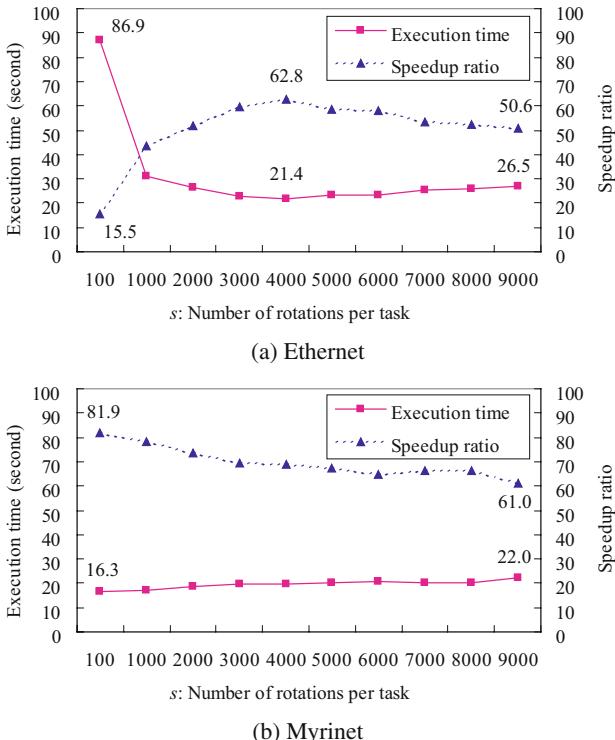


Fig. 7. Execution time on 128 processors with different grain size.

In the following, let s be the grain size of each task. We varied the values of three angles ϕ , θ , and ψ by 1, 1, and 20 degrees, respectively, so investigated $360 \times 180 \times 18 = 1,166,400$ rotations. Fig. 7 shows the execution time on 128 processors, where $100 \leq s \leq 9,000$.

On Ethernet, we obtained the worst speedup of 15.5 when $s = 100$, however, the best speedup reaches a factor of 62.8 when $s = 4,000$. This best speedup is near the result on Myrinet, which yields the worst and the best speedups of 61.0 and 81.9, respectively. Therefore, once we select an appropriate value for s , an intraoperative simulation can be performed on Ethernet, a widely used local area network.

Fig. 8 shows a distribution map of the detection time on Ethernet, where $\psi = 0$. This map gives us the reason why the worst speedup on Ethernet becomes extremely low. Most of rotations take below 2 ms for their detection. So, the grain size of each task is too small for this high latency network. Therefore, the master node suffers from network congestion and the slave nodes easily become idle state. Actually, the total time of send routine (`MPI_Send`) called by the master node accounts for 90% of the execution time (86.9 seconds) and the averaged total time of receive routine (`MPI_Recv`) called by the slave nodes accounts for 80%.

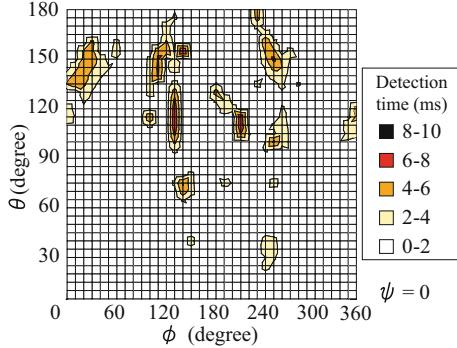


Fig. 8. Distribution map of detection time on Ethernet.

We also measured the execution time of an implementation that statically assigns the equal number of rotations in a block distribution manner. The execution time on Ethernet and Myrinet were 41.6 and 26.6 seconds, respectively. Here notice that the M/S implementation loses its load balancing facility with the increase of s . That is, the behavior of the M/S implementation becomes similar to that of the static implementation. However, there exists a gap between their execution time: 41.6-26.5 on Ethernet and 26.6-22.0 on Myrinet, where $s = 9,185 (\equiv \lfloor 1,166,400/127 \rfloor)$. This gap is also caused by the network congestion mentioned above. While the static implementation gathers the distributed safe ROMs all at once, the M/S implementation allows every slave node to return their safe ROMs as soon as they complete their assigned task. Therefore, compared to the M/S implementation, the static implementation can easily cause network congestion while gathering the safe ROM.

5.3 Turnaround Time of Remote Parallel Processing

Table 1 lists the turnaround times of ROM simulations when processing in sequential and remote parallel. To make clear the contribution of data compression, we compared three implementations: LZO, zlib [17] and no compression.

Although our system takes additional times required for remote parallel processing, times t_1-t_7 and t_9-t_{11} , it reduces the turnaround time from 1,354 seconds on a single system to the best of 23.4 seconds on Myrinet and to that of 28.5 seconds on Ethernet. These turnaround times below a half minute are rapid enough for intraoperative processing.

We also see that both data compression algorithms successfully reduce the turnaround time by approximately 33% on Ethernet and 37% on Myrinet. The significant gap between LZO and zlib appears only at time t_b on Ethernet. That is, compared to LZO, zlib takes more time to compress data but generates smaller data as shown in Table 2. Therefore zlib is superior to LZO when transmitting data via a low bandwidth network or when broadcasting data to the nodes in the cluster. On the other hand, LZO takes an advantage to zlib when transmitting data via a high bandwidth network or when compressing data that achieves high-rate compression such as the safe ROM.

Table 1. Turnaround time of sequential and remote parallel processing.

	Breakdown of turnaround time	Execution time in second			
		1CPU	128CPU		
			Compression algorithm		
			No ET / MY	LZO ET / MY	zlib ET / MY
Before surgery	t_1 : Compress polygon data (1.)	—	—	0.7	2.7
	t_2 : Transmit polygon data (2.-3.)	—	5.5	2.8	1.5
	t_3 : Distribute polygon data (4.)	—	16.5 / 5.6	7.9 / 2.2	4.7 / 1.9
	t_b : Total (preoperative)	—	22.0 / 11.1	11.4 / 5.7	8.7 / 5.9
During surgery	t_4 : Compress T_{pc} & T_{sf} (1.)	—	—	0.1	0.1
	t_5 : Transmit and distribute T_{pc} & T_{sf} (2.-4.)	—	0.5 / 0.5	0.5 / 0.5	0.5 / 0.5
	t_6 : Invoke parallel processes (5.-6.)	—	4.8 / 4.8		
	t_7 : Decompress all input data (7.)	—	—	0.4	0.4
	t_8 : ROM simulation (8.-9.)	1,354	21.4 / 16.3		
	t_9 : Compress the safe ROM (10.)	—	—	0.4	1.2
	t_{10} : Transmit the safe ROM (11.)	—	13.0	0.5	0.1
	t_{11} : Decompress the safe ROM (12.)	—	—	0.4	0.4
	t_d : Total (intraoperative)	1,354	39.7 / 34.6	28.5 / 23.4	28.9 / 23.8

ET and MY represent Ethernet and Myrinet, respectively.

Table 2. Results on data compression.

Data	Size in KB		
	No	LZO	zlib
Pelvis polygon	5,616	2,781 (49%)	1,634 (29%)
Femur polygon	1,454	781 (53%)	470 (32%)
Safe ROM	22,781	379 (2%)	146 (1%)

Numbers in brackets represent the compression rate.

6 Conclusions

We presented a remote parallel processing system for medical imaging, the MI-Cluster system, developed for the goal of the cluster-assisted surgery. Our current system provides an intraoperative ROM simulation by parallel processing based on the master/slave dynamic load balancing. It also realizes a framework for remote parallel processing, which includes a secure execution environment by a public key cryptography and a high-speed data transmission by a real-time lossless data compression algorithm.

As a result, MI-Cluster accelerates the simulation that takes a half hour on a single system in less than a half minute and thereby realizes intraoperative surgical planning without dropping the accuracy of the simulation. One remaining issue is to achieve fault tolerance, which enables highly reliable systems.

Acknowledgements. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C)(2) (14580374), for Young Scientists (B)(15700030), and JSPS Research for the Future Program JSPS-RFTF99I00903. We would like to thank the anonymous reviewers for their valuable comments.

References

1. Jaramaz, B., DiGioia, A.M., Blackwell, M., Nikou, C.: Computer assisted measurement of cup placement in total hip replacement. *Clinical Orthopaedics and Related Research* **354** (1998) 70–81
2. Sato, Y., Sasama, T., et al.: Intraoperative simulation and planning using a combined acetabular and femoral (CAF) navigation system for total hip replacement. In: Proc. 3rd Int'l Conf. on Medical Image Computing and Computer-Assisted Intervention (MICCAI'00). (2000) 1114–1125
3. Hajnal, J.V., Hill, D.L., Hawkes, D.J., eds.: *Medical Image Registration*. CRC Press (2001)
4. Takeuchi, A., Ino, F., Hagiwara, K.: An improvement on binary-swap compositing for sort-last parallel rendering. In: Proc. 18th ACM Symp. Applied Computing (SAC'03). (2003) 996–1002
5. Rohlfing, T., Maurer, C.R.: Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees. *IEEE Trans. Information Technology in Biomedicine* **7** (2003) 16–25
6. Warfield, S.K., Ferrant, M., Gallez, X., Nabavi, A., Jolesz, F.A., Kikinis, R.: Real-time biomechanical simulation of volumetric brain deformation for image guided neurosurgery. In: Proc. High Performance Networking and Computing Conf. (SC2000). (2000)
7. Warfield, S.K., Jolesz, F.A., Kikinis, R.: Real-time image segmentation for image-guided surgery. In: Proc. High Performance Networking and Computing Conf. (SC98). (1998)
8. Boden, N.J., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.K.: Myrinet: A gigabit-per-second local-area network. *IEEE Micro* **15** (1995) 29–36
9. PC Cluster Consortium: SCORE Cluster System Software, (2002)
<http://www.pccluster.org/>
10. O'Carroll, F., Tezuka, H., Hori, A., Ishikawa, Y.: The design and implementation of zero copy MPI using commodity hardware with a high performance network. In: Proc. 12th ACM Int'l Conf. on Supercomputing (ICS'98). (1998) 243–250
11. Message Passing Interface Forum: MPI: A message-passing interface standard. *Int'l J. Supercomputer Applications and High Performance Computing* **8** (1994) 159–416
12. OpenSSL Project: <http://www.openssl.org/> (2003)
13. Oberhumer, M.F.X.J.: LZO data compression library, (2003)
<http://www.oberhumer.com/opensource/lzo/>
14. Hudson, T.C., Lin, M.C., Cohen, J., Gottschalk, S., Manocha, D.: V-COLLIDE: Accelerated collision detection for VRML. In: Proc. 2nd Symp. Virtual Reality Modeling Language (VRML'97). (1997) 117–124 http://www.cs.unc.edu/~geom/V_COLLIDE/.
15. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics (Proc. ACM SIGGRAPH'96)* **30** (1996) 171–180
16. Lorensen, W.E., Cline, H.E.: Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proc. ACM SIGGRAPH'87)* **21** (1987) 163–169
17. Gailly, J.L., Adler, M.: zlib general purpose compression library,
<http://www.gzip.org/> (2003)

Parallel Partitioning Techniques for Logic Minimization Using Redundancy Identification

B. Jayaram, A. Manoj Kumar, and V. Kamakoti

Indian Institute of Technology Madras, India
kama@iitm.ernet.in

Abstract. Redundancy identification is a challenging open problem in logic optimization of Boolean circuits. Partitioning techniques are employed successfully to solve the redundancy identification problem with less time and higher scalability. Any heuristic/algorithm for the Logic optimization problem, and hence the redundancy identification problem is compute-intensive, especially when very high approximation to the optimal solution is demanded. This is because the problems are NP-complete. This necessitates parallel heuristics/algorithms to speed-up the computation process. In this paper, we present a parallel partitioning approach for the logic optimization problem using the concept of redundancy identification. This result finds extensive applications in the area of VLSI CAD tool design.

1 Introduction

Redundancy identification and removal is a very important and crucial step in logic synthesis and optimization. Logic optimization algorithms based on various notions of redundancy have been proposed. While the topic of two-level logic minimization has been perfected to give optimal solutions, multi-level logic optimization continues to be a challenging and time-consuming problem. Two main approaches are being followed in this field namely - the Boolean Logic based approaches and the Netlist based approaches. Boolean methods analyze the logic representing the circuit and try to optimize it. While they are effective in identifying the redundancies, they are not efficient in terms of execution time. Netlist based approaches depend on the notion of faults for the characterization of redundancy. Such approaches are based on the fact that *redundant circuit elements are equivalent to undetectable faults* [8]. Netlist based approaches are faster and generally detect a large proportion of redundancies. These approaches may be categorized into two - Fault-oriented algorithms, and Fault-independent algorithms.

Fault-oriented algorithms use ATPG techniques to identify undetectable faults [1,2,3]. They use test pattern generators to determine whether or not a target fault has a test that can detect it. A complete test generation, thus identifies all undetectable faults and hence all the redundant circuit elements. Standard test pattern generators like the D-algorithm, the 9V-algorithm and the PODEM are well known [8]. On the other hand, Fault-independent algorithms,

analyze the structure of the topology of the circuit and do not target a specific fault for the identification of redundancies. Menon *et al* have proposed a method based on the analysis of reconvergent fanout structures [4]. This method requires repeatedly computing the *controlling value* paths associated with every reconvergent gate of every circuit stem. Iyer and Abramovici give another technique to identify redundancy by propagating *uncontrollability* values in a combinational circuit [5]. Their algorithm, called FIRE, works by forcing conflicts on lines to identify undetectable faults.

The determination of all undetectable faults in a circuit is a proven NP-complete problem. A trade-off between *efficiency*, in terms of runtime, and *effectiveness*, in terms of the number of faults detected, can be seen in the above methods. The test pattern generators used in Fault-oriented algorithms exhibit worst case behavior when identifying undetectable faults. Fault-independent approaches can identify only a subset of undetectable faults. Moreover, the scalability of the above methods has not been discussed in the literature. As we deal with larger circuits of the order of millions of gates, we need algorithms that are fast and effective. Berkelaar and Eijk have reported a technique that works on big circuits [6]. They have used carefully tuned ATPG pre-processing methods and a modified version of the D-algorithm to achieve the desired scalability.

Partitioning based approaches for redundancy identification can be used for large circuits. *This is possible because of the fact that a redundant fault in the context of a sub-circuit remains redundant in the context of the circuit as well.* However, partitioning is an expensive task and we must ensure that the improvement obtained by partitioning should not be offset by the time taken for partitioning. Hence, clever partitioning techniques, that enable redundancy identification within the sub-circuit itself, should be used. In [11], we have shown that corolla based partitioning is an effective strategy for redundancy identification with scalability. The advantages of partitioning have also been discussed in it.

Most of the Logic Minimization approaches are compute-intensive. However, its interesting to note that most of them are inherently parallelizable. This observation can be used to design parallel heuristics/algorithms that result in a significant speed-up in the execution time. The exponential complexity of solving the Logic Minimization problem makes these approaches very crucial in the context of very large circuits for which sequential algorithms cannot provide solutions in reasonable time. In this paper, we propose a parallelized partitioning based scheme for redundancy identification. Besides parallelizing the redundancy identification procedures, we have also parallelized the partitioning scheme to a large extent.

2 Redundancy and ATPG

ATPG deals with the generation of test patterns for the detection of physical faults in the circuit. These physical faults are represented through logical models. Stuck-fault model is one such model which has been found to describe a wide range of physical faults. In this model, a faulty net is assumed to be stuck

permanently at a logical value, either 0 or 1. The test vector v for a fault f is a combination of input values to a circuit C , such that the output of C on application of v , is different in the presence and absence of f . If none of the combinations of input values is a test vector for a fault f , then f is *functionally undetectable*. The faulty net can be assumed to be permanently stuck-at the logical value 0 or 1 without any loss of functionality and can be replaced by Gnd and Vdd respectively. The sub-circuit that was driving only this redundant net (in other words, the part of the circuit that gets disconnected due to these replacements) may be removed [8]. Hence absence of a test vector to test a fault indicates the redundancy of the net.

A general type of redundancy [7] exists in a circuit when it is possible to cut a set of r lines and to connect $q \leq r$ of the cut lines to some other signals in the circuit without changing its function. However, in this paper, we will restrict ourselves to the definition of redundancy related to the presence of undetectable stuck faults.

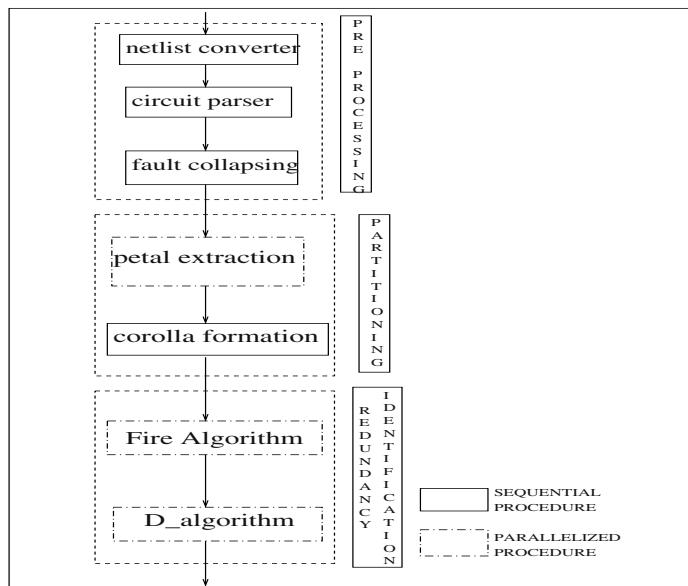


Fig. 1. Overall strategy

3 Overall Approach

We now give an overview of our parallelized approach for logic minimization. There are three main stages in the method, namely- the Input Preprocessing Stage, the Partitioning Stage and the Redundancy Identification Stage. In the

Input Preprocessing Stage, we convert the circuit given in some standard netlist format into a format which is accepted by the parser. The parser converts this format into a DAG representation of the circuit. Fault collapsing is performed on this structure. This stage is inherently sequential and hence it has not been parallelized. In the next stage, petals are extracted and the circuit is partitioned into corollas as explained in Section 4. The extraction of petals is parallelized. Each processor is assigned the task of extracting the petals of a sub-set of the fanout-stems in the circuit. These steps constitute the Partitioning Stage. Finally, we perform redundancy identification on the partitioned circuit (the undetectable faults are identified as explained in Section 6). This stage also involves parallel computation in the implementation of the two algorithms mentioned in Section 6. All the steps in the approach are automated and integrated into a single flow which is shown in Figure 1.

4 Corolla Partitioning

Partitioning of the circuit is based on the identification of corollas by performing a reconvergence analysis of the circuit. Intuitively, a corolla is a set of overlapping reconvergent fanout regions. This partitioning technique has been proposed by Dey *et al* in [9] and has been used in the fields of resynthesis for load balancing and parallel simulation. It is based on the reconvergence analysis proposed by Maamari and Rajski [10]. We explain below, the key concepts used in corolla partitioning.

The given circuit is modeled as a DAG, $G(V, E)$.

Definition 1 (Reconvergent fanout stem and reconvergent node). Let $s \in V$ be a stem node in the graph. If there exists more than one disjoint paths from s to another node $t \in V$, then s is a *reconvergent fanout stem* and t is a *reconvergent node* of s .

Definition 2 (Closing reconvergent node). A *closing reconvergent node* of a reconvergent fanout stem s is a reconvergent node of s that does not drive any other reconvergent node of s .

In the circuit shown in Figure 2, b is a reconvergent fanout stem and g_5, g_6 and g_7 are its reconvergent nodes. g_7 is the only closing reconvergent node of b .

Definition 3 (Primary Stem Region). The *primary stem region* of a reconvergent fanout stem s consists of all the nodes v and v 's output edges such that v is located on a path from s to any of the closing reconvergent nodes of s .

Definition 4 (Petal). Every biconnected component of the graph corresponding to the primary stem region of a reconvergent fanout stem s is called a *petal* of s .

In Figure 2, the primary stem region of fanout stem v consists of the nodes in $\{v, w, g_3, g_4, g_5, g_6, g_7\}$ and all their output lines. This primary stem region also forms a petal.

Definition 5(Overlap of Petals). Two petals P_i and P_j are said to *overlap* either if (i) $P_i \cap P_j \neq \emptyset$ or (ii) $P_i \cap P_k \neq \emptyset$ and P_k overlaps with P_j .

Given two petals P_i and P_j , $P_i \cap P_j$ represents the set of all nodes n such that

n is present in both P_i and P_j . The overlap relation defines a set of equivalence classes C_1, C_2, \dots, C_k on the circuit. Two petals belong to the same equivalence class only if they overlap.

Definition 5 (Corolla). A *corolla* is a collection of overlapping petals.

Definition 6 (Maximal Corolla). A *maximal corolla* is an equivalence class C_i formed by the relation *overlap*.

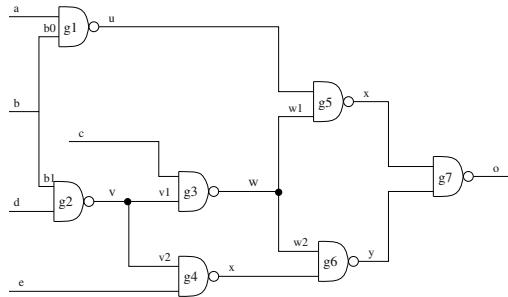


Fig. 2. Gate level circuit

Corolla partitioning divides the circuit into many *non-overlapping* corollas. Typically, a size constraint is imposed on each of the corollas to ensure that the partition is uniform and reasonable. A restriction on the number of inputs of the corolla was chosen for this purpose. This would ensure a significant decrease in run time if random test generation approaches are used on the corolla. Exhaustive test generation could also be considered for smaller corollas.

The algorithm for circuit partitioning involves formation of petals and then combining them to form corollas. The formation of petals is parallelized. One of the processors is designated the *master* and the rest of them run as *slave* processors. Each *slave* processor extracts the petals corresponding to a sub-set of the fanout stems of the circuit. These sub-sets are carefully formed to ensure load balancing. It has been observed that fanout stems that are closer to primary inputs have large petal regions. Hence, the BFS levels of all the fanout stems are calculated and a uniform distribution of fanout stems with respect to these levels is ensured among the different *slave* processors. The *master* processor collects information about the petals in the circuit from the *slaves* and forms corollas iteratively as indicated in the algorithm given above.

The *slave* processors run an algorithm similar to the one given in [10] for the identification of petals.

```

Corolla_set =  $\phi$ ;
Unprocessed_petals = petals collected from slave processors;
while there are more unprocessed petals do
    Select an unprocessed petal  $P$ ;
    if  $P$  overlaps with only one corolla  $Q$  then
        try to add  $P$  to  $Q$ ;
    else if  $P$  overlaps with a set of corollas
         $\{Q_1, Q_2, \dots\}$  then
            try to combine corollas  $\{Q_1, Q_2, \dots\}$  into a single corolla  $Q$  and add  $P$  to  $Q$ ;
    else
        put  $P$  in a new corolla;
    end if
    Remove  $P$  from unprocessed petals;
end while
Broadcast petals to all the processors;

```

Fig. 3. Partitioning algorithm

5 Motivation for Corolla Partitioning

The presence of redundant lines is closely related to the presence of reconvergent regions. In fact, a circuit without reconvergent fanouts can be proven to have all non-redundant lines. Fault-oriented algorithms work by attempting to identify conflicting mandatory assignments necessary for detection of faults. A conflict is said to occur when two incompatible logic values have to be assigned to the same net. A set of such conflicts can indicate the undetectability of a fault. In Fault-oriented algorithms, these conflicts occur at the fanout stems of a reconvergence region. Given a net x , the potential conflict points for a given fault can be classified into two categories.

- The fanout stems of the petals which belong to the maximal corolla containing net x .
- The fanout stems of the petals which do not belong to the maximal corolla containing net x .

Corolla partitioning aims to put the fanout stems belonging to the first category along with the net x in the same partition. This improves the chance of detecting redundancy on x within the partition itself. We now show that *the possibility of conflicts on fanout stems of second category leading to undetectability of a fault on net x is low*. Hence, these fanout stems need not be put in the same partition. We give a set of lemmas that are useful for justifying the above statement. Let C be a maximal corolla in the given circuit and let net $x \in C$.

Lemma 1. *Let $\{O_1, O_2, \dots\}$ be the set of outputs of C driven by net x . Then the output cones of each of these outputs are mutually disjoint.*

Lemma 2. Let $\{I_1, I_2, \dots\}$ be the set of inputs of C that drive net x . Then the input cones of each of these inputs are mutually disjoint.

Lemma 3. Let y and z be two nets $\notin C$ such that y drives x and z is driven by x . Then all the paths between y and z pass through C.

The proofs for each of the above lemmas are quite simple and an intuitive proof can be worked out by looking at the Figure 4.

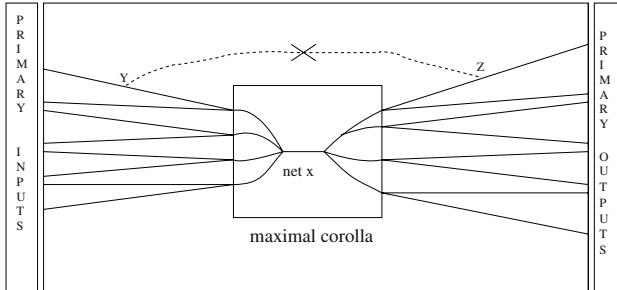


Fig. 4. Maximal corolla

Theorem 1. If a fault on a net x is detectable within its maximal corolla C and undetectable globally, then at least one input or output of C, that is redundant, (i.e. there exists at least one undetectable fault on the input or output lines)

Proof. We give a proof by contradiction. Suppose all the inputs and outputs of C are non-redundant. Let $\{O_1, O_2, \dots, O_m\}$ be the set of outputs of C driven by net x and let $\{I_1, I_2, \dots, I_n\}$ be the set of inputs of C that drive net x . Since the fault is detectable within the maximal corolla, we have an input vector $V = < v_1, v_2, \dots, v_n >$ that excites the fault on x and the fault effect can be propagated to a set of outputs $\{O_{l1}, O_{l2}, \dots, O_{lp}\}$.

Now since all input cones are disjoint(by Lemma 2) and since all inputs of the maximal corolla are non redundant, each of the values in the vector V can be justified to the primary inputs of the circuit independently. Since the output cones of each output are disjoint(by Lemma 1) and since all the outputs of C are non redundant, we can propagate the fault effect from at least one output O_{lr} to the primary outputs. Let the corresponding output cone be O_{lr} . The input cones of the set of inputs $\{I_1, I_2, \dots, I_n\}$ do not drive O_{lr} (by Lemma 3). Therefore, the justifications required for propagating the fault through O_{lr} can be done independent of the justifications done in the maximal corolla and the input cones. Such a justification exists because all the outputs are non-redundant.

Therefore, given that the fault is detectable within its maximal corolla, we have given a method to construct a test vector for its detection in the circuit, which is a contradiction. Hence, the theorem. \square

A redundancy at the inputs or outputs of a maximal corolla indicates redundancy in the corresponding cones. If the cone does not drive any other cones in the circuit, then the whole of it becomes redundant. We can see that such a case is unlikely. Therefore, the possibility that a fault is detectable within its maximal corolla and undetectable in the circuit is low. This implies that undetectable faults are most likely to be detected within their maximal corolla itself. Hence, conflicts on fanout stems of petals outside the maximal corolla need not be considered while partitioning.

Corolla partitioning also reduces the number of interconnections among the partitions. Intuitively, a fanout stem increases the span of the circuit while a reconvergent node decreases it. Therefore by avoiding a cut in a reconvergent region we are likely to decrease the number of interconnections among partitions. This makes it possible to use exhaustive test generation techniques on partitions with lesser number of inputs.

```

 $\alpha$  = set of corollas assigned to this processor;
for each of the corollas in  $\alpha$  do
     $\Theta$  = set of faults in the corolla after equivalence and dominance fault collapsing;
     $\phi$  = set of undetectable faults in the corolla determined using FIRE;
    for each of the faults in  $\Theta - \phi$  do
        determine their detectability within the corolla using  $D$ -algorithm.
    end for
end for
if processor is master then
    collect results from slave processors and display the results;
else
    send undetectable faults to the master processor;
end if
```

Fig. 5. Redundancy identification algorithm

6 Redundancy Identification

A two phase approach is followed for redundancy identification. Once the given circuit has been partitioned into corollas, we can identify redundant faults in them using a combination of fault independent and fault oriented algorithms. We use the FIRE[5] algorithm as the first stage in redundancy identification. Parallelization is achieved by distributing the fanout stems to be processed amongst various processors. To ensure load balancing, the strategy used during petal extraction is followed. At the end of this stage, the processors broadcast the results and the undetectable faults detected by FIRE are removed from the fault list.

Table 1. Results for benchmark circuits using the parallelized approach (Number of Processors = 4) and sequential approach

Circuit	# gates	Tot Red	Sequential		Parallelized	
			# faults identified	Time taken (in sec)	# faults identified	Time taken (in sec)
c3540	1669	131	113	31.47	128	8.74
c5315	2307	59	48	51.42	46	21.54
c6288	2416	34	33	103.2	32	31.28
c7552	3512	131	92	158.36	86	42.3
b14_C	8812	-	125	275	131	79.8
b15_C	8371	-	169	345	172	95.2
b20_C	17648	-	435	510	415	152
b21_C	17972	-	495	464	512	145

Table 2. Results for benchmark circuits using Parallel approach using 'n' processors

Circuit	Time Taken(in sec)				
	n = 2	n = 4	n = 6	n = 12	n = 18
c3540	15.58	8.74	7.34	4.64	3.96
c5315	40.44	21.54	15.72	8.54	6.41
c6288	54.3	31.28	27.96	15.29	10.86
c7552	70.93	42.3	28.52	15.98	11.32
b14_C	148.7	79.8	55.63	28.1	19.9
b15_C	182.4	95.2	58.84	33.4	22.1
b20_C	267	152	102.6	64.3	40.7
b21_C	240	145	96.8	56.7	35.2

The *D*-algorithm is then performed on the remaining set of faults. A processor works with a sub-set of corollas. It tries to generate tests for all the faults that lie within its sub-set of corollas. At the end of this stage, a *master* processor collects all the redundant results and displays them.

The following conclusions hold good for the results of the algorithm presented in Figure 5.

- A fault found to be undetectable in its corolla is globally redundant.
- A fault found to be detectable in its corolla may still be globally redundant.

7 Results

We have applied the parallelized approach discussed in this paper to the ISCAS85 benchmark circuits and to the combinational profiles of four circuits from the ITC99 benchmark suite. The algorithms were coded in C and the program was

run on a 64-node SGI system. The Message Passing Interface(MPI) was used in the implementation of the program.

In Table 1, we compare the results of running the program with 4 processors and running the sequential counterpart of the parallel approach(i.e. with one processor) on the benchmark circuits. A backtrack limit of 10 was used in the implementation of *D*-algorithm. The corollas formed in the partitioning stage were constrained to have less than 100 input lines. *Tot Red* is the total number of redundant faults in the circuit. The next two columns give the number of redundant faults identified and the time taken by the program respectively. It can be seen that in all the cases we get a speed-up in the execution times by a factor slightly less than 4. This indicates the efficiency of the parallelized approach. In some cases, we even observe an increase in the number of faults identified. This is probably due to formation of corollas in a more effective manner.

In Table 2, we give the results of running the program with varying number of processors. We can see an almost linear correlation between the number of processors and the run times.

References

1. S.T.Chakradhar, V.D.Agrawal and S.G.Rothweiler, "A Transitive Closure Algorithm for Test Generation," *IEEE Trans. Computer Aided Design*, vol.12, pp. 1015–1028, July 1993.
2. H.Cox and J.Rajski, "On Necessary and Nonconflicting Assignments in Algorithmic Test Pattern Generation," *IEEE Trans. on Computer Aided Design*, vol. 13, pp.515–530, April 1994.
3. W.Kunz and D.K.Pradhan, "Recursive Learning: An Attractive Alternative to the Decision Tree for Test Generation in Digital Circuits," in *Proceedings of the IEEE International Test Conference*, pp.816–825, September 1992.
4. P.R.Menon and H.Ahuja, "Redundancy Removal and Simplification of Combinational Circuits," in *Proceedings of the 10th IEEE VLSI Test Symposium*, pp.268–273, April 1992.
5. M.A.Iyer and M.Abramovici, "FIRE: A Fault Independent Combinational Redundancy Identification Algorithm," in *IEEE Trans. on Very Large Scale Integration(VLSI) Systems*, Vol.4, No.2, June 1996.
6. Michel Berkelaar and Koen van Eijk, "Efficient and Effective Redundancy Removal for Million-Gate Circuits", *Proc. International Workshop on Logic Synthesis 2001*, pp.249ff.
7. J.P.Hayes, "On the properties of Irredundant Logic Networks," *IEEE Trans. on Computer*, Vol. C-25, No.9, pp. 884–892, September,1976.
8. Miron Abramovici, Melvin.A.Breuer and Arthur.D.Friedman, "Digital Systems Testing and Testable Design," IEEE press, 2000.
9. S.Dey, F.Brglez and G.Kedem, "Corolla-based circuit partitioning and resynthesis," in *ACM/IEEE 27th Design Automation Conference*, pp.607–612, June 1990.
10. Fadi Maamari and Janusz Rajski, "A Reconvergent Fanout Analysis for Efficient Exact Fault Simulation of Combinational Circuits," In *18th International Symposium on Fault Tolerant Computing*, June 1988.
11. B. Jayaram, E. V. Vijuraj, R. Manimegalai, John.P.Joseph and V. Kamakoti, "Corollas Based Partitioning for Logic Minimization Using Redundancy Identification," Technical Report, IIT Madras, April 2003.

Parallel and Distributed Frequent Itemset Mining on Dynamic Datasets^{*}

Adriano Veloso^{1,2}, Matthew Eric Otey², Srinivasan Parthasarathy², and Wagner Meira Jr.¹

¹ Computer Science Department, Universidade Federal de Minas Gerais, Brazil
`{adrianov,meira}@dcc.ufmg.br`

² Department of Computer and Information Science, The Ohio State University, USA
`{otey, srini}@cis.ohio-state.edu`

Abstract. Traditional methods for data mining typically make the assumption that data is centralized and static. This assumption is no longer tenable. Such methods waste computational and I/O resources when the data is dynamic, and they impose excessive communication overhead when the data is distributed. As a result, the knowledge discovery process is harmed by slow response times. Efficient implementation of incremental data mining ideas in distributed computing environments is thus becoming crucial for ensuring scalability and facilitating knowledge discovery when data is dynamic and distributed. In this paper we address this issue in the context of frequent itemset mining, an important data mining task. Frequent itemsets are most often used to generate correlations and association rules, but more recently they have been used in such far-reaching domains as bio-informatics and e-commerce applications. We first present an efficient algorithm which dynamically maintains the required information in the presence of data updates without examining the entire dataset. We then show how to parallelize the incremental algorithm, so that it can asynchronously mine frequent itemsets. We also propose a distributed algorithm, which imposes low communication overhead for mining distributed datasets. Several experiments confirm that our algorithm results in excellent execution time improvements.

1 Introduction

The field of knowledge discovery and data mining (KDD), spurred by advances in data collection technology, is concerned with the process of deriving interesting and useful patterns from large datasets. Frequent itemset mining is a core data mining task. Its statement is very simple: to find the set of all subsets of items that frequently occur together in database transactions. Although the frequent itemset mining task has a simple statement, it is CPU and I/O intensive, mostly because the large number of itemsets that are typically generated and the large size of the datasets involved in the process.

Now consider the problem of mining frequent itemsets on a dynamic dataset, like those found in e-commerce and web-based domains. The datasets in such domains are constantly updated with fresh data. Let us assume that at some point in time we have computed all frequent itemsets for such a dataset. Now, if the dataset is updated, then

* This work was done while the first author was visiting the Ohio State University

the set of frequent itemsets that we had previously computed would no longer be valid. A naive approach to compute the new set of frequent itemsets would be to re-execute a traditional algorithm on the updated dataset, but this process is not efficient since it is memoryless and ignores previously discovered knowledge, essentially replicating work that has already been done, and possibly resulting in an explosion in the amount of computational and I/O resources required. To address this problem, several researchers have proposed incremental algorithms [15,6,9,14,4], which essentially re-use previously mined information and try to combine this information with the fresh data to efficiently re-compute the new set of frequent itemsets. The problem now is that the size and rate of dataset updates are so large that existing incremental algorithms are ineffective. Therefore, to mine such large and high-velocity datasets, we must rely on high-performance multi-processing computing.

Two dominant approaches for using multiple processors have emerged: distributed memory (where each processor has a private memory), and shared memory (where all processors access common memory). The performance-optimization objectives for distributed memory approaches are different from those of shared memory approaches. In the distributed memory paradigm synchronization is implicit in communication, so the goal becomes communication optimization. In the shared memory paradigm, synchronization stems from locks and barriers, and the goal is to minimize these points. However, the majority of the parallel mining algorithms [1,3,5] suffer from high communication and synchronization overhead. In this paper we propose an efficient parallel and incremental algorithm for mining frequent itemsets on dynamic datasets. Our algorithm makes use of both shared and distributed memory advantages, and it is extremely efficient in terms of communication and synchronization overhead. Extensive experimental evaluation confirm that it results in excellent execution time improvements.

1.1 Problem Definition

The frequent itemset mining task can be stated as follows: Let \mathcal{I} be a set of distinct attributes, called items. Let \mathcal{D} be a set of transactions, where each transaction has a unique identifier (tid) and contains a set of items. A set of items is called an *itemset*. An itemset with exactly k items (where k is a nonnegative integer) is called a k -itemset. The *tidset* of an itemset C corresponds to the set of all transaction identifiers ($tids$) in which the itemset C occurs. The *support count* of C is the number of transactions of \mathcal{D} in which it occurs as a subset. Similarly, the *support* of C , denoted by $\sigma(C)$, is the percentage of transactions of \mathcal{D} in which it occurs as a subset. The itemsets that meet a user specified *minimum support* are referred to as *frequent* itemsets. A frequent itemset is *maximal* if it is not subset of any other frequent itemset.

Using \mathcal{D} as a starting point, a set of new transactions d^+ is added, forming the dynamic dataset Δ (i.e., $\Delta = \mathcal{D} \cup d^+$). Let $s_{\mathcal{D}}$ be the minimum support used when mining \mathcal{D} , and $F_{\mathcal{D}}$ be the set of frequent itemsets obtained. Let Ω be the information kept from the current mining that will be used to enhance the next mining operation. In our case, Ω consists of $F_{\mathcal{D}}$ (i.e., all frequent itemsets, along with their support counts, in \mathcal{D}). An itemset C is frequent in Δ if $\sigma(C) \geq s_{\Delta}$. Note that an itemset C not frequent in \mathcal{D} may become a frequent itemset in Δ . In this case, C is called an *emerged* itemset. If a frequent itemset in \mathcal{D} remains frequent in Δ it is called a *retained* itemset.

The dataset Δ can be divided into n partitions, $\delta_1, \delta_2, \dots, \delta_n$. Each partition δ_i is assigned to a site S_i . We say that Δ is horizontally distributed if its transactions are distributed among the sites. In this case, let $C.sup$ and $C.sup_i$ be the respective support counts of C in Δ and δ_i . We will call $C.sup$ the *global support count* of C , and $C.sup_i$ the *local support count* of C in δ_i . For a given minimum support s_Δ , C is *global frequent* if $C.sup \geq s_\Delta \times |\Delta|$; correspondingly, C is *local frequent* at δ_i , if $C.sup_i \geq s_\Delta \times |\delta_i|$. The set of all maximal global frequent itemsets is denoted as MFI_Δ , and the set of maximal local frequent itemsets at δ_i is denoted as MFI_{δ_i} . The task of mining frequent itemsets in distributed and dynamic datasets is to find F_Δ , with respect to a minimum support s_Δ and, more importantly, using Ω and minimizing access to \mathcal{D} (the original dataset) to enhance the algorithm's performance.

1.2 Related Work

Incremental Mining. Some recent effort has been devoted to the problem of incrementally mining frequent itemsets [9,14,15,4,6]. Some of these algorithms cope with the problem of determining when to update the current model, while others update the model after an arbitrary number of updates [15]. To decide when to update, Lee and Cheung [9] propose the DELI algorithm, which uses statistical sampling methods to determine when the current model is outdated. A similar approach proposed by Ganti *et al* [6] monitors changes in the data stream. An efficient incremental algorithm, called ULI, was proposed by Thomas [14] *et al*. ULI strives to reduce the I/O requirements for updating the set of frequent itemsets by maintaining the previous frequent itemsets and the *negative border* [9] along with their support counts. The entire dataset is scanned just once, but the incremental dataset must be scanned as many times as the size of the longest frequent itemset. This work presents extensions to the ZIGZAG algorithm for incremental mining, presented in [15].

Parallel and Distributed Mining. Often, the size of a new block of data (i.e., d^+) or the rate at which it is inserted is so large that existing incremental algorithms are ineffective. In these cases, parallel or distributed algorithms are necessary. In [12] a parallel incremental method for performing 2-dimensional discretization on a dynamic dataset was presented. [10] gives an overview of a wide variety of distributed data mining algorithms for collective data mining, clustering, and association rule mining. In particular, there has been much research into parallel and distributed algorithms for mining association rules [16,8,11]. In [17] an overview of several of these methods was presented. In [1] three parallel versions of the apriori algorithm were introduced, namely, COUNT DISTRIBUTION, DATA DISTRIBUTION, and CANDIDATE DISTRIBUTION. They conclude that the COUNT DISTRIBUTION approach performs the best. The FDM (FAST DISTRIBUTED MINING) [3] and DMA (DISTRIBUTED MINING OF ASSOCIATION RULES) [5] algorithms result in fewer candidate itemsets and smaller message sizes compared to the COUNT DISTRIBUTION algorithm. Schuster and Wolff propose the DDM (DISTRIBUTED DECISION MINER) algorithm in [13]. They show that DDM is more scalable than COUNT DISTRIBUTION and FDM with respect to the number of sites participating and the minimum support.

2 Distributed, Parallel, and Incremental Algorithm

In this section we describe the algorithm developed to solve the problems defined in the previous section. We start by presenting our incremental algorithm, ZIGZAG. Next, we present a parallel approach for mining the maximal frequent itemsets. Finally, we describe our distributed, parallel and incremental algorithm. We also prove the correctness of the algorithm and present an upper bound for the amount of communication necessary in the distributed and incremental mining operation.

Incremental Algorithm. Almost all algorithms for mining frequent itemsets use the same procedure – first a set of candidates is generated, the infrequent ones are pruned, and only the frequent ones are used to generate the next set of candidates. Clearly, an important issue in this task is to reduce the number of candidates generated. An interesting approach to reduce the number of candidates is to first find MFI_{Δ} . Once MFI_{Δ} is found, it is straightforward to obtain all frequent itemsets (and their support counts) in a single dataset scan, without generating infrequent (and unnecessary) candidates. This approach works because the downward closure property (all subsets of a frequent itemset must be frequent). The number of candidates generated by this approach is generally much smaller than the number of candidates generated to directly find all frequent itemsets. The maximal frequent itemsets has been successfully used in several data mining tasks, including incremental mining of evolving datasets [15].

In [15] an efficient incremental algorithm for mining evolving datasets, ZIGZAG, was proposed. The main idea is to incrementally compute MFI_{Δ} using previous knowledge Ω . This avoids the generation and testing of many unnecessary candidates. Having MFI_{Δ} is sufficient to know which itemsets are frequent; their exact support can be obtained by examining d^+ and using Ω , or, where this is not possible, by examining Δ .

ZIGZAG employs a backtracking search to find MFI_{Δ} . Backtracking algorithms are useful for many combinatorial problems where the solution can be represented as a set $I = \{i_0, i_1, \dots\}$, where each i_j is chosen from a finite *possible set*, P_j . Initially I is empty; it is extended one item at a time, as the search space is traversed. The length of I is the same as the depth of the corresponding node in the search tree. Given a k -candidate itemset, $I_k = \{i_0, i_1, \dots, i_{k-1}\}$, the possible values for the next item i_k comes from a subset $R_k \subseteq P_k$ called the *combine set*. If $y \in P_k - R_k$, then nodes in the subtree with root node $I_k = \{i_0, i_1, \dots, i_{k-1}, y\}$ will not be considered by the backtracking algorithm. Each iteration of the algorithm tries to extend I_k with every item x in the combine set R_k . An extension is valid if the resulting itemset I_{k+1} is frequent and is not a subset of any already known maximal frequent itemset. The next step is to extract the new possible set of extensions, P_{k+1} , which consists only of items in R_k that follow x . The new combine set, R_{k+1} , consists of those items in the possible set that produce a frequent itemset when used to extend I_{k+1} . Any item not in the combine set refers to a pruned subtree. The backtracking search performs a depth-first traversal of the search space, as depicted in Figure 1. In this example the minimum support is 30%. The framed itemsets are the maximal frequent ones, while the cut itemsets are the infrequent ones.

The support computation employed by ZIGZAG is based on the associativity of itemsets, which is defined as follows. Let C be a k -itemset of items $C_1 \dots C_k$, where $C_i \in I$. Let $\mathcal{L}(C)$ be its tidset and $|\mathcal{L}(C)|$ is the length of $\mathcal{L}(C)$ and thus the support count of

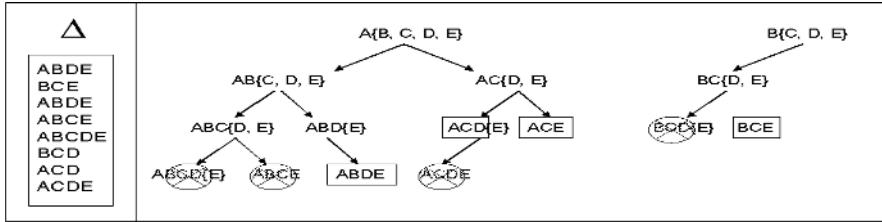


Fig. 1. Backtrack Trees for Items A and B on Δ

C. According to [7], any itemset can be obtained by joining its atoms (individual items) and its support count can be obtained by intersecting the tidsets of its subsets. In the first step, ZIGZAG creates a tidset for each item in d^+ and \mathcal{D} . The main goal of incrementally computing the support is to maximize the number of itemsets that have their support computed based just on d^+ (i.e., retained itemsets), since their support counts in \mathcal{D} are already stored in Ω . To perform a fast support computation, we first verify if the extension $I_{l+1} \cup \{y\}$ is a retained itemset. If so, its support can be computed by just using d^+ and Ω , thereby enhancing the support computation process.

Parallel Search for Maximal Frequent Itemsets. We now consider the problem of parallelizing the search for maximal frequent itemsets in the shared memory paradigm. An efficient parallel search in this paradigm has two main issues: (1) minimizing synchronization, and (2) improving data locality (i.e., maximizing access to local cache). The main idea of our parallel approach is to assign distinct backtrack trees to distinct processors. Note from Figure 1 that the two issues mentioned above can be addressed by this approach. First, there is no dependence among the processors, because each backtrack tree corresponds to a disjoint set of candidates. Since each processor can proceed independently there is no synchronization while searching for maximal frequent itemsets. Second, this approach is very efficient in achieving good data locality, since the support computation of an itemset is based on the intersection of the tidsets of the last two generated subsets. To achieve a suitable level of load-balancing, the backtrack trees are assigned to the processors in a *bag of tasks* approach. That is, we have a *bag* of trees to be processed. Each processor takes one tree, and as soon as it finishes the search on this tree, it takes another task from the *bag*. When the *bag* is empty (i.e., all backtrack trees were processed), all maximal frequent itemsets have been found.

Distributed, Parallel and Incremental Algorithm. We now consider the problem of parallelizing the ZIGZAG algorithm in the distributed memory paradigm. We first present Lemma 1, which is the basic theoretical foundation of our distributed approach.

Lemma 1. A global frequent itemset must be local frequent in at least one partition.

Proof. – Let C be an itemset. If $C.sup_i < s_{\Delta} \times |\delta_i|$ for all $i = 1, \dots, n$, then $C.sup < s_{\Delta} \times |\Delta|$ (since $C.sup = \sum_{i=1}^n C.sup_i$ and $|\Delta| = \sum_{i=1}^n |\delta_i|$), and C cannot be globally frequent. Therefore, if C is a global frequent itemset, it must be local frequent in some partition δ_i . \square

In the first step each site S_i independently performs a parallel and incremental search for MFI_{δ_i} , using ZIGZAG on its dataset δ_i . In the second step each site sends its local MFI to the other sites, and then they join all local MFIs. Now each site knows the set $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$, which is an upper bound for MFI_Δ . In the third step each site independently performs a top down incremental enumeration of the potentially global frequent itemsets, as follows: Each itemset present in the upper bound $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$ is broken into k subsets of size $(k - 1)$. This process iterates generating smaller subsets and incrementally computing their support counts until there are no more subsets to be checked. At the end of this step, each site will have the same set of potentially global frequent itemsets (and the support associated with each of these itemsets).

Lemma 2. $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$ **determines all global frequent itemsets.** *Proof.* – We know from Lemma 1 that if C is a global frequent itemset, so it must be local frequent in at least one partition. If C is local frequent in some partition δ_l , so it must be determined by MFI_{δ_l} , and consequently by $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$. \square

By Lemma 2 all global frequent itemsets were found, but not all itemsets generated in the third step are globally frequent (some of them are just locally frequent). The fourth and final step makes a reduction operation on the local support counts of each itemset, to verify which of them are globally frequent in Δ . The process starts with site S_1 , which sends the support counts of its itemsets (generated in the third step) to site S_2 . Site S_2 sums the support count of each itemset (generated in the third step) with the value of the same itemset obtained from site S_1 , and sends the result to site S_3 . This procedure continues until site S_n has the global support counts of all potentially global frequent itemsets. Then site S_n finds all itemsets that have support greater than or equal to s_Δ , which constitutes the set of all global frequent itemsets, (i.e., F_Δ).

An Upper Bound for the Amount of Communication. Now we present an upper bound for the amount of communication performed during the distributed mining operation. The upper bound calculation is based just on the local MFIs and on the size of the upper bound for MFI_Δ . We divide the upper bound construction into two steps. The first step is related to the local MFI exchange operation. Since each one of the n sites must send its MFI to the other sites, the first term is given by: $\sum_{i=1}^n \sum_{j=1}^{|\text{MFI}_{\delta_i}|} |C_{i,j}|$, where $|C_{i,j}|$ is the size of the j^{th} itemset of the local MFI of site S_i .

The second step is related to the local support count reduction operation. In this operation $n - 1$ sites have to pass their local support counts. The amount of communication for this operation is given by: $(n - 1) \times \sum_{i=1}^n |\text{UB}| (2^{|C_j|} - 1)$, where UB is $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$, and the term $\sum_{i=1}^n (2^{|C_j|} - 1)$ represents the local support counts of all subsets of all itemsets in UB . In our data structure a k -itemset is represented by a set of k integers (of 4 bytes). So, in the worst case (when each itemset is subset of only one itemset in UB), the total amount of communication is given by: $(\sum_{i=1}^n \sum_{j=1}^{|\text{MFI}_{\delta_i}|} |C_{i,j}| + (n - 1) \times \sum_{i=1}^n (2^{|C_j|} - 1)) \times 4$ bytes. This upper bound shows that our approach is extremely efficient in terms of communication overhead, when compared with the amount of communication necessary to transfer all data among the sites.

3 Experimental Evaluation

Our experimental evaluation was carried out on an 8 node dual PENTIUM processor SMP cluster. Each SMP has 1GB of main memory and 120GB of disk space. We assume that each dataset is already distributed among the nodes, and that updates happen in a periodic fashion. We have implemented the parallel program using the MPI message-passing library (MPICH over GM), and POSIX PTHREADS.

We used both real and synthetic datasets for testing the performance of our algorithm. The WPORTAL dataset is generated from the click-stream log of a large Brazilian web portal, and the WCUP dataset is generated from the click-stream log of the 1998 World Cup web site, which is publicly available at <ftp://researchsmp2.cc.vt.edu/pub/>. We scanned each log and produced a respective transaction file, where each transaction is a session of access to the site by a client. Each item in the transaction is a web request. Not all requests were turned into items; to become an item, the request must have three properties: (1) the request method is GET; (2) the request status is OK; and (3) the file type is HTML. A session starts with a request that satisfies the above properties, and ends when there has been no click from the client for 30 minutes. All requests in a session must come from the same client. We also used a synthetic dataset, which has been used as benchmarks for testing previous mining algorithms. This dataset mimics the transactions in a retailing environment [2]. WPORTAL has 3,183 distinct items comprised into 7,786,137 transactions (428MB), while WCUP has 5,271 distinct items comprised into 7,618,927 transactions (645MB). T5I2D8000K has 2,000 distinct items comprised into 8,000,000 transactions (1,897MB).

Performance Comparison. The first experiment we conducted was to empirically verify the advantages of incremental and parallel mining. We compared the execution time of the distributed algorithms (non-incremental, incremental, parallel non-incremental, and parallel incremental search). We varied the number of nodes (1 to 8), the number of processors per node (1 and 2), and the increment size (10% and 20% of the original dataset). In the incremental case, we first mine a determined number of transactions and then we incrementally mine the remaining transactions. For example, for increment sizes of 20%, we first mine 80% of the dataset and then we incrementally mine the remaining 20%. Each dataset was divided into 1, 2, 4, and 8 partitions, according to the number of nodes employed. Figure 2 shows the execution times obtained for different datasets, and parallel and incremental configurations. As we can see, better execution times are obtained when we combine both parallel and incremental approaches. Further, when the parallel configuration is the same, the execution time is better for smaller increment sizes (since the dataset is smaller), but in some cases the parallel performance is greater than the incremental performance, and better results can be obtained by applying the parallel algorithm with larger increment sizes. This is exactly what happens in the experiments with the WCUP and WPORTAL datasets. The algorithm using the parallel search, applied to an increment size of 20% is more efficient than the algorithm with sequential search, applied to an increment size of 10%, for any number of nodes.

The improvements obtained on the real datasets are not so impressive as the improvement obtained on the synthetic one. The reason is that the real dataset has a skewed

data distribution, and therefore the partitions of the real datasets have a very different set of frequent itemsets (and therefore very different local MFIs). On the other hand, the skewness of the synthetic data is very low, therefore each partition of the synthetic dataset is likely to have a similar set of frequent itemsets. From the experiments in the synthetic dataset we observed that $\bigcup_{i=1}^n \text{MFI}_{\delta_i}$ was very similar to each local MFI. This means that the set of local frequent itemsets is very similar to the set of global frequent itemsets, and therefore few infrequent candidates are generated by each node.

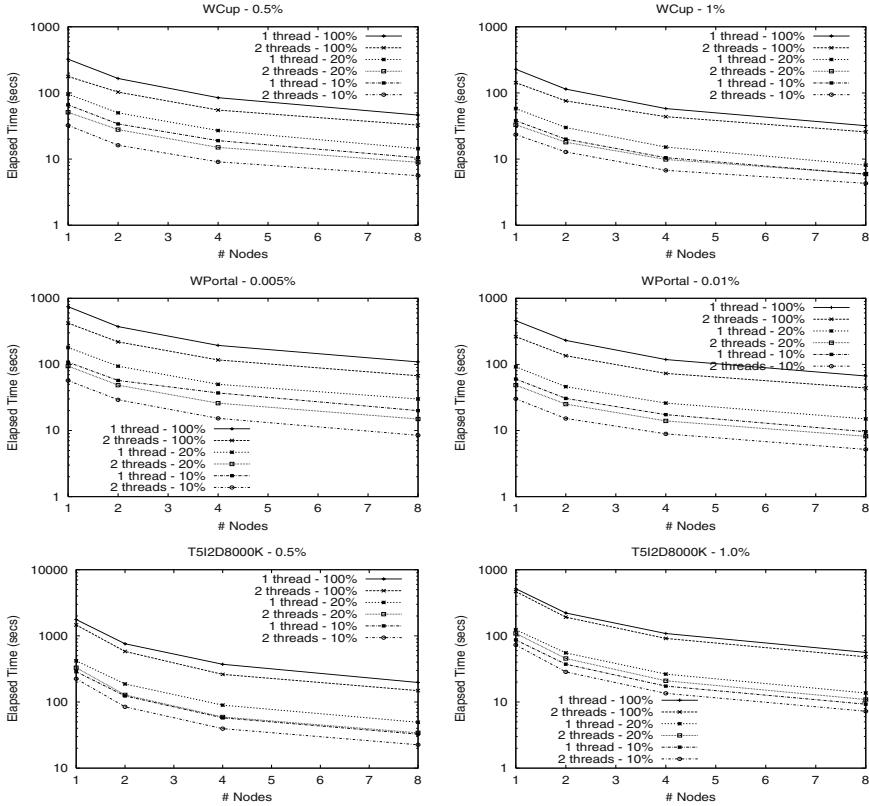


Fig. 2. Total Execution Times on different Datasets.

Parallel Performance. We also investigated the performance of our algorithm in experiments for evaluating the speedup of different parallel configurations. We used a fixed size dataset with increasing number of nodes. The datasets were divided into 1, 2, 4, and 8 partitions, according to the number of nodes employed. With this configuration we performed speedup experiments on 1, 2, 4, and 8 nodes. In order to evaluate only the parallel performance, we used the parallel (two processors) non-incremental algorithm and varied the number of nodes. The speedup is in relation to the sequential non-incremental

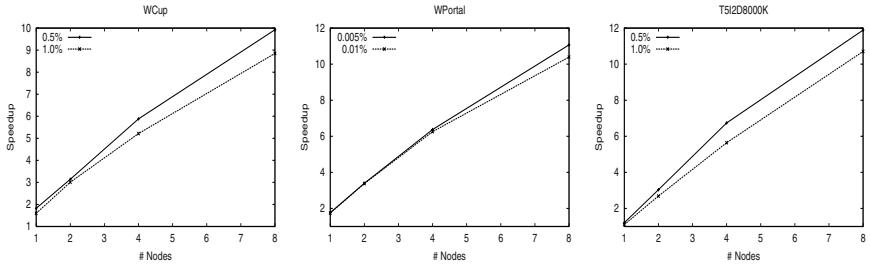


Fig. 3. Speedup of the Parallel Algorithm

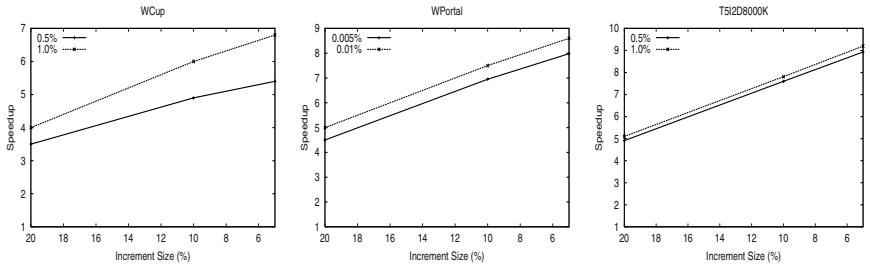


Fig. 4. Speedup of the Incremental Algorithm

algorithm. Figure 3 shows the speedup numbers of our parallel algorithm. The “super-linear” speedups are due to the parallel MFI search (remember that our environment has 8 dual nodes). Also note that the speedup is inversely proportional to the minimum support. This is because for smaller minimum supports the MFI search becomes more complex, and consequently the parallel task becomes more relevant.

Incremental Performance. We also investigated the performance of our algorithm in experiments for evaluating the speedup of different incremental configurations. In this experiment, we first mined a fixed size dataset, and then we performed the incremental mining for different increment sizes (5% to 20%). In order to evaluate only the incremental performance, we used the incremental algorithm with sequential MFI search. We also varied the number of nodes, but the speedup was very similar for different number of nodes, so we show only the results regarding one node. Figure 4 shows the speedup numbers of our incremental algorithm. Note that the speedup is in relation to re-mining the entire dataset. As is expected, the speed is inversely proportional to the size of the increment. Also note that better speedups are achieved by greater minimum supports. We observed that, for the datasets used in this experiment, the proportion of retained itemsets (itemsets that are computed by examining only d^+ and Ω) is larger for greater minimum supports.

4 Conclusions

In this paper we considered the problem of mining frequent itemsets on dynamic datasets. We presented an efficient distributed and parallel incremental algorithm to deal with this problem. Experimental results confirm that our algorithm results in execution time improvement of more than one order of magnitude when compared against a naive approach. The efficiency of our algorithm stems from the fact that it makes use of the MFI, reducing both the number of candidates processed and the amount of communication necessary. The MFI is updated by an efficient parallel and asynchronous backtracking search.

References

1. R. Agrawal and J. Shafer. Parallel mining of association rules. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 962–969, 1996.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int'l Conf. on Very Large Databases*, SanTiago, Chile, June 1994.
3. D. Cheung, J. Han, V. Ng, A. Fu, , and Y. Fu. A fast distributed algorithm for mining association rules. In *4th Int'l. Conf. Parallel and Distributed Info. Systems*, 1996.
4. D. Cheung, S. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. of the 5th Int'l. Conf. on Database Systems for Advanced Applications*, pages 1–4, April 1997.
5. D. Cheung, V. Ng, A. Fu, , and Y. Fu. Efficient mining of association rules in distributed databases. In *IEEE Trans. on Knowledge and Data Engg.*, volume 8, pages 911–922, 1996.
6. V. Ganti, J. Gehrke, and R. Ramakrishnan. Demon: Mining and monitoring evolving data. In *Proc. of the 16th Int'l Conf. on Data Engineering*, pages 439–448, San Diego, USA, 2000.
7. K. Gouda and M. Zaki. Efficiently mining maximal frequent itemsets. In *Proc. of the 1st IEEE Int'l Conf. on Data Mining*, San Jose, USA, November 2001.
8. E.-H. Han, G. Karypis, , and V. Kumar. Scalable parallel data mining for association rules. In *ACM SIGMOD Conf. Management of Data*, 1997.
9. S. Lee and D. Cheung. Maintenance of discovered association rules: When to update? In *Research Issues on Data Mining and Knowledge Discovery*, 1997.
10. Byung-Hoon Park and Hillol Kargupta. Distributed data mining: Algorithms, systems, and applications. In Nong Ye, editor, *Data Mining Handbook*, 2002.
11. J.S. Park, M. Chen, , and P. S. Yu. CACTUS - clustering categorical data using summaries. In *ACM Int'l. Conf. on Information and Knowledge Management*, 1995.
12. S. Parthasarathy and A. Ramakrishnan. Parallel incremental 2d discretization. In *Proc. IEEE Int'l Conf. on Parallel and Distributed Processing*, 2002.
13. A. Schuster and R. Wolff. Communication efficient distributed mining of association rules. In *ACM SIGMOD Int'l. Conf. on Management of Data*, 2001.
14. S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules. In *Proc. of the 3rd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, August 1997.
15. A. Veloso, W. Meira Jr., M. Bunte, S. Parthasarathy, and M. Zaki. Mining frequent itemsets in evolving databases. In *Proc. of the 2nd SIAM Int'l Conf. on Data Mining*, USA, 2002.
16. M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New parallel algorithms for fast discovery of association rules. *Data Mining and Knowledge Discovery: An International Journal*, 4(1):343–373, December 1997.
17. M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, December 1999.

A Volumetric FFT for BlueGene/L

Maria Eleftheriou, José E. Moreira, Blake G. Fitch, and Robert S. Germain

IBM Thomas J. Watson Research Center

Yorktown Heights, NY 10598-0218 {mariae,jmoreira,bgf,rgermain}@us.ibm.com

Abstract. BlueGene/L is a massively parallel supercomputer organized as a three-dimensional torus of compute nodes. A fundamental challenge in harnessing the new computational capabilities of BlueGene/L is the design and implementation of numerical algorithms that scale effectively on thousands of nodes. A computational kernel of particular importance is the Fast Fourier Transform (FFT) of three-dimensional data. In this paper, we present the approach we are taking in BlueGene/L to produce a scalable FFT implementation. We rely on a volume decomposition of the data to take advantage of the toroidal communication topology. We present experimental results using an MPI-based implementation of our algorithm, in order to test the basic tenets behind our decomposition and to allow experimentation on existing platforms. Our preliminary results indicate that our algorithm scales well on as many as 512 nodes for three-dimensional FFTs of size $128 \times 128 \times 128$.

1 Introduction

BlueGene/L [1,2] is a massively parallel computer system being developed at the IBM Thomas J. Watson Research Center in collaboration with Lawrence Livermore National Laboratory. A fully configured BlueGene/L machine will consist of 65,536 dual-processor nodes interconnected as a $32 \times 32 \times 64$ three-dimensional torus topology, and is projected to deliver 360 Teraflops of peak computing power when using both processor cores on each node. BlueGene/L will represent a significant advance in resources for scientific simulation, enabling computational experiments far beyond the capacity of existing machines. In particular, we expect BlueGene/L to lead to new advances in the computational life sciences, with a special focus on the mechanisms of protein folding.

A fundamental challenge in harnessing the new computational capabilities of BlueGene/L is the design and implementation of numerical algorithms that scale effectively to thousands of nodes. A computational kernel of particular importance is the Fast Fourier Transform (FFT) of three-dimensional data. FFTs are at the core of many important applications, including computational fluid dynamics, electronic structure calculation, and structural analysis.

In the context of protein science, three-dimensional FFTs are a key component of the Particle Particle Particle Mesh Ewald (P3ME) [4,12] for computing the evolution of molecular systems. An analytical study of the scalability in BlueGene/L of molecular dynamics applications that simulate protein systems, utilizing three-dimensional FFTs for the calculation of the long range forces, has been reported in a previous work [7].

The most common approach to computing the FFT of an $N \times N \times N$ data array in parallel on multiple nodes is to use a technique called slab decomposition [8,10]. With

slab decomposition, the data is partitioned along a single axis. For example, to perform the computation on N nodes, each node would be assigned a slab of size $N \times N \times 1$. While effective from a performance perspective, the scalability of this method is limited by N , the extent of the data along a single axis. This becomes a problem when one wants to scale to very large numbers of nodes, such as in BlueGene/L. Slab decomposition algorithms also do not map naturally to a torus interconnected machine like BlueGene/L.

In this paper, we present the approach we are taking in BlueGene/L to produce a more scalable implementation of three-dimensional FFTs. We rely on a volume decomposition of the data. With volumetric decomposition, when an $N \times N \times N$ array is distributed on a $P \times P \times P$ node grid, each node is assigned an $\frac{N}{P} \times \frac{N}{P} \times \frac{N}{P}$ subcube of data. The volumetric decomposition achieves better scalability, since all data is decomposed along all three axes. Volumetric decomposition algorithms map more naturally to toroidal topologies.

We validate the effectiveness of our approach to computing three-dimensional FFTs though measurements on real machines. Prior to the availability of large counts of BlueGene/L nodes, we use a 512-processor RS/6000 SP to play the role of an $8 \times 8 \times 8$ BlueGene/L configuration. We compare the performance of our volumetric decomposition implementation with that of the FFTW library [9]. FFTW is a well established library for computing FFTs. It relies on the slab decomposition approach for running on multiple nodes, and it is widely recognized as a high performance implementation. Our experiments show that for a given problem size N , FFTW has better performance on small number of nodes, but that our implementation scales better and overtakes FFTW on large number of nodes.

The rest of the paper is organized as follows. Section 2 provides background information on the computation of the three-dimensional FFTs and gives a brief overview of the BlueGene/L machine. Section 3 introduces our volumetric decomposition algorithm for computing the three-dimensional FFT on BlueGene/L. Section 4 describes our experimental measurements and presents the corresponding results. Section 5 gives an overview of related work on parallel FFTs. Finally, Section 6 presents our conclusion.

2 Background

To better explain how we compute three-dimensional FFTs on BlueGene/L, we provide the following background information. First, we give a mathematical description of the computation of three-dimensional FFTs. Then, we provide an overview of the architecture of BlueGene/L to the level relevant to our volumetric decomposition implementation.

2.1 The Three-Dimensional FFT

Let $x(0 : n - 1)$ be a vector of n complex numbers. The one-dimensional FFT of vector x is a n -element vector $y(0 : n - 1) = \text{fft}_1(x(0 : n - 1))$ computed by

$$y(k) = \sum_{j=0}^{n-1} x(j) e^{-2\pi i \frac{jk}{n}} \quad (1)$$

Let $X(0 : N_x - 1, 0 : N_y - 1, 0 : N_z - 1)$ be a three-dimensional array of $N_x \times N_y \times N_z$ complex numbers. The three-dimensional FFT of array x is an $N_x \times N_y \times N_z$ array $Y(0 : N_x - 1, 0 : N_y - 1, 0 : N_z - 1) = \text{fft}_3(x(0 : N_x - 1, 0 : N_y - 1, 0 : N_z - 1))$ of complex numbers computed by

$$Y(k_x, k_y, k_z) = \sum_{j_x=0}^{N_x-1} \left(\sum_{j_y=0}^{N_y-1} \left(\sum_{j_z=0}^{N_z-1} X(j_x, j_y, j_z) e^{-2\pi i \frac{j_z k_z}{N_z}} \right) e^{-2\pi i \frac{j_y k_y}{N_y}} \right) e^{-2\pi i \frac{j_x k_x}{N_x}}. \quad (2)$$

This computation can be decomposed in three stages. First, we compute $Y_z(j_x, j_y, k_z)$ such that

$$Y_z(j_x, j_y, k_z) = \sum_{j_z=0}^{N_z-1} X(j_x, j_y, j_z) e^{-2\pi i \frac{j_z k_z}{N_z}}, \quad \forall j_x, j_y, 0 \leq j_x < N_x, 0 \leq j_y < N_y. \quad (3)$$

That is, $Y_z(j_x, j_y, :)$ is the one-dimensional FFT of $X(j_x, j_y, :)$ for all (j_x, j_y) pairs. In the second stage, we compute $Y_{zy}(j_x, k_y, k_z)$ such that

$$Y_{zy}(j_x, k_y, k_z) = \sum_{j_y=0}^{N_y-1} Y_z(j_x, j_y, k_z) e^{-2\pi i \frac{j_y k_y}{N_y}}, \quad \forall j_x, k_z, 0 \leq j_x < N_x, 0 \leq k_z < N_z. \quad (4)$$

That is, $Y_{zy}(j_x, :, k_z)$ is the one-dimensional FFT of $Y_z(j_x, :, k_z)$ for all (j_x, k_z) pairs. Finally, in the last stage, we complete the computation of the three-dimensional FFT by evaluating

$$Y(k_x, k_y, k_z) = \sum_{j_x=0}^{N_x-1} Y_{zy}(j_x, k_y, k_z) e^{-2\pi i \frac{j_x k_x}{N_x}}, \quad \forall k_y, k_z, 0 \leq k_y < N_y, 0 \leq k_z < N_z. \quad (5)$$

That is, $Y(:, k_y, k_z)$ is the one-dimensional FFT of $Y_{zy}(:, k_y, k_z)$ for all (k_y, k_z) pairs. It is easy to verify that if we substitute first Equation (4) and then Equation (3) into Equation (5), we get back Equation (2).

Figure 1 illustrates this approach to computing the three-dimensional FFT of an array of size $N_x \times N_y \times N_z$. We first (a) evaluate $N_x \times N_y$ independent one-dimensional FFTs of size N_z along the z dimension. We then (b) evaluate $N_x \times N_z$ independent one-dimensional FFTs of size N_y along the y dimension. And finally (c), we evaluate $N_y \times N_z$ independent one-dimensional FFTs of size N_x along the x dimension.

2.2 The BlueGene/L Supercomputer

The computational core of the BlueGene/L supercomputer is organized as a $32 \times 32 \times 64$ three-dimensional torus of compute nodes. Each of these 65,535 compute nodes consists of two PowerPC 440 processors sharing 256 Mbytes of memory. Each processor is capable of performing two fused multiply-adds per cycle, for a total of 8 floating-point operations per cycle per compute node. At the target operating frequency of 700 MHz, this corresponds to a peak computing power of 5.6 Gflops per compute node, or approximately 360 Teraflops peak for the entire system.

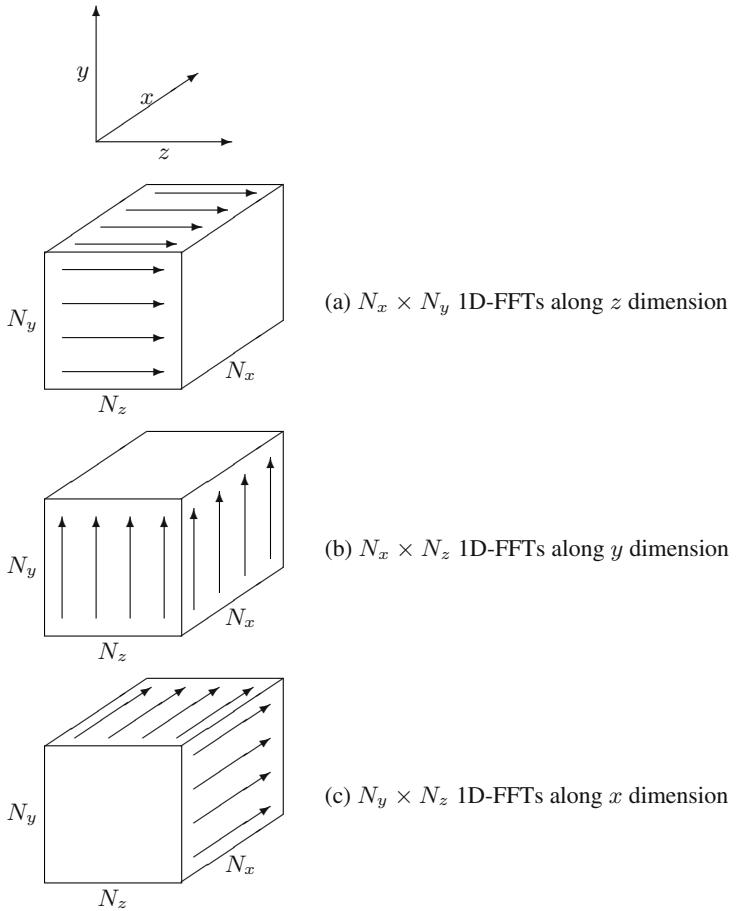


Fig. 1. The computation steps of a three-dimensional FFT.

Inter-node communication is performed exclusively through message-passing. Data packets are routed entirely in hardware, by the torus network, from the source node to the destination node. In the default mode of operation of BlueGene/L, one processor of a compute node is dedicated to running computation code, while the other is dedicated to handle communications (injecting and retrieving packets to and from the torus network). In this default mode, the maximum achievable application performance is 2.8 Gflops per compute node, or approximately 180 Teraflops for the entire system.

The BlueGene/L system software infrastructure provides three layers of inter-node communication services. At the lowest level, an active packets library maps directly to the hardware capabilities. An active packet is limited in size to 256 bytes, which is the maximum size of hardware packets for inter-node communication. When an active packet is received at its destination node, it causes a corresponding handler code to be executed. The next level of communication services is represented by active messages.

Like active packets, a handler code is executed when an active message is received at its destination node. The difference is that active messages can be of arbitrary length. The third and highest level of communication services is an optimized MPI library, which provides users with a familiar and portable environment.

3 A Volumetric FFT for BlueGene/L

We designed a three-dimensional FFT for the BlueGene/L machine based on a volumetric decomposition of the data. We used the FFTW library for performing one-dimensional FFTs on a single processor. This leverages the high-performance routines in the FFTW library. Data communication in our FFT can use either the MPI layer or the active message layer from the BlueGene/L communication services.

Let $A(0 : N_x - 1, 0 : N_y - 1, 0 : N_z - 1)$ be an $N_x \times N_y \times N_z$ array of complex numbers block distributed onto a $P_x \times P_y \times P_z$ grid of nodes. That is, each node (x, y, z) stores a section of size $n_x \times n_y \times n_z$ ($n_x = \frac{N_x}{P_x}$, $n_y = \frac{N_y}{P_y}$, $n_z = \frac{N_z}{P_z}$) of array A in its local memory. More precisely, each node has a local array $A'(0 : n_x - 1, 0 : n_y - 1, 0 : n_z - 1)$ of size $n_x \times n_y \times n_z$. If we denote by $A'(i, j, k)[x, y, z]$ the element (i, j, k) of local array A' in node (x, y, z) , then

$$A'(i, j, k)[x, y, z] \equiv A(xn_x + i, yn_y + j, zn_z + k). \quad (6)$$

For clarity of explanation, we impose the following restrictions on the sizes of the local array A' : $n_x = \alpha \times P_z$, $n_y = \beta \times P_z$, and $n_z = \gamma \times P_x$, where α , β , and γ are integers.

We wish to perform an FFT on the three-dimensional array A . We do that following the approach described in Section 2.1. We compute $N_x \times N_y$ one-dimensional FFTs along the z dimension, followed by $N_x \times N_z$ one-dimensional FFTs along the y dimension, followed by $N_y \times N_z$ one-dimensional FFTs along the x dimension. We describe how we perform the one-dimensional FFTs along the z dimension. The computations along the y and x dimensions are similar.

Since the $N_x \times N_y$ one-dimensional FFTs along the z dimension are all independent, we can consider the case of a one-dimensional grid of P_z nodes that has to compute $n_x \times n_y$ one-dimensional FFTs of size N_z . Let $A(0 : n_x - 1, 0 : n_y - 1, 0 : N_z - 1)$ be an $n_x \times n_y \times N_z$ array of complex numbers, block distributed along the z dimension onto P_z nodes. That is, if $A'(0 : n_x - 1, 0 : n_y - 1, 0 : n_z - 1)[p]$ is the local array in node p , then

$$A'(i, j, k)[p] \equiv A(i, j, pn_z + k). \quad (7)$$

This initial distribution of array A is shown in Figure 2.

We compute each one-dimensional FFT (of size N_z) in a single node. That is, we do not parallelize each one-dimensional FFT. Instead, we exploit parallelism across the multiple independent one-dimensional FFTs. We accomplish that by first redistributing the data along the y dimension onto the P_z nodes. If $A''(0 : n_x - 1, 0 : \beta - 1, 0 : N_z - 1)[p]$ is the local array in node p , then

$$A''(i, j, k)[p] \equiv A(i, p\beta + j, k). \quad (8)$$

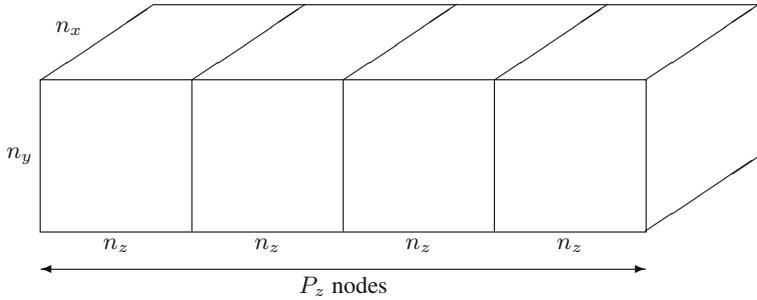


Fig. 2. Initial data distribution of an $n_x \times n_y \times N_z$ array onto P_z nodes.

This new distribution of array A is shown in Figure 3. Once the data is in this new distribution form, each node p computes $\beta \times n_x$ independent one-dimensional FFTs of size N_z that are stored in its local array A'' . For these one-dimensional FFT computations, we use the services provided by the FFTW library. Once the FFTs are computed, data is redistributed back to the original distribution described by Equation (7).

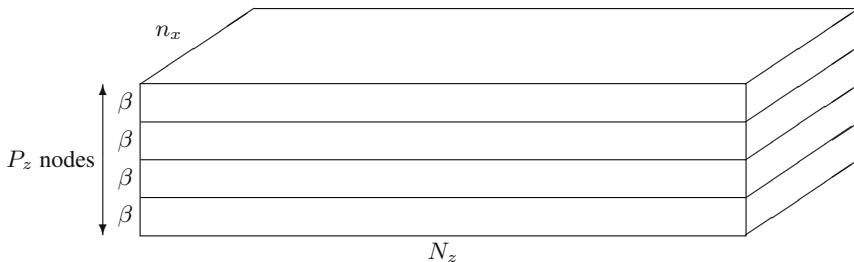


Fig. 3. Data redistributed for computation of $\beta \times n_x$ one-dimensional FFTs one each node.

Summarizing, the computation of a three-dimensional FFT is performed in three steps, along the z , y , and x dimensions. In each step, we first redistribute the data, perform one-dimensional FFTs, and redistribute the data back to its original form. To minimize the number of communication operations, in our implementation we combined the post-FFT data redistribution of one step with the pre-FFT data redistribution of the next step. That way, we only have to perform four data redistributions, as opposed to six.

4 Performance Evaluation

We validated the correctness and evaluated the performance of our three-dimensional FFT implementation by executing our code on a real machine. Since only a very limited number of BlueGene/L nodes are available at this time, we conducted our experiments

on a cluster of IBM POWER4 servers. Each node in this cluster is an 8-way symmetric multiprocessor, with the processors running at 1.3 GHz. The nodes are interconnected by a high-performance interconnection network. Each node attaches to this network through two 320 Mbyte/s bidirectional channels.

We performed various runs of the MPI version of our FFT code on this platform. We varied both the number of MPI tasks and the size of the data being transformed. The MPI tasks were organized in a three-dimensional grid using the MPI topology constructs. At all times, there were enough physical processors to dedicate one processor per task.

We also performed experiments on the same platform using the FFTW library to compute the three-dimensional FFT of data of varying sizes. This allows us to compare the performance of our implementation with that of a well established library.

Figure 4 shows the performance of the volumetric FFT of three different sizes ($128 \times 128 \times 128$, $256 \times 256 \times 256$, and $512 \times 512 \times 512$) on different numbers of tasks. The running times reported here are averages of 9 runs. Each experiment was run 10 times and the first run was discarded. The graph shows that the volumetric FFT scales well with task count. The speedup for the smallest size FFT ($128 \times 128 \times 128$) on 128, 256 and 512 processors, are about 28, 74 and 93, respectively. This demonstrates good scalability of our code for this problem size.

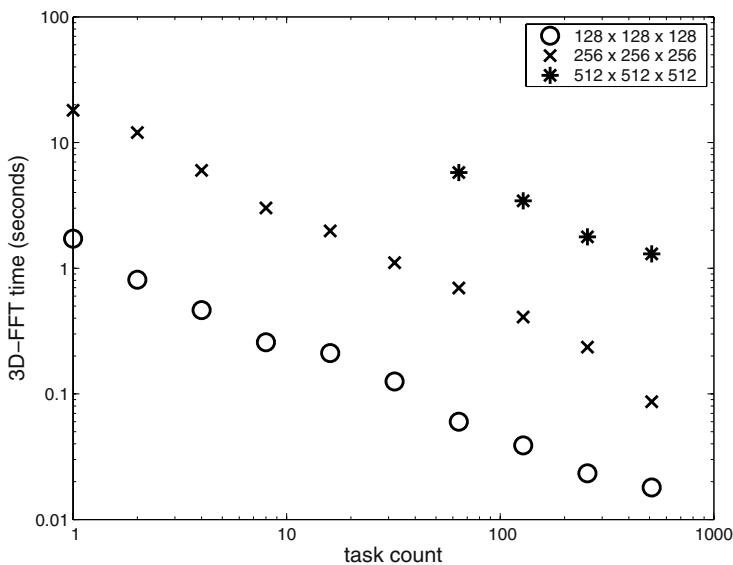


Fig. 4. Performance measurements for the volumetric FFT, for three different problem sizes (128^3 , 256^3 , and 512^3) and various task counts.

Figure 5 compares the performance of volumetric FFT with FFTW. The results show that the FFTW is faster on small task counts and the volumetric FFT is faster on large

task counts. This is due to the superior scalability of a volumetric approach as compared with a slab decomposition.

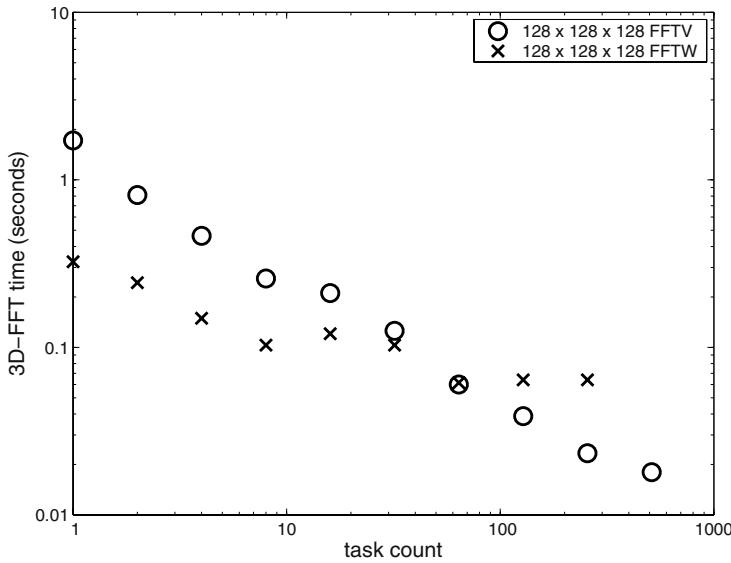


Fig. 5. Comparing the performance of FFTW with the volumetric FFT (FFFT-V). Results are for the smallest problem size (128^3) on varying number of tasks.

5 Related Work

The parallel computation of three-dimensional FFTs has been studied from two different view points. The first one is aimed at parallelizing the basic one-dimensional FFT. Examples include the work of Zapata et al [11], which present an algorithm that adapts an optimum sequential FFT algorithm to a specific parallel architecture, and the work of Edelman et al [6], which propose an algorithm that reduces the communication overhead.

In the second group, three-dimensional FFTs are carried out as successive, independent one-dimensional local FFTs. Frigo and Johnson [8,9] developed an FFT package that computes one- or multi-dimensional FFTs and achieves superior performance. Recently, Crammer and Board developed a three-dimensional FFT for clusters of workstations [3]. Ding et al [5] proposed an algorithm with an in-place data transpose, which reduces the memory requirements for the operation. The above approaches use the slab decomposition as a mean of parallelism. Although slab decomposition has been shown to be optimal in conventional parallel machines, it is not suitable for exploiting parallelism on massively parallel computers such as BlueGene/L. The main limitation in slab decomposition comes from the fact that we cannot take advantage of more processors than the number of elements along a single dimension of the three-dimensional FFT.

Another approach is to use volumetric decomposition. Haybes and Cote [10] present an algorithm that minimizes the latency cost by using a volumetric decomposition scheme.

In this work we presented a volumetric decomposition FFT that belongs to the second group. (Parallelism across multiple independent FFTs.) The three-dimensional FFT is carried out as three successive sets of independent one-dimensional FFTs. The one-dimensional FFTs are performed locally. This reduces the communication operations to four data redistribution operations. In addition, we use highly optimized one-dimensional FFTs from FFTW. The major advantage of the volumetric decomposition approach we adopt is that we can achieve better scalability, since the data is decomposed along all three axis.

6 Conclusions

We have discussed the design and implementation of a three-dimensional FFT for the BlueGene/L supercomputer. We base our approach on a volumetric decomposition of data. This decomposition maps more naturally to the torus topology of BlueGene/L. We rely on the FFTW library to perform the basic one-dimensional FFT operations in our algorithm.

Experiments performed on an IBM POWER4 cluster show that our approach scales well with the number of tasks, even when computing a $128 \times 128 \times 128$ FFT on as many as 512 tasks. Our implementation also performs better on large number of nodes than FFTW, which is based on a slab decomposition approach.

Acknowledgment. The authors would like to thank Bob Walkup, of the IBM Thomas J. Watson Research Center, for his help in running the FFT benchmarks.

References

1. N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.
2. G. Almasi, G. S. Almasi, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castaños, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, A. Deutsch, M. B. Dombrowsky, W. Donath, M. Eleftheriou, B. Fitch, J. Gagliano, A. Gara, R. Germain, M. E. Giampapa, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, A. P. Lanzetta, D. Lieber, M. Lu, M. Mendell, L. Mok, J. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, R. Rand, R. Regan, R. Sahoo, A. Sanomiya, E. Schenfeld, S. Singh, P. Song, B. D. Steinmacher-Burow, K. Strauss, R. Swetz, T. Takken, P. Vranas, and T. J. C. Ward. Cellular supercomputing with system-on-a-chip. In *Proceedings of International Solid-State Circuits Conference (ISSCC'02)*, 2002.
3. C. E. Cramer and J. A. Board. The development and integration of a distributed 3D FFT for a cluster of workstations. In *4th Annual Linux Showcase and Conference*, pages 121–128, Atlanta, GA, October 2000.
4. M. Deserno and C. Holm. How to mesh up Ewald sums. I. A theoretical and numerical comparison of various particle mesh routines. *J. Chem. Phys.*, 109(18):7678–7693, 1998.

5. H. Q. Ding, R. D. Ferraro, and D. B. Gennery. A portable 3D FFT package for distributed-memory parallel architecture. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1995.
6. A. Edelman, P. McCorquodale, and S. Toledo. The future fast Fourier transform? In *SIAM J. Sci. Comput.*, volume 20, pages 1094–1114, 1999.
7. B. G Fitch, R. S. Germain, M. Mendell, J. Pitera, A. Rayshubskiy, Y. Sham, F. Suits, W. Swope, Y. Zhestkov, and R. Zhou. Blue Matter, an application framework for molecular simulation on Blue Gene. *Journal of Parallel and Distributed Computing*, 2003. To appear.
8. M. Frigo and S. G. Johnson. The fastest Fourier transform in the west. Technical Report MIT-LCS-TR-728, Laboratory for Computing Sciences, MIT, Cambridge, MA, 1997.
9. M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, volume 3, pages 1381–1384, 1998.
10. P. D. Haynes and M. Cote. Parallel fast Fourier transforms for electronic structure calculations. *Comp. Phys. Comm.*, 130:121, 2000.
11. E. L. Zapata, F. F. Rivera, J. Benavides, J. M. Garazo, and R. Peskin. Multidimensional fast Fourier transform into fixed size hypercubes. *IEE Proceedings*, 137(4):253–260, July 1990.
12. R. Zhou, E. Harder, H. Xu, and B. J. Berne. Efficient multiple time step method for use with Ewald and particle mesh Ewald for large biomolecular systems. *J. Chem. Phys.*, 115:2348–2358, 2001.

A Nearly Linear-Time General Algorithm for Genome-Wide Bi-allele Haplotype Phasing*

Will Casey¹ and Bud Mishra^{1,2,3}

¹ Courant Institute of Mathematical Sciences, 251 Mercer St.,
New York, New York USA

{wcasey, mishra}@cims.nyu.edu

² Cold Spring Harbor Lab,

1 Bungtown Rd,

Cold Spring Harbor, New York, USA

³ Tata Institute of Fundamental Research,

Homi Bhabha Road,

Mumbai 400 005, India

Abstract. The determination of feature maps, such as STSs (sequence tag sites), SNPs (single nucleotide polymorphisms) or RFLP (restriction fragment length polymorphisms) maps, for each chromosome copy or *haplotype* in an individual has important potential applications to genetics, clinical biology and association studies. We consider the problem of reconstructing two haplotypes of a diploid individual from genotype data generated by mapping experiments, and present an algorithm to recover haplotypes. The problem of optimizing existing methods of SNP phasing with a population of diploid genotypes has been investigated in [7] and found to be NP-hard. In contrast, using single molecule methods, we show that although haplotypes are not known and data are further confounded by the mapping error model, reasonable assumptions on the mapping process allow us to recover the co-associations of allele types across consecutive loci and estimate the haplotypes with an efficient algorithm. The haplotype reconstruction algorithm requires two stages: Stage I is the detection of polymorphic marker types, this is done by modifying an EM-algorithm for Gaussian mixture models and an example is given for RFLP sizing. Stage II focuses on the problem of *phasing* and presents a method of local maximum likelihood for the inference of haplotypes in an individual. The algorithm presented is nearly linear in the number of polymorphic loci. The algorithm results, run on simulated RFLP sizing data, are encouraging, and suggest that the method will prove practical for haplotype phasing.

1 Introduction

Diploid organisms carry two mostly similar copies of each chromosome, referred to as *Haplotypes*. Variations in a large population of haplotypes at specific loci

* Work reported in this paper is funded by grants from NSF Cubic program, DARPA, HHMI biomedical support research grant, US DOE, US Air Force, NIH, New York office of Science and Technology & Academic Research

are called *Polymorphisms*. The co-associations of these variations across the loci indices are of intense interest in disease research. Genetic markers such as RFLPs and SNPs are the units to which this paper's association studies apply.

The problems and difficulties of inferring diploid haplotypes through the use of population data have been extensively investigated ([4], [6], [10], [13], [7]), and widely acknowledged.

Our approach focuses on the use of multiple independent mapping experiments (on, for example, a collection of large DNA fragments) as the base data to infer haplotypes. Single molecule methods and technologies, such as optical mapping and polony ([9]), may accommodate the high-through-put for haplotyping diploids in a population.

We consider the problem of reconstructing two haplotypes from genotype data generated by general mapping techniques, with a focus on single molecule methods. The genotype data is a set of observations $D = \langle d_i \rangle_{i \in [1 \dots N]}$. Each observation is derived from one of the two distinct but unknown haplotypes. Each observation $d_i = \langle d_{ij} \rangle_{j \in [1 \dots M]}$ is a set of observations over the loci index j with $d_{ij} \in \mathbb{R}^r$.

Mapping processes are subject to noise and we assume a Gaussian model $d_{ij} \sim N(\mu, \sigma)$ with parameter μ depending on the underlying haplotype of d_i . Mapping processes shall be designed to discriminate the polymorphic allele types in the data space for each loci; hence the set of observation points $\langle d_{ij} \rangle_{i \in [1 \dots N]}$ are derived from a mixed distribution which displays bi-modal characteristics in the presence of a polymorphic feature. By estimating the parameters of the distribution, we can assign a posteriori distribution that a particular point in \mathbb{R}^r is derived from an allele type.

Since the mapping errors for d_{ij} and $d_{ij'}$ are assumed to be independent, computing the posteriori distribution for haplotypes with product allele types is straightforward, and is a major advantage of utilizing single molecule methods in association studies.

The *Phasing* problem is to determine which haplotypes are most likely generating the observed genotype data. The challenge is to infer the most likely parameter correlations across the loci index accounting for the posteriori.

2 Mapping Techniques

We wish to present applications for a wide spectrum of mapping techniques, to allow a large number of polymorphic markers (SNPs, RFLPs, micro-insertions and deletions, microsatellite copy numbers) to be used in an association study.

This paper focuses on mapping techniques capable of 1) discriminating alleles at polymorphic loci and 2) providing haplotype data at multiple loci. A mapping technique designed for association studies should be discriminating: for each polymorphic loci, data points in the data space \mathbb{R}^r which are derived from separate allele types should form distinct clusters in the data. A technique which allows observation of a single haplotype over multiple loci may be necessary for an efficient phasing algorithm. Thus single molecule methods are of particular

interest to us. Our models and analysis are influenced by their applicability to association studies.

As an example, consider the length between two restriction fragments. The observable x is modeled as a random variable depending on the actual distance μ .

$$P(x|\mu) = \frac{1}{\sqrt{2\pi\mu}} \exp\left(\frac{-(x-\mu)^2}{2\mu}\right)$$

Isolate a specific pair of restriction sites on one of the haplotypes H_1 , and let the distance between them be given by μ_1 . The distance between the homologous pair on the second haplotype H_2 is given by μ_2 . An observation x from the genotype data is then either derived from H_1 or H_2 , denoted $x \sim H_1$ and $x \sim H_2$ respectively.

$$\begin{aligned} P(x) &= P(x|x \sim H_1)P(x \sim H_1) + P(x|x \sim H_2)P(x \sim H_2) \\ &= \frac{1}{\sqrt{2\pi\mu_1}} \exp\left(\frac{-(x-\mu_1)^2}{2\mu_1}\right) P(x \sim H_1) \\ &\quad + \frac{1}{\sqrt{2\pi\mu_2}} \exp\left(\frac{-(x-\mu_2)^2}{2\mu_2}\right) P(x \sim H_2) \end{aligned}$$

With the RFLP sizing mapping technique, observable $d_{ij}, d_{ij'}$ have independent error sources depending on loci-specific parameters. The set $\{d_{ij}, i \in [1 \dots N]\}$ provides points in \mathbb{R} which may be discriminated using a Gaussian Mixture model. Due to the uncertainty of mapping and underlying haplotypes we have chosen to model data as posteriori distribution $\alpha(x) = [P(x \sim H_1), P(x \sim H_2)]$ rather than determined allele types.

This paper is organized into five sections: section 1 defines the problems we are addressing; section 2 explains the EM-Algorithm application; section 3 discusses the phasing problem; section 4 outlines algorithm implementation and provides examples; and section 5 briefly discusses results, technologies and future work.

2.1 EM-Algorithm for Detection of Bi-allelic Polymorphisms

The use of the EM-Algorithm for inferring parameters of a Gaussian mixture model is a well-known method (see [5] [12]), and useful in this context as well. We postulate that in the presence of polymorphisms at loci j , informative mapping data will display a bi-modal distribution in the data space \mathbb{R}^r . Detailed computations for the E-Step and M-Step are provided in the full paper. For each locus j the EM-algorithm is run until convergence occurs; the result being: $\langle \alpha_k(x), \hat{\Phi} = \langle \hat{\mu}_1, \mu_2, \dots, \mu_K, \sigma \rangle \rangle$. Here α is a posteriori probability that data point x is derived from allele type $k \in [1, 2, \dots, K]$.

Criteria for Polymorphisms. Let $\hat{\Phi}(D)$ denote the limit of the EM-algorithm with data set D at the loci j . A critical question is: When will a loci

exhibit 2 specific allele variations? Setting $K = 2$ for the remainder of the paper (hence $\hat{\Phi} = \langle \mu_1, \mu_2, \sigma \rangle$, we define polymorphic loci as events:

$$X(D) = \begin{cases} 1 & \text{if } \hat{\Phi}(D) : |\hat{\mu}_1 - \hat{\mu}_2| - \delta > 0 \\ 0 & \text{o.w.} \end{cases}$$

The individual experiment data $\{d_{ij} : i \in [1\dots N]\}$ is mapped to posteriori probability measures over the allele classes producing a probability function $\alpha(y)$ reflecting our confidence (in the presence of mapping error) that point y corresponds to one of our allele types. For polymorphism assignments, false positives are unlikely to disturb the phasing, while false negatives affect the size of phased contigs.

3 Phasing Genotype Data

Phasing is the problem of determining co-association of alleles, due to linkage on the same haplotype. Letting Λ_j be the allele space at loci j , a haplotype may be considered an element of the set: $\prod_{j \in [1, 2, \dots, M]} \Lambda_j$.

In phasing polymorphic alleles for an individual's genotype data (a mix of two haplotypes), we assume that half of the data is derived from each of the underlying haplotypes H_1 and H_2 . In this context haplotypes have a complementary structure in that the individual's genotype must be heterozygote at each polymorphic loci.

3.1 Haplotype Space and Joint Distributions

The full space of haplotypes is the product over all allele spaces $\{1, 2, \dots, M\}$; in the problem under discussion the haplotype space is in one-to-one correspondence with $\mathcal{M} = \{-1, 1\}^M$. The discrete-measure space $\langle \mathcal{M}, 2^{\mathcal{M}} \rangle$ will be used to denote the haplotypes, while $\mathcal{M}[j_1, j_2, \dots, j_v]$ denotes the haplotypes over the range of loci j_1, j_2, \dots, j_v . The result of phasing genotype data is a probability measure on the space $\langle \mathcal{M}, 2^{\mathcal{M}} \rangle$. Noiseless data may result in a measure assigning $\frac{1}{2}$ to each of the complementary haplotypes, and 0 to all others. This uniform measure over complements corresponds to perfect knowledge of what the haplotypes are. Our algorithm is consistent in that the correct result is achieved for suitably large data sets.

Let Λ_j be the allele set for the polymorphic loci j . Consider two bi-allelic loci j and j' . For clarity, we will assume that $\Lambda_j = \{A, a\}$ while $\Lambda_{j'} = \{B, b\}$. A data observation d_i is derived from one of the four classes: AB, Ab, aB, ab . Because the mapping noise at loci j and j' are independent, we can assess the probability based on the loci posteriori that the observation is derived from the following four classes:

$$\begin{aligned} P(d_i \sim AB) &= \alpha_{jA}(d_{ij})\alpha_{j'B}(d_{ij'}) \\ P(d_i \sim Ab) &= \alpha_{jA}(d_{ij})\alpha_{j'b}(d_{ij'}) = \alpha_{jA}(d_{ij})(1 - \alpha_{j'B}(d_{ij'})) \end{aligned}$$

$$\begin{aligned} P(d_i \sim aB) &= \alpha_{ja}(d_{ij})\alpha_{j'B}(d_{ij'}) = (1 - \alpha_{jA}(d_{ij}))\alpha_{j'B}(d_{ij'}) \\ P(d_i \sim ab) &= \alpha_{ja}(d_{ij})\alpha_{j'b}(d_{ij'}) = (1 - \alpha_{jA}(d_{ij}))(1 - \alpha_{j'B}(d_{ij'})) \end{aligned}$$

We define $\alpha_{jj'}^{(i)}$ as the *estimated probability distribution for observation i on haplotypes over the loci j, j'*:

$$\alpha_{jj'}^{(i)} = [\alpha_{jj'AB}(d_i), \alpha_{jj'A b}(d_i), \alpha_{jj'a B}(d_i), \alpha_{jj'ab}(d_i)]$$

We define $\alpha_{jj'}$ as the *estimated probability distribution over the data set on haplotypes over the loci j, j'*:

$$\alpha_{jj'}(D) = \frac{1}{N} \sum_{i=1}^N \alpha_{jj'}^{(i)}$$

For $\rho \in \mathcal{M}[j_1, j_2, \dots, j_M]$ and $\alpha_{j_w \rho_w}(d_i) = \text{Prob}(d_i \sim \rho_w)$ with $\rho_w \in \Lambda_{j_w}$, we can extend the estimates to any set of indices producing:

$$\begin{aligned} \alpha_{j_1 j_2 \dots j_v}^{(i)} &= \left[\prod_{w \in [1 \dots v]} \alpha_{j_w \rho_w}(d_i) \right]_{\rho \in \mathcal{M}[j_1, j_2, \dots, j_v]} \\ \alpha_{j_1 j_2 \dots j_v} &= \frac{1}{N} \sum_i \alpha_{j_1 j_2 \dots j_v}^{(i)} \end{aligned}$$

3.2 Complementarity

In phasing the diploid genotype data into two haplotypes $\rho_1, \rho_2 \in \mathcal{M}$ there is a special property: haplotype ρ_2 is complementary to haplotype ρ_1 , denoted $\bar{\rho}_2 = \rho_1$. The complementary pair of haplotypes may be represented by a change of variables, $w \in \{-1, 1\}^{M-1}$, and the transformation to the haplotypes is given by the map:

$$\rho_1(b) = \begin{cases} -1 & \text{if } b = 1 \\ -1 \prod_{j=1:(b-1)} w(j), b \in [2 \dots M] \end{cases} \quad \rho_2(b) = \begin{cases} 1 & \text{if } b = 1 \\ 1 \prod_{j=1:(b-1)} w(j), b \in [2 \dots M] \end{cases}$$

In evaluating the data, there are a possible 2^{M-1} complementary pairs of allele types to search.

The confidence of a set of complementary haplotypes is modeled as a probability distribution on the discrete measure space $\langle \mathcal{M}, 2^{\mathcal{M}} \rangle$, which is the convex hull of the following set of extremal points which correspond to certain knowledge of complementary haplotypes.

$$A = \left\{ \theta_\rho : \theta_\rho(\delta) = \begin{cases} \frac{1}{2} & \text{if } \delta = \rho \\ \frac{1}{2} & \text{if } \delta = \bar{\rho} \\ 0 & \text{o.w.} \end{cases} \quad \text{for } \delta \in \mathcal{M} \right\}$$

These values are the uniform distribution over complementary haplotypes and geometrically are vertices of a high dimensional hyper-cube. Let $A[j_1, j_2, \dots, j_v]$ be the corresponding distribution over the haplotype space $\mathcal{M}[j_1, j_2, \dots, j_v]$.

3.3 Maximum Likelihood Problem

We assume that for every loci j , the data $\{d_{ij} : i \in [1 \dots N]\}$ contains an equal distribution of data from the underlying haplotypes H_1, H_2 that can be inferred. Using the estimated values α for the joint distribution over loci product spaces, we will compute the haplotypes most likely producing α . We formulate the corresponding maximum likelihood problem as follows: Let the likelihood function be given by:

$$L(\Theta) = P(D|\Theta) = \frac{\Gamma(N)}{\prod_{\rho \in \mathcal{M}} \Gamma(N\alpha_\rho)} \prod_{\rho \in \mathcal{M}} \Theta_\rho^{\alpha_\rho N}$$

MLE 1 Find $\rho \in A$ so that $L(\rho) \geq L(\omega) \forall \omega \in A$.

Similarly, for any specified set of loci $\{j_1, j_2, \dots, j_v\}$ we may define a likelihood function $L_{\mathcal{M}[j_1, j_2, \dots, j_v]}$ as the most likely to produce posterior $\alpha_{j_1, j_2, \dots, j_v}$ over the space $\mathcal{M}[j_1, j_2, \dots, j_v]$.

Lemma 1 If $d(\alpha, A) < \epsilon$ for some ϵ small enough, and $d(\alpha, A) = \min_{\theta \in A} \|\alpha - \theta\|_2$. Maximizing $\prod_{\rho \in \{0,1\}^{M-1}} \Theta_\rho^{\alpha_\rho N}$ over $\Theta \in A$ is equivalent to minimizing $\sum_{j=1}^M \frac{(\alpha_j - \Theta_j)^2}{\alpha_j}$ over $\Theta \in A$.

The proof in the full paper is derived from a Taylor-series expansion of the likelihood function. It demonstrates that the MLE result in set A is the vertex of a 2^{M-1} hyper-cube closest to our estimated joint probability function α , measured by a modified L_2 norm.

With this result we assume the following function to be used in the algorithms presented later:

Algorithm 1

MLE-COLLAPSE(j_1, j_2, \dots, j_v)

COMPUTE $\rho \in A[j_1, j_2, \dots, j_v]$ MINIMIZING $\sum_{j \in [j_1, j_2, \dots, j_v]} \frac{(\alpha_j - \Theta_j)^2}{\alpha_j}$ OVER $\Theta \in A$
RETURN ρ

4 Algorithms

The algorithms focus on growing disjoint-phased contiguous sets of loci called *Contigs*. All loci are assigned an arbitrary phase and begin as a singleton phased contig. A JOIN operation checks if these phased contigs may be phased relative to one another using a function called VERIFY-PHASE. VERIFY-PHASE can be designed to check a phasing criteria, for example refuting a hypothesis of Hardy-Weinberg Equilibria is discussed in the full paper.

If a pair of phased contigs can be joined by passing the VERIFY-PHASE function then the disjoint sets are combined into a single phased contig and the

joint distribution over the set is computed with the MLE–COLLAPSE function. Having completed a successful join operation, we may regard the distribution function as the most likely haplotypes generating the observed data over the specified loci. Because the growth of contigs is monotonic and depends on local information available at the time of the operation, in the full paper we also consider an ADJUST operation that fractures a contig and re-join's using a larger locality of data than what was available during the JOIN.

We describe the operations in detail, analyze the results and indicate how to avoid incorrect operations.

Collapse : In the previous section of this paper, we discussed the collapse operation as the MLE–COLLAPSE function. It may be used to update a joint probability distribution over a set of contigs; it has the effect of keeping the contig structures bound to haplotype states which simplifies the computing of a phase.

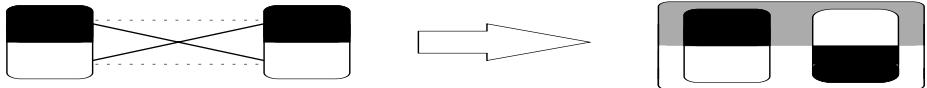


Fig. 1. Collapse

Join: Let K be a parameter denoting neighborhood size. Letting $C_1 = \{j_1, j_2, \dots, j_v\}$ and $C_2 = \{j'_1, j'_2, \dots, j'_w\}$, the join operation is as follows:

Given joint-probability functions $\alpha_{j_1}, \alpha_{j_2}, \dots, \alpha_{j_v}, \alpha_{j'_1}, \alpha_{j'_2}, \dots, \alpha_{j'_w}$, compute the joint probability function α_{C_1, C_2} with formula

$$\begin{aligned} \alpha_{C_1, C_2} &= \alpha_{(j_v)}(j'_1 j'_2 \dots j'_w) \\ &= w_{j_v, j'_1} \alpha_{(j_v)}(\bar{j}'_1 j'_2 \dots j'_w) + w_{j_v, j'_2} \alpha_{(j_v)}(j'_1 \bar{j}'_2 \dots j'_w) \\ &\quad + \dots + w_{j_v, j'_K} \alpha_{(j_v)}(j'_1 j'_2 \dots \bar{j}'_K \dots j'_w) \end{aligned}$$

with

$$\begin{aligned} \alpha_{(j_v)}(j'_1 j'_2 \dots \bar{j}'_x \dots j'_w) &= \sum_{i=1:N} \alpha_{j_v j'_x}^{(i)} = \sum_{i=1:N} \alpha_{j_v}^{(i)}(d_{ij_v}) \alpha_{j'_x}^{(i)}(d_{ij'_x}) \\ w_{j_v, j'_x} &= \kappa \frac{1}{d(j_v, j'_x)} \end{aligned}$$

Here, $\kappa = \frac{1}{d(j_v, j'_1) + d(j_v, j'_2) + \dots + d(j_v, j'_K)}$ and $d(j_v, j'_x)$ is proportional to genomic distance between loci j_v and j'_x .

Algorithm 2

COMPUTE-PHASE($C_1 = \{j_1, j_2, \dots, j_v\}$, $C_2 = \{j'_1, j'_2, \dots, j'_w\}$, K)

ASSUME j_v IN C_1 IS SUCH THAT $d(j_v, C_2) \leq d(j, C_2) \forall j \in C_1$:
 COMPUTE $\alpha_{(j_v)(j'_1 j'_2 \dots j'_w)}$ USING PARAMETER K .
 RETURN $\alpha_{(j_v)(j'_1 j'_2 \dots j'_w)}$

Algorithm 3

```
JOIN(  $C_1 = \{j_1, j_2, \dots, j_v\}$  ,  $C_2 = \{j'_1, j'_2, \dots, j'_w\}$  ,  $K$  )  

  COMPUTE-PHASE(  $C_1 = \{j_1, j_2, \dots, j_v\}$  ,  $C_2 = \{j'_1, j'_2, \dots, j'_w\}$  ,  $K$  )  

  IF( VERIFY-PHASE(  $\alpha_{j_1 j_2 \dots j_v}$  ) )  

     $\alpha_{j_1 j_2 \dots j_v} \leftarrow \text{MLE-COLLAPSE}( j_1, j_2, \dots, j_v )$  ;
```

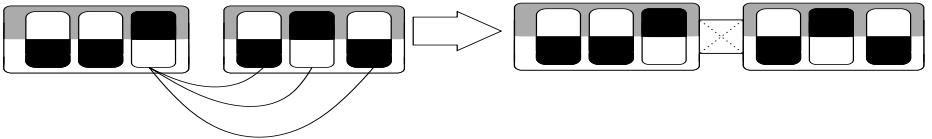


Fig. 2. Compute-Phase

Our algorithm estimates the haplotypes by solving an ordered set of local MLE problems. The rationale of the chosen function is discussed in the full paper.

4.1 Implementation

Input

The input is a set of data points $\{d_{ij} \in \mathbb{R}^r : i \in [1 \dots N], j \in [1 \dots M]\}$. We make the following assumptions about the input:

- For each j the points $d_{1j}, d_{2j}, \dots, d_{Nj}$ are derived from the Gaussian mixture model corresponding to mapping data at polymorphic loci j .
- For each i points $d_{i1}, d_{i2}, \dots, d_{iM}$ are independent random variables with parameters associated to underlying haplotypes.

Knowing the mapping order of polymorphic loci, we assume the positions of the genome to be $[x_1, x_2, \dots, x_M]$.

Pre-process

The EM-algorithm is run for each loci: $\{d_{ij} : i \in [1 \dots N] \text{ observable}\} \rightarrow \{\hat{\Phi}_j : \alpha_j\} \quad \forall j \in [1, \dots, M]$.

The result is a set of estimates for bi-allelic loci, $\{\hat{\Phi}_1, \hat{\Phi}_2, \dots, \hat{\Phi}_M\}$, as well as a set of functions estimating the probability that any data point derives from the distinct alleles $\{\alpha_1, \alpha_2, \dots, \alpha_M\}$.

Next we construct a join schedule. Letting $\beta_j = x_{j+1} - x_j$, we sort the results into an index array giving an increasing sequence: $\{j_1, j_2, \dots, j_v, \dots, j_{M-1}\}$.

Main Algorithm and Data Structure

Contigs are maintained in a modified union-find data structure designed to encode a collection of disjointed, unordered sets of loci which may be merged at any time. Union-find supports two operations, UNION and FIND [14]: union merges two disjoint sets into one larger set, FIND identifies the set containing a particular element. Loci j is represented by the estimated distribution α_j , and may reference its left and right neighbor. At any instant, a phased contig is represented by:

- A MLE distribution or haplotype assignment for the range of loci in the contig (if one can be evaluated).
- Boundary loci: Each contig has a reference to left- and right-most loci.

In the v th step of the algorithm, consider the set of loci determined by β_v , $\{j_v, j_{v+1}\}$: If FIND(j_v) and FIND(j_{v+1}) are in distinct contigs C_p and C_q , then 1) attempt to UNION C_p and C_q , by use of the JOIN operation and 2) update the MLE distribution and boundary loci at the top level if the JOIN is successful.

Output

Output is a disjointed collection of sets, each of which is a phased contig. It represents the most likely haplotypes over that particular region.

4.2 Time Complexity

The preprocess may involve using the EM-algorithm once for each loci. The convergence rate of the EM-algorithm is a topic of research ([8]) and depends on the amount of overlap in the mixture of distributions. For moderate-sized data sets we have noticed no difficulties with convergence of the EM-algorithm.

First we estimate the time complexity of the main algorithm implementing the K -neighbor version. For each β_j , there are two find operations. The number of union operations cannot exceed the cardinality of the set $\{\beta_j : j \in [j_1, j_2, \dots, j_{M-1}]\}$, as contigs grow monotonically. The time cost of a single find operation is at most $\gamma(M)$, where γ is the inverse of Ackermann's function. Hence the time cost of all union-find operations is at most $O(M\gamma(M))$. The join operation, on the other hand, requires running the K -neighbor optimization routine, at a cost of $O(K)$. Thus the main algorithm has a worst-case time complexity of

$$O\left(M(\gamma(M) + K)\right) = O\left(M\gamma(M)\right)$$

and may be regarded as almost linear in the number of markers, M for all practical purposes since K is almost invariably a small constant.

4.3 Examples

The full paper contains two examples illustrating the implementation for two simulated RFLP data sets, subject to extensive random errors.

5 Conclusions and Future Work

The simulation results are found to be encouraging, as they demonstrate that locally the phasing may be highly accurate. When local coverage derived from one haplotype is low, then the detection of polymorphisms become difficult. In the first data set a false negative detection is found on the eight marker from the left, this is due to zero coverage from one of the haplotypes at that point. The ninth marker is a false negative detection and is attributed to zero coverage from one haplotype and low coverage (2 molecules) from the alternative haplotype. Note that the false positive does not cause errors in the phase information for correctly detected polymorphic loci in the phased-contig achieved over marker index in the set {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}. Designing a mapping experiment targeting a polymorphic marker in the set {6, 7, 8, 9, 10} could allow one to phase the two contigs into a single contig.

In the full paper we shall explore how the order of *local* maximum likelihood problem solutions relate to the *global* maximum likelihood problem. We further discuss mapping technologies and single molecule methods that may be used to generate data suited for the diploid haplotyping problem. In particular SNPs, micro-arrays, and DNA-PCR-Colonies or polonies ([9]) are under investigation by the authors. Further analysis and examples will be presented.

References

1. T.S. ANANTHARAMAN, B. MISHRA AND D.C. SCHWARTZ. "Genomics via Optical Mapping II: Ordered Restriction Maps," *Journal of Computational Biology*, **4(2)**:91–118, 1997.
2. V. BAFNA,D. GUSFIELD,G. LANCIA,S. YOOSEPH. "Haplotyping as Perfect Phylogeny, A Direct Approach," *Technical Report UC Davis CSE-2002-21*
3. W. CASEY, B. MISHRA, AND M. WIGLER. "Placing Probes on the Genome with Pairwise Distance Data," *Algorithms in Bioinformatics: first international workshop: proceedings WABI 2001*, 52–68, Springer, New York, 2001.
4. A. CLARK. "Inference of Haplotypes from PCR-Amplified Samples of Diploid Populations," *Mol. Biol. Evol.*, **7**:111–122,1990.
5. A. DEMPSTER, N. N. LAIRD, AND D.RUBIN. "Maximum likelihood from incomplete data via the EM algorithm," *J.R. Stat. Soc.*, **39**:1–38, 1977.
6. L. EXCOFFIER, M. SLATKIN. "Maximum-Likelihood Estimation of Molecular Haplotype Frequencies in a Diploid Population," *Mol. Biol. Evol.*, **12**:921–927, 1995.
7. D. GUSFIELD. "Inference of Haplotypes from Samples of Diploid Populations: Complexity and Algorithms," *Journal of Computational Biology*, **8-3**:305–323, 2001.
8. J. MA, L. XU, AND M. JORDAN. "Asymptotic Convergence Rate of the EM-Algorithm for Gaussian Mixtures," *Neural Computation*, **12-12**:2881–2907, 2000.
9. R. MITRA, AND G. CHURCH. "In situ localized amplification and contact replication of many individual DNA molecules," *Nucleic Acids Research*, **27-24**:e34-e34, 1999.
10. T. NIU, Z. QIN, X. XU, AND J. LIU. "Bayesian Haplotype Inference for Multiple Linked Single-Nucleotide Polymorphisms," *Am. J. Hum. Genet.* **70**:156–169, 2002.

11. L. PARIDA, AND B. MISHRA. "Partitioning Single-Molecule Maps into Multiple Populations: Algorithms And Probabilistic Analysis," *Discrete Applied Mathematics*, (The Computational Molecular Biology Series), **104**(1-3):203–227, August, 2000.
12. S. ROWEIS, AND Z. GHAHRAMANI. "A Unifying Review of Linear Gaussian Models," *Neural Computation*, **11**(2):305–345, 1999.
13. M. STEPHENS, N. SMITH, AND P. DONNELLY. "A new statistical method for haplotype reconstruction from population data," *Am. J. Hum. Genet.* **68**:978–989, 2001.
14. R. E. TARJAN. *Data Structures and Network Algorithms*, CBMS 44, SIAM, Philadelphia, 1983.
15. B. WEIR. *Genetic Data Analysis II*, Sinauer Associates, Sunderland, Massachusetts, 1996.

Appendix

A RFLP Examples

We demonstrate our algorithm on two simulated data sets composed of ordered restriction fragment lengths subject to sizing error. Figure 3 below is presented in bands: Parameters of the simulations are summarized in the table:

Parameter	Symbol	Data Set 1	Data Set 2
Number of molecules	M	80	150
Number of fragments RFLP and non RFLP	F	20	100
Size of the genome	G	12000	50000
Expected molecule size	EMS	2000	2000
Variance in molecule size	VMS	50	500
Variance in fragment length size	VFS	1	20
P-value that any given fragment is an RFLP	P-BIMODE	.5	.3
Expected separation of means for RFLP	ERFLPSEP	10	50
Variance in the separation of means for RFLP	VRFLPSEP	.01	6

Any parameter with both an expectation and variance is generated with a normal distribution.

We used a simple VERIFY-PHASE function which merely checked that our posteriori distribution α_{C_a, C_b} is separated by a distance of $C > 0$ from the point $[\frac{1}{2}, \frac{1}{2}]$. In practice we discovered that the parameter C should depend on the local coverage.

For the first simulation on data set I seen in figure 3, a relatively small set is chosen so that the limitations of the algorithm can be seen. Here the neighborhood size is set to $k = 5$. There is no guarding against false positive RFLP detections, still phasings are computed and one can see that mistakes are due to the low coverage library.

In the second simulation on data set II seen in figure 4 we illustrate that good phasing results may be achieved on large, sparse data sets.

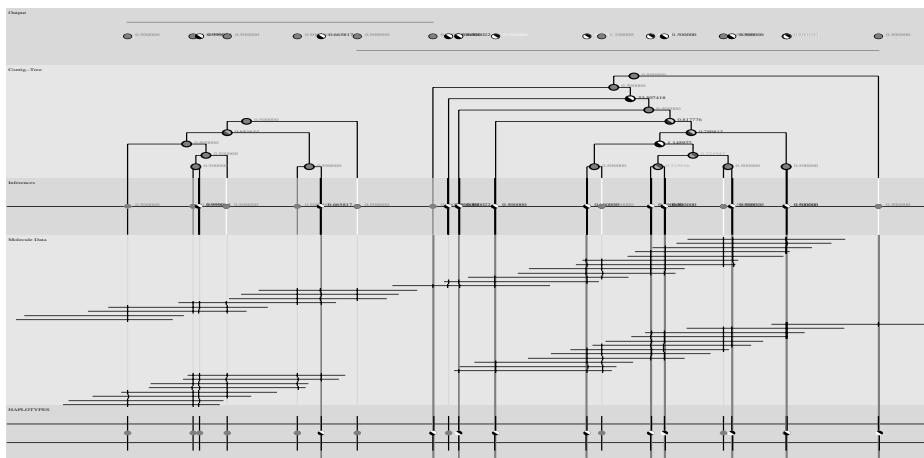


Fig. 3. Data set I, The bottom most band is the actual haplotypes, with polymorphic sites depicted by polarized color. Above that in the band labeled Molecule data is a set of two haplotype marker maps which are mixed to form the genotype data. In the inference band we show the results of polymorphic detection. The tree indicates the order of operations and in the top band you can view the resulting haplotyped regions.

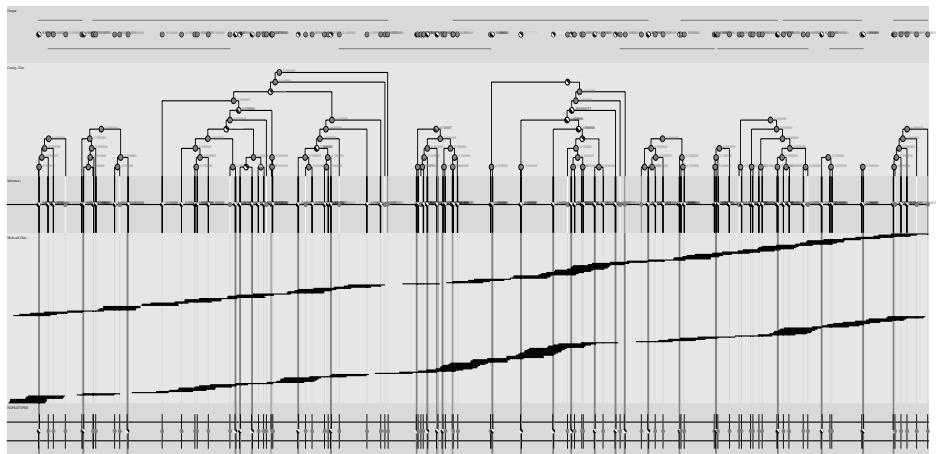


Fig. 4. Data set II, The bottom most band is the actual haplotypes, with polymorphic sites depicted by polarized color. Above that in the band labeled Molecule data is a set of two haplotype marker maps which are mixed to form the genotype data. In the inference band we show the results of polymorphic detection. The tree indicates the order of operations and in the top band you can view the resulting haplotyped regions. These results display how the algorithm scales in parameters of density of polymorphic markers and haplotype molecular coverage.

Energy Aware Algorithm Design via Probabilistic Computing: From Algorithms and Models to Moore's Law and Novel (Semiconductor) Devices

Krishna V. Palem

Center for Research in Embedded Systems and Technology
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA
palem@ece.gatech.edu

Abstract. The energy consumed by computations is becoming an increasing concern both within the context of high-performance systems well as embedded systems, on par with the past focus on raw speed or its derivative performance. In this talk, we will outline a novel framework for designing and analyzing algorithms wherein the figure of merit is the energy complexity - a measure of the (physical) energy consumed. Using the formulation of an energy-aware switch, and a network of such switches, fundamental limits will be established for the energy needed for switching deterministically, as well as energy savings derived from probabilistic switching, with a probability of being correct, p . Specifically, it is shown that a single deterministic switching step for computing a BIT consumes at least $(-(\kappa \cdot T \ln(2)))$ Joules of energy, whereas the energy consumed by a single probabilistic switching step to compute a PBIT can be as low as $(-(\kappa \cdot T \ln(2p)))$ Joules. These results are developed within the context of an idealized switching device introduced here, constrained by the laws of classical (statistical) thermodynamics (of Maxwell, Boltzmann and Gibbs), as well as by the constraints of idealized semiconductor devices. Based on this notion of switching, models for algorithm analysis and design, as well as upper- and lower- bounds on energy complexity and hence, for the first time, asymptotic energy savings via the use of probabilistic computing will be established. Possible approaches to realizing these probabilistic switches using conventional CMOS technology, as well as their potential for accelerating the current semiconductor roadmap that is based on deterministic computing, including the projections implied by Moore's law, will be outlined. This work draws upon basic concepts from computer science, microelectronics and classical thermodynamics, and in the interest of being self-contained, the presentation will include a brief survey of the relevant thermodynamics.

Designing SANs to Support Low-Fanout Multicasts

Rajendra V. Boppana^{*,1}, Rajesh Boppana¹, and Suresh Chalasani²

¹ CS Department, The Univ. of Texas at San Antonio, San Antonio, TX 78249
`{boppana, brajesh}@cs.utsa.edu`

² School of Business and Technology, Univ. of Wisconsin-Parkside, Kenosha, WI 53141
`suresh.chalasani@uwp.edu`

Abstract. System area networks (SANs) need to support low-fanout multicasts efficiently in addition to broadcasts and unicasts. A critical component in SANs is the switch, which is commonly designed around crossbars. We present new switch designs using a combination of low-cost multistage switching fabrics and input and output buffering with hardware based packet scheduling mechanism. Using detailed simulations, we show that the proposed designs can scale to 512-ports and outperform crossbar based designs in a 4-switch SAN.

1 Introduction

System area networks (SANs), designed using high-speed switches with direct links, are used to interconnect and provide high-throughput, low-latency communication among workstations. Similar switched networks are also used for high-speed local area networks (LANs) and storage area networks (SANs) to accommodate a variety of needs ranging from parallel computing, storage area networking, to video conferencing. These applications often use low-fanout multicast messages in addition to unicasts and broadcasts. It is important that future switches be designed to handle these diverse communication needs.

A multicast on a switched network is facilitated by replicating a multicast packet at appropriate switches as the packet progresses from its source to destinations. A multicast packet arriving at the input port of a switch may require more than one output port of that switch. Since multicast packets from different inputs may have one or more common outputs, the conflicts for the output ports of a switch increase. Often, a packet could be sent to some but not all output ports it needs to reach, owing to the output conflicts. A number of different architectures have been proposed to handle this problem.

In this paper, we focus on designing switches that can handle multicast traffic efficiently without compromising the unicast traffic. The key component of a switch is the switching fabric used to provide the datapath for packets to move within a switch from input ports to output ports. The most commonly used

* Boppana's research has been partially supported by NSF grant EIA-0117255.

choices are shared-bus and crossbar. The shared-bus designs are not scalable owing to performance constraints, and the crossbar-based designs are not scalable owing to cost considerations. Our design uses a multistage network, specifically an Omega multistage interconnection network, as the switching fabric with input and output buffers and a hardware based packet scheduling. This design handles both unicast and multicast traffic. We also look into the performance of SANs based on proposed switches. For a simple network of four switches, we show that the proposed switches give as much as 50% more throughput than crossbar-based switches.

2 Background and Related Work

We assume that fixed-size packets (called cells) are used for communication. If a message is too large to fit in a single packet, then multiple cells are sent from source to intended set of destinations. Such a message appears as bursty traffic or correlated traffic to the network. The amount of time a cell takes to move from input ports of a switch to its output ports is called a time-slot. For balanced design, this should be about the same as the transmission time for a cell. For example, the slot-time for a 100-byte cell with a line rate of 10 GHz is 80 ns. We are primarily interested in cut-through or store-and-forward switching with best-effort delivery. So excess packets may be dropped by switches when traffic load is high. For applications that require reliable delivery of data, a transport layer such as TCP will need to be used. It is noteworthy that the majority of parallel computing applications are currently designed to work on top of a standard TCP/IP protocol stack.

We assume switches are of size $N \times N$. Many switched networks and standards for the same were proposed in the literature [8,9,10,11,12]. These networks use crossbar based switches designed primarily to handle unicast traffic. Several designs to handle multicasts were proposed in the computer networks community [1,2,3,4,5,6]. Of particular interest is the Weight-based algorithm (WBA) for crossbar-based switches with input buffering, which is shown to achieve high throughput for multicasts [1]. In this design, each multicast cell is assigned a weight using the following formula $a \times \text{age} - f \times \text{fanout}$, where age is the number of time-slots for which the cell is queued in the current switch input buffers and fanout is the number of output ports of the switch it needs; a and f are tunable parameters and are often chosen as 1 and 2, respectively. However, the cost of crossbars in terms of crosspoints increases at $O(N^2)$. For work conservation, a technique known as cell-splitting is often used. In cell-splitting, when a multicast cell competes for outputs with other cells and is granted to reach only a partial set of its outputs, a copy of the cell is retained in the input buffer with the remaining outputs as its destination list.

3 Ω Switch

Our design attempts to simplify the design complexity of switching fabric and output ports at the cost of more complex cell scheduling. We use a combination

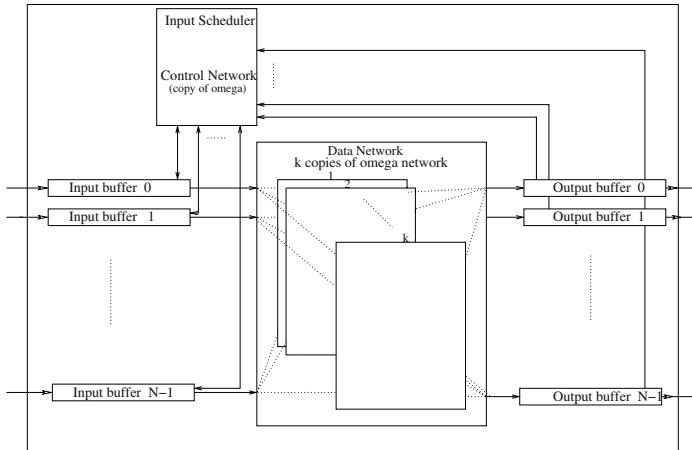


Fig. 1. Switch architecture. The switch constitutes of input and output queues, control network and data network. The control network is based on the Omega MIN and acts as input scheduler. The data network consists of k copies Omega MIN for routing the cells through the switch. Normally $k \leq 3$.

of input and output buffers and multiple copies of the Omega multistage interconnection network (MIN) [7] as the data network (switching fabric). We denote the switch as the Ω switch. Figure 1 shows the block diagram of Ω switch.

The cells arriving from the input lines are buffered at the input port buffers. Once a cell is in the input buffer, it is guaranteed to reach all of its destinations (switch outputs). Hence there is no cell loss within the switch fabric or output buffers. Cells are lost only when they arrive at the switch and are rejected by the input ports. This happens when the switch is saturated. Based upon the header of the cell, the destinations of the cell are determined by table lookup and a routing tag is assigned for routing through the switch fabric. For the purpose of the discussion, we assume the routing tag to be an N bit vector with each bit denoting a particular output port. A cell is destined to an output port if its corresponding bit in the vector is set. Routing tags could be precomputed or prepared as a cell is placed in an input buffer.

We select cells in such a way that there are no path conflicts within the switch fabric. Because cells are carefully selected to eliminate path conflicts, there is no need for buffering at any of the switching elements inside the switch fabric. We believe that, by moving all the necessary buffering away from the data path of the switch, we can simplify the switch fabric design.

The input scheduler determines the inputs and the cells they need to send through the switch fabric. Contention for paths within the switch fabric and outputs affect the performance of the switch. When crossbars are used as switch fabrics, there is no contention for data paths through the switch fabrics. It is a severe problem, however, when blocking switch fabrics like Omega MIN are used. On the other hand using crossbars could be expensive. To alleviate this, we use

multiple copies say, k , of the Omega MIN in the switch fabric. So conflicts for a path in the switch fabric can be handled by sending contending cells through different copies of the network. This increases the bandwidth inside the switch and achieves high throughput.

Since we use k copies of the Omega MIN as the switch fabric, each output could receive up to k cells during a single slot time. Hence the output buffers need to operate at k times the line speed. We can restrict the maximum number of cells sent to an output port in a single slot time to some l , where $l \leq k$. When k is small, it is simpler to have $l = k$. In each slot time, one cell is sent out of the output port. Excess cells are buffered in the output queues. This would achieve high switch utilization since any output that did not receive a cell due to blocking nature of the switch fabric could send out a cell as long as its buffer is not empty. This is particularly useful for correlated traffic in which the demand for an output tends to be bursty. Although, we use multiple copies of Omega MIN we limit the complexity or speedup requirements of output buffers by controlling the output port contention through cell scheduling so that the switch is scalable.

3.1 Scheduling Cells

It is known that multistage interconnection network based switches need elaborate cell selection techniques to achieve high throughput. Software based cell selection schemes do not work well for high line rates or large switches. Therefore, we use the idea of hardware specific selection technique for unicast cells by Boppana and Raghavendra [15]. In this method, a copy of the underlying network for the switch fabric (that is, the Omega multistage network itself) as the control network to aid the selection of cells. However this control copy network needs to route the routing tags (N -bit tags for an $N \times N$ switch) of the cells for scheduling purposes. The conflicts for data paths through a switch fabric will appear as contention for a switching element's output links in the Omega MIN. These conflicts are resolved randomly or by using a suitable priority mechanism. Cells whose routing tags go through all the stages of the control network successfully will be actually routed through the switch fabric in the following time slot. The self-routing techniques can be used to set up the switching elements in the switch fabric since the cells going through the data network will not have conflicts. To improve the number of cells that can be sent in a time slot, this selection process may be repeated several times for each copy of the data network.

Round-Robin Scheduling. The simplest way to schedule multicast cells is to allow one or more input ports to send their multicast cells in a round-robin (RR) fashion during each slot time. Since our proposed switch has k data networks, it is intuitive to select k input ports to route their multicast cells through the k data networks (one through each copy).

Hence, the simplest scheduling policy is a prioritized round robin algorithm in which, during each slot, k input ports have the priority and can send their multicast cell unrestricted through a distinct copy of the data network. If a

prioritized input port doesn't have a multicast cell the next input port (in a circular fashion) gets a chance. For the next slot, next k inputs will be the priority inputs. This policy basically routes k multicast cells through the switch during each slot time. So switch utilization $\rho = \frac{k \times f}{N}$, where k is number of copies of data network, f is the fanout of the multicast traffic and N is the size of the switch. This scheme does not work well if f is small and N is large.

Scheduling Additional Cells. To improve the performance, we attempt to schedule more than k multicast cells during each slot time; some Omega MINs send two or more multicast cells without path conflicts. This can be achieved by performing a round of selection for additional multicast cells that could go through each copy of the data network. So, after determining the k input ports that would send their multicast cells through the k data networks, using the round-robin scheduling, we let the remaining input ports send their output requests (routing tags) through the control network and determine if any other multicast cells could be scheduled through each of the data networks without conflicting with the internal route of the earlier selected multicast cell. Hence there is one round of selection for each of the k copies. It is noteworthy that this additional scheduling improves the switch utilization by using additional data paths. Each input request is either satisfied or rejected in entirety. That is, there is no cell splitting. If the fanout of the multicast cells is high, the probability of finding a cell which uses paths that do not conflict with those of an already selected multicast cell (using round-robin scheme) decreases and this scheduling does not achieve any improvement over the simple round-robin policy. So for multicast traffic with larger fanout, it becomes necessary to split the cell to achieve high switch utilization.

Scheduling Additional Cells with Fanout Splitting. In this policy, after determining the k input ports that would send their multicast cells through the k data networks, the remaining input ports send their routing tags through the control network to determine if any other multicast cells could be scheduled through each of the data networks without conflicting with the internal routes of earlier selected multicast cells. If a multicast cell is able to obtain paths to reach some but not all of its destinations, we split the destination list and let a copy of the multicast cell go to the available outputs while retaining a copy with the remaining destinations at the input. Once an input port obtains paths for one or more destinations for its multicast cell, it will not compete for paths in the other copies of the data network. A split cell with reduced destination list is treated as a normal multicast cell for scheduling in later rounds.

We use output buffers to store cells to accommodate multiple cells arriving (via multiple data networks) at an output in a slot time. So the output queues could overflow and result in high cell loss. To prevent this, a back pressure mechanism is used. If the output queue size exceeds a threshold it accepts only one cell during each slot time until the number of cells at the output is less than the threshold.

4 Performance Analysis of Switch Designs

We simulated a $N \times N$, $8 \leq N \leq 1024$, Ω -based and crossbar-based switches with separate buffers for unicast and multicast traffic. Owing to space considerations, we present only a subset of the results; for additional results see [16].

While each input queue of crossbar can buffer up to 256 cells, Ω can buffer 128 cells at the input and 128 cells at the output. So the total buffer space used was the same for both designs. The control network of the Ω switch used two rounds (one for each copy) for additional multicast cell selection. Weights of 1 and 2 were used the age and fanout for WBA simulations.

Each simulation was run for multiple batches of 100,000 cycles, after a warmup of 50,000 cycles. Simulation was stopped when the half width of 95% confidence interval fell within 5% the computed mean latencies and throughputs.

Traffic patterns. In this paper, we present results for correlated traffic, which is considered to be more representative of the traffic on computer networks [13, 14]. In correlated arrivals multicast cells generated in 16 consecutive slot times. Each injected cell has a constant fanout, specified as input to the simulation. The first cell of a stream of correlated cells chooses the destinations randomly; the remaining cells in the stream use the same destinations.

Performance metrics. We use average cell latency and output port utilization as the performance metrics. Cell latency is the time elapsed from the time a cell is injected to the time it is delivered to all of its destinations. Output port utilization is the average number of cells delivered by the switch (or SAN) per output per slot-time. The maximum possible utilization is 1. Load offered to the switch (or SAN) is expressed as a fraction of the maximum utilization.

We use the following notations for all the plots. Omega, Omega2 and Omega-split denote, respectively, the simple round-robin, round-robin with one round of scheduling and Omega2 with fanout splitting. WBA denotes the weight-based scheduling for crossbar-based switches.

4.1 Small Switches

Figure 2 presents the results for correlated traffic. WBA saturates at about 64% while Omega2 can attain up to 80% and Omega-split saturates at 86%. The delay curves for Omega2 and Omega-split are significantly lower than that of WBA when the loads are in the range 50% to 90%. We observed that with uncorrelated arrivals (not shown here), utilizations improve for Omega2, Omega-split and WBA [16]. The utilization of round-robin Omega switch is oblivious to traffic pattern. Similar performances were observed for 16×16 switches (see Figure 3).

4.2 Large Switches

For small switch sizes, the Ω switch with fanout splitting performs well compared to the crossbar-based WBA switch. To see if this holds for larger switch sizes, we simulated 512×512 Omega-split and WBA switches. For Ω -switch simulations,

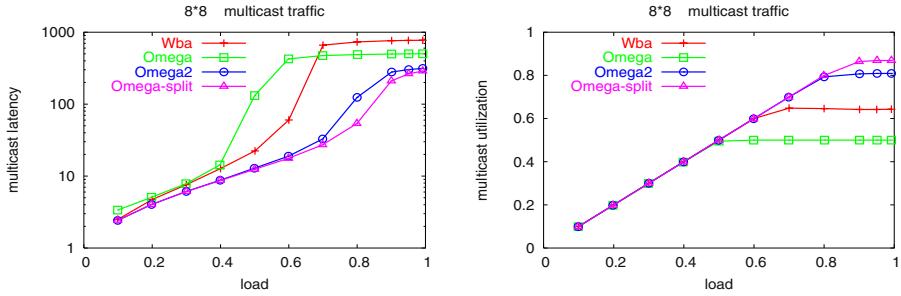


Fig. 2. Performance of 8×8 switches for correlated multicast traffic with fixed fanout of 2.

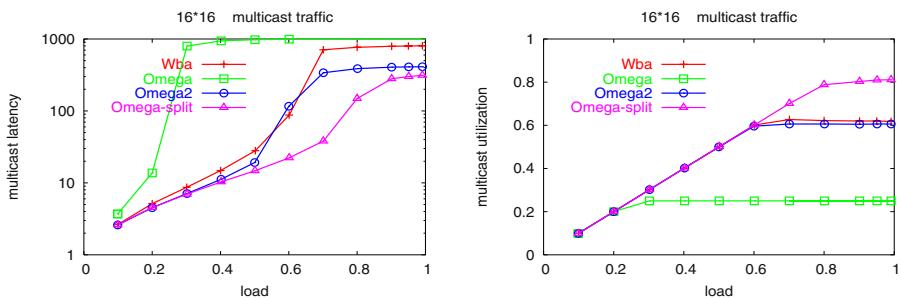


Fig. 3. Performance of 16×16 switches for correlated multicast traffic with fixed fanout of 2.

we varied k , the number of data networks, to determine the benefit of more copies of the data network. Since the cell selection in the Ω switch is more complicated than that in a WBA switch, we assume that cell selection is pipelined using additional hardware and that it takes k time slots to determine the cells that can go through a single slot. So in Ω switch designs, cell selection starts k time-slots prior to the time slot for which the scheduling is done. For WBA the cell delay is still 2 slots in the absence of contention or waiting.

Figure 4 gives the simulation results for 512×512 switches. We observe that, while 2 copies are not sufficient to obtain good performance, one or two additional copies of switch fabric provide much higher utilizations. The utilization is improved by 20 percentage points when 3 copies (split-3) are used instead of 2 (split-2). The difference in the performances of 3 and 4 copies decreases with increasing fanout. For fanout 8 and above, they are almost identical [16].

Though Ω switches with fanout splitting outperform WBA for large switches, they are less expensive in terms of number of crosspoints: Split-3 requires fewer than 11% ($\frac{3 \times \frac{512}{2} \times 9 \times 4}{512 \times 512}$) of the crosspoints used for a crossbar.

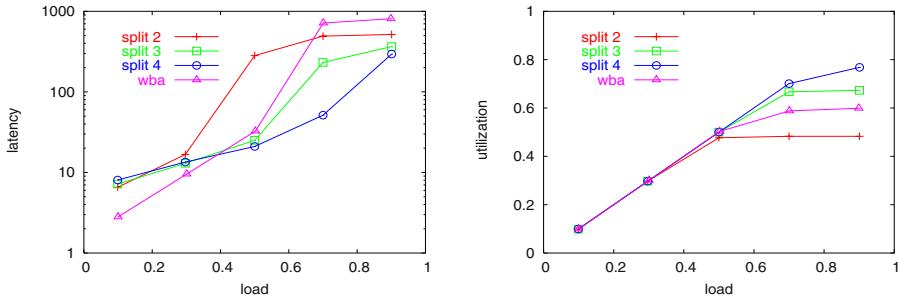


Fig. 4. Performance of 512×512 switches for correlated multicast traffic with fixed fanout of 2. Split- n indicates Omega-split with n copies of switch fabric.

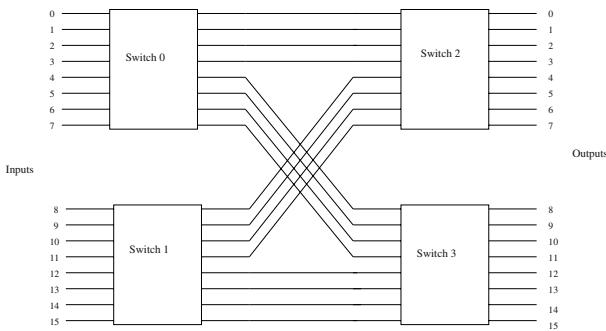


Fig. 5. Topology of SAN 1.

5 Performance Analysis of Switched Networks

Most published studies evaluate proposed switch designs as single components, but do not evaluate their switch designs in the context of a SAN. We used Omega and WBA switches as building blocks to form SANs. We have evaluated the performance of the SANs so designed. In particular, we analyzed the performance of the switches for two network topologies. The performance of one of the networks, shown in Figure 5, which provides the functionality of a 16×16 switch, is presented here. The performance of the second network is given in [16].

Figure 6 presents the network utilizations and latencies achieved by the switches for network in Figure 5 for multicast traffic of fanout 2. Though it uses more complex cell splitting, Omega-split performs only marginally better than Omega2. On the other hand, both Omega designs give 50% higher throughput than WBA. It is instructive to compare the performance of a single 16×16 switch given in Figure 3 with that of the network simulated. We see that Omega2 achieves a utilization of over 70% in the SAN configuration, but only 60% utilization as a single switch. On the other hand, WBA switch performs worse in a SAN

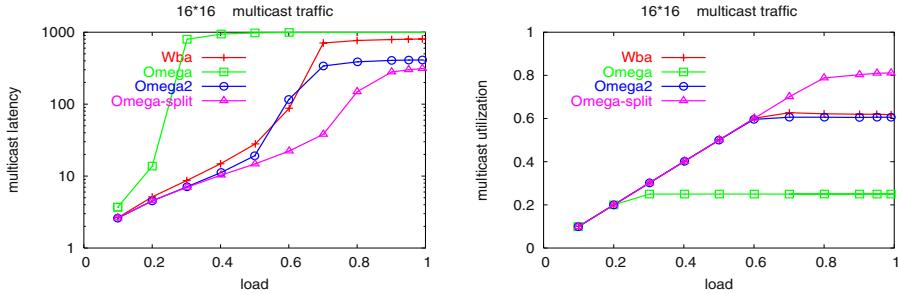


Fig. 6. Performance of SAN 1 for correlated multicast traffic with fixed fanout of 2

configuration (at 40% utilization) than as a single switch (at 60% utilization). The reasons for the performance divergence is as follows.

The network utilization depends on utilization of each of the switches in the network. By arranging the switches in stages we have reduced the effective fanout of the cells passing through each stage of the switches. Conversely, the rate of cell arrivals at the second stage of switches is increased. For a multicast load of 0.4 with fanout 2 on the network in Figure 5, the cell rate (rate at which cells are injected in to the inputs) at switches 0 and 1 is $\frac{0.4}{2} = 0.2$. Since the effective fanout at these switches is 1.53, assuming all outputs are uniformly used, the cell rate at switches 2 and 3 is $0.2 \times 1.53 = 0.31$. This is about the cell rate at which a single 8×8 WBA switch starts to lose cells for fanout 2 (see Figure 2). This shows the inherent limitation on the cell rate that the WBA can sustain on the crossbars irrespective of the fanout for correlated traffic. On the other hand, Omega2, which can handle a higher rate of cells in 8×8 size, clearly benefits from the reduced fanout seen by each stage of switches and provides higher utilization than it does as a single 16×16 switch.

This clearly demonstrates the inherent weakness of input buffered crossbar designs for correlated traffic. This also illustrates the advantage of using limited output buffering, especially for correlated multicast traffic. For this reason, the Ω switch designs take advantage of reduced effective fanout in the traffic that occurs as cells go through multiple switches.

6 Conclusions

The increasing demand for various multicast applications requires system area networks based on high-speed multicast switches. These switches should handle both unicast and multicast traffic efficiently. In this paper, we have presented the design of a multicast switch based on multistage interconnection network with input and output buffers. Being based on multistage network, the design is cost effective and scalable.

We have illustrated three possible designs based on the cell selection policy. They are Omega, which has a simple round-robin scheme, Omega2, which has an additional round of scheduling in addition to the round-robin scheduling,

and Omega-split, which is basically Omega2 with cell splitting. We have developed a modular simulator in java to evaluate the performance of the switches for the proposed scheduling policies. We have observed that the Omega-split outperforms WBA on crossbar for switch sizes ranging from 8×8 to 512×512 .

We have analyzed the performance of a simple 4-switch SAN built from the Ω and WBA switches. In particular, we simulated a 16×16 switch by interconnecting four 8×8 switches and observed that the throughput achieved is more than that of a single 16×16 switch for Omega based designs, while WBA performs worse in the SAN configuration. An interesting outcome of our study is that Omega2, which does not use cell-splitting, performs as well as Omega-split and better than WBA for small switches. On the other hand published results indicate that crossbar-based switches do not work well without fanout splitting [1]. Another contribution of our study is that WBA which uses crossbars as switch fabrics suffers as its not able to sustain the cell rate for low fanout multicast traffic that occurs as cells go through multiple switches in a network.

References

1. Balaji Prabhakar, Nick McKeown, Ritesh Ahuja, "Multicast Scheduling for Input-Queued Switches," IEEE Journal on Selected Areas in Communications, vol 15, No. 15, pp. 885–866, June 1997.
2. Y. Yeh, M. Hluchyj, and A. Acampora, "The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching," IEEE Journal on Selected Areas in Communications, Vol SAC-5, No. 8, October 1987, pp. 1274–1283.
3. Rein J. F. de Vries, "ATM Multicast connections using the Gauss switch," in Proc. GLOBECOM 1990, pp. 211–217.
4. D. X. Chen and J. W. Mark, "Multicasting in SCOQ Switch," INFOCOM '94, pp. 290–297, 1994.
5. H. J. Chao and B. S. Choe, "Design and Analysis of large-scale multicast output buffered ATM switch," IEEE/ACM Trans. Networking, vol. 3, pp. 126–138, April 1995.
6. K. L. E. Law and A. Leon-Garcia, "A large scalable ATM multicast switch," IEEE J. Selected Areas Commun., vol15, no. 5, pp. 844–854, 1997.
7. Duncan H. Lawrie, "Access and Alignment of Data in an Array Processor" IEEE Transactions on Computers, Dec. 1975, pp 1145–1155.
8. Michael D. Schroeder et. al., "Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point links," IEEE Journal of selected areas in communications. vol. 9. no.8, Oct. 1991.
9. N. J. Boden et. al., "Myrinet: A Gigabit-per-second local area network," IEEE Micro, pages 29–36, Feb. 1995.
10. D. Garcia and W. Watson, "Servernet II," Proceedings of the 1997 Parallel Computer, Routing and Communication Workshop, June 1997.
11. R. Sheifert, "Gigabit Ethernet," Addison-Wesley, April 1998.
12. Infiniband Trade Assoc., "Infiniband Architecture Specification, Release 1.0," Infiniband Trade Association, 2000.
13. W. E. Leland, M. S. Taqqu, W. Willinger and D. V. Willson, "On the self-similar nature of Ethernet traffic (extended version)," IEEE/ACM Transactions on Networking, vol. 2, pp 1–15, Feb. 1994.

14. V. Paxson and S. Floyd, "Wide area traffic: The failure of Poisson modeling," IEEE/ACM Transactions on Networking, vol. 3, pp. 226–244, June 1995.
15. Rajendra V. Boppana, C. S. Raghavendra, "Designing efficient Benes and Banyan based input buffered ATM switches," ICC 1999.
16. Rajesh Boppana, "Design of Multicast Switches for SANs." Masters thesis, University of Texas at San Antonio, May 2003.

POMA: Prioritized Overlay Multicast in Ad Hoc Environments*

Abhishek Patil¹, Yunhao Liu¹, Lionel M. Ni², Li Xiao¹, and A.-H. Esfahanian¹

¹Department of Computer Science and Engineering
Michigan State University

East Lansing, MI 48824
{patilabh, liyunha, lxião, esfahanian}@cse.msu.edu

²Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Kowloon
Hong Kong
ni@cs.ust.hk

Abstract. Overlay Multicast networks in mobile ad-hoc environments have received much attention due to their increasing number of practical applications. In many applications, some participating nodes might be members of more than one overlay trees or may wish to build a temporary tree in order to perform certain important tasks. The priority of these trees can be defined by the importance of the service. For the success of such an application, it is necessary that nodes belonging to more than one tree are smart enough to ignore incoming messages from members in low priority trees while they are listening to members from a higher priority tree. In this paper, we present a prioritized overlay multicast in ad-hoc environments (POMA). POMA builds priority trees with certain nodes carrying important tasks in overlay networks, and rearranges low priority trees whenever some nodes temporarily move to a high priority network. We study the feasibility of POMA by identifying a suitable unicast routing protocol, exploring to use location information to build efficient trees, and studying the impact of density of wireless nodes to the system performance.

1 Introduction

Multicast communication is increasingly being used in many applications. Although application layer (overlay) multicast networks are less efficient compared to IP based multicast networks, overlay networks are much easier to implement. The current implementation of IP networks provides very limited provision for forming multicast networks. The scene in mobile ad-hoc domain is further complicated due to its wireless nature and node mobility. Nodes in a Mobile Ad-hoc Network (MANET) have a limited coverage range and hence multiple network hops may be needed for one node

* This work was partially supported by Michigan State University IRGP Grant 41114 and by Hong Kong RGC Grant HKUST6161/03E.

to exchange information with another mobile node. However, these weaknesses have not reduced the attention received by overlay networks in MANET. Due to its ease of implementation and flexibility to adapt, overlay networks are finding many practical applications. In some situations, participating nodes may be able to carry out several different functions and, as a result, can be associated with more than one overlay tree. There may be times when some member nodes may decide to form a short-term network to perform certain important tasks. The different networks can have different levels of priority depending on the importance of the service they perform. In this paper, we first introduce the idea of building role-based priority trees in overlay networks and also present a way to rearrange low priority trees when certain nodes temporarily move to a high priority network.

We propose a prioritized overlay multicast in ad-hoc environments (POMA). In many practical environments, building a massive, fixed network infrastructure may not be a judicious idea. Instead, implementing a dynamic need-based system would be a better solution. POMA builds need-based priority trees with certain nodes carrying important tasks in overlay networks, and rearranges low priority trees whenever some nodes temporarily move to a high priority network. This paper explores the possibility of building efficient overlay trees by utilizing location information about member nodes. Our simulation results show the effectiveness of overlay trees that are built by using location information.

The paper is organized as follows. Section 2 discusses some of the related work in the area of overlay multicast and building location based trees. Section 3 presents the idea of role-based priority trees. Section 4 gives a description of the POMA system. Section 5 presents results and analysis of simulations. Finally, Section 6 describes the conclusion and future work for the paper.

2 Related Research

Overlay multicast in wired networks has been a popular area of research for the last few years. There have been several papers addressing the issue of forming an efficient overlay multicast tree. NICE [2] presents an application layer multicast protocol that arranges the end host into sequentially numbered layers, which defines the multicast, overlay data path. Two metrics are used to analyze the goodness of data paths in an overlay multicast – stress defined as the number of identical copies of a packet carried by that link and stretch, which measures the relative increase in delay incurred by the overlay path between pairs of members with respect to direct unicast path. However, the tradeoff for stress and stretch has been studied for wired networks only.

PAST-DM [1], Progressively Adaptive Sub-Tree in Dynamic Mesh, is an overlay multicast protocol defined for mobile ad-hoc networks that tries to eliminate redundant physical links so that the overall bandwidth consumption of the multicast session is reduced. The virtual mesh in PAST-DM constantly adapts to the changes in the underlying network topology. Each node implements a neighbor discovery protocol using the extended ring search algorithm. The nodes periodically exchange link state information with their neighbors in a non-flooding manner. Thus, by looking at the

link state of each node, a node gets a view of the entire topology. This information is used to build a source-based tree. PAST-DM yields a stable tree quality at the cost of higher overhead, which increases with the periodicity of the link state updates.

Location Guided Tree (LGT) [7] construction scheme builds overlay multicast tree using geometric distance between member nodes as the heuristic of link costs. Two tree construction algorithms are proposed: greedy k -ary tree construction (LGK) and Steiner tree construction (LGS). The algorithms are based on the assumption that longer geometric distances require more network-level hops to reach destination. With LGK, source node selects k nearest neighbors as its children, and partitions the remaining nodes according to their distance to the children nodes. LGS constructs the Steiner tree using link costs as their geometric lengths. LGT is a small group multicast scheme and hence, the scalability of the protocol is doubtful.

3 POMA Model

In overlay networks since the topology is built at the application layer, it provides greater flexibility in the design. Building role-based priority trees is one such advantage of overlay networks. Different priority trees can be built in the same environment to provide different kinds of services. With multiple priority trees, nodes belonging to different trees can switch among networks depending on what functionality they want to provide. At any given time, a node would associate itself only with one priority tree (the highest active priority tree in its set) and it would have to ignore incoming messages/data from members in low priority trees. A node that initiates the formation of a new priority tree can supply priority tokens. The value of the priority token can determine the priority of the tree. Thus if a node is currently a member of priority tree ' i ', it would not listen to message/data from member nodes belonging to ' $i-1$ ' or lower priority trees. Once the ' i ' priority tree is dissolved, a member node will down-shift to the next highest priority in its set. When a node decides to form a high priority tree or receives an invitation to join a high priority network, it may leave behind several 'orphan' nodes. These orphan nodes would now have to attach to another node in the original network to maintain their connectivity with the original tree. One way to tackle this problem would be for the departing node to send a control message to its children informing them of its intention of leaving the network. Along with this message, the exiting node will give them its parent address. The child node can thus contact its grandparent node, connect to it and start getting data from it as illustrated in Fig. 2a. The member nodes use multi-hop means of communicating with each other. Thus referring to Fig. 2a again, nodes F and I may not be close to each other and yet would be neighbors to each other in the logical topology. Later sections of the paper describe the idea of location-aware overlay trees, where the logical topology would try to match the physical topology; thus making F and I close to each other in the physical topology too. If for some reason the grandparent is unable to support the new node(s), it will pass on the connection request to the source node (node A in case of Fig. 2a). In location-aware trees, the source node has location information of the entire topology and should be able to redirect the orphan node's connection to a suitable node.

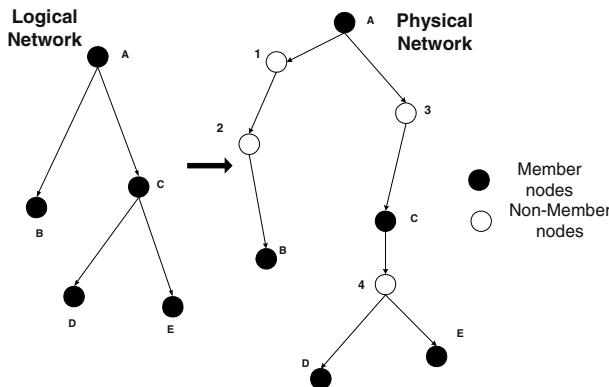


Fig. 1. Logical and physical topologies

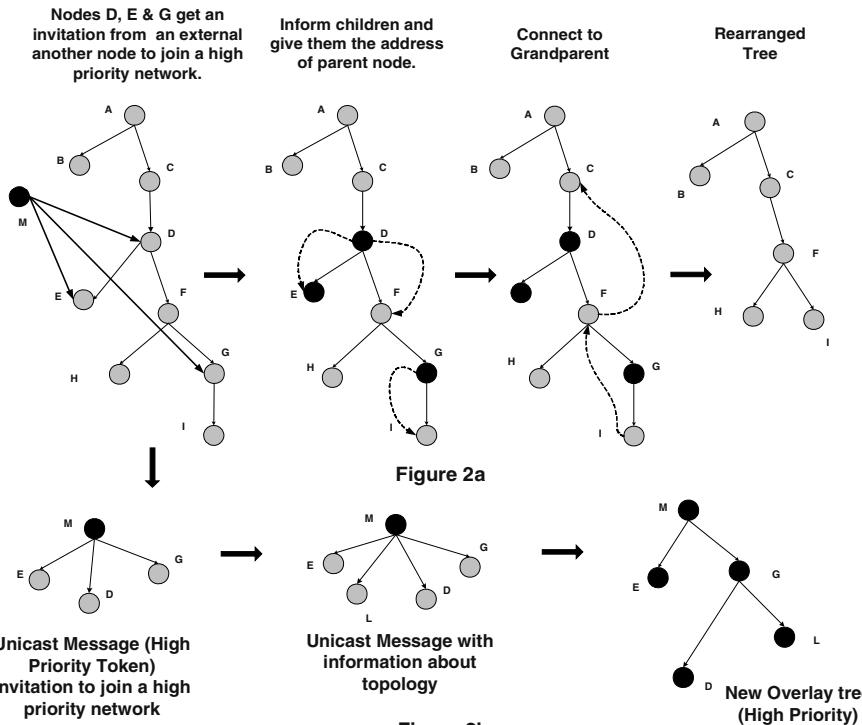


Fig. 2. Formation of a high priority tree and rearrangement of the old tree

Fig. 2b shows the formation of the new (high priority) tree. At the beginning, the external node M contacts nodes D, E & G of the original tree and another external node L to form a high priority network. In the first step, M asserts its priority by sending a unicast token message to each of the desired nodes. In Step 2, M exchanges

information about the formation of the tree topology. The topology information can be based on the location information of nodes with respect to each other. Step 3 is the final formation of the tree. The number of steps can vary depending on the implementation or the algorithm used for tree formation. In our approach, the location information is available to the source node and hence it decides on the topology and informs the other member nodes.

4 A Security System Example

This section describes the design and implementation of POMA with a specific example – implementing security in a crowded stadium. Consider a football field with thousands of spectators. It would be unwise to have all the security personnel at one place waiting for something to go wrong. A more sensible approach would be to have security personnel scattered across the stadium to monitor any suspicious activity in the crowd. At anytime, a security guard can request assistance from nearby guards if he/she observes any suspicious activity in the crowd. Usually, suspects are described by their color, height, clothes and facial description. This information can be ambiguous and insufficient when monitoring a large crowd. Recently, security systems have started using advanced (multimedia) ways to describe suspects. Images or video clips (live or recorded) can be used to accurately identify suspects. In POMA, this multimedia information will be exchanged between security personnel's wireless handheld devices using overlay multicast networks. POMA makes use of the concept of role-based priority tree described earlier. It also utilizes location information while building the overlay trees. Mobile devices carried by the event organizers and the security personnel would be tuned to communicate over the same wireless network. Under normal conditions, security nodes and nodes belonging to event organizers and stadium management would all belong to the same tree.

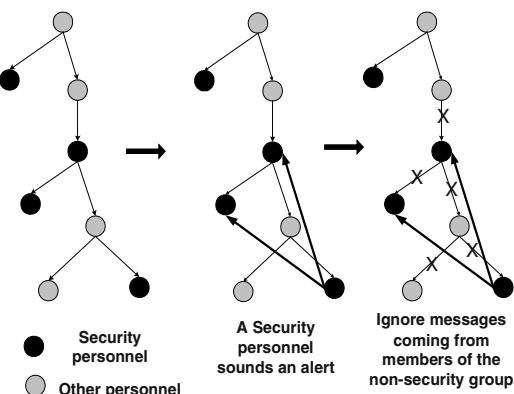


Fig. 3. Role-based partitioning of an overlay network

If a security node observes any suspicious activity, it can initiate the formation of a high priority network consisting of only security nodes. This (initiating) node would implement some form of neighbor discovery protocol to identify mobile devices carried by near-by security personnel. This neighbor discovery protocol can be made proactive where it periodically checks for neighboring devices or user initiated where the user sets off the search. The received alert message will move the near-by security nodes to a higher priority network. This will make them ignore messages from non-security nodes that would (now) belong to a lower priority network. Fig. 3 shows such a scenario. The formation of such a trees and the rearrangement of orphan nodes is similar to the description mentioned in Section 3. It is important to note that with higher density of nodes in a given area, there is more overlapping between the coverage areas of nodes, which would result in better (multi-hop) communication between member nodes and result in lower latency. Simulation results showing the effect of density of nodes to the latency will be presented in Section 5.3.

5 Simulation Methodology and Results Analysis

The simulations were carried out on ns (2.1b7a) simulator using CMU extension (for Ad-Hoc Network) on a P4 Dell Precision 330 running RedHat Linux 7.3. As of this writing, ns does not have any extension for simulating overlay multicast. With the help of bash scripting, the traffic pattern generated by CMU's cbrgen utility was modified to represent an overlay network. CMU's setdest utility was used to generate different node positions and movement patterns. The parameters considered were number of nodes, pause time, speed, time of simulation and the area of simulation. The nodes in the simulation move according to the 'random waypoint' model [4]. Since the performance of the protocols is very sensitive to the movement pattern, the result values are average of 10 different movement patterns (for every combination considered). In the rest of the section, we first identify a good unicast routing protocol for SOMA. We then explore the use of location information to build more efficient overlay tree for SOMA. We further study impact of density of wireless nodes to the system performance. Due to space limitation, detailed simulation setup parameters and other performance results can be found in [9].

5.1 Protocol Identification

Since an overlay network forms a logical network consisting of multicast member nodes, the underlying network looks at the data exchange between member nodes as a unicast communication. This unicast communication can make use of the various ad-hoc routing protocols DSR [4], AODV [5], DSDV [6] TORA [3] available for MANET. In most cases, the logical network remains more or less static even though the underlying physical topology is changing with time. The ad-hoc routing protocols are designed to adapt to the changing network dynamics. The change in physical topology is transparent to the overlay members and they are required to provide multi-cast functionality only. This paper examines different unicast ad-hoc routing protocols

in an attempt to find an efficient protocol with low latency, less drop rate and minimal overhead. In order to identify a good ad-hoc routing protocol, simulation results for two overlay trees (Fig. 4) were analyzed. TORA [3] was one of the candidate protocols; however, due to its very poor performance it was eliminated in the initial rounds of simulations. The speeds considered in the simulation were 1m/sec (human walking) and 5m/sec (human running). The average time to complete the transfer of a 100K file to all the member nodes along with the average drop ratio (ratio of total number of packet dropped to the total number of sent packets) and average protocol ratio (ratio of total number of protocol message packets to the total no of sent packets) was observed and analyzed. For all of the above three, lower values mean better performance. Fig. compares the performance of the three protocols in terms of the average completion time. It is clear that DSDV shows poor performance. Fig. 6 shows that AODV has a very high drop ratio compared to the other two protocols, while Fig. 7 indicates that DSDV has a very high protocol overhead. The simulation results for the twin topology (see Figs. 6–9) show the same trends as for the generic tree.

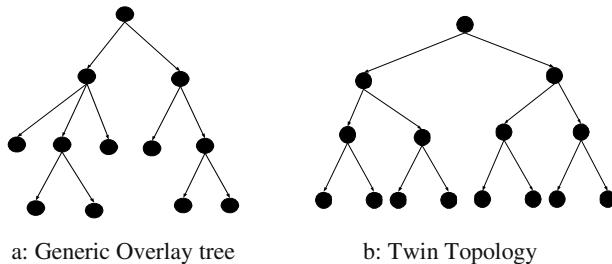


Fig. 4. Topologies used for testing different protocols

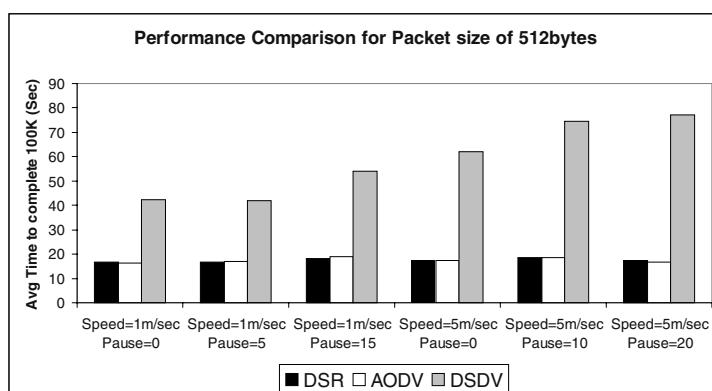


Fig. 5. Comparison of avg completion time for packet size of 512 for 3 protocols

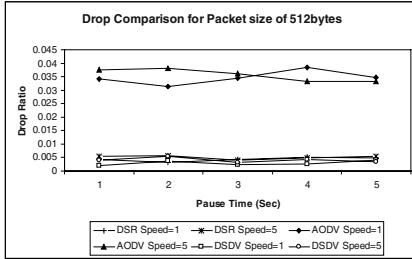


Fig. 6. Performance in terms of drop ratio

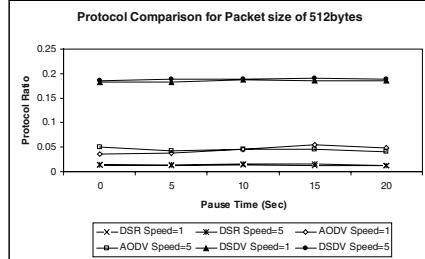


Fig. 7. Performance in terms of protocol overhead

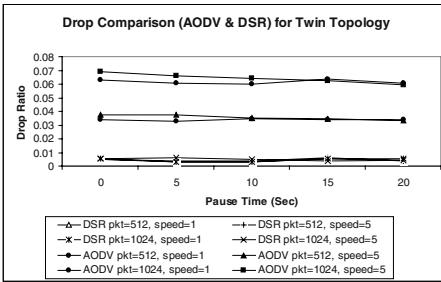


Fig. 8. Drop ratio comparison for DSR & AODV (Twin Topology).

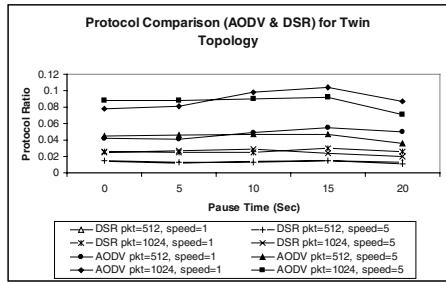


Fig. 9. Performance in terms of protocol overhead for twin topology.

DSR and AODV show similar performance in terms of transfer time. However, AODV has a very high drop ratio and the drop ratio increases with the packet size. It was also observed that higher packet size reduces the transfer time [9]. AODV has higher protocol overhead compared to DSR. AODV normally requires periodical transmission of a *hello* message (with a default rate of once per sec). DSR carries with it the advantage of source routing where the packets carry the information about the route to the destination. As a result, aside from the initial route discovery overhead, DSR does not exhibit a high routing overhead. On the other hand, in case of AODV, each node participating between the source and the destination needs to maintain information about the route. With these factors in mind, *packet size of 1024 and DSR (underlying routing protocol) were chosen for further simulations*.

5.2 Location-Based Trees

With advances in location sensing techniques, it is now possible to inexpensively establish the position of an object within 2 meters in an indoor environment [8]. Improvement in GPS systems has enhanced the accuracy in outdoor location positioning. Ad-hoc networks are characterized by dynamically changing topology and an exciting

approach would be the use of location sensing technologies to identify near-by member nodes during tree formation.

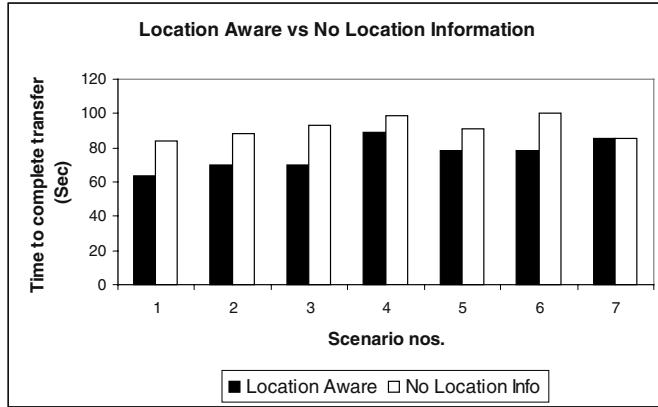


Fig. 10. Performance of location aware overlay tree and an overlay tree without any location information about member nodes.

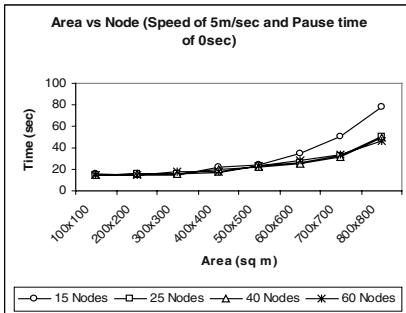


Fig. 11. Performance comparison for different number of nodes and areas.

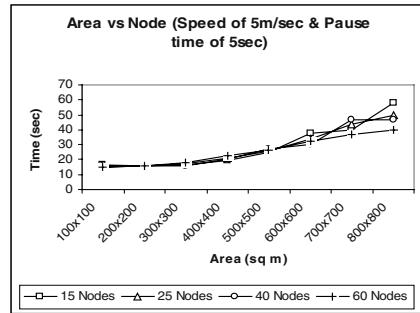


Fig. 12. Performance comparison for different number of nodes and areas.

An approach to use the geometric distance as heuristic in tree formation in ad-hoc networks has been suggested in [7]. Simulations were carried out on location aware overlay trees. Fig. 10 compares the performance of overlay trees built with and without location information in seven different movement scenarios. It can be seen that location-aware overlay trees have a lesser latency compared with trees built without any location information.

5.3 Density of Wireless Nodes

Wireless nodes have a limited coverage area. As a result, the density of wireless nodes in a given area greatly influences the performance of the network. With higher density of nodes in a given area, there are more nodes to perform multi-hop forwarding re-

sulting in improvement in the overall performance. In our simulation setup [9], the area was incrementally changed from $100 \times 100\text{m}^2$ to $800 \times 800\text{m}^2$. Also, a total of 15, 25, 40 and 60 nodes were tested with the different areas. The results are presented in Fig. 11 and 8. In smaller areas, even with fewer nodes, the coverage region of nodes overlaps and hence the performance is hardly affected by the number of nodes in the environment. However, as the area in the simulation is increased, the nodes are scattered and the coverage areas have very little overlap. The number of hops from source to destination increases and, as a result, the latency increases.

6 Conclusion and Future Work

This paper presented a new infrastructure-less, need-based, security system named POMA. The idea of building several role-based priority trees in the same environment can find many interesting applications in the future. Overlay trees with lower latencies can be designed by making use of location information. In the future, some form of location information will be used along with other factors in building efficient overlay trees. The concept of file fragmentation and simultaneous fragment download from multiple hosts has been around in the peer-to-peer world for quite a while. However, this concept has not been tried in wireless environments and will be investigated.

References

- [1] C. Gui and P. Mohapatra, "Efficient Overlay Multicast for Mobile Ad Hoc Networks," Wireless Communications and Networking Conference (WCNC), 2003
- [2] S. Banerjee and B. Bhattacharjee. Analysis of the NICE Application Layer Multicast Protocol. Technical report, UMIACSTR 2002-60 and CS-TR 4380, Department of Computer Science, University of Maryland, College Park, June 2002.
- [3] Vincent D. Park and M. Scott Corson, "Temporally-Ordered Routing Algorithm (TORA)," Internet-Draft, draft-ietf-manet-tora-spec-00.txt, November 1997.
- [4] Johnson, D. and Maltz, D. (1996). "Dynamic source routing in ad hoc wireless networks," in Mobile Computing (ed. T. Imielinski and H. Korth), Kluwer Academic Publishers, Dordrecht, The Netherlands.
- [5] C. E. Perkins, "Ad-hoc on-demand distance vector routing," in MILCOM '97 panel on Ad Hoc Networks, Nov. 1997.
- [6] Charles E. Perkins and Pravin Bhagwat. Highly dynamic Destination-Sequenced Distance-Vector routing (DSDV) for mobile computers. In Proceedings of the SIGCOMM '94 Conference on Communications Architectures, Protocols and Applications, pages 234–244, August 1994.
- [7] K. Chen and K. Nahrstedt, "Effective Location - Guided Tree Construction Algorithm for Small Group Multicast in MANET." In Proceedings of IEEE/Infocom'02, May, 2002.
- [8] Lionel M. Ni, Yunhao Liu, Yiu Cho Lau and Abhishek P. Patil, "LANDMARC: Indoor Location Sensing Using Active RFID", IEEE PerCom 2003.
- [9] Abhishek Patil, Yunhao Liu, Lionel Ni, Li Xiao, A-H Esfahanian, "SOMA: Security Using Overlay Multicast in Ad-Hoc Environments", Technical Report, MSU-CSE-03-21, Michigan State University, May 2003 (www.cse.msu.edu/~patilabh/tr/soma.pdf)

Supporting Mobile Multimedia Services with Intermittently Available Grid Resources

Yun Huang and Nalini Venkatasubramanian

Dept. of Information & Computer Science,
University of California, Irvine, CA 92697-3425, USA
{yunh, nalini}@ics.uci.edu

Abstract. Advances in high quality digital wireless networks and differentiated services have enabled the development of mobile multimedia applications that can execute in global infrastructures. In this paper, we introduce a novel approach to supporting mobile multimedia services by effectively exploiting the intermittently available idle computing, storage and communication resources in a Grid infrastructure. Specifically, we develop efficient resource discovery policies that can ensure continuous access to information sources and maintain application Quality-of-Service (QoS) requirements, e.g. required network transmission bandwidth on the mobile clients. Our performance studies indicate that mobility patterns obtained via tracking or user-supplied itineraries assist in optimizing resource allocation. The proposed policies are also resilient to dynamic changes in the availability of grid resources.

1 Introduction

The emerging digital wireless network infrastructure is being fueled by a number of factors such as the development of small and powerful mobile devices and the rising use of such mobile devices for a variety of business and entertainment applications. Mobile multimedia applications still present a significant challenge due to (a) large storage, bandwidth and computation requirements and (b) limited memory, computation and power on handheld devices. Furthermore, the mobile nature of the clients leads to frequent tearing down and reestablishment of network connections; the latency introduced by this process causes jitters which may be unacceptable for delay-sensitive streaming multimedia applications. Solutions such as *Mobile IP* introduce additional latency. While buffering at the client is a usual solution to jitter problems, mobile hosts often have limited memory resources.

One possible solution to achieve real time delivery guarantees in mobile environments is to use connection-oriented services combined with stream buffering in the path of service [18]. Another popular technique is to transcode [3] the incoming stream from the server at a local proxy that can customize the MM content based on user characteristics. Since different users require information at different QoS levels and devices may have varying resource/power capabilities, personalized customization of applications can achieve QoS assurance while prolonging the lifetime of the mobile device. Our objective is to use locally available (idle) grid

resources to customize multimedia applications for mobile users based on user requirements and device limitations.

Several complications arise in ensuring the effective utilization of grid resources for mobile multimedia services. Firstly since grid resources are intermittently available; optimal scheduling policies must take the availability of grid resources into account. Secondly, the heterogeneity of grid resources and clients [12] complicates the resource management issue. Thirdly, user mobility patterns may not be known. Furthermore, since MM applications have QoS requirements (e.g. required network transmission bandwidth, accuracy and resolution of displayed images); an effective resource allocation policy must address the performance-quality tradeoff. This tradeoff arises since finding optimal local resources for a mobile host that lowers overall network traffic (i.e. improves performance) can introduce frequent switches in the multimedia stream possibly leading to increased jitter (lower QoS).

The rest of this paper is organized as follows. Section 2 describes a grid-based mobile environment for a video streaming service, and introduces the middleware architecture for such an environment. Section 3 introduces GRAMS (Grid Resource Allocation for Mobile Services), a generalized resource discovery algorithm and proposes a family of GRAMS policies to address the dynamic nature of the mobile grid environment. We evaluate the performance of the GRAMS policies under different resource and mobility conditions in Section 4, and conclude with related work and future research directions in Section 5.

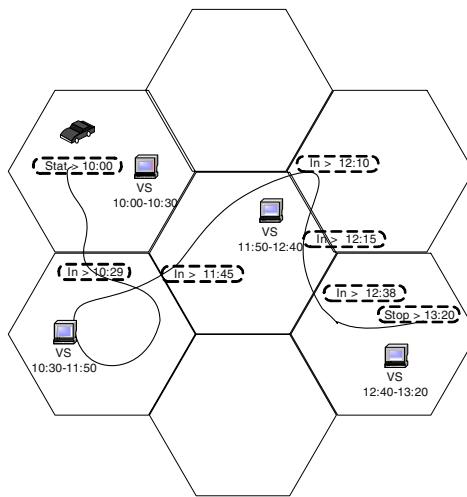


Fig. 1. System environment: Volunteer servers in the grid are used to provide multimedia services for mobile clients.

2 A Middleware Framework for Mobile Grid Services

Figure 1 depicts the system environment in which a mobile user submits a request for streaming video data and traverses through a series of “cells” before arriving at a final

destination¹. Components within a cell that might be involved in the MM session include base stations, *Volunteer Servers* (VSs) and mobile clients. *Volunteer Servers* (VS) participate in the grid by supplying idle resources. A VS may be a PC, work station, server, cluster, etc, which can provide high capacity storage devices to store the multimedia data, CPU processor for decompression and/or transcoding, buffer memory, and NIC (network interface card) resources for real-time multimedia retrieval and transmission. The VSs studied in this paper are fixed wired machines, whereas the mobile hosts connect to the infrastructure using a locally available wireless network. The mobile clients can be a PDA, handheld, laptop, or any wireless devices that can download and play video. They move around the cells, communicate with the base station, and hand in requests for the multimedia services.

In this paper, we address the issue of discovering and scheduling intermittently available grid resources to streaming video applications on mobile clients. Our approach is to divide the whole service period into non-overlapping chunks (possibly of different sizes). Subsequently, we attempt to map each chunk to an appropriate VS, for example one that is geographically close and lightly loaded. Video objects are also divided into equal sized segments and corresponding video segments are downloaded on the selected VS. The VS processes the request by transcoding the video segments and transmits the video stream to bandwidth limited and performance limited mobile clients, such as PDAs via wireless links. For example, in Fig 1(a), a mobile user goes through 5 cells. The service may be divided into 4 chunks that are served by various VSs. By choosing a VS close to the mobile client for performing the adaptation, the network transmission delay and overhead is reduced. Note that this approach has the additional benefit of accommodating varying wireless bandwidth in the local region. Video segmentation [4, 10], intelligent caching [6], synchronization with handoffs [14] and high speed wireless transmission [21] make our approach feasible. In this paper, we assume that play-out buffer is correctly used to mask jitters by VS switches.

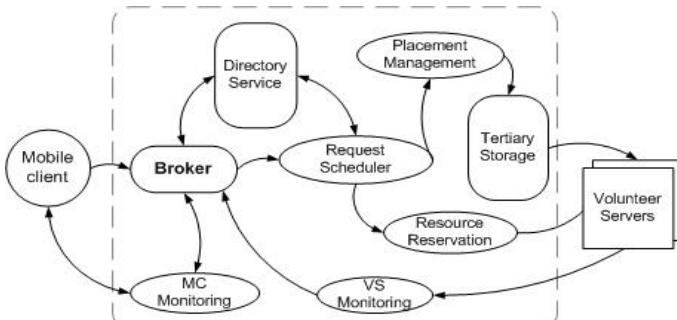


Fig. 2. A middleware service architecture

Fig 2 illustrates the various middleware components for discovering intermittently available resources for mobile multimedia applications. The key components are the *Broker* that performs admission control for incoming requests and manages their rescheduling; and the *Directory Service* module that stores information about

¹ Mobile wireless clients operate in infrastructure mode. Furthermore, we do not address issues of wireless channel allocation (typically handled at the MAC layer).

resources availabilities at the VSs and clients. When a mobile client (*MC*) hands in a request, the *Broker* sends the request to the *Request Scheduler* which executes the resource discovery algorithm based on the resource and network information retrieved from *Directory Service*. If no scheduling solution exists the *Broker* rejects the request; otherwise, the *Resource Reservation* module reserves the resource and the *Placement Management* replicates video segments from *Tertiary Storage* on each selected VS accordingly. The *Directory Service* is then updated with the new allocation information. If VS availability is changed, multiple preassigned requests may be invalidated. The *VS Monitoring* module detects dynamic changes in VS availabilities; the *Broker* is informed of these changes and the new configuration is registered in the *Directory Service*. Requests preassigned to VSs that fail unpredictably will need to be rescheduled, but if such requests cannot be rescheduled they will experience completion failures. The *MC Monitoring* module keeps track of the client's moving patterns. Changes detected by monitors can be reported to the *Broker* so that the necessary rescheduling processes will be triggered.

3 Grid Resource Allocation for Mobile Services (GRAMS)

In this section, we build upon our prior work [17] that uses a graph-theoretic approach to address grid resource scheduling for static clients executing QoS-sensitive applications. We now propose the GRAMS (Grid Resource Allocation for Mobile Services) algorithm to find suitable VSs for mobile hosts in a grid system. The objectives of the GRAMS process is to (a) satisfy user QoS requirements; (b) discover localized grid resources so as to minimize network overhead(c) bound the number of VS switches during the lifetime of a request; and (d) adapt to dynamic changes in user mobility and VS availabilities. We now define terms and notational conventions that will be used in the rest of this paper.

LoadFactor of a VS: In order to deal with the capacity of each VS over time in a unified way and represent how much a request will affect the server during the requested period of time, we define the notion of a *LoadFactor*. Since the amount of available resources on a VS can vary over time, we divide the entire duration of a day into time units, e.g. 10-minute time units. For each time unit, we consider four resources on a VS that must be allocated for every incoming request - CPU, memory (MEM), network bandwidth (NBW) and disk bandwidth (DBW). The Load Factor LF_t for a request R on VS j at a particular time unit t , as:

$$\begin{aligned} LF_t &= \text{Max } [\text{CPU}_a, \text{MEM}_a, \text{NBW}_a, \text{DBW}_a] \text{ where:} \\ X_a &= R_X / S_{\text{AvailX}}(t); R_X \text{ is the requested resource X} \\ \text{and } S_{\text{AvailX}}(t) &= \text{the amount of resource X available at time t.} \end{aligned} \quad (1)$$

Thus, the *LoadFactor* LF_t is determined by the bottleneck resource during time unit t . However, the duration of one requested period T may cover multiple time units; we therefore use the average *LoadFactor* over the time units.

Focus of the request for a given time period: represents a central point in space where a *MC* is likely to spend the maximum amount of time during a given chunk of a request. When the mobility patterns of clients are unknown, we use the starting position as the focus, and adjust it at runtime using information from mobility tracking. Note that the entire request can be divided into chunks with each chunk having its own focus. Section 3.3 presents an algorithm for calculating the focus.

VS factor: In order to evaluate the benefit of choosing VS_j for a time period T of request R , we define a *VS factor* based on VS_j availability, the current load and the distance between the VS_j and the focus of the request R as:

$$VS\ factor(j, R, T) = \frac{ava(j, T)}{LoadFactor(j, R, T) * dist(j, R)}, \text{ where} \quad (2)$$

$$ava(j, T) = \begin{cases} 1, & \text{if } VS_j \text{ is available at } T; \\ 0, & \text{otherwise} \end{cases}$$

$$dist(j, R) = dist \text{ from } VS_j \text{ to } R's \text{ focus during } T.$$

MM Segment: A multimedia object with total service duration R_d (if continuous execution) is divided into N uniform sized segments of size S_d . ($R_d = N * S_d$).

3.1 A Generalized GRAMS Solution

Given a request $R(VID, itinerary)$, where VID identifies the video object requested by the MC and the *itinerary* contains mobility information of the host (NULL, if no mobility information is available), we determine an appropriate *VS* allocation. A generalized GRAMS algorithm (See Fig 3) has three main steps. (1) *PartitionServicePeriod()* partitions the whole request service period into a fixed number of chunks that can potentially determine the number of *VSs* to be used for the request. (2) *VolunteerServerAllocation()* maps the chunks to specific *VSs* based on the load, availability and proximity. (3) *MobilityBasedRescheduling()* tracks the location of the mobile client (MC) and triggers rescheduling when the MC moves far away from the preassigned *VS*. We will describe each step in the following sections.

```

GeneralizedGRAMS(Request) {
    BOOLEAN found = true;
    PartitionServicePeriod(); // step 1
    FOR each chunk
        IF (VolunteerServerAllocation() == null) THEN found = false;
        IF (found) THEN MobilityBasedRescheduling() ELSE Reject the request;
    }
}

```

Fig. 3. A generalized GRAMS Algorithm

Step 1. Partition Service Period: We initially partition the service period using one of two possible strategies: (a) divide the whole service into uniform sized chunks, in multiple of S_d , (b) use a *Faststartup* Partition, which attempts to reduce the replication latency for the first segment of the video replica by choosing a short initial chunk and treating the remaining service time as another large chunk.

Step 2. Volunteer Server Allocation: We choose multiple *VSs* to startup the continuous media service immediately; this is implemented by a network flow algorithm devised in [17]. Specifically, for each chunk of the service period, we choose the *VS* with the largest value of *VS factor*. If it is possible to find *VSs* for all chunks, there exists a scheduling solution; otherwise, the request is rejected. To calculate the *VS factor* (defined in section 3.1), we use the Euclidean distance

between the *VS* and the chunk focus. Without knowledge of mobility pattern, we use the starting point to be the focus, and we present optimizations to the *VS* selection process in section 3.3 when knowledge of the host itineraries is available.

Step 3. Mobility-based Rescheduling: Generally, randomness of client mobility will not affect the service completion ratio once the request has been scheduled, as the *VSs* connect to *MC*'s access point (base station) via a wired network. However, the pre-assigned *VS* may no longer be optimal due to changes in client itinerary; increased path lengths can introduce additional delays, jitter and network traffic. The system architecture can be enhanced by adding a mobility tracking module, which keeps track of *MC*'s location, and informs the broker to trigger rescheduling.

3.2 GRAMS Optimizations Using Client Mobility Information

In this section, we propose optimizations to step 1 and step 2 of the generalized GRAMS algorithm given knowledge of client mobility patterns.

Optimizing Step 1 (Partition Service Period): In this case, the itinerary is specified as a series of times when a host moves into a new cell, e.g. $[(Time_1, Cell_1), \dots, (Time_N, Cell_N)]$. We first count the duration the *MC* stays in each cell and then partition the entire service period into chunks using one of two policies. A temporally-biased, *MajoritySpreadover* Partition policy attempts to minimize the number of *VS* switches. Here, we first choose the majority interval (cell with longest D_i) as the centerpoint of the service time; then spread the service time forward until the request submission time. If the entire service duration is not satisfied, we spread beyond the centerpoint until service can be completed. Note this policy may bring a period of startup latency between the submission and the startup time. Policy 2: The second policy is a spatially biased, *Distance* Partition, which partitions the whole service time into chunks based on the distances between the cells traversed. We apply a well known unsupervised neural learning technique, self-organizing map (SOM) [19] to partition the whole moving area into a number of regions; therefore, the service period will be partitioned into chunks accordingly.

Optimizing step 2 (Volunteer Server Allocation): Given the *MC*'s itinerary, we can determine the chunk focus in step 2 more accurately. If (a_i, b_i) represents the coordinates of the center for cell i , and D_i represents the time duration spent in cell i , we convert the problem of locating the chunk Focus into a Minisum Planar Euclidean Location Problem [15]. Our objective is to minimize the overall service time $f(x, y)$, where (x, y) represent the coordinates of the Focus position for this chunk that is composed of N cells. We calculate $f(x, y)$ by applying Weiszfeld's algorithm. [15]

$$f(x, y) = \sum_{i=1}^N D_i \sqrt{(x - a_i)^2 + (y - b_i)^2}. \quad (3)$$

A schedulable request may not be completed eventually due to dynamic changes in *VS* availability. To achieve system robustness, we reallocate other *VSs* for the interrupted requests to fulfill the remaining service. When a specific *VS* becomes unavailable, the broker retrieves information about requests that are scheduled on the unavailable *VS*, and triggers the re-scheduling process for each invalidated service. If requests cannot be rescheduled, the broker reports a request failure, in which case any

resources reserved for this failure request on other selected VSs for the remaining service should be released. Devising optimal online approaches to deal with dynamic changes in VS availability is beyond the scope of this paper.

4 Performance Evaluation

In this section, we analyze the performance of the resource discovery strategies under various VS configurations and host mobility patterns.

The Simulation Environment: We simulate a cellular network system with 100 cells, with 50 VSs distributed evenly. Each VS has a storage of 100 GB and network bandwidth of 100Mbps. For simplicity, CPU and memory resources of the VSs are assumed not to be bottlenecks. We set the duration of each video that ranges from 1 to 3 hours; each video replica requires 2 GB disk storage, and network transmission bandwidth that ranges from 500 kbps to 2 Mbps. The shortest segment of a video object can run for 10 minutes. These parameters can be varied to simulate different configurations.

Incoming multimedia requests are characterized by using a Zipfian distribution [9]. The request arrivals per day for each video V_i is given by:

$$\Pr(V_i \text{ is requested}) = \frac{K_M}{i}, \text{ where } K_M = \left(\sum_{i=1}^M \frac{1}{i} \right)^{-1}. \quad (4)$$

The probability of request arrivals in an hour j will be:

$$p_j = c / j^{1-\phi}, \quad c = 1 / (\sum (1/j^{1-\phi})) \text{ for } 1 \leq j \leq 24; \quad (5)$$

where Φ is the degree of skew and is assumed to be 0.8.

We use the incremental mobility model [13] to characterize random mobility of each MC, and study three main host mobility patterns: (a) *Random movement*: the MC will travel at any speed, heading in any direction. (b) *Constant velocity with intermediate halt*: a linear and straight moving pattern with one long stop where the MC remains stationary. (c) *Clustered movement*: the MC's moving area can be geographically partitioned into at least 2 clusters.

To model the intermittent availability of grid resources (Vss), we use a uniform availability policy where all VSs are available (or unavailable) for an equal amount of time (6 hours in the illustrated results). Furthermore, the VSs are divided into groups such that within each group the time distribution covers the entire 24-hour day. For a more detailed analysis of the performance of GRAMS under various other time-map and grid availability models, see the technical report[22].

Experimental Results: We evaluate the performance of various policies by using the number of rejections over execution time as the main metric. The average distance from the selected VS and request and the service completion ratio are also used in evaluations.

We discuss the performance of the various GRAMS policies under different request models (Fig 4 a,b,c). The time map of each VS in Fig 4 (a,b,c) follows a *Uniform availability* pattern. The *Single VS* views the itinerary as a whole chunk with the service time R_d . Two policies (*Single VS* and *Faststartup*) assume no knowledge of mobility patterns, while the remaining two policies (*MajoritySpreadover* and

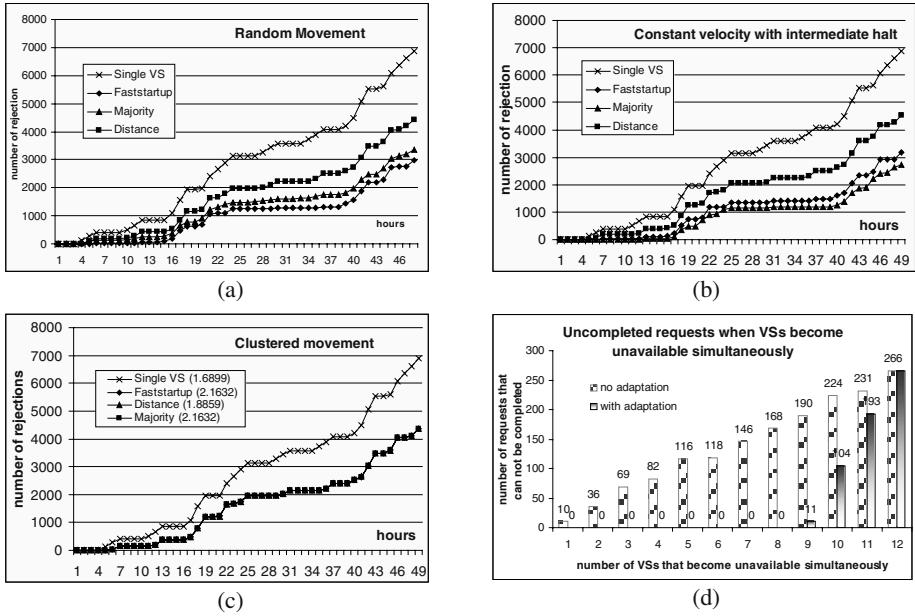


Fig. 4. (a), (b) and (c) illustrate experimental results for various policies under different mobility patterns; (d) shows experiment results under dynamic changes in VS availability

Distance Partitioning) exploit the knowledge of user itineraries. When using the *Random* mobility model, the *Faststartup* policy performs the best with the least number of rejections (Fig 4a). However under the *Constant velocity with intermediate halt* model, the *MajoritySpreadover* policy has the least number of rejections (Fig 4b). Finally, when using *Clustered movement*, three of the policies (except *Single VS*) have similar rejection ratios, but *Distance* partition performs better by selecting VSs closer to the request (In Fig 4c, the number in the parenthesis marks for each policy represents the average distance from selected VS to the request). In other words, the results match the intuition behind the development of these policies. As can be observed, knowledge of the mobility model can help improve GRAMS performance significantly, note that knowledge of user itineraries can help avoid unnecessary advance replications to VSs that may be unused due to mobility-based rescheduling.

When dynamic changes to VS availability occur, a number of factors affect the overall request completion ratios - the initial time maps of each VS, and the current load. Given the limited space, we only present the experiment results under a *Uniform availability* time map, *Random* mobility pattern, and a *Faststartup* policy. During the simulation, we have a total of 12 VSs available at the moment when changes happen. We gradually increase the number of simultaneous VS failures and observe its impact on request completion ratios. As Fig 4d illustrates, we can significantly decrease the number of requests that fail to complete due to dynamic changes in VS availability by applying the adaptation strategy proposed in section 3.4. When the number of simultaneous VS failures increases to the extent that a large portion of the grid is unavailable (more than 8 VSs in our case), there are fewer overall resources available,

causing increasing numbers of request completion failures. We also study the impact of various time map models, cell sizes and host velocities on different policies (See [22] for more details. In summary, the GRAMS approach and adaptations are resilient to dynamic changes in VS availabilities and can significantly improve request completion ratios under unpredictable system conditions.

5 Related Work and Concluding Remarks

Streaming multimedia information to mobile devices has been addressed in [20]. Battery power sensitive video processing and very-low-bit-rate video delivery techniques have been devised for streaming video information in a heterogeneous mobile environment [1]. Proxy-based systems have been developed for web browsing in mobile environments [16]. Resource management in the grid has been addressed by several metacomputing projects such as Legion [8], Globus [11], AppLes [7], GrADS [5] and Nimrod [2], etc. Support for MM applications in the grid environment has been addressed in the context of Globus via the definition of a QoS component called Qualis, where low level QoS mechanisms can be integrated and tested.

In this paper, we have proposed service partitioning algorithms for mobile environments that effectively utilize available grid resources to significantly improve system performance. We show how to take advantage of apriori knowledge of mobility patterns to tailor the scheduling policies for better overall performance and enhanced user QoS. Resource discovery mechanisms for mobile grid environments must be resilient to changes in user mobility patterns and server time maps. We have developed effective extensions to the GRAMS policies to deal with the dynamic changes in VS availability and mobile host itineraries. Future work will address the degree of location awareness required by the middleware for efficient allocation of grid resources in a scalable fashion. A specific extension is to allow for a distributed brokerage service that will allow for localized resource discovery. We are also looking into tradeoffs that arise when timeliness requirements interfere with other application requirements such as security and reliability.

References

1. P. Agrawal, S. Chen, P. Ramanathan and K. Sivalingam, *Battery Power Sensitive Video Processing in Wireless Networks*, Proceedings IEEE PIMRC'98, 1998.
2. D. Abramson, J. Giddy, and L. Kotler, *High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?* International Parallel and Distributed Processing Symposium, 2000.
3. P. Shenoy and P. Radkov, *Proxy-assisted Power-friendly Streaming to Mobile Devices*, In Proceedings of the Multimedia Computing and Networking (MMCN) Conference, 2003.
4. S. Chen, B. Shen, S. Wee, and X. Zhang, *Adaptive and lazy segmentation based proxy caching for streaming media delivery*, Proc. of ACM International Workshop on Network and Operating Systems Support for Design Audio and Video, 2003.
5. F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, et. al. *The GrADS Project: Software Support for High-Level Grid Application Development*, International Journal of High Performance Computing Applications, 2001 pp. 327–344.

6. W. H. O. Lau, M. Kumar and S. Venkatesh, *A cooperative cache architecture in support of caching multimedia objects in MANETs*, Proc. ACM international workshop on Wireless mobile multimedia, 2002.
7. F. Berman and R. Wolski. *The AppLeS project: A status report*. Proceedings of the 8th NEC Research Symposium, 1997.
8. S.J.Chapin, D. Katramatos, J.F. Karpovich, A. Grimshaw, Resource Management in Legion, University of Virginia Technical Report CS-98-09, 1998.
9. A. Dan and D.Sitaram. *An online video placement policy based on bandwidth to space ration (bsr)*. SIGMOD, 376–385, 1995.
10. A. Dan, M Kienzle, D Sitaram, *Dynamic Policy of Segment replication for load-balancing in video-on-demand servers*. ACM Multimedia Systems, 1995.
11. I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke. *Software Infrastructure for the I-WAY High Performance Distributed Computing Experiment*. Proc. IEEE Symposium on High Performance Distributed Computing1997
12. D. Xu, D. Wichadakul, and K. Nahrstedt., *Multimedia Service Configuration and Reservation in Heterogeneous Environments*. International Conference on Distributed Computing Systems, 2000.
13. Z. Haas. *A new routing protocol for the reconfigurable wireless networks*, In Proceedings of the IEEE Int. Conf. on Universal Personal Communications, 1997.
14. C. K. Hess, D. Raila, R. H. Campbell, D. Mickunas. *Design and Performance of MPEG Video Streaming to Palmtop Computers*. Multimedia Computing and Networking 2000.
15. Philip M. Kaminsky, IEOR 251, Logistics Modeling.
<http://www.ier.berkeley.edu/~kaminsky/ier251/notes/2-3-03b.pdf>
16. A. Joshi. *On proxy agents, mobility and web access*. ACM Journal of Mobile Networks and Applications, 2000.
17. Y. Huang, N. Venkatasubramanian, *QoS-based Resource Discovery in Intermittently Available Environments*, 11th IEEE High Performance and Distributed Computing, 2002.
18. R. H. Katz, *Adaptation and Mobility in Wireless Information Systems*, IEEE Personal Communications, 1994.
19. T. Kohonen, K. Mäkisara, and T. Saramäki, *Phonotopic maps - insightful representation of phonological features for speech recognition*. Proc. International Conference on Pattern Recognition, 1984
20. K. Keeton, B. Mah, S. Seshan, R.H. Katz, D. Ferrari, *Providing Connection-Oriented Network Services to Mobile Hosts*, Proceedings of USENIX Symposium on Mobile and Location-Independent Computing, 1993.
21. *3rd Generation Mobile Wireless*, A presentation on the Opportunities and Challenges of Delivering Advanced Mobile Communications Services, 2002.
22. Y. Huang, N.Venkatasubramanian, *Supporting Mobile Multimedia Services using Intermittently Available Grid Resources*, Technical report, 2003.

Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks

Vinod Tippuraj¹, Manojkumar Krishnan¹, Jarek Nieplocha¹,
Gopalakrishnan Santhanaraman², Dhabaleswar Panda²

¹Pacific Northwest National Laboratory, Richland, WA 99352, USA

{vinod.tippuraj, manojkumar.krishnan,
jarek.nieplocha}@pnl.gov

²Ohio State University, Columbus, OH 43210, USA
{santhana, panda}@cis.ohio-state.edu

Abstract. This paper describes a comparative performance study of MPI and Remote Memory Access (RMA) communication models in context of four scientific benchmarks: NAS MG, NAS CG, SUMMA matrix multiplication, and Lennard Jones molecular dynamics on clusters with the Myrinet network. It is shown that RMA communication delivers a consistent performance advantage over MPI. In some cases an improvement as much as 50% was achieved. Benefits of using non-blocking RMA for overlapping computation and communication are discussed.

1 Introduction

In the last decade message passing has become the predominant programming model for scientific applications. The current paper attempts to answer the question to what degree performance of well tuned application benchmarks coded in MPI can be improved by using another related programming model, remote memory access (RMA) communication. In the past RMA programming model was popular on the Cray T3D/E system where it was offered through the Cray SHMEM library [1]. The global address space architecture of these two Cray systems supported RMA communication very well. In fact, several of the current MPP systems only now can compete with latency and bandwidth of the RMA operations on the Cray T3E. In this comparative study, we are focusing on commodity Linux clusters with Myrinet. We chose this platform not because of its merits in supporting RMA but because of its popularity. In fact, Myrinet offers good support for message passing but rather limited support for RMA communication-- only the put operation has a native implementation in hardware. However, the next version of the Myricom GM programming interface will support get operation.

We use several popular scientific benchmarks and applications such as NAS CG and MG, SUMMA matrix multiplication, and Lennard Jones molecular dynamics to evaluate the effectiveness of RMA communication. In each case, two versions of the benchmark were derived: one based on blocking and the other non-blocking commu-

nication. The goal was to determine what additional performance benefit non-blocking RMA can offer in each individual benchmark. This paper demonstrates even on a network with limited support for RMA, this communication paradigm can offer consistent performance advantages over message passing. These results are quite encouraging especially since the network vendors are offering increasing level of support for RMA communication and the expectation that the MPI-2 1-sided implementations will eventually become more widespread (not yet offered by Myricom in their MPICH-GM library). Note that in this paper, we are not trying to evaluate the rather complex model of the MPI-2 one-sided operations or investigate its potential for high-performance implementation [2] on modern networks such as Myrinet. Instead, by using ARMCI, a low-level high-performance portable RMA library with simple progress model, published implementation approach and performance results for Myrinet [3,4], we study what benefits the RMA communication can offer in general. In addition, ARMCI could be used in MPI codes as a high-performance alternative interface to the MPI-2 one-sided operations.

This paper is organized as follows. Section 2 describes RMA communication and the Myrinet network. Section 3 describes the benchmarks used in the study and gives a synopsis of how they were converted to use RMA. Section 4 presents experimental results. Section 5 summarizes related work and the paper is concluded in Section 6.

2 Remote Memory Access Communication on Myrinet

Remote memory access operations offer support for an intermediate programming model between message passing and shared memory. This model combines some advantages of shared memory, such as direct access to shared/global data, and the message-passing model, namely the control over locality and data distribution. RMA is sometimes considered a form of message passing; however, an important difference over the MPI-1 message-passing model is that RMA does not require an explicit receive operation and thus offers increased asynchrony of data transfers. The availability of non-blocking RMA operations presents additional opportunities for overlapping data transfers and computations. Although prefetching and poststoring instructions are often supported by the shared memory hardware and are exploited by compilers to overlap computations with data movement, a scientific programmer on shared memory systems typically faces difficulties when attempting to manage explicitly overlapping of computations and communication due to the lack of precise APIs. Such explicit non-blocking APIs are present in the most RMA interfaces.

We have been developing a portable RMA interface called ARMCI [5]. It is a rather low-level interface primarily intended as a run-time system for other programming models [21] such as Global Arrays [6], Co-Array Fortran [7] or UPC [8] compilers, or portable SHMEM library [9]. However, we also use it as the RMA communication layer for the benchmarks studied in this work. For Fortran codes, appropriate wrappers were added to access the needed functionality.

In the last few years, Myrinet has become a primary network for building medium and large-scale clusters based on commodity processing nodes due to its good scalability and relatively moderate cost. GM is a low-level message-passing system for the Myrinet network [10]. The GM system includes a driver, the Myrinet-interface

control program, a network mapping program, and the GM API, library, and header files. GM features include 1) concurrent, protected, user-level access to the Myrinet interface; 2) reliable, ordered delivery of messages; 3) automatic mapping and route computation; 4) automatic recovery from transient network problems; 5) scalability to thousands of nodes; and 6) low host-CPU utilization. GM has certain limitations including the inability to send messages from or receive messages into non-DMA-able memory, and offers no support for gather or scatter operations. Moreover, memory registration operations in GM under Linux are quite expensive relative to other systems [3].

The implementation issues of an extensive set of RMA interfaces on Myrinet clusters like those offered by ARMCI have been described before [3,4]. As Myrinet GM 1.x offers only support for the put operation, other RMA operations, such as get, are implemented using a client-server approach and the GM put operation. Recently ARMCI has been expanded to support non-blocking RMA operations. Their implementations extend the original client-server architecture in a manner that reduces the host CPU involvement in the communication. This is important for applications that attempt to overlap communication with computation.

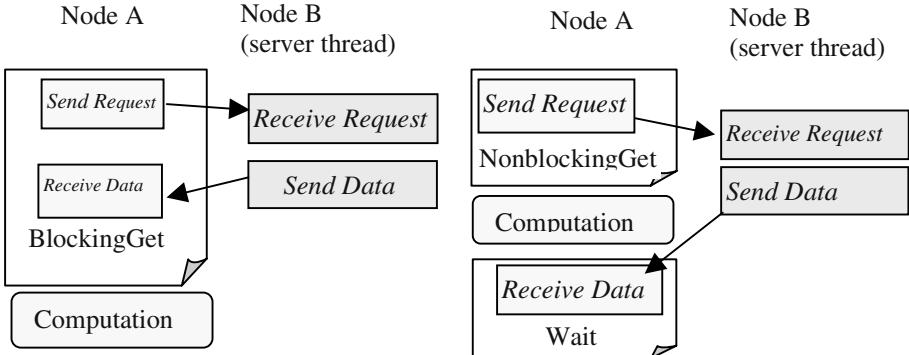


Fig. 1. Coupling of computation and communication in blocking (left) and nonblocking (right) get operation in ARMCI.

3 Application of RMA Communication in Scientific Benchmarks

To evaluate the benefits of RMA communication, we used multiple benchmarks representing a diverse set of algorithms used in scientific computing: conjugate gradient (CG) and multigrid (MG) kernel benchmarks from the NAS suite, SUMMA matrix multiplication, and a molecular dynamics application.

3.1 NAS Parallel Benchmarks

The NAS parallel benchmarks are a set of programs designed as a part of the NASA Numerical Aerodynamic Simulation (NAS) program originally to evaluate supercomputers. They mimic the computation and data movement characteristics of large-scale computations. NAS parallel benchmark suite consists of five kernels (EP,

MG, FT, CG, IS) and three pseudo applications (LU, SP, BT) programs. Our starting point was NPB 2.3 [11] implementation written in MPI and distributed by NASA. We modified two of the five NAS kernels, MultiGrid (MG) and Conjugate Gradient (CG), to replace point-to-point blocking and non-blocking message-passing communication calls with first blocking and then non-blocking RMA communication. This is just a mere replacement of the point-to-point message passing communications part of the current message passing version of CG and MG NAS kernels using ARMCI RMA blocking and non-blocking operations. Other benchmarks (e.g., FFT, IS) rely on collective communication, thus limiting the appropriateness of RMA (point-to-point) communication without reformulating the underlying mathematical algorithms. In our view, RMA is an alternative model to point-to-point message passing and a complementary model to collective operations. This view was shared by the authors of the Cray SHMEM library that offered both RMA and collective operations [1].

MG Benchmark

The NAS-MG multigrid benchmark solves Poisson's equation in 3D using a multigrid V-cycle. The multigrid benchmark carries out computation at a series of levels and each level of the V-cycle defines a grid at a successively coarser resolution. This implementation of MG from NAS is said to approximate the performance a typical user can expect for a portable parallel program on a distributed memory computer [11].

Most of the work in MG is done in four functions. Each of these functions is implemented using one or more 27-point stencils. “resid” is a function that computes the residual and operates at the same level of hierarchy. “psinv” is the smoother and also operates on the same levels of hierarchy. “interp” interpolates and “rpj3” projects between adjacent levels of hierarchy. The NPB 2.3 code uses a three-step dimensional exchange algorithm to satisfy boundary conditions. This is implemented with point-to-point message passing communication. In addition to this, point-to-point communication is used in the parallel implementation of these stencils to update every processors boundary values for each dimension that is distributed.

Our primary modification involved replacing these point-to-point communications with ARMCI RMA operations. For our implementation using ARMCI blocking operations, point-to-point communication was effectively replaced with the ARMCI_Put_notify operation. This blocking function call transfers the data to the destination processor memory and updates an internal (to the library) notify flag in the destination process memory. At the destination, arrival of this message can be (optionally) verified by making a call to ARMCI_Notify_wait that accesses the value of the notify flag. For our implementation using the corresponding non-blocking API, we attempt to achieve overlap by issuing non-blocking update in the next dimension before actually working on the data in the current dimension. This required us to use an additional set of buffers. Any explicit acknowledgement indicating the buffer availability is avoided by taking advantage of the periodic nature of the algorithm and alternating between these two sets of buffers.

CG Benchmark

In NAS CG benchmark, a conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite ma-

trix. This kernel benchmark tests irregular long distance communications and employs sparse matrix vector multiplication. The CG benchmark involves multiple iterations of a solution to the system of linear equations, $Az=x$, using the conjugate gradient method. It computes the residual norm at the end of each of these iterations. After each iteration, the eigenvalue is estimated with a shift λ . The size of the system, number of iterations involved and shift applied to the eigenvalue estimate is determined as a part of the initial setup of each class of the problem and is shown below.

Table 1. Problem sizes in the CG benchmark.

CLASS	N	Iterations	NonZeroes	λ
A	14000	15	11	20
B	75000	75	13	60
C	150000	75	15	110

Each of the CG iterations involves the following steps:

```

for i=1 to 25 {
    q = A.p          (A matrix vector product)
     $\alpha = \rho / (p^T q)$  (dot product of result of the above
                           product and transpose of p)
    z = z +  $\alpha p$ 
     $\rho_0 = \rho$ 
    r = r -  $\alpha q$ 
    ...
}
Computation of the residual norm ||r||
```

Of the above, the steps that involve most of the communication are: matrix vector multiplication, vector dot product and computation of the residual norm. The computation of the matrix vector product involves a recursive doubling based pairwise exchange. This is implemented in the original algorithm using point-to-point communication. Since the recursive doubling based pairwise exchange is a barrier in itself, it is expected that replacing them with RMA operations would not give much benefit. Even the computation of the residual norm, which is a recursive reduction based pairwise exchange, synchronizes all the processes. Hence replacing the point-to-point communication with RMA blocking operations offers limited room for improvement. However using non-blocking RMA operations, we can overlap data exchange with sum of partial sub-matrices. This is done by overlapping communication and computation with in each exchange and between different exchanges by dividing the data into two parts and overlapping a communication operation involved in the exchange of one part of the data with the sum of partial sub-matrix vector product on the second part.

3.2 SUMMA – Matrix Multiplication

SUMMA is a highly efficient, scalable implementation of common matrix multiplication algorithm proposed by van de Geijn and Watts [12]. The MPI version is the SUMMA code developed by its authors, which is modified to use a more efficient matrix multiplication dgemm routines from BLAS rather than equivalent C code distributed with SUMMA. We implemented two other SUMMA variants to use blocking and non-blocking RMA. The matrix is decomposed into sub-matrices and distributed among processors with a 2D block distribution. Each sub-matrix is divided into chunks. Overlapping is achieved by issuing a call to get a chunk of data while computing the previously received chunk, see Figure 2. The minimum chunk size was 128 for all runs, which was determined empirically and the maximum chunk size was determined dynamically, depending on memory availability and the number of processors.

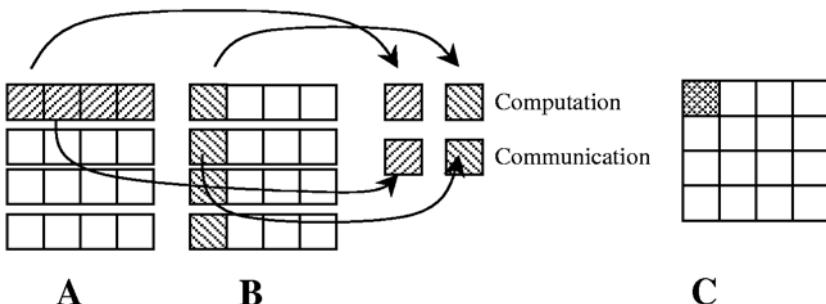


Fig. 2. Using two sets of buffers to overlap communication and computation in matrix multiplication

3.3 Molecular Dynamics of Lennard-Jones System

Parallel molecular dynamics of a Lennard-Jones system is a benchmark problem that has been extensively used by various researchers [13-15]. Molecular dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion. The force between two atoms is approximated by Lennard-Jones potential energy function $U(r)$, where r is the distance between two atoms. Using Newton's laws of equation and Velocity-Verlet algorithm, the velocities and coordinates of the atoms are updated for the next time step. The physics of the molecular dynamics problem is described in [13].

$$U(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^6 \right] \quad (1)$$

where, σ and ϵ are constants. The N atoms are simulated in a 3-D parallelepiped with periodic boundary conditions at the Lennard-Jones state point defined by the reduced density $\rho = 0.8442$ and reduced temperature $T = 0.72$ [13]. The simulation is begun with the atoms on an fcc lattice with random velocities and with a time step of 0.004 in reduced units. For simplicity reasons, there are no neighbor lists or cutoff limits.

There are three main classes of parallelization for classical molecular dynamics: atom, force and spatial decomposition. In this paper, a parallel algorithm based on force decomposition is tested on a standard Lennard-Jones benchmark for problem size ranging from 256 – 100,000 atoms. There are three variants of the problem: message-passing, one-sided blocking and one-sided non-blocking RMA. The RMA versions were implemented using Global Arrays that manages distributed arrays and ARMCI for all communication. Force decomposition is based on a block decomposition of the force matrix F distributed among processors, where each processor computes a fixed subset of inter-atomic forces. The entire force matrix ($N \times N$) is divided into multiple blocks ($m \times m$), where m is the block size and N is the total number of atoms. Each process owns N/P atoms, where P is the total number of processors. Newton's third law is exploited as it halves the amount of computation.

In the MPI implementation, the force matrix owned by each processor P_z is of size $(N/\sqrt{P}) \times (N/\sqrt{P})$. As these elements are computed they will be accumulated into the corresponding force sub-vectors and finally folded together to get the total forces on its N/P atoms [13]. In the RMA implementation of Lennard-Jones, the force matrix and atom coordinates are stored in a global array. A centralized task list is maintained which stores the information of the next block that needs to be computed. The issue of load imbalance is a serious concern for force decomposition MD algorithm. Processors will have equal work only if the force matrix distribution is regular and equally sparse. In order to address load imbalance, a simple and effective dynamic load balancing technique called fixed-size chunking is used [16]. Initially, all the processes get a block from the task list. Whenever a process finishes computing its block, it gets the next available block from the task list. Overlapping of computation and communication is achieved by issuing a get call to the next available block in the task list, while computing a block.

4 Experimental Results

The experiments were performed on the 2.4GHz Pentium-4 Linux cluster with Myrinet-2000 at the State University of New York at Buffalo. It employs the most recent versions of GM and MPICH-GM libraries provided by Myricom.

We ran our MG tests for class A (problem size: 256X256X256, iterations: 4), B (problem size: 256X256X256, iterations: 20) and C (problem size: 512X512X512, iterations: 20). They are three production grade problem sizes for the MG benchmark.

For Class A, a smaller problem size with fewest iterations, ARMCI blocking code outperforms the reference MPI implementation by 7 to 30%. ARMCI non-blocking version achieves an additional overlap of 10 to 23% over the ARMCI blocking implementation and 28 to 46% improvement over the reference MPI implementation. Most of the overlap achieved over the blocking implementation is just by mere issue of the update in the next dimension while working on the current one. For Class B, with the same problem size as class A but more iterations, ARMCI blocking implementation outperforms MPI by 10 to 37%, see Figure 3 (left). ARMCI non-blocking implementation achieves an additional overlap of 5 to 20% over the blocking version and shows a 30 to 45% improvement over the reference MPI implementation. For Class C, ARMCI blocking implementation outperforms MPI by 10 to 32%. ARMCI non-blocking implementation achieves an additional overlap of 2 to 21% over the

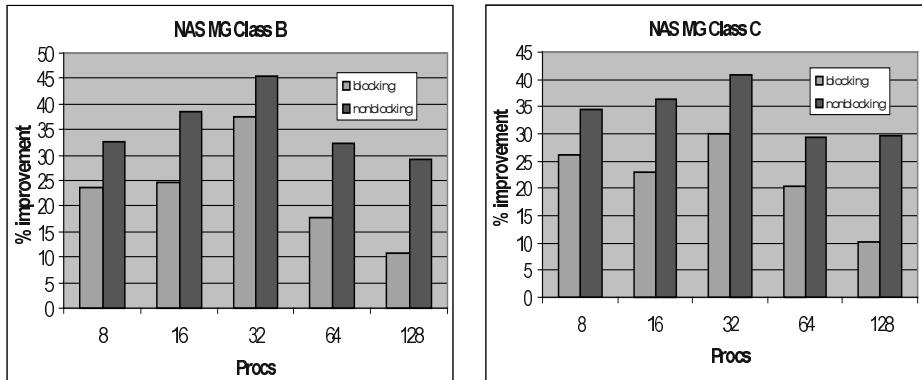


Fig. 3. Performance improvement in NAS MG for class B (left) and class C (right)

blocking implementation and shows a 30 to 40% improvement over MPI. Since coarser levels of multi-grid do not carry enough work to hide all the communication, an improvement achieved by using non-blocking over blocking API is limited for small processor configurations. With the increase of the number of processors for the problem size, the improvement is amplified.

Due to the synchronous nature of data transfers in the CG algorithm, the performance improvement over MPI, although consistent is rather limited, see Figure 4. As expected the main source of performance improvement is due to increased efficiency of RMA operations over the message passing (e.g., due to overheads associated with tag-matching, early message arrival etc that MPI must do). However, the non-blocking RMA offers an additional performance improvement. For example, for 128 processors it exceeds 10% over MPI.

Experiments with matrix multiplication were run by varying the matrix size and the number of processors, with one and two processes per node. The results show that the

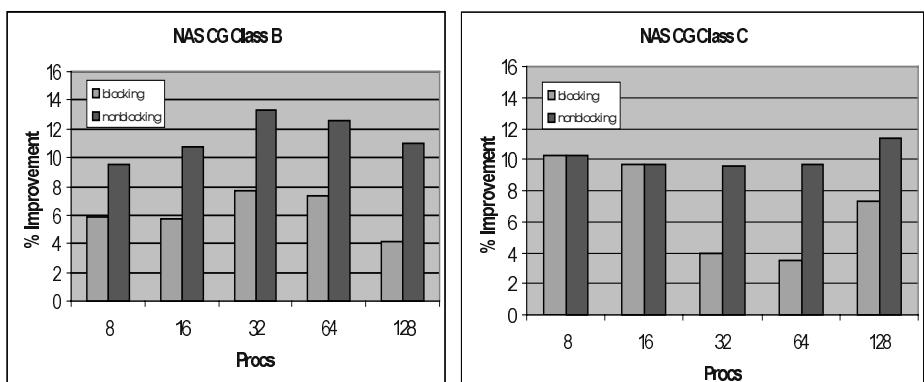


Fig. 4. Performance improvement in NAS CG class B (left) and class C (right)

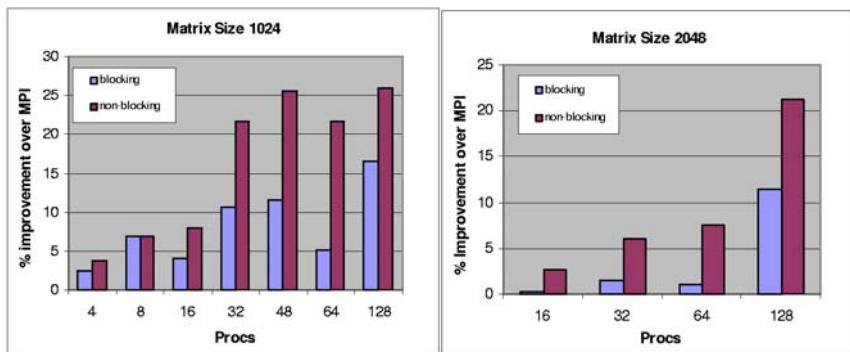


Fig. 5. Performance improvement in the matrix multiplication for matrix size 1024 (left) and 2048(right)

RMA-based matrix multiplication consistently outperforms its message-passing counterpart by 10-25%. For a matrix size of 1024, as the number of processors increases, the amount of local computation is less and hence lesser overlap. On the other hand, for a large matrix size (e.g. 2048), initially the computation cost is very high when compared to the communication cost. As the number of processors increases the impact of communication cost comes into picture. The graphs in Figure 5 indicate that using RMA communication in SUMMA resulted in improved application performance over message passing. This performance benefit is mainly due to the efficiency of the communication layer (in this case, ARMCI), which reduced the data transfer cost when compared to message passing. However, for a very large problem size, the effect of overlapping computation is not perceived due to very high computation cost.

The experimental results of the molecular dynamics benchmark indicate that using RMA resulted in improved application performance over message passing, see Figure 6. This benchmark problem scales well when the number of processors and/or the problem size is increased, thus proving the solution is cost-optimal. In some cases, the

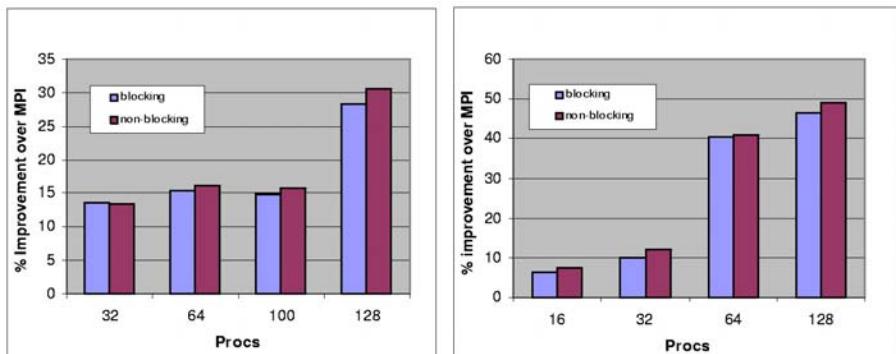


Fig. 6. Performance improvement in the molecular dynamics simulation involving 12000 (left) and 65536 (right) atoms.

performance improvement over MPI is greater than 40%. However, improvement in using non-blocking over blocking is not significant here as the potential for overlapping is limited in this benchmark problem.

5 Related Work

There have been multiple studies comparing effectiveness of different programming models to MPI [17-20]. For example, paper [18] studies MPI, SHMEM and shared memory in the context of adaptive applications dynamic remeshing and the n-body problem, all on a shared-memory machine. Another related paper [19] is comparing different parallel languages to MPI in the context of NAS MG parallel benchmark. In both of these studies, MPI was hard to outperform. Despite important merits of the other models (ease of use, reduced implementation complexity) none of them showed a consistent performance advantage over MPI across all the discussed benchmarks. In [20] several benchmarks were used to compare performance of the KeLP C++ runtime library to MPI. By exploiting SMP locality and non-blocking communication in the KeLP data mover to overlap communication with computations performance, improvement from 12 to 28% was measured on a DEC cluster.

6 Conclusions

This paper compared performance of MPI and RMA implementations of four scientific benchmarks: NAS MG, NAS CG, SUMMA matrix multiplication, and Lennard Jones molecular dynamics on clusters with the Myrinet network. Both blocking and non-blocking RMA versions of the benchmarks were studied. In all these benchmarks, RMA delivered a consistent performance advantage over MPI. In some cases an improvement as much as 50% was achieved. In parts of the algorithms where overlapping communication with computations is possible, non-blocking RMA provided an additional performance boost. The overall performance advantage of RMA over the send/receive model can be contributed to the fact that this approach can avoid the overheads associated with typical implementations of MPI such as management of message queues, tag matching, and dealing with early arrival of messages. In addition, since explicit cooperation with the remote data owner is not needed for the data transfer to complete, RMA offers a more asynchronous programming model than MPI. However, this approach usually requires a careful program design to assure that the remote data is in consistent state when it is being accessed by the RMA calls.

Acknowledgments. This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) and Ohio State University. PNNL is operated for DOE by Battelle. This work was supported by the Center for Programming Models for Scalable Parallel Computing project sponsored by the MICS/ASCR program in the DOE Office of Science.

References

1. R. Bariuso, Allan Knies, SHMEM's User's Guide., Cray Research,, SN-2516, 1994.
2. Glen R. Luecke, Silvia Spanoyannis, Marina Kraeva, Comparing the Performance and Scalability of SHMEM and MPI-2 One-Sided Routines on a SGI Origin 2000 and a Cray T3E-600, J. PEMCS, December 2002.
3. J. Nieplocha, V. Tipparaju, A. Saify, D. Panda, Protocols and Strategies for Optimizing Remote Memory Operations on Clusters, Proc. Communication Architecture for Clusters Workshop of IPDPS'02. 2002.
4. J. Nieplocha, V. Tipparaju, J. Ju, E. Apra, One-sided communication on Myrinet, Cluster Computing, 6, 115–124, 2003.
5. J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, Proc. RTSPP IPPS/SDP'99, 1999.
6. J. Nieplocha, RJ Harrison, and RJ Littlefield, Global Arrays: A portable `shared-memory' programming model for distributed memory computers. Proc. Supercomputing'94, 1994.
7. R. Numrich, J.K. Reid, Co-Array Fortran for parallel programming. ACM Fortran Forum, 17(2):1–31, 1998.
8. W. W. Carlson, J. M. Draper, D. E. Culler, K. Yellick, E. Brooks, and K. Warren. Introduction to UPC and language specification. Tech Report CCS-TR-99-157, Center for Computing Sciences, 1999.
9. K. Parzyszek, J. Nieplocha and R. Kendall, A Generalized Portable SHMEM Library for High Performance Computing, Proc PDCS-2000, 2000.
10. Myricom, The GM Message Passing System, 10/16/1999.
11. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, The NAS parallel benchmarks, Tech. Rep. RNR-94-007, NASA Ames Research Center, March 1994.
12. R. Van de Geijn and J. Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Concurrency: Practice and Experience, 9: 255–74, 1997.
13. S. J. Plimpton, “Fast Parallel Algorithms for Short-Range Molecular Dynamics”, J. Comp. Phys., 117:1–19, 1995.
14. S. J. Plimpton. Scalable Parallel Molecular Dynamics on MIMD supercomputers. In Proceedings of Scalable High Performance Computing Conference-92, 1992.
15. K. Esselink, B. Smit, and P. A. J. Hilbers. Efficient Parallel Implementation of Molecular Dynamics on a Toroidal Network: I. Parallelizing strategy. J. Comp. Phys., 106:101–107, 1993.
16. C. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. IEEE Transactions on Software Engineering, vol. SE-11, no. 10, 1985.
17. Robert W. Numrich, John Reid, and Kieun Kim. *Writing a multigrid solver using Co-array Fortran*. In Proceedings of the Fourth International Workshop on Applied Parallel Computing, Umea, Sweden, June 1998.
18. H. Shan, J. P. Singh, R. Biswas, and L. Oliker. A Comparison of Three Programming Models for Adaptive Applications on the Origin2000. Proc. SC'2000.
19. Bradford L. Chamberlain, Steven J. Deitz, Lawrence Snyder, A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. SC'2000.
20. Scott B. Baden and Stephen J. Fink, Communication overlap in multi-tier parallel algorithms, Conf. Proc. SC '98, Orlando FL, November 1998
21. Center for Programming Models for Scalable Parallel Computing, www.pmodels.org.

Scheduling Directed A-Cyclic Task Graphs on Heterogeneous Processors Using Task Duplication

Sanjeev Baskiyar and Christopher Dickinson

Department of Computer Science and Software Engineering
Auburn University, Auburn, AL
baskiyar@eng.auburn.edu
<http://www.eng.auburn.edu/users/baskiyar>

Abstract. In distributed computing, the schedule by which tasks are assigned to processors is critical to minimizing the finish time of the application. However, the problem of discovering the schedule that gives the minimum finish time is NP-Complete. By combining several innovative techniques, such as insertion-based scheduling and multiple task duplication, this paper presents a new heuristic called the Heterogeneous N-predecessor Decisive Path (HNDP), for statically scheduling directed a-cyclic weighted task graphs (DAGs) on a set of heterogeneous processors to minimize makespan. We compare the performance of HNDP, under a range of input conditions, with two of the best existing heuristics namely HEFT and STDS. We show that HNDP outperforms the two heuristics in finish time and the number of processors employed over a wide range of parameters.

1 Introduction

Proper scheduling of tasks is paramount to maximizing benefits of executing an application in a distributed computing environment. In this paper, the objective of scheduling tasks is to map them onto a set of processors such that the finish time is minimized, while observing precedence constraints. Since the scheduling problem is NP-Complete [3], much research has been done to discover heuristics that provide best approximations in a reasonable computational period.

This paper presents a new static list scheduling heuristic for the heterogeneous environment known as Heterogeneous N-Predecessor Duplication (HNPD). It is designed for a bounded number of processors but is also shown to work well for an unlimited number of processors. After assigning priorities according to the Decisive Path, originally presented in [13], each task, in order of priority, is assigned to the processor that completes the task the earliest. Then HNPD duplicate schedules the task's predecessors in the order of most to least favorite (defined in Section 2), on idle times of the processor, if it reduces makespan. It continues to recursively duplicate predecessors of any duplicated tasks until no more duplication can be performed.

This paper is organized as follows. Section 2 discusses background on scheduling tasks in a heterogeneous environment, including some definitions and parameters used by HNPD. Section 3 outlines related work. Section 4 defines the HNPD

heuristic. Section 5 presents the performance and simulation results. Section 6 has conclusions and suggestions for future work.

2 Problem Definition

The directed a-cyclic task graph (DAG) structure occurs frequently in many important applications. Its nodes represent tasks and edges data dependencies between tasks. In this paper, a DAG is represented by the tuple $G = (V, E, P, T, C)$, where V is the set of v nodes, E the set of e edges between the nodes, and P the set of p processors. $E(v_i, v_j)$ is an edge between nodes v_i and v_j . T is the set of costs $T(v_i, p_j)$, which represent the computation times of tasks v_i on processors p_j . C is the set of costs $C(v_i, v_j)$, which represent the communication cost associated with the edges $E(v_i, v_j)$. Since intra-processor communication is insignificant compared to inter-processor communication, $C(v_i, v_j)$ is considered to be zero if v_i and v_j execute on the same processor.

Node v_p is a *predecessor* of node v_i if there is a directed edge originating from v_p and ending at v_i . Likewise, node v_s is a *successor* of node v_i if there is a directed edge originating from v_i and ending at v_s . We can further define $\text{pred}(v_i)$ as the set of all predecessors of v_i and $\text{succ}(v_i)$ as the set of all successors of v_i . An *ancestor* of node v_i is any node v_p that is contained in $\text{pred}(v_i)$, or any node v_a that is also an ancestor of any node v_p contained in $\text{pred}(v_i)$.

The earliest execution start time of node v_i on processor p_j is represented as $EST(v_i, p_j)$. Likewise the earliest execution finish time of node v_i on processor p_j is represented as $EFT(v_i, p_j)$. $EST(v_i)$ and $EFT(v_i)$ represent the earliest start time upon any processor and the earliest finish time upon any processor, respectively. $T_{\text{Available}}[v_i, p_j]$ is defined as the earliest time that processor p_j will be available to begin executing task v_i . We can mathematically define these terms as follows.

$$\begin{aligned} EST(v_i, p_j) &= \max \{ T_{\text{Available}}[v_i, p_j], \max [(EFT(v_p, p_k) + C(v_p, v_i), k \neq j ; EFT(v_p, p_k), k = j), v_p \text{ in } \text{pred}(v_i), EFT(v_p, p_k) = EFT(v_p)] \} \\ EFT(v_i, p_j) &= T(v_i, p_j) + EST(v_i, p_j) \\ EST(v_i) &= \min (EST(v_i, p_j), p_j \in P) \\ EFT(v_i) &= \min (EFT(v_i, p_j), p_j \in P) \end{aligned}$$

The maximum clause finds the latest time that a predecessor's data will arrive at processor p_j . If the predecessor finishes earliest on a processor other than p_j , then communication cost must also be included in this time.

The *critical path* (*CP*) is the longest path (length is sum of computation and communication times) from an entry node to an exit node. The *critical path excluding communication cost* (*CPEC*) is the longest path from an entry node to an exit node, not including the communication cost of any edges traversed. For our problem, we assume that each task's mean execution cost across all processors is used to calculate

the *CP*. We also assume that each task's minimum execution cost from any processor is used to calculate the *CPEC*.

The *top distance* for a given node is the longest distance from an entry node to the node, excluding the computation cost of the node itself. The *bottom distance* for a given node is the longest distance from the node to an exit node, including the computation cost of the node. Again, we assume that each task's mean execution cost is used to calculate the *top* and *bottom distances*.

The *Decisive Path (DP)* of any node is defined as the sum of its *top* and *bottom distances*. The *CP* then becomes the largest *DP* for an exit node.

3 Related Work

In list scheduling (e.g. [4 - 8], [9 - 18]) tasks are ordered in a scheduling queue according to some priority. Then, in order, they are removed from the queue and scheduled on the optimal processor. Two main design points of the list scheduling heuristic are: how to build the scheduling queue and how to choose the optimal processor [1]. List scheduling algorithms have been shown to have good cost-performance trade-off [9].

There has been considerable work in scheduling DAGs to minimize finish time on homogeneous processors [2, 10, 13, 19], and the Dominant Sequence Clustering (DSC) [20] heuristic schedules any DAG yielding a finish time that is bounded within twice that of optimal for coarse grain graphs. Although there has been some work done (see below) on scheduling DAGs on heterogeneous multiprocessors systems, such a bound has not been established yet.

There are several algorithms for scheduling directed a-cyclic task graphs on heterogeneous multiprocessor systems namely Dynamic Level Scheduling (DLS) [21], Generalized Dynamic Level (GDL) [21], Best Imaginary Level (BIL) [12], Hybrid Re-mapper [11], Mapping Heuristic (MH) [17], Levelized Min Time (LMT) [17], Heterogeneous Earliest Finish Time, (HEFT) [18], Task Duplication Scheduling (TDS-1) [15-16], Task Duplication Scheduling (TDS-2) [9-10], Fast Critical Path (FCP) [14] and Fast Load Balancing (FLB) [15]. The performance and complexity of the algorithms have been summarized in Table 1, where v is the number of tasks in the directed a-cyclic task graph, e the number of edges and p the number of processors.

3.1 Decisive Path Scheduling (DPS)

The Decisive Path Scheduling (DPS) algorithm [13] has been shown to be very efficient for homogeneous case. HNPD leverages the benefits of DPS on the heterogeneous processor case. To build the scheduling queue, DPS first calculates the *top* and *bottom distance* from each node, the sum of which gives the *DP* for each node. The *top* and *bottom distances* are calculated as per Section 2, using the mean computation value for each node. After building the *DP* for each node, DPS begins creating the scheduling queue in a top-down fashion starting with the DAG's entry

Table 1. Comparison of complexity and schedule length of prominent heuristics.

<i>Algorithm, A</i>	<i>Complexity</i>	<i>Schedule length, L(A)</i>
BIL	$O(v^2 + plogp)$	$L(BIL) < L(GDL)$ by 20%
TDS-2	$O(v^2)$	$L(TDS-2) < L(BIL)$ for CCRs within 0.2 and 1
FLB	$O(vlogp + e)$	$L(HEFT) < L(FLB)$ by 63% when processor speed variance is high. Otherwise FLB performs equally well.
FCP	$O(vlogp + e)$	$L(HEFT) < L(FCP)$ by 32 % with high processor speed variance, otherwise identical.
HEFT	$O(v^2 p)$	HEFT better than DLS, MH, LMT by 8, 16, 52% respectively.

node and traversing down the *CP*. If all of a node's predecessors have been added to the scheduling queue, the node itself is added to the queue. If not, DPS attempts to schedule its predecessors in the same fashion. The first predecessors added to the queue are those that include the node in their *DP*. The time complexity of DPS is $O(v^2)$.

3.2 Selection of Heuristics for Comparison

From Table 1 it is clear that the HEFT, the Hybrid Remapper and STDS are the most important. We chose Heterogeneous Earliest Finish Time (HEFT) [17] and Scalable Task Duplication Scheduling (STDS) [15] to compare with HNPD. The Hybrid Remapper is basically a dynamic schedule to run after a static schedule has been obtained. We are interested in static scheduling in this work.

4 The HNPD Algorithm

4.1 Algorithm Description

The Heterogeneous N-Predecessor Duplication (HNPD) algorithm, combines the techniques of *insertion-based list scheduling* with *multiple task duplication* to minimize schedule length. The algorithm assigns tasks to the best available processor according to the order of tasks in a scheduling queue. The scheduling queue is populated in the order of *DP* construction [13].

In the order of tasks in the scheduling queue, HNPD uses $EFT(v_i)$ to select the processor for each task v_i . HNPD is an insertion-based algorithm; therefore it looks for a possible insertion between two already-scheduled tasks on the given processor without violating precedence relationships.

Once tasks have been assigned to processors, HNPD attempts to duplicate predecessors of the tasks. Tasks are ordered from most favorite to least and by descending *top distance*. The goal of duplicating predecessors is to decrease the length of time for which the node is awaiting data by making use of the processor's idle time. While execution of the duplicated predecessor may complete after its actual *EFT*, the avoided communication cost may make the duplication worthwhile.

If there is idle time between the recently assigned task v_i and the preceding task on the processor p_j , HNPD attempts to duplicate each predecessor v_p . If v_p is not already scheduled on processor p_j , it is duplicated if $EFT(v_p, p_j)$ is less than $EFT(v_p) + C(v_p, v_i)$. The duplication is retained if $EFT(v_p, p_j)$ decreases. Otherwise, it is discarded. The same duplication procedure is repeated for each predecessor in order of most favorite to least.

After HNPD attempts to duplicate each predecessor, it recursively attempts to duplicate the predecessors of any duplicated tasks. Thus, as many ancestors as allowable are duplicated, in a breadth-first fashion. Duplication recursively continues until no further duplication is possible.

4.2 Time Complexity

The time complexity of HNPD compares favorably with other heuristics. HEFT has a time complexity of $O(v^2 * p)$ [17]. STDS has a slightly better time complexity of $O(v^2)$ [15]. HNPD matches HEFT with a time complexity of $O(v^2 * p)$. DPS, used for ordering the scheduling queue, has a time complexity of $O(v^2)$ [13]. We can derive this complexity by considering the discovery of the *DP* for each node. Each node has to examine up to v nodes while building its *DP*. Therefore the time complexity of finding the *DP* is $O(v^2)$. When DPS builds the task queue, it must examine all edges in the DAG, so it is also bounded by $O(v^2)$.

5 Performance

This section presents the performance comparisons of HNPD against HEFT and STDS. Each algorithm was run against the same randomly generated set of DAGs. Results were then correlated to produce meaningful performance metrics.

The *Schedule Length Ratio (SLR)* is used as the main metric for comparisons. The *SLR* of a DAG is defined as the *makespan* divided by the *CPEC* (as defined in Section 2). Since *CPEC* considers the minimum execution time for each node and ignores inter-processor communication costs, it is the best possible *makespan*. Therefore, *SLR* can never be less than one. The algorithm that produces the lowest *SLR* is the best algorithm with respect to performance. The *improvement percentage* of HNPD over one of the heuristics is defined as the percent by which the original *SLR* is reduced. We can mathematically define the metrics as below:

$$SLR = \text{makespan} / \text{CPEC}$$

$$\text{Improvement}(\%) = (\text{Comparison_SLR} - \text{HNPD_SLR}) / \text{Comparison_SLR} * 100$$

5.1 Generating Random Task Graphs

Simulations were performed on a set of random DAGs on the three heuristics. The method used to generate random DAGs is similar to that presented in [17]. The following input parameters were used to create the DAG.

- Number of nodes in the graph, v .
- Shape parameter of the graph, α . The height of the DAG is randomly generated from a uniform distribution with mean value equal to $\alpha * \text{sqrt}(v)$. The width of the DAG at each level is randomly selected from a uniform distribution with mean equal to $\text{sqrt}(v) / \alpha$. If $\alpha = 1.0$, the DAG is balanced. Using $\alpha \ll 1.0$, a dense DAG (short DAG with high parallelism) can be generated. Similarly, $\alpha \gg 1.0$ will generate a sparse DAG (long DAG with low parallelism).
- Out degree of a node, out_degree . Each node's out degree is randomly generated from a uniform distribution with mean value equal to out_degree .
- Communication to computation ratio, CCR . If the DAG has a low CCR, it can be considered as a computation-intensive application; if CCR is high, it is communication-intensive.
- Average computation cost in the graph, avg_comp . Computation costs are generated randomly from a uniform distribution with mean value equal to avg_comp . Therefore, the average communication cost is calculated as $CCR * \text{avg_comp}$.
- Percentage variation of computation costs on processors, β . A high β value causes a wide variance between a node's computation cost across the processors. Let w be the average computation cost of v_i selected randomly from a uniform distribution with mean value equal to avg_comp . The computation cost of v_i on any processor p_j will then be randomly selected from the range $[w * (1 - \beta/2)]$ to $[w * (1 + \beta/2)]$.
- Processor availability factor, m . The number of available processors, p , is equal to $m * v$. A scheduling algorithm cannot use more processors than tasks; therefore m should never be larger than one.

A set of random DAGs was generated as the study test bed. The input parameters described above were varied with the following values.

- $v = \{10, 20, 40, 60, 80\}$
- $CCR = \{0.1, 0.5, 1.0, 5.0, 10.0\}$
- $\alpha = \{0.5, 1.0, 2.0\}$
- $\text{out_degree} = \{1, 2, 3, 4, 5, 100\}$
- $\beta = \{0.1, 0.25, 0.5, 0.75, 1.0\}$
- $m = \{0.25, 0.5, 1.0\}$
- $\text{avg_comp} = \{100\}$

These values produce 6,750 different combinations of parameters. Since we generated 25 random DAGs for each combination, the total number of DAGs in the study is around 168,750.

5.2 Results

The performances of HEFT, STDS and HNPD were compared using average *SLR* for different test sets of random DAGs. The test sets are created by combining results from DAGs with similar properties, such as the same number of nodes or same *CCR*. The results below show performance when attempting to duplicate all predecessors. In this section, we will present the test set combinations that provide the most meaningful insight into performance variances.

The first test set is achieved by combining DAGs with respect to number of nodes. The *SLR* value was averaged from 33,750 different DAGs with varying *CCR*, α , β , *out_degree* and *m* values, and *avg_comp* equal to 100. Fig. 1 shows the average *SLR* value with respect to number of nodes. The performance ranking is {HNPD, HEFT, STDS}. HNPD performs slightly better than HEFT at all points. The percentage improvement increases as the number of nodes increases, with HNPD outperforming HEFT by 6.4% for 10 nodes to around 7.9% for 80 nodes. HNPD performs much better than STDS, with the performance difference widening as the DAG size increases. HNPD outperforms STDS by 8.0% for 10 nodes up to 18.9% for 80 nodes.

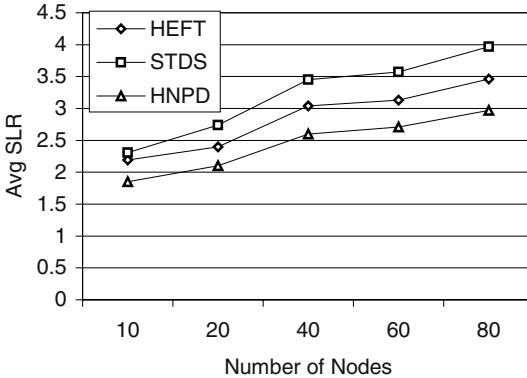


Fig. 1. Average SLR vs. Number of Nodes

The second test set combines DAGs with respect to *CCR*. Each *SLR* value for this set is averaged from 33,750 different DAGs with varying *v*, α , β , *out_degree* and *m* values, with *avg_comp* again equal to 100. In Fig. 2, average *SLR* is plotted with respect to varying *CCR*. Again, the performance ranking is {HNPD, HEFT, STDS}. For *CCR* values less than or equal to one (i.e. DAGs that are computation-intensive), HNPD and HEFT perform almost exactly even. However, as *CCR* becomes larger

than one (i.e. DAGs that are communication-intensive), HNPd clearly outperforms HEFT. For CCR values of 0.1, HEFT actually outperforms HNPd by 3.8%. For CCR value of 0.5, the difference is less than 1%. For CCR value of 1.0, HNPd outperforms HEFT by 2.8%. Then, the true savings begin to be achieved; for CCR values of 5.0 and 10.0, HNPd realizes savings of around 14.1% and 19.9%, respectively. HNPd outperforms STDS at each point with the performance difference largest for CCR values less than one. HNPd outperforms STDS by 19.8% for CCR equal to 0.1 down to 11.1% for CCR equal to 5.0. The savings returns to 19.0% for CCR equal to 10.0. This trend is the opposite of that displayed with HEFT.

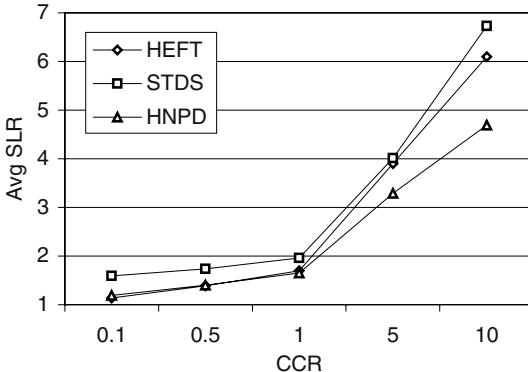


Fig. 2. Average SLR vs. CCR

5.3 Remarks

Interestingly enough, HNPd is outperformed only at the two extremes [7]. STDS performs close to HNPd (and outperforms HEFT) for unlimited processors and high CCR values. HEFT outperforms HNPd (and STDS) for a small set of processors and low CCR values. However, these scenarios are small enough that HNPd still performs better than STDS overall for an unlimited number of processors, and better than HEFT overall for a small set of processors.

6 Conclusions

We have proposed a new task-scheduling algorithm, Heterogeneous N-Predecessor Duplication (HNPd), for scheduling parallel applications on heterogeneous processors. By running a suite of tests on randomly generated DAGs, we showed that HNPd outperforms both HEFT and STDS in terms of scheduling efficiency. HNPd has been shown to be very consistent in its scheduling results, performing well with both an unlimited number of processors and a small set of processors. Further research may be done to extend HNPd so that it performs even better in the extreme cases, such as high CCR values on unlimited processors and low CCR values for the small set of processors.

References

1. Adam, T.L. et. al: A Comparison of List Schedules for Parallel Processing Systems. Communications of ACM. Vol. 17. No. 12. (1974) 685–690.
2. Baskiyar, S.: Scheduling DAGs on Message Passing m-Processors Systems. IEICE Transactions on Information and Systems. Vol. E-83-D. No. 7. Oxford University Press. (2000).
3. Baskiyar, S.: Scheduling Task In-Trees on Distributed Memory Systems. IEICE Transactions on Information and Systems. Vol. E 84-D. No. 6. (2001) 685–691.
4. O. Beaumont, V. Boudet, and Y. Robert: A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors. Proc. IPDPS. (2002).
5. Chiang, C., Lee C. and Chang, M.: A Dynamic Grouping Scheduling for Heterogeneous Internet-Centric Metacomputing System. ICPADS. (2001) 77–82.
6. Chan W.Y. and Li C.K.: Scheduling Tasks in DAG to Heterogeneous Processors System. Proc. 6th Euromicro Workshop on Parallel and Distributed Processing. Jan.1998.
7. Dickinson, C.: Scheduling Directed A-cyclic Task Graphs on a Bounded Set of Heterogeneous Processors Using Task Duplication. MSSwE Project Report. CSSE Dept. Auburn University. (2003).
8. Dogan and Ozguner, F.: Stochastic Scheduling of a Meta-task in Heterogeneous Distributed Computing. ICPP Workshop on Scheduling and Resource Management for Cluster Computing. (2001).
9. Kwok, Y. and Ahmad, I.: Benchmarking the Task Graph Scheduling Algorithms. Proc. IPDPS. (1998).
10. Kwok, Y., Ahmad, I. and Gu, J.: FAST: A Low-Complexity Algorithm for Efficient Scheduling of DAGs on Parallel Processors. Proc. ICPP. (1997).
11. Maheswaran, M. and Siegel, H.J.: A Dynamic Matching and Scheduling Algorithm for Heterogenous Computing Systems. Proc. 7th HCW. IEEE Press. 1998 (57–69).
12. Oh, H. and Ha, S.: A Static Scheduling Heuristic for Heterogeneous Processors. Euro-Par., V.2. (1996) 573–577.
13. Park, G. and Shirazi, B., Marquis, J. and Choo, H.: Decisive Path Scheduling: A New List Scheduling Method. Proc. ICPP. (1997).
14. Radulescu and van Gemund, A.J.C.: Fast and Effective Task Scheduling in Heterogeneous Systems. Proc. HCW. (2000) 229–238.
15. Ranaweera and Agrawal, D.P.: A Scalable Task Duplication Based Algorithm for Heterogeneous Systems. Proc. International Conference on Parallel Processing. 2000 (383–390).
16. Ranaweera and Agrawal, D.P.: A Task Duplication Based Algorithm for Heterogeneous Systems. Proc. IPDPS. (2000) 445–450.
17. Topcuoglu, H., Hariri, S. and Wu, M.-Y.: Task Scheduling Algorithms for Heterogeneous Processors. Proc HCW. (1999) 3–14.
18. Topcuoglu, H., Hariri, S. and Wu, M-Y.: Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing Parallel and Distributed Systems. IEEE Transactions on Parallel and Distributed Systems. V. 13. No. 3. (2002).
19. Sarkar,V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. The MIT Press. Cambridge MA. (1989).
20. Yang, T. and Gerasoulis, A.: DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors. IEEE TPDS. V. 5. No. 9. (1994).
21. Sih, G. and Lee, E. A Compile Time Scheduling Heuristic for Interconnection Constrained Heterogeneous Processor Architectures. IEEE TPDS. V. 4. No. 2. (1993) 175–187.

Double-Loop Feedback-Based Scheduling Approach for Distributed Real-Time Systems

Suzhen Lin and G. Manimaran

Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011, USA
`{linsz,gmani}@iastate.edu`

Abstract. The use of feedback control techniques has been gaining importance in real-time scheduling as a means to provide predictable performance in the face of uncertain workload. In this paper, we propose and analyze a feedback scheduling algorithm, called *double-loop feedback scheduler*, for distributed real-time systems, whose objective is to keep the deadline miss ratio near the desired value and achieve high CPU utilization. This objective is achieved by an integrated design of a local and a global feedback scheduler. We provide the stability analysis of the double-loop system. We also carry out extensive simulation studies to evaluate the performance and stability of the proposed double-loop scheduler. Our studies show that the proposed scheduler achieves high CPU utilization with low miss ratio and stays in steady state after a step change in workload, characterized by change in actual execution time of tasks.

1 Introduction and Motivation

In real-time systems, the correctness of the system depends not only on the logical correctness of the result, but also on the time at which the results are produced [1][2]. Traditional real-time scheduling algorithms are based on estimations of the (pessimistic) worst-case execution time (*WCET*) of tasks. However, this will result in an extremely underutilized system. In many cases, it is preferable to base scheduling decisions on a more correctly estimated execution time (*EET*) and to be ready to deal with changes in execution times dynamically. This approach is especially preferable as it provides a firm performance guarantee in terms of deadline misses while achieving high throughput.

One of the very successful areas in addressing performance in the presence of uncertainty is control theory. The feedback strategy is useful primarily if uncertainty is present and the advantages toward performance far outweigh the associated complexity in implementation.

In dynamic real-time systems, three steps are involved in task management: Schedulability check (admission control), Scheduling, and Task execution. The scheduler is responsible for performing the first two steps and processor(s) are responsible for executing the tasks. The first step decides if the tasks can be admitted into the system, and the scheduling algorithm decides where (which processor) and when to execute the admitted tasks. Since the system resources are limited, some tasks may be rejected during the schedulability check. Even after tasks are admitted to the system for execution, there may be still some tasks that miss their deadlines due to the uncertainty in execution

times. *Task rejection ratio (RR)* is defined as the ratio of the number of tasks rejected by the admission controller to the number of tasks arrived in the system. *Deadline miss ratio (MR)* is defined as the ratio of the number of tasks that miss their deadlines to the number of tasks admitted by the admission controller. Ideally, one would prefer a low *RR* if possible, low *MR*, and high *CPU utilization (U)* in an overloaded system. *U* can be defined as $\frac{CPU_{busystime}}{CPU_{busystime} + CPU_{idletime}}$ in a certain time interval.

When the scheduler performs the admission test and scheduling, it uses task parameters, such as execution time and deadline. Among the parameters, the (actual) execution time of a task depends on conditional statements and data dependent loops that are influenced by the dynamics of the environment in which the system is operating. Thus, task execution time creates workload uncertainty in the system. Consider image processing (includes digital filtering for noise reduction and edge detection) [1] for object moving on a conveyor belt. Task execution time may change due to environmental factors such as lighting condition and object orientation. For example, if the light of the environment becomes dark, the image processing will take longer time, since more work needs to be done for noise reduction and edge detection.

There are three ways to deal with the uncertainty in execution time: (1) Schedule tasks based on *WCET* obtained through pessimistic static analysis. This will result in high *RR* (also low *U*), but zero/less *MR*; (2) Schedule tasks based on best-case execution time (*BCET*) obtained through analysis based on optimistic assumptions. This will result in low *RR*, but may incur high *MR*. Thus, the task execution time introduces a trade-off between *RR* and *MR*. (3) *Schedule tasks based on an estimate of actual execution time (AET) obtained by using feedback control algorithm*. Unlike approaches (1) and (2), approach (3) has the potential to capture this trade-off in order to minimize both *RR* and *MR*, by making a good estimate of *AET*.

For convenience, we define *CPU utilization factor (UF)* as $(1 - \text{requested } CPU \text{ utilization})$. The *requested CPU utilization (U}_r\)* is defined as the summation of *AET* over the corresponding period of admitted tasks. Ideally, *UF* should be close to 0. In the feedback control context, in order to achieve low *MR* and high *U*, we monitor *MR* and *UF* and feed them back to the controller, and regulate the two performances to desired values by adjusting appropriate parameters like *EET*. The reason that we do not choose *RR* as the regulated variable is because it depends not only on the estimation of execution times, but also on the number of tasks.

Scheduling in distributed systems involves two main components: (1) local scheduling which schedules tasks on a given node; (2) global scheduling which migrates tasks to a lightly loaded node if the current node is overloaded [2].

Our objective is to develop a feedback-based scheduling algorithm for distributed real-time systems (DRTS), with the goal of achieving low *MR* and high *U*. We propose a double-loop feedback control-based scheduling algorithm of which the local controller (inner loop) is associated with the local scheduler and the global controller (outer loop) is associated with the global scheduler. The local controller will be used to estimate the execution time of tasks and the global controller will be used to adjust the set points (desired values of the output) of the local system as a means to facilitate load balancing.

The rest of the paper is organized as follows. In Section 2, we discuss the related work. Section 3 proposes the double-loop feedback scheduling architecture. Section 4 validates the result through simulation. Section 5 makes some concluding remarks.

2 Related Work

In [3][4], the authors present a feedback control EDF scheduling algorithm for real-time uniprocessor systems. In [5][6], authors present a closed-loop scheduling algorithm based on execution time estimation for multiprocessor systems. In [7], load balancing is achieved by monitoring load information of each host. [8] assumes that each task consists of n sequential replicable sub-tasks. The controller adjusts the number of the replicas of sub-tasks to achieve low MR and high resource utilizations when the system is overloaded. However, [3][4][5][6] are about local systems and [7][8] suffer from the scalability problem due to the global information collection. Our scheme achieves better scalability than [7][8] since each node gets information only from neighbors and migrates tasks to neighbors if overloaded. [9] proposed a two-level feedback control scheduling architecture. The parameter adjusted is service level. The stability analysis of the control system did not combine the two loops together. Our analysis combines the two loops together: at first, we analyze the stability of the local system, then we analyze the stability of the double-loop system. [8][9] assume special tasks type to adjust, this paper focuses on adjusting the estimated execution times to achieve high U and low MR during overloading, which is more suitable to many applications.

3 Proposed Double-Loop Feedback Scheduling Architecture

As we state in Section 1, in DRTS, each node is responsible for local scheduling and global scheduling. The double-loop scheduling architecture is shown in Figure 1. The various components and their corresponding algorithms or functions in the double-loop architecture are as follows:

- Local PID controller: PID control law is used.
 - Local actuator: changing the estimated execution factor.
 - Local scheduler: EDF, RMS, or Spring scheduling algorithm.
 - Processor: executing tasks.
 - MR , U and UF measurers: calculating the MR , U and UF . UF can be gained by calculating U_r . By measuring the tasks execution time in a past interval, we can calculate U_r .
 - Feedback information collector: collecting the performances and feeding back to local and global controllers.
 - Average Calculator: Calculating the average performances of the neighborhood.
 - Global PID controller: PID control law is used.
 - Global actuator: changing the set points for the local control system.
 - Global scheduler: deciding which tasks and where to migrate based on load index.
- We employ two controllers: *global controller (or distributed controller)* and *local controller* [9]. Each node in the distributed system has these two controllers. The global

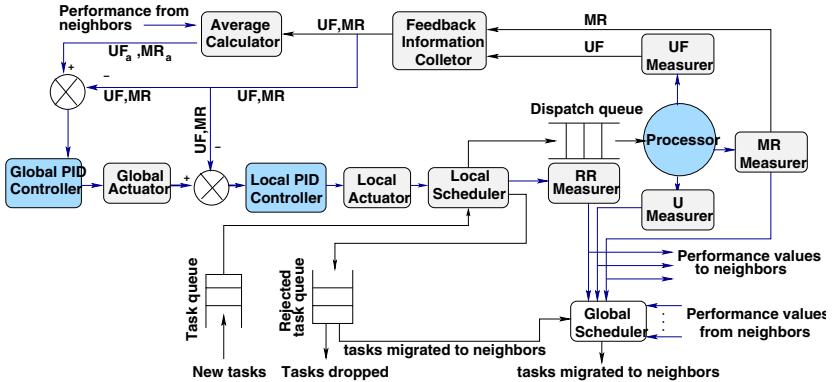


Fig. 1. Double-loop feedback scheduling architecture

controller gets information from its neighbors and itself, and then gives outputs according to the control law it uses. The outputs of the global controller are used as set points for the local controller. In other words, the local controller and the local system are treated as the controlled system, controlled by the global controller. The global controller cooperates with the local controller to adjust the performances of the local node in terms of deciding the number of tasks to be rejected by the local system. The rejected tasks that have not missed their deadlines may be migrated to other nodes. That is, in some sense, the global controller is responsible to achieve load balance in the distributed system.

3.1 Feedback Control

System Variables: In real-time scheduling, we choose MR and UF as regulated variables and measured variables, since these are the metrics indicating how well the system is behaving and resources are being used. The set points are desired values for MR and UF . We ignore the disturbance in our discussion. The control variable is the estimation factor for the task execution time, since the change of task execution time may lead to poor performance if the estimated execution time remains unchanged.

Control Law: The control law used in the controllers is PID [10]. PID stands for Proportional, Integral and Derivative. Controllers are used to adjust EET to hold UF and MR at the set point. The set point is desired value of the system output. Error (the controller input) is defined as the difference between set point and measurement. The output of a controller changes in response to a change in measurement or set point, since the error changes when measurement or set point changes.

Controller Algorithms: We consider PI controller for the local controller and PI controller for the global controller.

Local System: Assume the task set is $\{T_1, T_2, \dots, T_i\}$. EET_i , $AvCET_i$, and $BCET_i$ are the estimated execution time, the average case execution time ($AvCET$) and the best case execution time of task T_i respectively. The set points for the local controller are MR_{LS} and UF_{LS} . UF and MR of the node are fed back to the controller periodically for every time interval T . So UF and MR are collected at regular

intervals: $0, T, \dots, kT$. In our algorithm, we use an estimation factor (etf) to estimate the execution time of tasks, the estimation factor adjust the estimated execution time by increasing or decreasing the execution time from the average case execution time. At instance k , this adjustment is shown in Equation 1.

$$(EET_i)_k = AvCET_i + etf_k(AvCET_i - BCET_i) \quad (1)$$

etf_k is the estimation factor at time instance k . When the system starts to work, we set the etf_0 value to be 0, that is, we use the average case execution time to perform the admission test and schedule tasks. Then we get the feedback performances (MR_k and UF_k , which are MR and UF at time instance k respectively); according to these information, we get the amount of change for the estimation factor (Δetf_k) and then change the estimation factor. This is shown in Equation 2 and 3.

$$\Delta etf_k = K_m(MR_{LS} - MR_{k-1}) - K_u(UF_{LS} - UF_{k-1}) \quad (2)$$

$$etf_k = etf_{k-1} - \Delta etf_k \quad (3)$$

In Equation 2, let K_m and K_u be positive values. If MR_{k-1} is greater than MR_{LS} , this means the estimation for the execution time is too small, hence MR is high. We have to increase etf_k in order to make MR small. $K_m(MR_{LS} - MR_{k-1})$ contributes a negative part to Δetf_k . According to Equation 3, this will contribute a positive part to etf_k . Thus, the feedback information of MR_{k-1} will lead to the increase of etf_k , so as to decrease MR . Similarly, if UF_{k-1} is greater than UF_{LS} , this means the estimation for the execution time is too large, CPU will allocate too much time for each task and U_r is low. We have to decrease etf_k in order to make UF close to zero. $-K_u(UF_{LS} - UF_{k-1})$ contributes a positive part to Δetf_k . According to Equation 3, this will contribute a negative part to etf_k . Thus, the feedback information of UF_{k-1} will lead to the decrease of etf_k , so as to decrease RR . For the case that MR_{k-1} is less than MR_{LS} or UF_{k-1} is less than UF_{LS} , the analysis is the same.

Global System: Compared with the local controller, the global controller acts slowly. It gets information from its neighbors and computes the set points for the local controller. At every time instance, the global controller gets MR and UF from its neighbors and itself, it uses the averages of MR s and UF s as the global set points. Assume these two averages are MR_A and UF_A , then the outputs of the global controller are given by Equation 4.

$$\begin{bmatrix} MR_{LS} \\ UF_{LS} \end{bmatrix}_k = \begin{bmatrix} MR_{LS} \\ UF_{LS} \end{bmatrix}_{k-1} + \begin{bmatrix} K_{11} & -K_{12} \\ -K_{21} & K_{22} \end{bmatrix} \left(\begin{bmatrix} MR_A \\ UF_A \end{bmatrix}_{k-1} - \begin{bmatrix} MR \\ UF \end{bmatrix}_{k-1} \right) \quad (4)$$

MR_{LS} considers the influence not only from MR_A but also from UF_A , so does UF_{LS} . So we will let K_{11} , K_{12} , K_{21} and K_{22} be positive values. These values are coefficients, which can be determined experimentally. According to Equation 4, we have $MR_{LSk} = MR_{LS(k-1)} + K_{11}(MR_{A(k-1)} - MR_{k-1}) - K_{12}(UF_{A(k-1)} - UF_{k-1})$. Consider if the current node's MR is less than MR_A of the whole distributed system, then the current node needs to increase MR set point. The increased part is: $K_{11}(MR_{A(k-1)} - MR_{k-1})$. At the same time, in order to increase MR of the system,

the current node can decrease U_r , so that MR can be increased further. The increased part is: $-K_{12}(UF_{A(k-1)} - UF_{k-1})$. Since we only need to decrease UF when UF is greater than the average UF of the whole distributed system. So, this part is greater than zero in such situation. Other analysis is similar.

3.2 Stability Analysis

For discrete system, the sufficient and necessary condition for stability is that all the eigen values of the characteristic equation lie within the unit circle in Z plane (assuming no zeros and poles cancellation). Figure 2 is the block diagram for the local system.

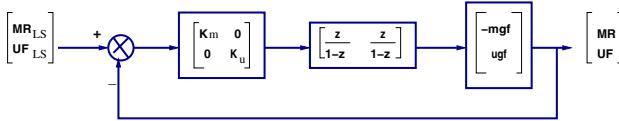


Fig. 2. Block diagram for the local system

The characteristic equation of the local control system is given in Equation 5. $G_L(z)$ and $G_K(z)$ are the transfer functions of the local system and controller respectively.

$$I + G_L(z)G_K(z) = 0 \quad (5)$$

Assume that mgf and ugf are the gains that map the estimated factor to MR and UF respectively. Then the transfer function of the local system is given in Equation 6.

$$G_L(z) = \begin{bmatrix} -mgf \\ ugf \end{bmatrix} \quad (6)$$

We can get the characteristic equation as shown in Equation 7.

$$\begin{bmatrix} 1 + \frac{z}{z-1} K_m mgf & \frac{z}{z-1} K_u mgf \\ \frac{z}{z-1} K_m ugf & 1 + \frac{z}{z-1} K_u ugf \end{bmatrix} = 0 \quad (7)$$

The eigen values are $z_{1,2} = 0$, $z_3 = \frac{1}{1+K_m mgf}$, $z_4 = \frac{1}{1+K_u ugf}$. Since $K_m mgf > 0$, $K_u ugf > 0$, all the eigen values lie within the unit circle. So, our local control scheduling system is stable.

In the double-loop control system, the inner loop will respond to changes much more quickly than the outer loop. So, when we consider the global controller, we can treat the local system as a model that has transfer function I (identity matrix). Then using PI controller, the analysis for the global controller will be the same as the local one. The block diagram for the double-loop system is shown in Figure 3.

Similarly, we can get the characteristic equation as shown in Equation 8.

$$\begin{bmatrix} 1 + \frac{z}{z-1} K_{11} & -\frac{z}{z-1} K_{12} \\ -\frac{z}{z-1} K_{21} & 1 + \frac{z}{z-1} K_{22} \end{bmatrix} = 0 \quad (8)$$

The eigen values are $z_{1,2} = 0$, $z_3 = \frac{1}{1+K_{11}}$, $z_4 = \frac{1}{1+K_{22}}$. Since $K_{11} > 0$, $K_{22} > 0$, all the eigen values lie within the unit circle. So, our double-loop control system is stable.

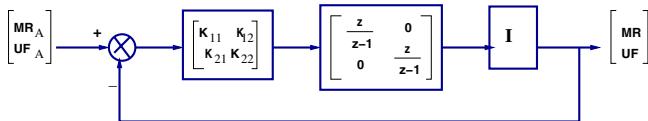


Fig. 3. Block diagram for the global system

3.3 Scheduling Algorithms

In DRTS, task scheduling involves local task scheduling and global task scheduling. For local task scheduling, since the adjustment of execution time of tasks are performed by controller and actuator, the scheduler can schedule tasks by using any real-time scheduling algorithms such as RMS (Rate Monotonic Scheduling) [11], EDF (Earliest Deadline First) [11], Spring Scheduling Algorithm [12], etc. The global scheduler needs to migrate tasks from overloaded nodes to underloaded nodes. A good load index should be defined to precisely characterize the system state.

Load Index: We consider U , RR and MR to form the load index, which will be used to guide task migration in the distributed system. U can tell us if the CPU of the node is used effectively. However, we also need to consider the incoming load, that is, the tasks submitted to the node, since these tasks will take the CPU time if they are admitted at the node. Thus, we consider two factors to form the load information. Besides, for nodes that have the same U and RR , if the computation times of tasks change, increase for example, then the load of the system will increase too. Since the increase of tasks' computation times will lead to the increase of not only U but also MR , we also need to consider MR . Thus large U , RR , or MR indicates high load of the node, and we can use the following linear combination of the three factors to get the load information L (load index) for each node:

$$L = \alpha U + \beta RR + \gamma MR \quad (9)$$

where α , β and γ are coefficients in the interval $[0,1]$ and $\alpha + \beta + \gamma = 1$. For example, if we set $\alpha = \frac{1}{3}$, $\beta = \frac{1}{3}$, and $\gamma = \frac{1}{3}$, this means we consider $\frac{1}{3}$ influence from each of U , RR , and MR . The choice of α , β and γ is application dependent.

Load Balancing: Each node maintains the load information of nodes in its neighborhood in load information table. When the load L is greater than a certain threshold value $L_{threshold}$ (overloaded), the node can migrate tasks to its neighbors based on the load information table. The load information is exchanged periodically. If some nodes have a large number of neighbors, we can only maintain $\min(d, h)$ neighbors' load information. d is the number of neighbors, and h denotes the computing capability of a node with respect to the least compute-capable node whose h value is 1. Each node can have the same or different h value. For example, node i has $h = 2$, and node j has $h = 4$, this means the computing capability of node i is twice of that of node j . If d is greater than h , the node can choose h neighbors arbitrarily to maintain the load information.

Now, the remaining problem is how to allocate tasks to the neighbors according to the load information table. Our main idea is to allocate tasks to the node's neighbors who are underloaded. Assume that S is the set of the current node's neighbors and whose

load index are less than $L_{threshold}$, and $i \in S$, we have Equation 10:

$$N_i = N \times \frac{L_{threshold} - L_{vi}}{|S| \times L_{threshold} - \sum_{j \in S} L_j} \quad (10)$$

where N is the total number of tasks to be migrated from an overloaded node of which N_i is the number of tasks migrated to node v_i , and $|S|$ is the number of nodes in S .

[9] proposed two logic network structures for task migration: hierarchical structure and neighborhood structure. The difference between these two structures is the way the global controller gives set points to local controller. The hierarchical structure does not scale well and the neighborhood structure scales better.

We notice that the two structures are logical network structures, which incur high complexity and overhead to maintain them. In contrast, our paper bases on the physical network structure, and each node contacts its neighbors to get MR , RR , and U . We do not need to maintain the logical structure, and hence less overhead. Besides, we allocate the number of migrated tasks according to neighbors' remaining ability, that is, neighbor whose load index is less than the threshold load index by a large value will get more migrated tasks than those whose load index is less than the threshold load index by a small value.

4 Simulation Studies

Since the local and global controllers work in different time scale, we study the performances of the local and global system separately. The simulation model is as follows:

- EDF algorithm is used for local scheduling.
- A task set with average $U_r > 1$ is generated at the beginning of every time interval T . Each task has a $WCET$, $BCET$, period, arrival time, and $AvCET$ with $AvCET = \frac{BCET + WCET}{2}$.
- The feedback is obtained for every T time units and is taken to be 200 in the simulation.
- To study the ability that the controller adapts the parameters, we use step load request in task execution time. We define a *load indicator*, L_{ind} , to indicate the load in terms of execution time of tasks. L_{ind} is equal to 1 when $AvCET$ is used. $L_{ind} < 1$ indicates underload, and $L_{ind} > 1$ indicates overload.
- We use UF and MR as set points. But due to the discrete property and the length of task execution times, the performances may not be exactly equal to the set points. So, we only require the performances to be close to the set points. We use $MR_{LS} = 0$ and $MR_{LS} = 0$ as set points values. The measured performances are UF and MR .

4.1 Simulation Studies – Local System

We study the performances of local system in two cases, one is when the task execution time increases and the other is when the execution time decreases. In our studies, we compare the performances of three approaches: (1) our feedback approach, (2) open-loop approach based on average computation time, and (3) open-loop approach based on worst-case computation time.

Task Execution Time Increases: In this case, at time instance 200 we give a step load request, AET of all tasks are increased by some value, that is, we let $L_{ind}=1.5$, then we observe the change of MR and UF of the system. Figure 4(a) shows the change of MR and U_r factor when the load indicator changes from 1 to 1.5 at time 200 for the three approaches. In the feedback approach, at time 400, the UF and MR measurers detect that $UF = -0.32$ and $MR = 0.48$. The controller calculates the error (difference between the set points and the feedback values) and begins to adjust the estimation factor to achieve good UF and MR values. From the figure, we see that at time 800, UF becomes stable, and the final value is 0.039. This means U is equal to $1 - 0.04 = 0.96$. MR becomes stable at time 600 and the final value is 0.0. This means no tasks miss deadline. Thus, we get a high U . In the open-loop approach based on the average computation time, UF stays at -0.32 and MR stays at 0.48 after the step load is applied. This means U is high (close to 1) since negative UF means the system is overloaded, but 48% tasks miss their deadlines. In the open-loop approach based on $WCET$, UF increases from 0.2 to 0.24 and stays at this value. MR stays at 0. This means U is 76%. Comparing the three approaches, we see that when the load increases, the feedback approach can achieve high U and low MR . The open-loop approach based on average computation time can achieve high U , but MR is also high. The open-loop approach based on $WCET$ can achieve low MR , but U is low.

Task Execution Time Decreases: In this case, we change L_{ind} from 1 to 0.7. Then we observe the change of MR and UF of the system. Figure 4(b) shows the change of MR and UF when the load indicator changes from 1 to 0.7 at time 200 for the three approaches. Comparing these three approaches, we see that when the load decreases, the feedback approach offers high U and low MR . The other two approaches offer low MR , but U is lower than that of feedback approach.

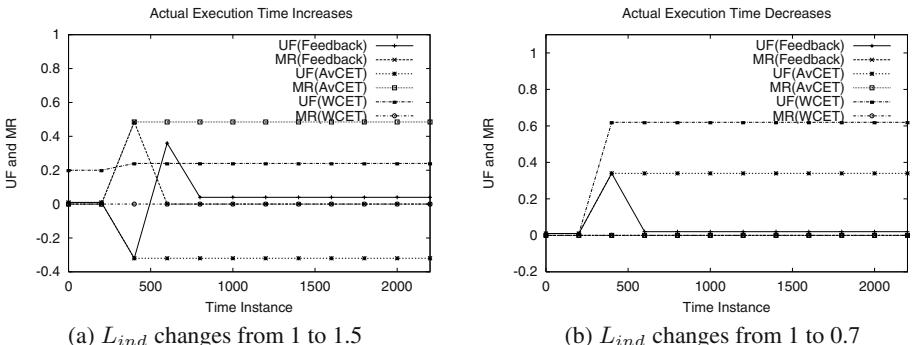


Fig. 4. UF and MR of local system

So the PI controller can adjust the etf of tasks according to the load change (computation time change) and maintain U and MR near the desired values.

4.2 Simulation Studies – Global System

Since the local system acts very quickly compared with the global controller. We can treat the transfer function of the local system as an Identity matrix, that is, the outputs of the local system are approximately equal to the set points from the global system.

Our simulation shows that the global controller can adjust MR and UF for the local system and achieves load balancing for the distributed system.

Task Execution Time Increases: Figure 5(a) shows the changes of UF and MR when AET of tasks of the local system increases at time 2000. This increase causes U_r to be higher than the new average U_r , that is, UF is less than the new average UF . The increase also causes MR to be higher than the new average MR . The global controller pulls MR and UF to new average values due to the feedback control.

Task Execution Time Decreases: Figure 5(b) shows the changes of UF and MR when AET of the local system decreases at time 2000. This decrease causes U_r to be lower than the new average U_r of the distributed system. This means UF is increased to be larger than the new average UF of the distributed system. The global controller pulls UF to new average value due to the feedback control.

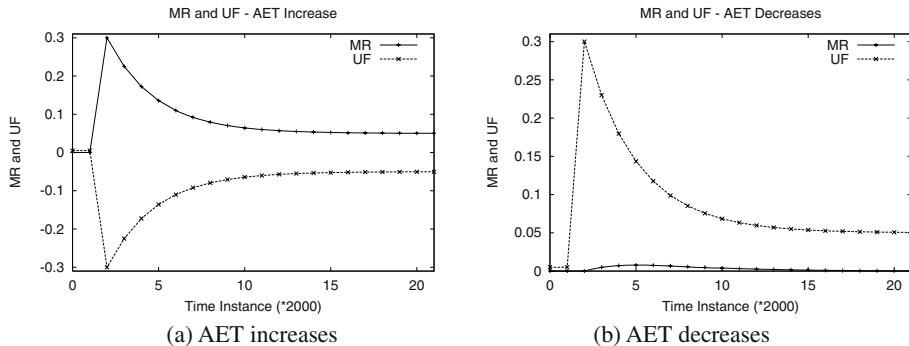


Fig. 5. MR and UF - Actual execution time of tasks changes

Therefore, the global controllers can adjust the set points of the local systems and let the systems become stable quickly by working with local controllers.

5 Conclusions

In this paper, a double-loop feedback scheduling approach is proposed for distributed real-time systems. By using feedback control theory, the inner loop feedback control system adjusts the estimated execution time of tasks to achieve desired U and MR for the local system, and the outer loop control system adjusts the set points for each local system. Then we analyzed the stability of the double-loop feedback-based scheduling system in Z domain. We also proposed a novel load index, which considers MR , RR , and U . The performance and stability of the proposed double-loop scheduler are evaluated through

simulation studies. Our studies show that good performances can be achieved. There are several possible avenues for further work in this emerging area of research, which includes the following: (1) feedback-based scheduling algorithm that takes into account network link load in addition to CPU load for global scheduling, and (2) feedback-based fault-tolerant scheduling algorithm that monitors system fault rate and performs redundancy management accordingly.

References

1. G. C. Buttazzo, "Hard Real-Time Computing Systems", *Kluwer Academic Publisher*, 1997.
2. C. Siva Ram Murthy and G. Manimaran, "Resource Management in Real-Time Systems and Networks", *MIT Press*, April 2001.
3. C. Lu, J. A. Stankovic, G. Tao, and S.H. Son, "Design and evaluation of feedback control EDF scheduling algorithm", In Proc. *IEEE RTSS*, pp. 56–67 1999.
4. J. A. Stankovic, Chenyang Lu, S. H. Son, and G. Tao "The case for feedback control real-time scheduling", In Proc. *Euromicro Conference on Real-Time Systems*, pp. 11–20, 1999.
5. D. R. Sahoo, S. Swaminathan, R. Al-Omari, M. V. Salapaka, G. Manimaran, and A. K. Soman, "Feedback control for real-time scheduling", In Proc. *American Controls Conference*, vol.2, pp. 1254–1259, 2002.
6. R. Al-Omari, G. Manimaran, M. V. Salapaka, and A. K. Soman, "New algorithms for open-loop and closed-loop scheduling of real-time tasks based on execution time estimation", In Proc. *IEEE IPDPS*, 2003.
7. D. R. Alexander,D. A. Lawrence and L. R. Welch, "Feedback control resource management using a posteriori workload characterizations", In Proc. *IEEE Conference on Decision and Control*, vol.3, pp. 2222–2227, 2000.
8. B. Ravindran, P. Kachroo, and T. Hegazy, "Adaptive resource management in asynchronous real-time distributed systems using feedback control functions", In Proc. *Intl. Symposium on Autonomous Decentralized Systems*, pp. 39–46, 2001.
9. J. A. Stankovic, T. He, T. F. Abdelzaher, M. Marley, G. Tao, S.H.Son, and C.Lu, "Feedback control scheduling in distributed systems", In Proc. *IEEE RTSS*, pp. 59–70, 2001.
10. K. Ogata, "Modern Control Engineering", *Prentice Hall*, Upper Saddle River, New Jersey, 2002.
11. C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment", *Journal of ACM*, vol.20, no.1, pp. 46–61, January 1973.
12. J. A. Stankovic and K. Ramararitham, "The Spring Kernel: a new paradigm for real-time systems", *IEEE Software*, vol.8, no.3, pp. 62–72, May 1991.

Combined Scheduling of Hard and Soft Real-Time Tasks in Multiprocessor Systems

B. Duwairi and G. Manimaran

Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50011, USA
`{dbasheer, gmani}@iastate.edu`

Abstract. Many complex real-time applications involve combined scheduling of hard and soft real-time tasks. In this paper, we propose a combined scheduling algorithm, called *Emergency Algorithm*, for multiprocessor real-time systems. The primary goal of the algorithm is to maximize the schedulability of soft tasks without jeopardizing the schedulability of hard tasks. The algorithm has the inherent feature of dropping already scheduled soft tasks by employing a technique that switches back and forth between two modes of operation: *regular mode*, in which all tasks (hard and soft) are considered for scheduling without distinction; *emergency mode*, in which one or more already scheduled soft tasks can be dropped in order to accommodate a hard task in the schedule. The proposed algorithm has been evaluated by comparing it with the iterative server rate adjustment algorithm (ISRA) [6]. The simulation results show that emergency algorithm outperforms ISRA by offering better schedulability for soft tasks while maintaining lower scheduling overhead.

1 Introduction

Many real-time applications involve combined scheduling of hard and soft real-time tasks. Hard real-time tasks have critical deadlines that are to be met in all working scenarios to avoid catastrophic consequences. In contrast, soft real-time tasks (e.g., multimedia tasks) are those whose deadlines are less critical such that missing the deadlines occasionally has minimal effect on the performance of the system.

In military applications, such as attack helicopters, multimedia information is being used to provide tracking and monitoring capabilities, that can be used directly to engage a threat and avoid crashing unexpectedly [6]. It is possible to allocate separate resources for each of hard and soft tasks. However, sharing the available resources among both types of tasks would have enormous economical and functional impact. Therefore, it is necessary to support combined scheduling of hard and soft real-time tasks in such systems, in which multiprocessors are increasingly being used to handle the compute-intensive applications.

Real-time tasks, beside being hard or soft, can be periodic or aperiodic. Therefore, in combined scheduling we may encounter the following task combinations: (1) periodic hard tasks with aperiodic soft tasks, (2) periodic hard tasks with periodic soft tasks, (3) aperiodic hard tasks with aperiodic soft tasks, or (4) aperiodic hard tasks with periodic soft tasks. Scheduling any of these task combinations in multiprocessor systems is a

challenging problem due to heterogeneous mix of tasks and dynamic nature of workload and multiple processors.

Maximizing the schedulability of soft tasks without jeopardizing the schedulability of hard tasks, and minimizing the scheduling overhead are among the most important design objectives that a combined scheduling algorithm attempts to achieve. It is obvious that achieving both objectives at the same time is not trivial as it requires consideration of many parameters and constraints.

Many researchers have investigated mechanisms to support combined scheduling of hard and soft real-time tasks. A number of scheduling approaches were adopted to handle a mix of hard and soft real-time tasks in uniprocessor systems. Examples are, Constant Bandwidth Server (CBS) [11] and [5], Total Bandwidth Server (TBS) [10] and [11], Constant Utilization Server [4], Deferrable Server (DS) [8], Priority Exchange (PE) [8], Extended Priority Exchange (EPE) [12], Dual priority Scheduling [3] and Dynamic Slack Stealing (DSS) [2]. Other scheduling schemes were presented in [1], and [7]. These approaches can be made applicable to multiprocessor systems by static allocation of tasks to processors. Obviously, the main problem of static allocation is that it is not able to reclaim unused time on one processor to schedule tasks allocated to another processor.

In this paper, we propose an algorithm for combined scheduling of hard and soft real-time tasks in multiprocessor systems. The primary goal of the proposed algorithm is to maximize the schedulability of soft tasks without jeopardizing the schedulability of hard tasks. The algorithm has the inherent feature of degrading QoS, by dropping already scheduled soft tasks, by employing a technique that switches back and forth between two modes of operation: regular mode in which all tasks (soft and hard) are considered for scheduling without distinction; emergency mode in which one or more already scheduled soft tasks can be dropped in order to accommodate a hard task in the schedule.

The rest of the paper is organized as follows: in section 2, we provide a description of the related work to this paper. The proposed algorithm is presented in section 3. Simulation studies are presented in section 4, followed by the conclusion in section 5.

2 Related Work

In multiprocessor systems, the multimedia server based combined scheduling algorithm [6] is the most relevant work. We first introduce the system model which is the same as used in [6], then we describe combined scheduling in the context of multimedia server.

2.1 System Model

The system model can be outlined as follows:

1. We consider multiprocessor system with (P) identical processors. The system has tasks with the following characteristics:
 - a) Aperiodic hard real-time tasks. Each task T_{ih} is characterized by its ready time (r_{ih}), worst-case computation time (c_{ih}), and absolute deadline (d_{ih}).

- b) n multimedia streams, each stream S_i is characterized by its period (P_i) and worst-case computation time (C_i).
- 2. All multimedia streams are mapped to one multimedia server, S_s , of period P_s and worst-case computation time C_s (multimedia server concept is described in subsection 2.2).
- 3. Each multimedia server instance is considered as a separate task. For example, the j^{th} server instance has ready time $(j - 1)P_s$, deadline jP_s , and computation time C_s .
- 4. No task migration or preemption is allowed.
- 5. The scheduling algorithm has complete knowledge about the currently active task set, but not about any new tasks that may arrive while scheduling the current set.

2.2 Multimedia Server [6]

The combined scheduling of aperiodic hard real-time tasks and multimedia streams in multiprocessor environment was addressed in [6] by introducing the concept of *Multimedia Server*. The server is a periodic task generated dynamically to accommodate multimedia streams and basically used for the purpose of reducing the total number of soft tasks considered for scheduling, because considering each multimedia task instance as a hard real-time task and scheduling it without having the multimedia server will increase the scheduling cost.

The creation of multimedia server can be done according to *proportional* or *individual* allocation schemes. For example, in proportional allocation, which is used in our experiments, the multimedia server is created as a periodic stream with period P_s equals to the smallest period of all multimedia streams in the system. Assuming that there are n different multimedia streams in the system, where each stream S_i is characterized by its period P_i and execution time C_i , then each stream instance is divided among P_i/P_s server instances such that the computation time of the multimedia server instance C_s is given by $\sum(C_i \times \frac{P_s}{P_i})$.

An incoming multimedia stream is initially mapped to an already existing multimedia server – otherwise a new server is created – to be scheduled with tasks waiting in the queue. If the whole task set is not schedulable, the incoming multimedia stream is rejected and removed from the multimedia server. The situation is different for incoming hard tasks; upon the arrival of a hard task, the scheduler tries to schedule this new task with all other already guaranteed tasks, hard and soft, if the hard task is found to be infeasible, either it is rejected directly – if hard tasks have no priority over already guaranteed tasks – or the QoS of already scheduled multimedia server must be degraded repeatedly until the new hard task become feasible. For this purpose, an **Iterative Server Rate Adjustment (ISRA)** algorithm is used to achieve the required QoS degradation. We refer the readers to [6] for more details about ISRA.

2.3 Motivation and Contribution

The multimedia server based scheduling strategy proposed in [6] performs QoS degradation outside the dynamic scheduler (i.e., using separate QoS degradation algorithm), which means that the scheduler has to be invoked several times to test the schedulability

of the task set after each time the QoS is degraded. In order to avoid all these complications included in quantifying the level of degradation, and to avoid the high scheduling overhead resulting from the fact of invoking the scheduler several times for the same task set, we propose an algorithm that is inherently capable of degrading QoS of multimedia server by dropping some of its instances, when necessary, while building the schedule (i.e., the scheduler is invoked only once for each task set).

3 Proposed Work

In this section, we describe the proposed emergency algorithm, which is a more sophisticated version of myopic algorithm [9]. The main difference is that our algorithm relies on a distinction between hard and soft real-time tasks when making the scheduling decisions in such a way that the schedulability of hard tasks is not affected by the existence of soft tasks. Therefore, by maintaining some additional information, the algorithm has the inherent feature for QoS degradation.

3.1 Main Idea

The emergency algorithm proposed in this paper, keeps track of the maximum *spare time* available on each processor. The value of this maximum spare time on processor P_j is defined as the difference between the earliest available time (EAT) of P_j in regular mode (EAT_{jr}) and that in emergency mode (EAT_{je}), where EAT of a processor is the earliest time at which the processor is ready for executing a task. EAT_{jr} and EAT_{je} are updated for each processor while building the schedule. The spare time of a certain processor represents the maximum amount of CPU time that can be reclaimed on that processor by dropping specific soft tasks. The dropping policy is explained below.

The algorithm is designed such that the spare time can be made available at the end of the schedule on each processor. When a new hard task is found to be infeasible (Task T_i is considered feasible if $EST_i + C_i \leq d_i$, where C_i is the worst-case execution time, d_i is the deadline, and EST_i is the earliest start time of task T_i . Otherwise, it is considered infeasible), the scheduler checks the information it keeps about already scheduled tasks to see if it is possible to accommodate the new hard task by reclaiming the spare time on one of the processors. A straightforward but expensive approach to do this is by initially dropping some of the already scheduled soft tasks on that processor and then reschedule the new hard task with the remaining already scheduled tasks.

In our algorithm, we follow a more computationally efficient approach to add the new hard task to the schedule. We start by selecting the processor that have enough spare time to make the new hard task feasible, then we reclaim that time as follows: the soft tasks that are scheduled after the last hard task on that processor are dropped directly. If their total computation time is equal to the spare time, the new hard task is added on that processor. Otherwise, the last hard task is shifted back by an amount of time equals to spare time minus total computation time of soft tasks dropped in the previous step. This may result in overlapping between the shifted task and other hard tasks. To resolve this overlapping, the next hard task is also shifted back such that it will have a finish

time equals to the start time of the previously shifted task. We keep doing this shifting process until there is no overlapping between hard tasks.

Any soft task that overlaps with the shifted hard tasks is dropped directly and its computation time is decremented from the spare time. After reclaiming the total spare time, the new hard task is added to the end of the schedule on that processor. We emphasize here that no task is shifted beyond its ready time because this was taken care of originally when the spare time is updated on each processor. It is also important to notice that the order of execution of the tasks scheduled on the selected processor remains the same and only some soft tasks are dropped on the way of shifting the hard tasks back.

3.2 Algorithm Details

Similar to myopic [9], emergency algorithm is a heuristic search algorithm that tries to construct a full feasible schedule by extending a partial schedule by one task at a time from window of k tasks (called the feasibility check window). The algorithm (shown on next page), starts its operation in the regular mode and stays there as long as all hard tasks encountered so far are feasible (step 2). For simplicity and convenience, the algorithm drops any infeasible soft task directly rather than backtracking or considering other options (step 2a). Among the feasible tasks considered in the feasibility check window, the algorithm always selects the one with the minimum heuristic value (the heuristic function for task T_i is given by $H_i = d_i + EST_i$). The selected task is scheduled on processor P_j that has the minimum earliest available time (EAT) to extend the current partial schedule (step 2b).

We mentioned earlier that this algorithm keeps track of resources reserved for soft tasks as they get scheduled such that they can be reclaimed if necessary. This can be achieved by building an array side by side with the schedule to keep the information about resources allocated for soft tasks. We adopted a simpler approach by maintaining two values for the minimum available time of each processor; the value of EAT_{jr} defines the earliest available time of processor P_j in regular mode, while the value EAT_{je} defines the earliest available time of processor P_j in emergency mode.

The difference between the two values defines the maximum spare time available on processor P_j . EAT_{jr} is always updated as follows: $EAT_{jr} = EAT_i + C_i$ for a selected task T_i . On the other hand, EAT_{je} is updated differently for hard and soft tasks: if the selected task T_i is hard, then $EAT_{je} = EAT_i + C_i$. EAT_{je} remains the same. In fact, the value of EAT_{je} tells the scheduler about the minimum available time of processor P_j if a specific soft tasks scheduled on it are to be dropped. This is true because the computation time allocated for these tasks on that processor is not included in this value. It is important to realize that task feasibility check is done based on EAT_{jr} in the regular mode.

The algorithm switches to emergency mode if at least one hard task in the feasibility check window is found infeasible (step 2). In this mode, task feasibility is checked based on the value of EAT_{je} of each processor. If all hard tasks in the feasibility check window are feasible (step 2a) we choose to extend the current partial schedule by the hard task with the minimum heuristic value on the processor P_j that provides the minimum EAT_{je} while dropping any infeasible soft task in the window (step 2a-i). To accommodate the new hard task on the selected processor P_j we defer tasks

already scheduled on that processor back to their start times to obtain the EAT_{je} of that processor (step 2a-ii). This requires the scheduler to drop one or more of the already scheduled soft tasks on that particular processor but not on any other processor. The worst situation occurs when at least one hard task is infeasible in the emergency mode (step 2b), in such case backtracking is required.

Emergency Algorithm ()

1. Tasks in the task queue are ordered in non-decreasing order of deadline.
2. if (all hard tasks in the feasibility check window are feasible) // Regular mode.
 - a) Drop any infeasible soft tasks from the window.
 - b) Extend the current partial schedule by the task with minimum heuristic value.
 - c) Update the earliest available times (EAT_{jr} and EAT_{je}) of the selected processor P_j accordingly.
3. else // At least one hard task in the feasibility check window is infeasible.
 - a) Switch to Emergency mode.
 - b) if (all hard tasks are feasible)
 - i. Drop any infeasible soft tasks from the window.
 - ii. Defer tasks scheduled on processor P_j (the one with minimum emergency earliest available time EAT_{je}). This forces us to drop one or more soft tasks that already scheduled on P_j to make a room for the new selected hard task.
 - iii. Extend the current partial schedule by the hard task with minimum heuristic value on processor P_j .
 - iv. Update the earliest available times (EAT_{jr} and EAT_{je}) of the selected processor P_j accordingly.
 - c) else // At least one hard task is infeasible.
 - i. Backtrack to the search level at which last hard task is scheduled.
 - ii. Extend the schedule with the hard task having the next best heuristic value.
4. Repeat steps (2- 4) until a feasible or hard feasible schedule is obtained.

It is to be noted that the correctness of emergency algorithm can be verified by following the algorithm steps one by one.

3.3 Complexity Analysis

It is easy to see that emergency algorithm requires $(n + m)$ steps to construct a schedule for n hard tasks and m soft tasks. In the worst case scenario, the algorithm will switch from regular mode to emergency mode each time it encounters a hard task. By recalling that the algorithm checks the feasibility and evaluates heuristic functions for at most k (the feasibility check window size) tasks in each mode at each step, the complexity of emergency algorithm can be expressed as $O(k(n + m)) + \text{number of backtracks}$. In this algorithm, we limit the number of backtracks to avoid exponential backtracking time.

A single invocation of myopic algorithm to schedule the same set of tasks takes $O(k(n + m)) + \text{backtracking time}$. Using myopic algorithm in the context of multilevel QoS degradation algorithm requires i invocations of myopic algorithm to schedule the same task set. Therefore, such QoS degradation algorithm takes $O(ik(n+m)) + i(\text{number of backtracks})$, which is higher than that required by emergency algorithm whenever a QoS degradation is required.

4 Simulation Studies

In our simulation experiments, we compare the proposed emergency algorithm with the Iterative Server Rate Adjustment (ISRA) algorithm described in [6] based on the following two metrics:

1. The multimedia server utilization (MSU) defined as the amount of time allocated to multimedia server to the total amount of time requested originally by the multimedia server during the scheduling interval.
2. The scheduling overhead ratio (OHR) defined as the number of scheduling steps performed by emergency algorithm to that performed by ISRA algorithm to schedule the same task set.

In the following subsections we describe load generation process, and simulation results.

4.1 Load Generation

We adopted a method similar to that presented in [6] for load generation. To generate a set of hard real-time tasks, a two dimensional matrix with time as one dimension and processors as the other dimension is used to generate a task set in such away a schedule of tasks is created by arranging these tasks in the matrix one after another. A task is generated by selecting one of the processors with the earliest available time. The generated task is assigned a ready time equals to that earliest available time and a computation time chosen randomly between two input parameters (MinC, MaxC). Also, it is assigned a deadline that is chosen randomly between two other input parameters (MinD, MaxD). The earliest available time of the selected processor becomes equal to the finish time of the generated task. This process continues until the remaining unused time for each processor, up to L (the schedule length), is smaller than the minimum processing time of a task, which means no more tasks can be generated to use the processors.

Multimedia streams used in our experiments are generated according to the following parameters:

- Baseline multimedia computation time, C_b .
- Baseline multimedia period, P_b .
- Period increase ratio, r .
- Number of multimedia streams, m .

The baseline multimedia stream period P_b is the smallest period of all generated streams. The periods of other streams are made larger than P_b by multiples of the period increase ratio r . That is, for any stream S_i , the period $P_i = (1 + (i - 1)r)P_b$. Also, the baseline multimedia stream is assigned a computation time equal to C_b , while other streams are assigned computation time larger than C_b by multiples of the period increase ratio r . That is, for any stream S_i , the computation time $C_i = (1 + (i - 1)r)C_b$. By this method, we can generate any number of streams with different periods and different computation times, such that all streams are related to one baseline stream. This enables us to study the effect of multimedia load by only changing the parameters of the baseline stream.

Load generation was done according to the default parameters shown in Table 4.1 unless otherwise specified, where H-Parameters refers to hard tasks generation parameters, and S-Parameters refers to soft tasks (i.e., multimedia streams) generation parameters. Each of the results presented in the following subsection represents the average of four simulation runs. Each simulation run involves 400 experiments (i.e., 400 different tasks sets were generated for each run). Throughout our experiments we used a system composed of 3 processors and we fixed the schedule length parameter (L) to 400. This corresponds to an average number of 60 hard tasks in each task set and about 13 multimedia server instances. Therefore, an average of 73 tasks were generated for each experiment.

Table 4.1. Simulation parameters

$H - Parameters$	Value	$S - Parameters$	Value
MinC	10	P_b	33
MaxC	30	C_b	2.5
MinD	60	r	0.2
MaxD	90	m	5

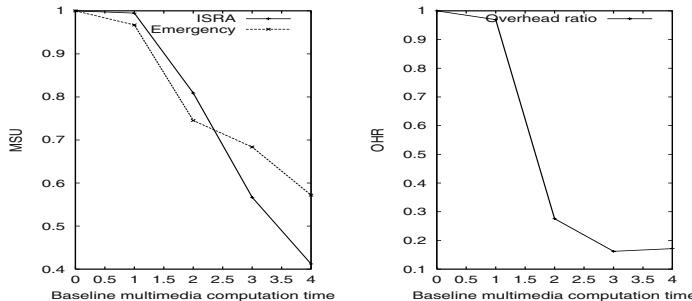


Fig. 1. Effect of baseline multimedia computation time on MSU and OHR

4.2 Simulation Results

Effect of Baseline Multimedia Computation Time: Fig.1 shows the effect of the baseline multimedia computation time, which reflects the load imposed by the multimedia streams on the system, on the multimedia server utilization (MSU) as well as on overhead ratio (OHR) for the same set of experiments. As expected, the value of MSU keeps decreasing by increasing C_b . When C_b is less than 1.5, both algorithms offer an MSU greater than 0.8. But after that, MSU drops to about 0.5. This can be explained by recalling that the main objective of both algorithms is to guarantee the schedulability of hard tasks. As the load of multimedia streams increases (represented by increasing C_b) more service degradation is required and therefore the fraction of time allocated to multimedia streams keeps decreasing.

At the same time, OHR curve suggests that the two algorithms incur almost the same amount of scheduling overhead when the multimedia load is light. However, by increasing the multimedia load, OHR decreases substantially because the number of rescheduling attempts made by ISRA increases to achieve the necessary QoS degradation, which emphasizes that emergency algorithm has much lower scheduling overhead than ISRA.

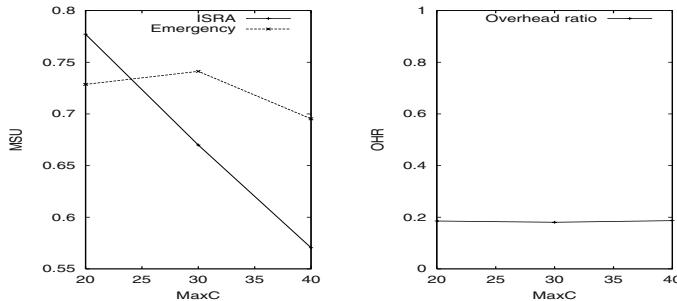


Fig. 2. Effect of average worst case execution time of hard real-time tasks on MSU and OHR

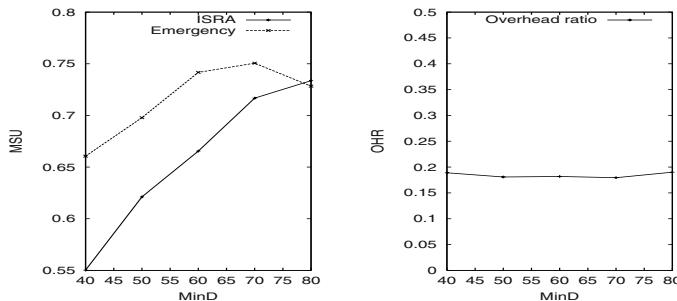


Fig. 3. Effect of average laxity of hard real-time tasks on MSU and OHR

Effect of Average Worst Case Execution Time of Hard Real-Time Tasks: Fig.2 illustrates the effect of changing parameter (MaxC) from 20 to 40 in steps of 10, while fixing MinC to 10. By increasing MaxC, the worst case computation time of the generated hard tasks increases as well. Therefore, it becomes more difficult to fit such hard tasks in the schedule without degrading the service of soft tasks either by dropping some of them as in emergency algorithm or by iterative adjustments of the server utilization as in ISRA. This explains the continuous decrease of MSU observed by both algorithms. However, the rate of decrease obtained by ISRA is higher than that obtained by emergency algorithm. This supports our theory about inherent QoS degradation by selective task dropping. For the same experiments, OHR remains almost constant.

For the same experiments, OHR remains almost constant. This can be interpreted by looking at the results of previous experiment when C_b was equal to 2.5. OHR value reported at that point is about 0.2 which is almost the same value obtained in this experiment. This observation suggests that rate of increase of scheduling overhead of both algorithms is relatively the same, and because of that OHR remains constant.

Effect of Average Laxity of Hard Real-Time Tasks: The effect of laxity (the maximum amount of time a task can wait before it can start its execution) of hard tasks can be captured by adjusting MinD and MaxD. Fig.3 shows our results for this experiment. Increasing MinD results in implicit increase in the laxity of hard tasks, and therefore, such tasks become more relaxed. As a result, soft tasks can be given more chances. This explains the continuous increase of MSU by both algorithms. For the same reasons explained in the previous experiment, OHR remains almost constant (about 0.2) in this experiment which also means that more scheduling steps are required by ISRA than required by emergency algorithm.

5 Conclusions

In this paper, we addressed the issue of combined scheduling of hard and soft real-time tasks in multiprocessor systems with a primary objective of maximizing the schedulability of soft tasks without jeopardizing the schedulability of hard tasks. For this objective, we proposed the emergency algorithm which has the inherent feature of QoS degradation of soft tasks. This feature removes the burden of repetitive QoS degradation and rescheduling, and therefore, avoids high scheduling overhead. Our algorithm was evaluated by comparing it to Iterative Server Rate Adjustment (ISRA) algorithm proposed in [6]. Simulation results show that the emergency algorithm offers higher schedulability for soft tasks, in most cases, and has much lower scheduling overhead compared to ISRA. Future work includes extending the proposed algorithm to allow task migration and preemption such that higher schedulability can be obtained.

References

1. G. Buttazzo and F. Sensini. Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments. *IEEE Transactions on Computers*. vol. 48, no. 10, pp. 1035–1051, October 1999.
2. R. Davis, K. W. Tindell, and A. Burns. Scheduling Slack Time in Fixed Priority Pre-emptive Systems. *Proceedings of the 14th Real Time System Symposium*, pp. 222–231, North Carolina, USA, December 1993.
3. R. Davis and A. Wellings. Dual Priority Scheduling. *Proceedings of Real-Time Systems Symposium*, pp. 100–109, 1995.
4. Z. Deng, J. W. S. Liu, and J. Sun. A Scheme for Scheduling Hard Real-Time Applications in Open System Environment. In *Proceedings of the 9th Euromico Workshop on Real-Time Systems*, 1997.
5. T.M. Ghazalie and T. Baker. Aperiodic Servers in a Deadline Scheduling Environment. *Real-Time Systems*, 9, 1995.
6. H. Kaneko, J.A. Stankovic, S. Sen, and K. Ramamritham. Integrated Scheduling of Multimedia and Hard Real-Time Tasks. *Proceedings of Real-Time Systems Symposium*, pp. 206–217, 1996.

7. S. Lee, H. Kim and J. Lee. A Soft Aperiodic Task Scheduling in Dynamic-Priority Systems. *Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, IEEE 1995.
8. J.P. Lehoczky, L. Sha, and J.k. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *Proceedings of Real-Time Systems Symposium*, pp. 261–270, 1987.
9. K. Ramamritham, J.A. Stankovic, and P.-F. Shian. Efficient Scheduling Algorithm for Real-Time Multiprocessor systems *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pp. 184–194, April 1990.
10. M. Spuri and G.C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. *Proceedings of IEEE Real-Time Systems Symposium*, San Juan, Puerto Rico, December 1994.
11. M. Spuri and G.C. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, vol. 10, no. 2, 1996.
12. B.Sprunt, J.Lehoczky, and L. Sha. Exploiting Unused Periodic Time for Aperiodic Service Using the Extended Priority Exchange Algorithm. *Proceedings of the 9th IEEE Real-Time Systems Symposium*, December 1988.

An Efficient Algorithm to Compute Delay Set in SPMD Programs

Manish P. Kurhekar¹, Rajkishore Barik², and Umesh Kumar³

¹ IBM India Research Lab,
Indian Institute of Technology Delhi, India
mkurhekar@in.ibm.com

² Institute of Computer Systems, ETH Zentrum, CH 8092, Zurich
barik@inf.ethz.ch, <http://www.inf.ethz.ch/~barikr>

³ Indian Institute of Technology Delhi, India
umesh@cse.iitd.ernet.in

Abstract. We present compiler analysis for single program multiple data (SPMD) programs that communicate through shared address space. The choice of memory consistency model is sequential consistency as defined by Lamport[9]. Previous research has shown that these programs require cycle detection to perform any kind of code re-ordering either at hardware or software. So far, the best known cycle detection algorithm for SPMD programs has been given by Krishnamurthy et al[5,6,8]. Their algorithm computes a *delay set* that is composed of those memory access pairs that if re-ordered either by hardware or software may cause violation of sequential consistency. This delay set is computed in $\mathcal{O}(m^3)$ time where m is the number of *read/write* accesses. In this paper, we present $\mathcal{O}(m^2)$ algorithm for computing analogous delay set for SPMD programs that are used in practice. These programs must be structured with the property that all the variables are initialized before their value is read.

1 Introduction

According to Lamport's definition[9], a multiprocessor system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. There are two aspects to the above definition: (1) maintaining program order among operations from individual processors, and (2) maintaining a single sequential order among operations from all processors. The second aspect talks about the *atomic* execution of a memory operation with respect to other memory operations. Let us consider the code segment shown in Figure 1a. It depicts an implementation of Dekker's algorithm for critical sections, involving two processors (P_1 and P_2) and two boolean variables ($flag_1$ and $flag_2$) that are initialized to 0. When P_1 attempts to enter the critical section, it updates $flag_1$ to 1, and checks the value of $flag_2$. The value of 0 to $flag_2$ indicates that P_2

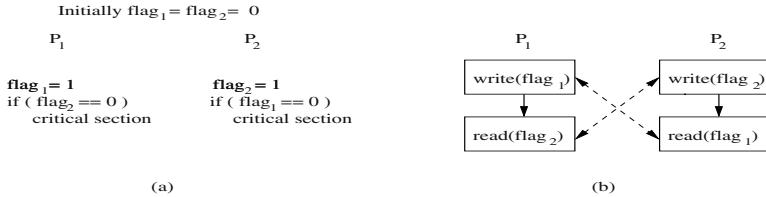


Fig. 1. Example of sequential consistency

has not yet tried to enter the critical section. Figure 1b shows the same program in terms of reads and writes of the program variables including the control flow edges. The bi-directional dotted lines in Figure 1b indicate the access conflicts of the same variable. It can be seen that if P_2 's read of $flag_1$ and P_1 's read of $flag_2$ both return a value of 0, then both the processors will enter the critical section and there will be a violation of sequential consistency. This is possible if any hardware or software reordering among the reads and writes is done without proper checking of violation of sequential consistency. A sequential compiler (typical single processor compiler) for instance might consider write to $flag_1$ and read of $flag_2$ in P_1 as potential target for reordering and as a result, end up violating sequential consistency.

If only the local dependencies within a processor are observed, then the program execution might not be sequentially consistent. To guarantee sequential consistency under reordering, an analysis called *cycle detection*[11,5,7,6,8] is required. The cycle detection problem is to detect access cycles such as one in Figure 1b (cycle comprising of $write(flag_1) \rightarrow read(flag_2) \rightarrow write(flag_2) \rightarrow read(flag_1) \rightarrow write(flag_1)$).

Cycle detection for multiple instruction multiple data (MIMD) programs was first described by Shasha et al[11] and later extended by Midkiff et al[10] to handle array indices. Their formulation gives an algorithm that is exponential in the number of processors and requires n copies of the program, where n is the number of processors.

Later on, Krishnamurthy et al[5,6] proved that the cycle detection problem for MIMD programs is NP-hard and proposed an $\mathcal{O}(m^3)$ algorithm for the restricted version of the problem arising in single program multiple data (SPMD) programs, where m is the number of memory accesses in the program. Their algorithm considers two copies of the program for the analysis.

The primary contribution of this paper is an efficient algorithm for detecting cycles in SPMD programs which are used in practice. Such programs are structured and we assume that before a value of a variable is read, the variable is initialized. We require only one copy of the program for our analysis. We provide a theoretical proof of the equivalence of our algorithm to that of Krishnamurthy et al's. Our algorithm is of $\mathcal{O}(m^2)$ complexity.

The paper is organized as follows. In next section, we briefly introduce the cycle detection problem as described by Shasha et al[11]. We then describe Krishnamurthy et al's cycle detection algorithm for SPMD programs in section 3.

In section 4, we present our algorithm along with correctness proofs. In section 5, promising future works and conclusions are discussed.

2 Cycle Detection

Let us consider a trace of a parallel program execution which consists of a collection of instruction sequences, one sequence for every processor. Each instruction is either a memory access (either a *read* or *write* of a program variable) or simple arithmetic operation. The program variables are located in the shared address space. There are no instructions for transfer of control.

The *program order*, P on n processors is given by n sequences of instruction executions of P_1, \dots, P_n . Each processor execution P_i is a sequence of a_{i1}, \dots, a_{im} where each a_{ij} indicates a read/write access to the shared variable or a simple arithmetic operation. Since we are considering trace of a parallel program execution, each P_i is a total order $a_{i1} < a_{i2} \dots < a_{im}$. P is the partial order formed by taking the union of these P_i 's.

An execution is *sequentially consistent* with respect to P if there exists a total order S of the operations of P , i.e., $P \subseteq S$, such that S is a correct sequential execution resulting in the same program behavior of read and write operations. Thus, S is a dynamic notion based on a particular execution. It exposes all details of a given execution.

Let V_v be the set of accesses initiated by the processors to the variable v . There exists a total order E_v over the accesses in V_v . E_v is sequentially consistent. This is true for hardware that does not allow any caching. Otherwise, it is hardware designer's responsibility to ensure this semantics through cache coherence protocol. The union of these E_v 's for all program variable v , defines the *execution order* E , which is partial. E encapsulates all the inter-processor interactions. It can be seen that E is a subset of S .

Thus, P and E serve as two different orderings on operations of a parallel execution. P orders operations using processor-centric view and E orders operations using memory-centric view. The correct execution of P obeying S is given in Theorem 1.

Theorem 1. *S is a correct execution of P if and only if S defines an execution order E such that the graph corresponding to the relation $P \cup E$ is acyclic[11].*

A cycle in $P \cup E$ graph corresponds to conflicting ordering relationships of two accesses and indicates the order in which the accesses were executed in memory is inconsistent with the order in which they were issued by the processor. Therefore, it violates sequential consistency.

To allow code reordering either at hardware or software that does not violate sequential consistency, a *delay set* D is defined which specifies some pairs of memory accesses that need to be ordered, such that the second memory access operation must be delayed until the first one is complete. D is a subset of the ordering given by P i.e., $D \subseteq P$. If D is forced to be P on a machine, this will always produce a sequentially consistent execution. The idea is to compute a

smaller D that still guarantees sequential consistency. D is sufficient for program order P , if $P \cup E$ is acyclic[5].

According to the definition, P and E are defined for a particular execution of a program. For compile time representation, P can be safely approximated using control flow graph of the program. It can be noted that each P_i is no longer a total order on accesses as the control flow graph can have branch conditions and loops. At compile time, the run-time ordering of accesses to a variable is also not known. Hence, E is approximated by undirected version of E , which are called *conflict edges*, C . C contains all unordered pairs of memory operations (a_{ij}, a_{st}) , such that $i \neq s$, and both access the same shared variable, and at least one of a_{ij} or a_{st} is a *write* operation. C can be obtained by performing a data dependence analysis over the program. However, determining C might get complicated due to alias analysis. Hence, C can further be approximated as long as $E \subseteq C$ for any execution is not violated.

During compile time analysis, Theorem 1 can be rewritten in terms of C instead of E . The primary idea is that if $P \cup C$ does not contain any cycle, then $P \cup E$ would not contain any cycle since C is conservative superset of E .

In order to construct D , it is sufficient to consider all the cycles in the $P \cup C$ and add the P edges in the cycles to D . However, the goal is to consider only those cycles that are necessary. These cycles are called *critical cycles* as defined in Definition 1 [11].

Definition 1. A cycle σ in $P \cup C$ is critical iff it fulfills the following conditions:
(i) σ contains at most two accesses from any processor; these accesses occur at successive locations in σ .

(ii) σ contains either zero, two, or three accesses to any variable; the accesses occur in consecutive locations in σ . The possible configurations for a program variable v are $\text{read}(v) \rightarrow \text{write}(v)$, $\text{write}(v) \rightarrow \text{read}(v)$, $\text{write}(v) \rightarrow \text{write}(v)$, or $\text{read}(v) \rightarrow \text{write}(v) \rightarrow \text{read}(v)$.

In the above definition the first condition specifies that σ comprises of at most two accesses from any processor. These two accesses must have a control flow path between them. These are the P edges in σ . The second condition specifies inter-processor conflict edges i.e., C edges. It also makes sure that there is no smaller cycle embedded in σ .

The delay set D is then defined as in Equation 1. It consists of all the P edges in σ .

$$D = \{[a_{is}, a_{it}] | \exists \sigma \text{ in } P \cup C \text{ such that } [a_{is}, a_{it}] \in \sigma\} \quad (1)$$

If D is obeyed during any kind of code reordering at compile time or at hardware level then the program execution is going to be sequentially consistent even on weaker memory systems. However, computing D for MIMD programs with n number of processors is proved to be NP-complete[5].

3 Cycle Detection for SPMD Programs[5,6,8]

For an SPMD program graph $P = \{P_1, \dots, P_n\}$, all P_i 's are identical. Since P can have branches, P is a partial order of the memory accesses i.e., $P = \{a_1, \dots, a_m\}$, where each a_i is a memory access. Let E be the set of directed edges in P and V be the set of vertices in P . Let the conflict set C_{SPMD} consists of pairs of memory accesses that belong to V such that both memory accesses access the same variable and at least one of them is a *write*. The conflict edges in C_{SPMD} are bi-directional, so unordered pair (u, v) means both ordered pair u to v i.e., $[u, v]$ and ordered pair v to u i.e., $[v, u]$.

The transformed graph P_{SPMD} with nodes V_{SPMD} and edges E_{SPMD} is defined in Definition 2 [5,6].

Definition 2. *The transformed graph, $P_{SPMD} = (V_{SPMD}, E_{SPMD})$, where*

$$V_{SPMD} = \{v_l \mid v \in V\} \cup \{v_r \mid v \in V\}, \text{ and}$$

$$E_{SPMD} = T_1 \cup T_2 \cup T_3, \text{ where}$$

$$T_1 = \{(u_l, v_r) \mid (u, v) \in C_{SPMD}\}$$

$$\cup \{(v_l, u_r) \mid (u, v) \in C_{SPMD}\},$$

$$T_2 = \{(u_r, v_r) \mid (u, v) \in C_{SPMD}\},$$

$$T_3 = \{[u_r, v_r] \mid [u, v] \in P\}$$

P_{SPMD} has two copies of the original program i.e., left copy and the right copy. V_{SPMD} is two copies of the accesses in V , which is labeled as l and r for left and right side copies respectively. The left and right side copies have vertices from the original program accesses. The T_1 edges connect the left and right set of nodes. The T_2 edges are conflict edges between the right set of nodes. The T_3 edges are control flow edges that link the right set of nodes. Note that the left set of nodes do not have any control flow edges. Hence, a path from v_l to u_l is composed of a T_1 edge, followed by a possible series of T_2 and T_3 edges and terminated with a T_1 edge.

For every edge $[u, v] \in P$, it is checked whether there exists a path from v_l to u_l in the graph P_{SPMD} . The delay set D_{SPMD} consists of all such edges $[u, v]$ having a path from v_l to u_l .

The algorithm by Krishnamurthy et al to calculate delay set runs in polynomial time i.e. if m is the number of accesses in the program, the delay set can be computed in $\bigcirc(m^3)$ time.

4 Our Algorithm

In this section, we present a more efficient algorithm for computing delay set in SPMD programs. Our algorithm uses a single copy of the program. The delay set computed by our algorithm is exactly the same as that computed by Krishnamurthy et al for practical SPMD programs.

Our algorithm works for structured SPMD programs. Arbitrary control branching is allowed as long the control flow graph produced by such a program is not irreducible[2]. Furthermore, we assume that before a value of a variable is

Algorithm 1 Reachable computation

```

1: Find the set  $\{S_1, S_2, \dots, S_t\}$  of maximal strongly connected regions in  $P$  (use Tarjan's algorithm). Note that every node in  $P$  has to belong to any of  $\{S_1, S_2, \dots, S_t\}$ .
2: Construct  $\Pi_P$  by reducing each  $S_i$  to a single node.
3: for all  $a_i \rightarrow a_j$  in  $P$  do
4:   Add the edge  $S_{SCR_{no}(a_i)} \rightarrow S_{SCR_{no}(a_j)}$  in  $\Pi_P$ .
5: end for
6: for all  $S_i \in \Pi_P$  do
7:   Perform a depth first traversal from  $S_i$  in  $\Pi_P$ .
8:   Set  $helper\_reachable[S_i][S_j] = 1$  if  $\exists$  a path from  $S_i$  to  $S_j$  in  $\Pi_P$ . Otherwise
      $helper\_reachable[S_i][S_j] = 0$ .
9: end for
10: for all  $a_i \in P$  do
11:   for all  $a_j \in P$  do
12:     if  $SCR_{no}(a_i) = SCR_{no}(a_j)$  then
13:        $reachable[a_i][a_j] = 1$ .
14:     else
15:        $reachable[a_i][a_j] = helper\_reachable[S_{SCR_{no}(a_i)}][S_{SCR_{no}(a_j)}]$ 
16:     end if
17:   end for
18: end for

```

read in the program, it must have been initialized. This assumption is valid for SPMD programs because before a variable is read in one processor, this variable must have at least one write performed by some other processor executing ahead of it or the same processor has a write preceding the read, failing both of which the read might contain some garbage value.

We reduce the complexity of delay set computation by using the concept of *strong connectivity* of directed graphs as defined by Tarjan[12]. In a directed graph, node a is strongly connected to node b if there exists two paths, one from a to b and another from b to a . A directed graph can be partitioned into maximal strongly connected regions using Tarjan's algorithm. Let each maximal strongly connected region be identified with a unique integer number. Nodes a and b belong to the same maximal strongly connected region if $SCR_{no}(a)$ and $SCR_{no}(b)$ are same where $SCR_{no}(m)$ denotes the unique integer associated with the strongly connected region to which node m belongs. A node which is not part of any strongly connected region according to Tarjan's algorithm has a unique SCR_{no} with respect to the strongly connected regions and can be assumed to be part of a strongly connected region formed by itself.

To keep track of all possible execution paths in P , we compute *reachable* information. $reachable(a_i, a_j)$ represents a true value if there is a control flow path from node a_i to node a_j in P .

$$reachable(a_i, a_j) = \begin{cases} 1 & \exists \text{ a path from } a_i \text{ to } a_j \text{ in } P \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

The *reachable* defined in Equation 2 can be implemented using a two dimensional boolean matrix. The *reachable* computation is given in Algorithm 1. The *reachable* computation must consider all possible paths that can be taken during execution time including the ones taken due to loop back edges in P . One can see that loop back edges add extra complexity while computing *reachable*. To counter this, we apply maximal strongly connected region algorithm to reduce P to an acyclic graph by treating each strongly connected region as a single node. In other words, Π_P is a directed acyclic graph derived from P by collapsing each strongly connected region to a single node. Edges in Π_P are inherited from P in a natural way. Note that, the strongly connected region can consist of only one node from P as mentioned before. Let $S_{SCR_{no}(a_i)}$ represents the strongly connected region in Π_P corresponding to access a_i in P . The formal definition for Π_P is given in Definition 3.

Definition 3. If $P = (V, E)$, then $\Pi_P = (V_{\Pi_P}, E_{\Pi_P})$ where $V_{\Pi_P} = \{S_1, S_2, \dots, S_t\}$ where each S_i , $1 \leq i \leq t$, is a maximal strongly connected region in P , and $E_{\Pi_P} = \{[S_1, S_2] \in V_{\Pi_P} \times V_{\Pi_P} \mid v_1 \in S_1, v_2 \in S_2, v_1 \in V, v_2 \in V, \text{ and } [v_1, v_2] \in P\}$

reachable for a pair of access (a_i, a_j) is then set to 1 if either $SCR_{no}(a_i)$ is same as $SCR_{no}(a_j)$ or there exists a path between the corresponding strongly connected regions $S_{SCR_{no}(a_i)}$ and $S_{SCR_{no}(a_j)}$ in Π_P . Our algorithm uses *reachable* information to decide which pairs need to be considered for delay set membership. It then checks if these pairs are involved in any cycle in our transformed graph P_{eff} . P_{eff} is defined in Definition 4.

Definition 4. The Transformed graph $P_{eff} = (V_{eff}, E_{eff})$, where

$$\begin{aligned} V_{eff} &= \{v \mid v \in V\}, \text{ and} \\ E_{eff} &= T_2 \cup T_3, \text{ where} \\ T_2 &= \{[u, v] \mid (u, v) \in CSPMD\} \\ &\quad \cup \{[v, u] \mid (u, v) \in CSPMD\}, \text{ and} \\ T_3 &= \{[u, v] \mid [u, v] \in P\} \end{aligned}$$

The above definition is similar to the construction of right hand side copy of P_{SPMD} defined by Krishnamurthy et al except that we add explicit bi-directional conflict edges. We ignore the T_1 edges from Krishnamurthy et al's algorithm as we do not consider two copies of the program. $CSPMD$ is the set of conflict edges as defined by Krishnamurthy et al.

To compute if a pair of access (a_i, a_j) having a control flow path from a_i to a_j i.e. $reachable(a_i, a_j) = 1$ belong to a cycle or not, we apply strong connectivity region search algorithm to P_{eff} graph. If a_i and a_j belong to the same strongly connected region then it is apparent that they are part of a cycle in P_{eff} . Hence, the delay set computed by our algorithm D_{eff} , consists of the set of pair of accesses $[a_i, a_j]$ for which $reachable(a_i, a_j)$ is true and $SCR_{no}(a_i)$ is same as $SCR_{no}(a_j)$.

The formal algorithm for computing delay set is given in Algorithm 2. The correctness of our algorithm is provided by proving that D_{SPMD} and D_{eff} are same. We show this in Theorem 2. The complexity of our algorithm is analyzed in Theorem 3.

Algorithm 2 Delay set computation

```

1: Compute reachable information using Algorithm 1.
2:  $C_{SPMD} = \{ (m, n) \mid m \in V \text{ and } n \in V \text{ and either } m \text{ or } n \text{ is a write access to the same variable}\}.$ 
3: Obtain  $P_{eff}$  by adding  $C_{SPMD}$  edges to  $P$ .
4: Use strongly connected region search algorithm to compute  $SCR_{no}(m)$ , for each  $m \in P_{eff}$ [12].
5: for all  $m \in P$  do
6:   for all  $n \in P$  and  $n \neq m$  do
7:     if  $reachable[m][n] = 1$  and  $SCR_{no}(m) = SCR_{no}(n)$  then
8:        $D_{eff} = D_{eff} \cup \{[m, n]\}.$ 
9:     end if
10:   end for
11: end for

```

Lemma 1. In Krishnamurthy et al's algorithm for any $u \in P$ and $v \in P$, if $(u_l, v_r) \in E_{SPMD}$ then $(u_r, v_r) \in E_{SPMD}$.

Proof. The pair $(u_l, v_r) \in E_{SPMD}$ implies that (u_l, v_r) is an edge of type T_1 . Since T_1 is constructed from C_{SPMD} , $(u, v) \in C_{SPMD}$. T_2 edge construction adds (u_r, v_r) to E_{SPMD} for every $(u, v) \in C_{SPMD}$. Hence, $(u_r, v_r) \in E_{SPMD}$.

Lemma 2. Based on our assumption for any $u \in P$, there is a path from u_l to u_r in P_{SPMD} .

Proof. Our assumption on SPMD programs is that before any variable is read, it must have been assigned some value. The node u can either be a read or a write of a program variable i.e., either

- 1) if u is a write of a program variable, then according to the construction of T_1 conflict edges there will be a edge from u_l to u_r in P_{SPMD} .
or
- 2) if u is a read of a program variable, then based on our assumption there exists a write to the same program variable in some node v in P which precedes u . Since v is a write to the program variable, there is a path from u_l to v_r in P_{SPMD} due to T_1 edges. T_2 conflict edge construction implies that there is a path from v_r to u_r . Hence there is a path from u_l to u_r in P_{SPMD} .

The above two cases indicate that for any $u \in P$ there is a path from u_l to u_r in P_{SPMD} .

Theorem 2. Pair of accesses $[u, v] \in D_{eff}$ iff $[u, v] \in D_{SPMD}$.

Proof. We present the proof in two parts:

Part 1: If $[u, v] \in D_{eff} \Rightarrow [u, v] \in D_{SPMD}$.

Consider $[u, v] \in D_{eff}$, so

$$SCR_{no}(u) = SCR_{no}(v) \quad (3)$$

and

$$reachable[u][v] = 1 \quad (4)$$

Equation 3 indicates that u and v are part of the same strongly connected region in P_{eff} . In particular this means there exists a path from v to u in P_{eff} . Since P_{eff} was constructed using directed version of T_2 edges and T_3 edges as defined by Krishnamurthy et al, P_{eff} represents the right hand side of P_{SPMD} graph with bi-directional conflict edges. Hence, there exists a path from v to u in right side of P_{SPMD} i.e., there is a path from v_r to u_r in P_{SPMD} . Also according to Lemma 2 there exists paths from v_l to v_r and u_l to u_r . Therefore, there exists a path from v_l to u_l in P_{SPMD} . Equation 4 implies that there exists a control flow path from u to v in P , so the pair (u_l, v_l) is considered for the computation of D_{SPMD} . Therefore we have shown that (u_l, v_l) is considered for D_{SPMD} and there exists a path from v_l to u_l . Hence, $[u, v] \in D_{SPMD}$.

Part 2: If $[u, v] \in D_{SPMD} \Rightarrow [u, v] \in D_{eff}$.

Consider the pair $[u, v] \in D_{SPMD}$. This implies there is a path from u_r to v_r and a path $v_l \rightarrow A_r \cdots B_r \rightarrow C_r \cdots D_r \rightarrow u_l$. By Lemma 1, the path $v_l \rightarrow A_r \cdots B_r \rightarrow C_r \cdots D_r \rightarrow u_l$ implies that there exist a path $v_r \rightarrow A_r \cdots B_r \rightarrow C_r \cdots D_r \rightarrow u_r$. Hence, the path u_r to v_r and the path v_r to u_r exist in the right hand side of P_{SPMD} . Therefore according to the construction of P_{eff} , the path from u to v and the path from v to u exist in P_{eff} . By the definition of strong connectivity, $SCR_{no}(u)$ is same as $SCR_{no}(v)$. Hence, $(u, v) \in D_{eff}$.

Theorem 3. *The time complexity of delay set Computation algorithm (Algorithm 2) is $\mathcal{O}(m^2)$, m is the number of accesses in the program.*

Proof. Step 1 of Algorithm 2 takes $\mathcal{O}(m^2)$ time. This is because the complexity of Algorithm 1 is bounded by the complexity of maximal strongly connected region search algorithm (in step 1 of Algorithm 1) and depth first traversal from each node (in step 7 of Algorithm 1) in Π_P . The strongly connected region search algorithm in step 1 of Algorithm 1 is $\mathcal{O}(m^2)$ using the algorithm suggested by Tarjan[12]. The depth first traversal in step 7 of Algorithm 1 takes $\mathcal{O}(m)$ time as it's complexity is $\mathcal{O}(|\Pi_E|)$ where $|\Pi_E|$, the number of edges in Π_P which is $\mathcal{O}(m)$. Loops from step 5 till 11 of Algorithm 2 take $\mathcal{O}(m^2)$ time. Hence the total complexity of our algorithm is $\mathcal{O}(m^2)$.

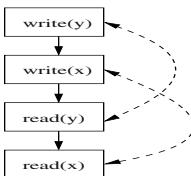


Fig. 2. Minimal delay set

5 Conclusion and Future Work

In this paper, we presented an efficient $\mathcal{O}(m^2)$ algorithm to compute delay set for SPMD programs. Our algorithm is also space efficient since it uses only

one copy of the program for analysis as compared to two copies suggested by Krishnamurthy. We provide a theoretical equivalence of our algorithm with that of Krishnamurthy's algorithm for practical SPMD programs. In practice, the delay set D_{eff} computed by our algorithm can be used along with the code generation module for languages like Split-C[4] to produce back-end codes which will guarantee sequential execution. The details of integration with the code generation module of Split-C is similar to [6,5].

Our delay set D_{eff} is not minimal as it can have information which is transitive. Consider the simple SPMD program fragment in Figure 2. Our algorithm will compute D_{eff} as $\{ [write(y), write(x)], [write(x), read(y)], [write(y), read(y)], [write(y), read(x)], [write(x), read(x)], [read(y), read(x)] \}$. However the minimal delay set is $\{ [write(y), write(x)], [write(x), read(y)], [read(y), read(x)] \}$. Future research can be done to eliminate transitive information from D_{eff} and proving the minimality criteria.

References

1. Sarita A. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley publishing company, 1986.
3. A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5), October 1966.
4. David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing*, pages 262–273, 1993.
5. Arvind Krishnamurthy. *Compiler analyses and system support for optimizing shared address space programs*. PhD thesis, University of California Berkeley, 1999.
6. Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel spmd programs. *Languages and Compilers for Parallel Computing*, 1994.
7. Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming Languages Design and Implementation (PLDI)*, pages 196–204, June 1995.
8. Arvind Krishnamurthy and Katherine Yelick. Analyses and optimizations for shared address space. *Journal of Parallel and Distributed Computing*, 1996.
9. Leslie Lamport. How to make multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9), September 1979.
10. S. P. Midkiff and D. A. Padua. Issues in the optimization of parallel programs. *International Conference on Parallel Processing - Vol II*, pages 105–113, 1990.
11. Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.
12. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

Dynamic Load Balancing for I/O-Intensive Tasks on Heterogeneous Clusters

Xiao Qin, Hong Jiang, Yifeng Zhu, and David R. Swanson

Department of Computer Science and Engineering
University of Nebraska – Lincoln, Lincoln, NE, USA.
`{xqin, jiang, yzhu, dswanson}@cse.unl.edu`

Abstract. Since I/O-intensive tasks running on a heterogeneous cluster need a highly effective usage of global I/O resources, previous CPU- or memory-centric load balancing schemes suffer significant performance drop under I/O-intensive workload due to the imbalance of I/O load. To solve this problem, we develop two I/O-aware load-balancing schemes, which consider system heterogeneity and migrate more I/O-intensive tasks from a node with high I/O utilization to those with low I/O utilization. If the workload is memory-intensive in nature, the new method applies a memory-based load balancing policy to assign the tasks. Likewise, when the workload becomes CPU-intensive, our scheme leverages a CPU-based policy as an efficient means to balance the system load. In doing so, the proposed approach maintains the same level of performance as the existing schemes when I/O load is low or well balanced. Results from a trace-driven simulation study show that, when a workload is I/O-intensive, the proposed schemes improve the performance with respect to mean slowdown over the existing schemes by up to a factor of 8. In addition, the slowdowns of almost all the policies increase consistently with the system heterogeneity.

1 Introduction

Dynamic load balancing schemes are widely recognized as important techniques for the efficient utilization of resources in networks of workstations or clusters. Many load balancing policies achieve high system performance by increasing the utilization of CPU [1], memory [2], or a combination of CPU and memory [3]. However, these load-balancing policies are less effective when the workload comprises a large number of I/O-intensive tasks and I/O resources exhibit imbalanced load. We have proposed two I/O-aware load-balancing schemes to improve overall performance of a distributed system with a general and practical workload including I/O activities [4][5]. However, it is assumed in [4][5] that the system is homogeneous in nature. There is a strong likelihood that upgraded clusters or networked clusters are heterogeneous, and heterogeneity in disks tends to induce more significant performance degradation when coupled with imbalanced load of memory and I/O resources. Therefore, it becomes imperative that heterogeneous clusters be capable of hiding the heterogeneity of resources, especially that of

I/O resources, by judiciously balancing I/O work across all the nodes in a cluster. This paper studies two dynamic load balancing policies, which are shown to be effective for improving the utilization of disk I/O resources in heterogeneous clusters.

There is a large body of literature concerning load balancing in disk systems. Scheuermann et al. [6] have studied the issues of striping and load balancing in parallel disk systems. To minimize total I/O time in heterogeneous cluster environments, Cho et al. [7] have developed heuristics to choose the number of I/O servers and place them on physical processors. In [8][9], we have studied dynamic scheduling algorithms to improve the read and write performance of a parallel file system by balancing the global workload. The above techniques can improve system performance by fully utilizing the available hard drives. However, these approaches become less effective under a complex workload where I/O-intensive tasks share resources with many memory- and CPU-intensive tasks.

Many researchers have shown that I/O cache and buffer are useful mechanisms to optimize storage systems. Ma et al. [10] have implemented active buffering to alleviate the I/O burden by using local idle memory and overlapping I/O with computation. We have developed a feedback control mechanism to improve the performance of a cluster by manipulating the I/O buffer size [11]. Forney et al. [12] have investigated storage-aware caching algorithms in heterogeneous clusters. Although we focus solely on balancing disk I/O load in this paper, the approach proposed here is capable of improving the buffer utilization of each node. The scheme presented in this paper is complementary to the existing caching/buffering techniques, thereby providing additional performance improvement when combined with active buffering, storage-aware caching, and a feedback control mechanism.

The rest of the paper is organized as follows. Section 2 describes the system model. Section 3, we propose two I/O-aware load-balancing policies. Section 4 evaluates the performance. Section 5 concludes the paper.

2 Workload and System Model

In this paper, we consider a cluster as a collection of heterogeneous nodes connected by a high-speed network. Tasks arrive at each node dynamically and independently, and share resources available there. Each node maintains its individual task queue where newly arrived tasks may be transmitted to other nodes or executed in the local node, depending on a load balancing policy employed in the system. Each node keeps track of reasonably up-to-date global load information by periodically exchanging load status with other nodes.

We address heterogeneity with respect to a diverse set of disks, CPUs, and memories. We characterize each node i by its CPU speed C_i , memory capacity M_i , and disk performance D_i . Let B_i^{disk} , S_i , and R_i denote the disk bandwidth, average seek time, and average rotation time of the disk in node i , then the disk performance can be approximately measured as the following equation:

$D_i = \frac{1}{S_i + R_i + d/B_i^{disk}}$ where d is the average size of data stored or retrieved by I/O requests.

The weight of a disk performance W_i^{disk} is defined as a ratio between its performance and that of the fastest disk in the cluster. Thus, we have $W_i^{disk} = \frac{D_i}{\max_{j=1}^n (D_j)}$. The disk heterogeneity level, referred to as H_D , can be quantitatively measured by the standard deviation of disk weights. Thus, H_D is expressed as:

$$H_D = \sqrt{\frac{\sum_{i=1}^n (W_{avg}^{disk} - W_i^{disk})^2}{n}} \quad (1)$$

where W_{avg}^{disk} is the average disk weight. Likewise, the CPU and memory heterogeneity levels are defined as follows:

$$H_C = \sqrt{\frac{\sum_{i=1}^n (W_{avg}^{CPU} - W_i^{CPU})^2}{n}}, H_M = \sqrt{\frac{\sum_{i=1}^n (W_{avg}^{mem} - W_i^{mem})^2}{n}} \quad (2)$$

where W_i^{CPU} and W_i^{mem} are the CPU and memory weights, and W_{avg}^{CPU} and W_{avg}^{mem} are the average weights of CPU and memory resources [3].

We also assume that the network provides full connectivity in the sense that any two nodes are connected through either a physical link or a virtual link. We assume that arriving tasks are either sequential or otherwise parallel applications can be broken into a number of tasks with individual and independent resource requirements.

Each task is assumed to read or write data locally, and this amount of data, referred to as initial data, is required to be shipped along with a migrated task. This assumption is conservative in the sense that it makes our approach less effective, because the migration overhead imposed by the initial data can be potentially avoided by replicating it across all the nodes. The memory burden of migrating data is not considered in our model, because data movement can be handled by storage and network interface controllers without local CPU's intervention and I/O buffer [13].

For simplicity, we assume that all I/O operations issued by tasks are blocking. This simplification is conservative in the sense that it underestimates the performance benefits from the proposed scheme because this assumption causes a number of undesired migrations with negative impact.

3 Load Balancing in Heterogeneous Clusters

3.1 Existing Load Balancing Policies

In principle, the load of a node can be balanced by migrating either a newly arrived job or a currently running job preemptively to another node if needed. While the first approach is referred to as Remote Execution, the second one is called preemptive migration [1][5]. Since the focuses of this study are effective usage of I/O resources and coping with system heterogeneity, we only consider

the technique of remote execution in this paper. In what follows, we briefly review two existing load-balancing policies.

(1) CPU-based Load Balancing (CPU-RE) [3][14]. We consider a simple and effective policy, which is based on a heuristic. The CPU load of node i , $load_{CPU}(i)$, is measured by the following expression [3]: $load_{CPU}(i) = L_i \times (\frac{\max_{j=1}^n C_j}{C_i})$ where L_i is the number of tasks on node i .

If $load_{CPU}(i)$ is greater than a certain threshold, node i is considered overloaded with respect to CPU resource. The CPU-based policy transfers the newly arrived tasks on the overloaded node i to the remote node with the lightest CPU load. This policy is capable of resulting in a balanced CPU load distribution for systems with uneven CPU load distribution [14]. Note that the CPU threshold is a key parameter that depends on both workload and transfer cost. In the experiments reported in Section 4, the value of CPU threshold is set to four [3].

(2) CPU-memory-based Load Balancing (CM-RE) [3]. This policy takes both CPU and memory resources into account. The memory load of node i , $load_{mem}(i)$, is the sum of the memory space allocated to the tasks running on node i . Thus, we have $load_{mem}(i) = \sum_{j \in N_i} mem(j)$ where $mem(j)$ represents the memory requirement of task j , and N_i denotes the set of tasks running on node i . If the memory space of a node is greater than or equal to $load_{mem}(i)$, CM-RE adopts the above CPU-RE policy to make load-balancing decisions. When $load_{mem}(i)$ exceeds the amount of available memory space, the CM-RE policy transfers the newly arrived tasks on the overloaded node to the remote nodes with the lightest memory load. Zhang et al. [3] showed that CM-RE is superior to CPU-RE under a memory-intensive workload.

3.2 IO-Aware Load Balancing in Heterogeneous Clusters

We now turn our attention to an I/O-aware load balancing policy (IO-RE), which is heuristic and greedy in nature. Instead of using CPU and memory load indices, the IO-RE policy relies on an I/O load index to measure two types of I/O access: the implicit I/O load induced by page faults and the explicit I/O requests resulting from tasks accessing disks. Let $page(i, j)$ be the implicit I/O load of task j on node i , and $IO(j)$ be the explicit I/O requirement of task j . Thus, node i 's I/O load index is:

$$load_{IO}(i) = \sum_{j \in N_i} page(i, j) + \sum_{j \in N_i} IO(j) \quad (3)$$

An I/O threshold, $threshold_{IO}(i)$, is introduced to identify whether node i 's I/O resource is overloaded. Node i 's I/O resource is considered overloaded if $load_{IO}(i)$ is higher than $threshold_{IO}(i)$. Specifically, $threshold_{IO}(i)$, which reflects the I/O processing capability of node i , is expressed as:

$$threshold_{IO}(i) = \frac{D_i}{\sum_{j=1}^n D_j} \times \sum_{j=1}^n load_{IO}(j) \quad (4)$$

where the first term on the right hand side of the above equation corresponds to the fraction of the total I/O processing power on node i , and the second term gives the accumulative I/O load imposed by the running tasks in the heterogeneous cluster. The I/O threshold associated with node i can be calculated using equations 3 to substitute for $load_{IO}(j)$ in equation 4.

For a task j arriving at a local node i , the IO-RE scheme attempts to balance I/O resources in the following four main steps. First, the I/O load of node i is updated by adding task j 's explicit and implicit I/O load. Second, the I/O threshold of node i is computed based on Equation 4. Third, if node i 's I/O resource is underloaded, task j will be executed locally on node i . When the node is overloaded with respect to I/O resource, IO-RE judiciously chooses a remote node k as task j 's destination node, subject to the following two conditions: (1) The I/O resource is not overloaded. (2) The I/O load discrepancy between node i and k is greater than the I/O load induced by task j , to avoid useless migrations. If such a remote node is not available, task j has to be executed locally on node i . Otherwise and finally, task j is transferred to the remote node k , and the I/O load of nodes i and k is updated in accordance with j 's load.

Since tasks' implicit and explicit I/O load will be used in Equation 4 and the above conditions, it is essential to derive the two types of I/O load. When the available memory space M_i is unable to fulfill the accumulative memory requirements of all tasks running on the node ($load_{mem}(i) > M_i$), the node may encounter a large number of page faults. Implicit I/O load depends on three factors: M_i , $load_{mem}(i)$, and the page fault rate pr_i . Thus, $page(i, j)$ can be defined as follows:

$$page(i, j) = \begin{cases} 0 & \text{if } load_{mem}(i) \leq M_i, \\ \frac{pr_i \times load_{mem}(i)}{M_i} & \text{otherwise.} \end{cases} \quad (5)$$

Explicit I/O load $IO(i, j)$ is proportional to I/O access rate $ar(j)$ and inversely proportional to I/O buffer hit rate $hr(i, j)$. The buffer hit rate for task j on node i is approximated by the following formula:

$$hr(i, j) = \begin{cases} \frac{r}{r+1} & \text{if } buf(i, j) \geq d(j), \\ \frac{r \times buf(i, j)}{(r+1) \times d(j)} & \text{otherwise,} \end{cases} \quad (6)$$

where r is the data re-access rate (defined to be the number of times the same data is accessed by a task), $buf(i, j)$ denotes the buffer size allocated to task j , and $d(j)$ is the amount of data accessed by task j , given a buffer with infinite size. The buffer size a task can obtain at run time heavily depends on the total available buffer size in the node and the tasks' access patterns. $d(j)$ is linearly proportional to access rate, computation time and average data size of I/O accesses, and $d(j)$ is inversely proportional to r . In some cases, where the initial data of a task j is not initially available at the remote node, the data migration overhead can be approximately estimated as $T_d(j) = d_{init}(j)/b_{net}$, where $d_{init}(j)$ and b_{net} represent the initial data size and the available network bandwidth, respectively.

3.3 IOCM-RE: A Comprehensive Load Balancing Policy

Since the main target of the IO-RE policy is exclusively I/O-intensive workload, IO-RE is unable to maintain a high performance when the workload tends to be CPU- or memory-intensive. To overcome this limitation of IO-RE, a new approach, referred to as IOCM-RE, attempts to achieve the effective usage of CPU and memory in addition to I/O resources in heterogeneous clusters.

More specifically, when the explicit I/O load of a node is greater than zero, the I/O-based policy will be leveraged by IOCM-RE as an efficient means to make load-balancing decisions. When the node exhibits no explicit I/O load, either the memory-based or the CPU-based policy will be utilized to balance the system load. In other words, if the node has implicit I/O load due to page faults, load-balancing decisions are made by the memory-based policy. On the other hand, the CPU-based policy is used when the node is able to fulfill the accumulative memory requirements of all tasks running on it. A pseudo code of the IOCM-RE scheme is presented in Figure 1 below.

Algorithm: IO-CPU-Memory based load balancing (IOCM-RE):

/* Assume that a task j newly arrives at node i */

if $IO(j) + \sum_{j \in N_i} IO(k) > 0$ then

The IO-RE policy is used to balance the system node; /* see Section 3.2 */

else if $page(i, j) + \sum_{j \in N_i} page(i, k) > 0$ then /* see Section 3.1(2) */

The memory-based policy is utilized for load balancing;

else /* see Section 3.1(1) */

The CPU-based policy makes the load balancing decision;

Fig. 1. Pseudocode of the IO-CPU-Memory based load balancing

4 Performance Evaluation

In this section, we experimentally compare the performance of IOCM-RE against that of three other schemes: CPU-RE [14][15], CM-RE [3], and IO-RE (Section 3.2). The performance metric by which we judge system performance is the mean slowdown. Formally, the mean slowdown of all the tasks in trace T is given below, where w_i and $l_C(i)$ are wall-clock execution time and computation load of task i . The implicit and explicit disk I/O load of task i are denoted as $l_{page}(i)$ and $l_{IO}(i)$, respectively.

$$slowdown(T) = \frac{\sum_{i \in T} w_i}{\sum_{i \in T} \left(\frac{nl_c(i)}{\sum_{j=1}^n C_j} + \frac{n(l_{page}(i) + l_{IO}(i))}{\sum_{j=1}^n D_j} \right)} \quad (7)$$

Note that the numerator is the summation of all the tasks' wall-clock execution time while sharing resources, and the denominator is the summation of all

tasks' time spent running on CPU or accessing I/O without sharing resources with other tasks. Since the clusters studied in the simulation experiments are heterogeneous in nature, we use the average values of CPU power, memory space, and the disk performance to calculate the execution time of each task exclusively running on a node.

4.1 Simulator and Simulation Parameters

Harchol-Balter and Downey [1] have implemented a simulator of a distributed system with six nodes. Zhang et. al [3] extended the simulator by incorporating memory recourses. Compared to these two simulators, ours possesses four new features. First, the IOCM-RE and IO-RE schemes are implemented. Second, a fully connected network is simulated. Third, a simple disk model is added into the simulator. Last, an I/O buffer is implemented on top of the disk model in each node. In all experiments, we used the simulated system with the configuration parameters listed in Table 1. The parameters are chosen in such a way that they resemble workstations such as the Sun SPARC-20.

Table 1. System Parameters. CPU speed and page fault rate are measured by Millions Instruction Per Second (MIPS) and No./Million Instructions (No./MI), respectively.

Parameters	Value	Parameters	Value
CPU Speed	100-400 MIPS	Page Fault Service Time	8.1 ms
RAM Size	32-256 MByte	Mean page fault rate	0.01No./MI
Buffer Size	64MByte	Data re-access rate,r	5
Context switch time	0.1 ms	Network Bandwidth	100Mbps

Disk I/O operations issued by each task are modeled as a Poisson Process with a mean arrival rate λ , which is set to 2.0 No./MI in our experiments. The service time of each I/O access is the summation of seek time, rotation time, and transfer time. The transfer time is equal to the size of accessed data divided by the disk bandwidth. Data sizes are randomly generated based on a Gamma distribution with the mean size of 256KByte.

The configuration of disks used in our simulated environment is based on the assumption of device aging and performance-fault injection. Specifically, IBM 9LZX is chosen as a base disk and its performance is aged over years to generate a variety of disk characteristics [12], which is shown in Table 2.

Table 2. Characteristics of Disk Systems. sk time: seek time, R time: Rotation time

Age	sk time	R time	Bandwidth	Age	sk time	R time	Bandwidth
1 year	5.3 ms	3.00ms	20.0MB/s	4 year	7.27ms	4.11ms	7.29MB/s
2 year	5.89ms	3.33ms	14.3MB/s	5 year	8.08ms	4.56ms	5.21MB/s
3 year	6.54ms	3.69ms	10.2MB/s	6 year	8.98ms	5.07ms	3.72MB/s

Table 3. Characteristics of Five Heterogeneous Clusters. CPU and memory are measured by MIPS and MByte. Disks are characterized by bandwidth measured in MByte/S. HL-Heterogeneity Level

Node	system A			system B			system C			system D			system E		
	cpu	mem	disk	cpu	mem	disk	cpu	mem	disk	cpu	mem	disk	cpu	mem	disk
1	100	480	20	100	480	20	100	480	10.2	50	320	10.2	50	320	5.21
2	100	480	20	150	640	20	150	640	20	200	800	20	200	800	14.3
3	100	480	20	150	640	20	150	640	20	200	800	20	200	800	20
4	100	480	20	50	320	20	50	320	10.2	50	320	14.3	50	320	5.21
5	100	480	20	100	480	20	100	480	20	50	320	14.3	50	320	7.29
6	100	480	20	50	320	20	50	320	10.2	50	320	10.2	50	320	3.72
HL	0	0	0	0.27	0.20	0	0.27	0.20	0.25	0.35	0.28	0.20	0.35	0.28	0.30

4.2 Overall Performance Comparisons

In this experiment, the CPU power and the memory size of each node are set to 200MIPS and 256MByte. Figure 2 plots the mean slowdown as a function of disk age. Disks are configured such that five fast disks are one year old, and a sixth, slower disk assumed an age ranging from 1 to 6 years.

Figure 2 shows that the mean slowdowns of four policies increase considerably as one of the disks ages. This is because aging one slow disk gives rise to longer I/O processing time. A second observation from Figure 2 is that IO-RE and IOCM-RE perform the best out of the four policies, and they improve the performance over the other two policies by up to a factor of 8. The performance improvement of IO-RE and IOCM-RE relies on the technique that balances I/O load by migrating I/O-intensive tasks from overloaded nodes to underloaded ones.

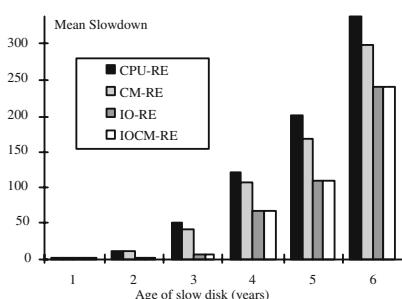


Fig. 2. Mean slowdown as a function of the age of a single disk

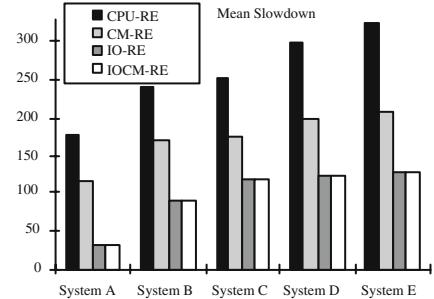


Fig. 3. Mean slowdown on five heterogeneous systems.

4.3 Impact of Heterogeneity on the Performance of Load-Balancing Policies

In this section, we turn our attention to the impact of system heterogeneity on the performance of the proposed policies. The five configurations of increasing heterogeneity of a heterogeneous cluster with 6 nodes are summarized in Table 3. As can be seen from Figure 3, IO-RE and IOCM-RE significantly outperform the other two policies. For example, IOCM-RE improves the performance over CPU-RE and CM-RE by up to a factor of 5 and 3, respectively.

Importantly, Figure 3 shows that the mean slowdowns of almost all policies increase consistently as the system heterogeneity increases. An interesting observation from this experiment is that the mean slowdowns of IO-RE and IOCM-RE are more sensitive to changes in CPU and memory heterogeneity than the other two policies. Recall that system B's CPU and memory heterogeneities are higher than those of system A. Compared the performance of system A with that of B, the mean slowdowns of IO-RE and IOCM-RE are increased by 196.4%, whereas the slowdowns of CPU-RE and CM-RE are increased approximately by 34.7% and 47.9%, respectively. The reason is that I/O-aware policies ignore the heterogeneity in CPU resources. When the heterogeneity of CPU and memory remain unchanged, IO-RE and IOCM-RE is less sensitive to the change in disk I/O heterogeneity than the other three policies. This is because both IO-RE and IOCM-RE consider disk heterogeneity as well as the effective usage of I/O resources.

5 Conclusion

In this paper, we have studied two I/O-aware load-balancing policies, IO-RE (I/O-based policy) and IOCM-RE (load balancing for I/O, CPU, and Memory), for heterogenous clusters executing applications that represent a diverse workload conditions, including I/O-intensive and memory-intensive applications. IOCM-RE considers both explicit and implicit I/O load, in addition to CPU and memory utilizations. To evaluate the effectiveness of our approaches, we have compared the performance of the proposed policies against two existing approaches: CPU-based policy (CPU-RE) and CPU-Memory-based policy (CM-RE). IOCM-RE is more general than the existing approaches in the sense that it can maintain high performance under diverse workload conditions. A trace-driven simulation provides us with empirical results to draw three conclusions: (1) When a workload is I/O-intensive, the proposed scheme improves the performance with respect to mean slowdown over the existing schemes by up to a factor of 8. (2) The slowdowns of the four policies considerably increase as one of the disks ages. (3) The slowdowns of almost all the policies increase consistently with the system heterogeneity. In this paper, we assumed that parallel applications can be broken into a number of individual tasks and, therefore, future research will deal with parallel jobs with inter-dependent tasks. We also plan to evaluate the performance of the proposed schemes using a set of real I/O-intensive application traces.

Acknowledgments. This work was partially supported by an NSF grant (EPS-0091900), a Nebraska University Foundation grant (26-0511-0019), and a UNL Academic Program Priorities Grant. Work was completed using the Research Computing Facility at University of Nebraska-Lincoln.

References

1. Harchol-Balter, M., Downey, A.: Exploiting process lifetime distributions for load balancing. *ACM Transactions on Computer Systems* **15** (1997) 253–285
2. Acharva, A., Setia, S.: Availability and utility of idle memory in workstation clusters. In: Proceedings of the ACM SIGMETRICS Conf. on Measuring and Modeling of Computer Systems. (1999)
3. Xiao, L., Zhang, X., Qu, Y.: Effective load sharing on heterogeneous networks of workstations. In: Proc. of International Symposium on Parallel and Distributed Processing. (2000)
4. Qin, X., Jiang, H., Zhu, Y., Swanson, D.: A dynamic load balancing scheme for I/O-intensive applications in distributed systems. In: Proceedings of the 32nd International Conference on Parallel Processing Workshops. (2003)
5. Qin, X., Jiang, H., Zhu, Y., Swanson, D.: Boosting performance for I/O-intensive workload by preemptive job migrations in a cluster system. In: Proc. of the 15th Symp. on Computer Architecture and High Performance Computing, Brazil (2003)
6. Scheuermann, P., Weikum, G., Zabback, P.: Data partitioning and load balancing in parallel disk systems. *The VLDB Journal* (1998) 48–66
7. Cho, Y., Winslett, M., S. Kuo, J.L., Chen, Y.: Parallel I/O for scientific applications on heterogeneous clusters: A resource-utilization approach. In: Proceedings of Supercomputing. (1999)
8. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.: Scheduling for improved write performance in a cost-effective, fault-tolerant parallel virtual file system (CEFT-PVFS). In: the Fourth LCI International Conference on Linux Clusters. (2003)
9. Zhu, Y., Jiang, H., Qin, X., Feng, D., Swanson, D.: Improved read performance in a cost-effective, fault-tolerant parallel virtual file system (ceft-pvfs). In: Proc. of the 3rd IEEE/ACM Intl. Symp. on Cluster Computing and the Grid. (2003)
10. Ma, X., Winslett, M., Lee, J., Yu, S.: Faster collective output through active buffering. In: Proceedings of the International Symposium on Parallel and Distributed Processing. (2002)
11. Qin, X., Jiang, H., Zhu, Y., Swanson, D.: Dynamic load balancing for I/O- and memory-intensive workload in clusters using a feedback control mechanism. In: Proceedings of the 9th International Euro-Par Conference on Parallel Processing (Euro-Par 2003), Klagenfurt, Austria (2003)
12. Forney, B., Arpacı-Dusseau, A.C., Arpacı-Dusseau, R.H.: Storage-aware caching: Revisiting caching for heterogeneous storage systems. In: Proceedings of the 1st Symposium on File and Storage Technology, Monterey, California, USA (2002)
13. Geoffray, P.: Opiom: Off-processor I/O with myrinet. *Future Generation Computer Systems* **18** (2002) 491–499
14. Franklin, M., Govindan, V.: A general matrix iterative model for dynamic load balancing. *Parallel Computing* **33** (1996)
15. Eager, D., Lazowska, E., Zahorjan, J.: Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Eng.* **12** (1986) 662–675

Standards Based High Performance Computing

David Scott

Intel Corporation

Abstract. While traditional supercomputers are proprietary designs, there has been a dramatic increase in the use of Intel processors in High Performance Computing. One indication of this is the number of Intel based systems appearing on the list of Top500 computers based on MPLinpack performance, which has grown from 3 eighteen months ago to 119 on the most recent list.

One of the enablers for this increase is the availability of standards and open software at almost every level of the solution stack. Standard rack mounted two or four processor servers are used as compute nodes. A large number of commercial off the shelf (COTS) networks are available to complete the hardware configuration. The open source operating system Linux is used on each node of the cluster. A number of different cluster configuration management systems are available including OSCAR and NPACI Rocks. Application level message passing is done using the standard MPI (Message Passing Interface) API. Standard compilers are then used to compile the application.

Given this infrastructure, a number of commercial applications have been released for Linux clusters including Oracle 9i RAC, the petroleum software systems from Landmark and Slumberger, and manufacturing codes such as MSC Nastran and LS-Dyna3D.

The weakest link in the software infrastructure for cluster computing is the lack of tools for porting applications from shared memory implementations to distributed memory message passing implementations, which requires distributing the data among the compute nodes and putting the message passing sends and receives in as needed. This has historically been a manual process involving significant changes to the source code of the application. Tools to help automate this process are underdevelopment and their availability will greatly accelerate the acceptance of cluster computing as mainstream High Performance Computing.

Delay and Jitter Minimization in High Performance Internet Computing

Javed I. Khan and Seung S. Yang

Media Communications and Networking Research Laboratory
Department of Computer Science, Kent State University
`{javed|syang}@kent.edu`

Abstract. Delay and jitter management in high performance active computation is a challenging problem in the emerging area of internet computing. The problem takes a complex new form from its classical counterpart. Here processing time becomes a major part of transit delay dynamics. Also the concept of path ‘bandwidth’ a classical notion used extensively in classical networking, now degenerates as data volume can change while in transit. The paper presents a dynamic feedback based scheme for this problem with a live internet computing based video transcoder experiment.

1 Introduction

The Internet and particularly the Web is increasingly becoming ‘active’. Several paradigms, though emerging from widely different origin, are already underway which are looking for more efficient means for performing active computations with in a network, where the Internet is viewed not only as a large network but also as a confederated platform for integrated computing and communication. The spectrum ranges from grid networking, web-based meta-computing, content services networking, active and programmable networks, sensor computing to the very recent automatic computing [4]. In one end of the spectrum, the Grid initiative is exploring technology so that distributed idle cycles of massive number of computers, including supercomputers, in the Internet can be used to perform advanced scientific tasks [10]. In the context of internet computation, a frequently appearing model of computation is *Active Information Streaming (AIS)*. AIS consider how a passing data stream can be arbitrarily processed while in transit. This data stream does not have to be conventional video or audio stream. Recently, we are investing on various issues related to AIS. Just like the current HTML based adaptation services, wide range of active services can be potentially built for streamed information ranging from channel multiplexing/ de-multiplexing, distributed simulation, remote visualization, automatic rate adaptation, sensor data flow aggregation, to security filtering etc. conforming to AIS model. A particularly interesting problem in AIS is the management of temporal stream characteristics. The problem is quite different and far more challenging from its counterpart in classical networks (we will explain the differences shortly). We believe that it will be a central concern in high performance networked computing irrespective of the framework. This will be a major and central concern for time sensitive application processing, including live simulation, distributed visualization, live processing of sensor data, instrument control besides media streaming.

Interestingly, also many other application processing, which are not normally known to be time sensitive-- may turn into one. The new variability introduced by uncertain compute resources available over a loosely federated resource pool can seriously destabilize synchronization, load balancing, and utilization efficiency of known distributed solutions. In this paper we present a jitter and delay control model for AIS including sharing results from a live experiments on an implemented concept system. We have conducted live video experiments on the recently launched ABONE test bed using a novel video transcoding system called *MPEG-2 Active Video Streaming* (AVIS) transport [5]. AVIS has been designed for arbitrary transformation (or filtering) of a passing video and can be used by any server/player application as a socket like transport. A particularly novel aspect of AVIS is that its processing capable components can utilize multiple processing capable junctions in the pathway to perform the required computation. The video stream can receive arbitrary transformation in a distribute manner in various processing capable nodes over an ‘active’ subnet while the packets diffuse via multiple paths by cooperative transcoding. Compared to the previous works in jitter and delay control (see [1], [2], [3], [6], [7], [8]) this paper addresses the problem with respect to joint communication and computation delay. No previous work could be found to address this problem. Active streaming adds three new distinct challenges. First, even in a real network environment, it is difficult to obtain the source traffic model. In active paradigm, network computation adds additional set of complex variability. All network nodes do not have same processing capability. The processing time can vary for different contents and for degree of customization. Secondly, the initial data can dramatically alter in size and time spacing at each stage of servicing. The capsule data unit can be of unequal size. All packets are not uniformly needed by the service capsules. Thirdly, also there is effect of non sequential access. Some of the packets should be used at the same time by the service module, while some others may not be accessed at all. In this paper, we demonstrate a joint buffering and scheduling based algorithm which corrects both computation and transmission difference to reduce the jitter.

2 Active Stream Computation Model

2.1 Graph Time Computing

We first present the framework called *Graph Time Computing Model*. A stream transformation occurs in one or more *processing capable subnet* in a path between the source and the sink. The processing of a flow involves an ordered set of transformations on a series of *application data units* (ADU) via a series of sub-task modules. In this *processing capable subnet* the application data units can spread into multiple *processing capable nodes*. The spread occurs to overcome the resource limitation, whether it is the scarcity of available compute cycle on a single node or of bandwidth in an involved pathway. We call the point where the flow enters this subnet as *GT-fork* and the point where it exits the subnet as *GT-joint*. Series nodes are called *GT-connect*. Any AIS layout can be decomposed in terms of these basic connection components. Graph in Fig-1 shows a two level recursive factored graph.

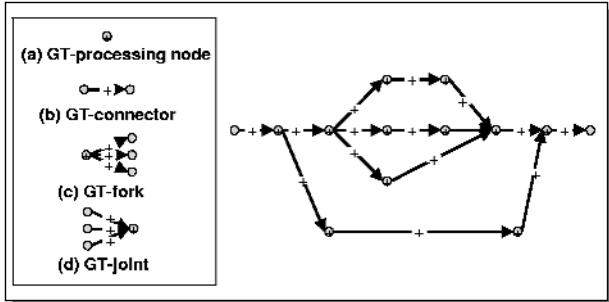


Fig. 1. Graph Time based decomposition

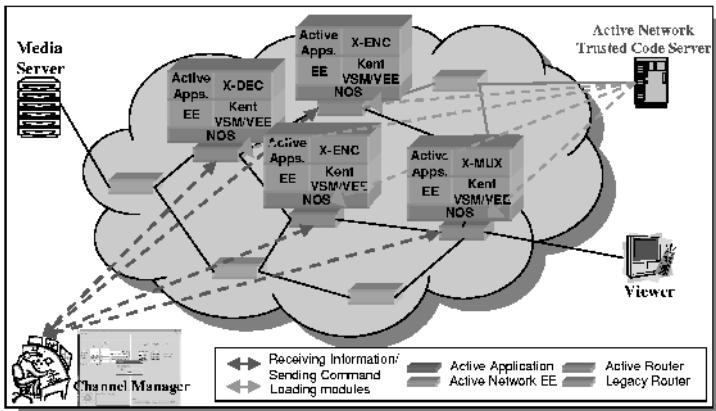


Fig. 2. AVIS system service model

2.2 Flow and Service Model

In active system, the size and representation of ADU is no longer a fixed quantity rather it can vary between the active hops due to active transformation. Each ADU must pass through the service sub-task modules in same pre-specified order irrespective of the sub-path it takes. We will denote the g-th ADU after the sub-task M as ADU^M_g . An active stream processing involves a *service*, a *processing capable subnet* platform and a *session* flow. A *service* (A) is characterized by a sequence of sub-task module and their dependency graph $\langle A = \{a^i\} \rangle$, with nodes representing the sub-tasks. The *processing capable subnet* is characterized by a graph $\langle N(V, L, C) \rangle$, where v^i is the processing capable node, l^{ij} are the overlay links and C is the capacity metric. In C each link has a bandwidth attribute B^{ij} bytes/sec and each node has a compute power attribute B^i flops. A *session* is modeled with a source flow rate $\langle F \rangle$ bytes/second. It is the size of ADUs at the source. Each of the process stages a^i is modeled using its computational and outflow requirements per unit of inflow respectively denoted by r_c^i flops/bps and r_d^i bytes/byte. If b_s bps is the rate at which

ADUs are arriving, then $r_c^i \cdot b_s$ flops is the required computational power and $r_d^i \cdot b_s$ bps is the outflow. We assume that the ratios are also applicable of the ADUs.

3 Description of the Test Application

3.1 AVIS Transport

The **Active VVideo Streaming** (AVIS) system appears as a custom transport between a video server and a set of receiver end-points. It is capable of arbitrary transformation of an MPEG-2 ISO-13818 video stream. The transcoder operates using the compute power of a node (or multiple nodes) in the stream's logical pathway. For this test scenario we demonstrate network embedded rate transformation with adaptive behavior at two levels. In the first level, it adapts video rate based on the available link local bandwidth. In the second level it also adapts with respect to the dynamic variation in available compute power in nodes.

3.2 Architecture of the Test Bed Application

For flexible deployment, AVIS have been designed with modular processing components. Also, the most computation intensive tasks can be performed dividedly in multiple nodes. It has three principle worker capsules (i) X-DEC (GT-fork), (ii) X-ENC (processing node) and (iii) X-MUX (GT-joint) [5]. Besides, it also has a service control capsule called AVIS-Manager. An X-DEC module decodes an input video stream to decoded frame images and schedule to distribute among processing paths. An X-ENC module receives decoded frame images and produces a GOP-ed encoding video stream slice. An X-MUX, is used for guaranteeing the transcoded stream is in sequence. Given the end-points and a network map, with an AVIS transport, the modules X-DEC, X-ENC, and X-MUX are logically connected in a pipe. A particular deployment may have more than one X-ENC between X-DEC, and X-MUX. Because of the heterogeneous of network environment, some X-ENCs are expected to run on high performance node, while the others on low powered one. The variation in the (i) active processor speed, (ii) the CPU load in the processors from other active processing, and (ii) the difference and variation on the network link/bandwidth in each path from X-DEC to X-MUX via one of X-ENC causes the variance in delay.

3.3 AVIS on ABONE

Active VVideo Streaming (AVIS) system runs on ABONE system using the Kent VSM/VEE execution environment. ABONE is an operational network and provides an Internet wide network of routing as well as processing capable nodes. The Kent VSM/VEE supports object oriented dynamic module management via dynamic loading and unloading of a service composed as a collection of modules, transfer of service configuration scripts, and log files.

4 Jitter Control

4.1 Multipath Jitter Model

First, we will explain the notation. For denoting the delays we use the following two level notations. We use subscripts to refer to the ADU's sequence number (g) and the path number (p). Each ADU is processed by a set of sub-task modules (M). There can be multiple instances of a module. Each sub-path should have a copy of each module. Also, for some services (such as tree-transcoding in a multicast distribution scenario [9]) a stream can encounter multiple services with recurrence of the whole set of transformations. Modules are ordered and have a stage index. Thus, in the superscript each module M is identified with its stage index (i), service number (s) and the sub-path number (sp) within this service. Thus, let g, p, sp, s denotes respectively the g-th ADU, path number, sub-path number, module name, and the delay stages in a service. Then the delay experienced by an ADU along a path p can be expressed as:

$$D_{g,p} = \sum_m^M (d_{g,p}^{m(i,sp,s)}) \quad (1a)$$

For example the transcoding service is defined by the sub-task processing stages $A \subseteq \{F, P, T\}$. Where, F, P, T respectively represents the computation delays in the fork, connect and joint units. Their orders are 1,2 and 3 respectively. Thus the total delay stages include the communication delays as well:

$$M \subseteq \{F, fp, P, pt, T\}$$

Here, fp and pt represent the communication delays in the first and second stages respectively. Thus the objective of the proposed algorithm is to reduce the variation in inter-departure time from the joint defined by (1b):

$$J = \sum_g^{\text{Stream}} |D_{g,p} - D_{g-1,p}| \quad (1b)$$

The computation delay of each module can be shown as in (2).

$$d_{g,p}^{a(i,sp,s)} = e^i \times r_c^i / B_i^i \quad (2)$$

Here, e^i is the input ADU size in bits, r_c^i is a computation needed for the module in flops per input bits. B_i is the processing power on the node allocated to the service in units of flops. The change in size after the stream flows via a processing capable module is represented by a *stage expansion factor*. The size after the i-th stage is thus:

$$f^i = I \times \prod_{j=0}^i r^j \quad (3)$$

Let f^i is an output and I is an initial input stream size and r^i is a *stage expansion factor* or output bits per input bits of a stage i. Their relation is shown in (4)

$$r^i \times e^i = f^i \quad (4)$$

We can get delay in a link as shown in (5).

$$d_{g,p}^{ij(i,sp,s)} = f^i / B^{ij} \quad (5)$$

Here, f^i is an output stream size as seen in (3) while B^{ij} is a bandwidth of link i.

4.2 Algorithm

4.2.1 Scheduling

Given the streaming rate (R) the algorithm estimates a relative *target arrival time* (T_g) at the destination for each ADU. A quantity *maximum allowed delay* is estimated for each ADU based on this deadline. The algorithm chooses a least weighted time path among the paths which has predicted delay less than maximum allowed delay. When there are multiple conforming sub-paths, the weighted time is a time based on *the average delay time* and the *delay variation* of the sub-path. If no sub-path could process a given ADU within the deadline for it, then the least delay time sub-path is chosen without considering the variations. At the start of the flow, the average delay is initialized to the lowest possible delay of the path and the delay variation is initialized to zero. The *joint* gathers individual delays and delay variations from each sub-path and informs the values to the scheduler in *fork* for subsequent updates.

4.2.2 Delay Estimation

An expected sub-path delay is the sum of (i) expected delay of transmission from fork to the first sub-path processor, (ii) sub-path processing, and (iii) transmission time from last sub-path processor to the joint. The following equation is used for deriving expected delays along each sub-path:

$$\tilde{d}_{g,p}^{sp(i,sp,s)} = \tilde{d}_{g,p}^{fp(i,sp,s)} + \tilde{d}_{g,p}^{P(i+1,sp,s)} + \tilde{d}_{g,p}^{pt(i+1,sp,s)} \quad (6)$$

Links Capacity Estimate: As seen in (5), $\tilde{d}_{g,p}^{fp(i,sp,s)}$ and $\tilde{d}_{g,p}^{pt(i+1,sp,s)}$ can be predicted based on f^i and B^{ij} , but f^i and B^{ij} may vary because of several reasons. The actual compression on each ADU can vary from the ideal compression ratio. The network activities on a link may cause different B^{ij} values from time to time. So, each of the nodes in a sub flow including the joint estimates the average.

$$\tilde{d}_{g,p}^{ij(i,sp,s)} = \tilde{e}^i / \tilde{B}_{g,p}^{ij(i,sp,s)} \quad (7)$$

The bandwidth for each incoming link is approximated by each receiving node, including the fork node, using the method shown in (8).

$$\tilde{B}_{g,p}^{ij(i,sp,s)} = \frac{1}{k} (B_{g,(k-2),p}^{ij(i,sp,s)} \times (k-1) + B_{g,(k-1),p}^{ij(i,sp,s)}) \quad (8)$$

Here k is the number of ADUs which arrived at the receiver node or arrived at the joint using sub-path sp. The $g(k)$ is a k-th ADU number which passed through the sub-path sp. The join estimates the quantity separately for each incoming flow. If the path has no history then last known or initially known bandwidth is used. The right hand side quantities of the equation are observed bandwidth at the joint not prediction.

Processing Capacity Estimate: Similar to the transmission delay, the module delay can also be different from the ideal expected value. Thus, averages is:

$$\tilde{d}_{g,p}^{a(i,sp,s)} = \frac{d_{g(k-2),p}^{a(i,sp,s)} \times (k-1) + d_{g(k-1),p}^{a(i,sp,s)}}{k} \times e^i + Q_{g,p}^{a(i,sp,s)} \quad (9)$$

Equation (9) is for delay of a module a ($a \subseteq A$). It is derived from the *average delay per bit* observed on the previous ADU's on the sub-path sp and the current input ADU size, e^i . Also, in each module, it has a queuing delay, $Q_{g,p}^{a(i,sp,s)}$. In Internet computing all the modules do not operate in identical speed. Each processing module thus maintains an incoming queue of unprocessed ADUs. For AVIS there is negligible queuing delay on the decoder. It is relatively fast comparing with encoding speed. The encoder's queuing delay is given to equation (10).

$$\begin{aligned} Q_{g,p}^{P(i,sp,s)} &= \tilde{d}_{c,p}^{P(i,sp,s)} - (T_c - T_s^{P(i,sp,s)}) + \sum_{w \in W} \tilde{d}_{w,p}^{P(i,sp,s)} \\ \tilde{d}_{c,p}^{P(i,sp,s)} &: \text{expected delay of current encoding ADU, } c, \text{ in encoder in sub-path } sp \\ T_c &= \text{Current time} \\ T_s^{P(i,sp,s)} &= \text{Start time of current encoding ADU, } c, \text{ in encoder in sub-path } sp \\ W &= \text{unstarted ADU scheduled in the node.} \end{aligned} \quad (10)$$

The delay variations of sub-paths are used to select a proper sub-path for a given ADU. Its use provides the worst expected delay time in each path and thus can help in selecting reliable path. Equations (11) and (12) are used to track delay variation.

$$\overline{\text{Avr}}(d_{g,p}^{sp(i,sp,s)}) = \frac{\text{Avr}(d_{g(k),p}^{sp(i,sp,s)}) \times k + \tilde{d}_{g,p}^{sp(i,sp,s)}}{k+1} \quad (11)$$

$$\overline{\text{Var}}(d_{g,p}^{sp(i,sp,s)}) = \frac{\text{Var}(d_{g(k),p}^{sp(i,sp,s)}) \times k + \left| \overline{\text{Avr}}(d_{g,p}^{sp(i,sp,s)}) - \tilde{d}_{g,p}^{sp(i,sp,s)} \right|}{k+1} \quad (12)$$

5 Experiment Results

In test environment, we used total five ABONE nodes, each machines running RedHat Linux 7.1. The nodes received authenticated transcoding service modules from a code server located at KSU Medianet Lab. Those are run on Athlon 1.4GHz, Athlon XP 1700+, and dual Pentium III 450MHz machines. The deployment, management and monitoring process was automatic and adaptive. Selected nodes have different computation powers to make sure that paths have different delay variations in transcoding a video stream. The selected source video streams have identical contents but were initially encoded with different frame and GOP size. There was dynamic variation on the node and link capacities since there were also other activities on the processing capable ABONE nodes.

5.1 Jitter and Delay Reduction

The fig. 3 plots the jitter performance for both with and without the technique. It shows the result of frame size 320x240 and 704x480. The x-axis is the GOP number in the video stream and y-axis are delay jitter in seconds. As seen delay jitters are reduced dramatically with the control scheduling. First few GOPs have more delay jitter variations because the scheduler doesn't know proper initial delays of each path. After some time, however, the scheduler adapts. A bigger GOP causes more delay jitter. This is caused by transcoding method. The encoders start only after all needed decoded video data has arrived. So, a bigger GOP size causes larger wait. Also, a bigger GOP needs more transcoding time than smaller GOP. It magnifies the delay jitter variation of delay. If the encoders can start before all needed video data is transferred to it, it will reduce delay jitter more. Also, the bigger frame stream has larger delay jitter variations because of the same reasons.

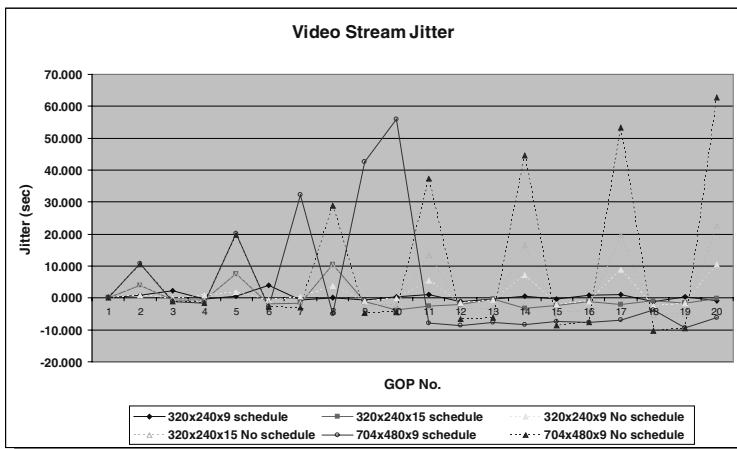


Fig. 3. Video stream jitter measurement

5.2 Cold Start Adaptation

Fig. 4 shows the frame-rate observed in their sample run on a small uncontrolled processing capable network, i.e. with background computational and communication load. We let the system auto deploy itself and find optimum mapping. It plots the performance for both 320x240 and 704x480 frame sizes streams at three different GOP sizes. The computation load heavily depends on the number of macro-blocks or frame size. Based on the frame size the frame transcoding rate varied from 30-5 frames/second. The adaptive behavior is noticeable at the step like increments at the beginning. Initially the channel used only one processing capable node. The single node was unable to sustain the target rate. Soon, it auto-deploys additional nodes.

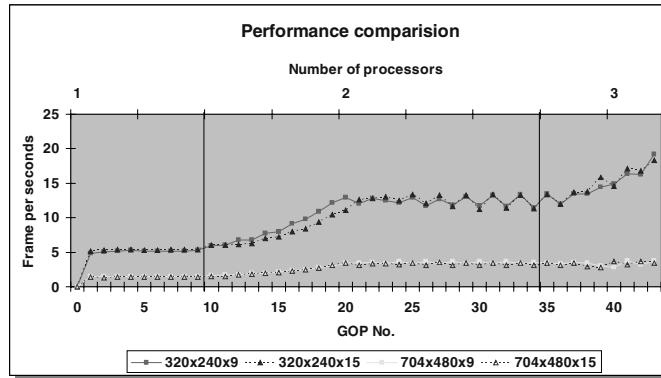


Fig. 4. Frame size and computation

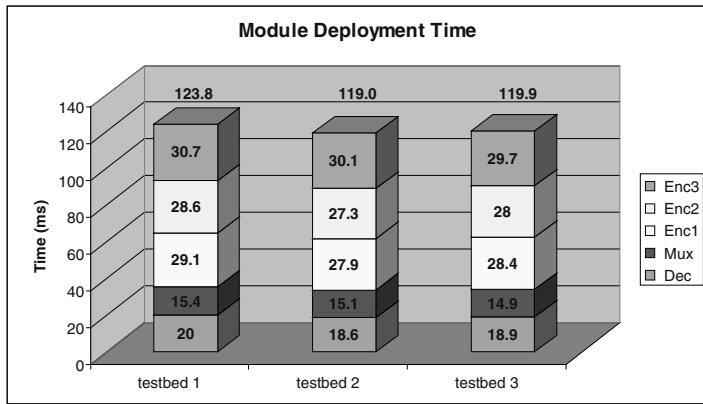


Fig. 5. AVIS system deployment overhead

5.3 System Deployment and Signaling Overhead

We run the system in three test bed scenarios. In the first scenario, the application as well as the AVIS components-- all were deployed in a single Autonomous System's LAN. In the second scenario the application end points (server and players) were in different networks but AVIS computation was performed in a single network. In the third setup application as well as each AVIS component were in distinct. The corresponding module deployment time of each test bed shows in fig. 5. The stack show how much time was taken by individual components of the system. In all three scenarios the total module deployment took about 1.2 seconds. It includes authentication, automatic module transfer and their activation. The entire synchronization was performed by inter modules signals. Clearly a concern was to how much communication resource was consumed by this. The signaling overhead is plotted in fig. 6. It plots individual AVIS components in scenario. For comparison the

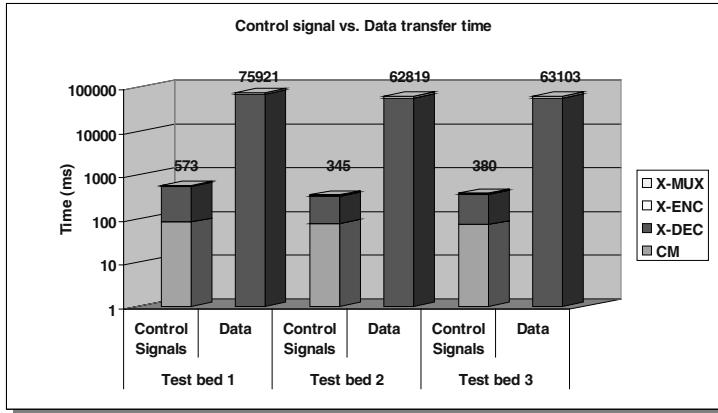


Fig. 6. AVIS system signaling overhead

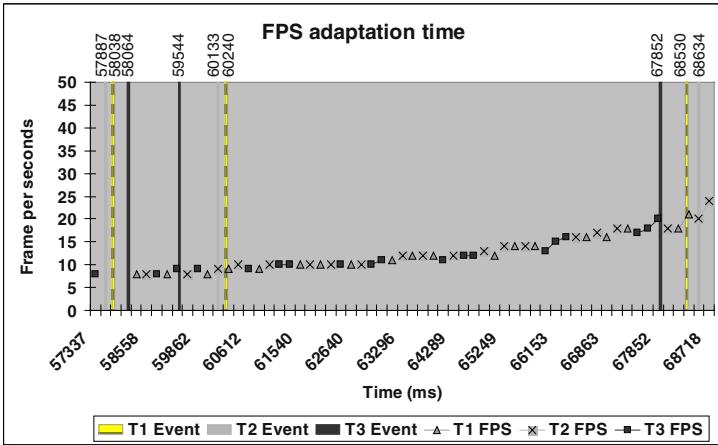


Fig. 7. FPS adaptation reaction time

second bar also shows the actual ADU data volume. Relatively small network resources are used for achieving coordination and controlling among the modules.

5.4 Adaptation Performance

Whenever there is a change in the network condition/ capacity the adaptive system responds. The AVIS system offers two forms of adaptation. The first is with the processing capable nodes and their computing powers. Here we provide an experiment on this (as it involves module reallocation) and emulated incremental allocation of additional CPU power in the processing capable nodes into the system in three steps (events T1, T2 and T3) by changing the *target frame rate* of the AVIS

system. The corresponding change in X-MUX buffers *throughput frame rate* (FPS) observed in the X-MUX unit is shown in fig. 7. It also shows the reaction time. More computation power gave more performance boost as expected. However, the first effects of the events on the throughput were reflected in about 1.5 to 2.2 seconds. It took little more time before the full effects took place.

6 Conclusions

It is interesting to note that the prominent target applications in the network computing paradigms (like grid computing, sensor computing or active networking) targets time sensitive systems such as multimedia, control, or synchronized transactions, where jitter and delay management will be a central issue. The scheme we presented seems to be one of the first to specifically address the issue. However, the temporal QoS required by these applications will be needed in very specific shades and sensitivities. Along with, it is highly unlikely that idealized statistical models (such as assumption of Poisson, Fractal, or Normal arrival), blind to applications', used abundantly in classical attempts, may not be very effective in characterizing AIS where flows are being deliberately manipulated. Therefore, in the design of next generation network for advanced applications it will be of advantage to provision the path scheduling decisions at higher levels- perhaps outside of network layers. The lower network layers should focus more on providing base network parameters to the higher layers to facilitate this decision process. The proposed scheme can be implemented by generic *application level framing* (ALF) with some modification to RTP/ RTCP model. The critical decision that provides the QoS actually lies in the scheduling process. It is possible a different application (such as one which is interested in hard deadline- than jitter) would like to use a different scheduling component in GT-fork. The work has been supported by the DARPA Research Grant F30602-99-1-0515.

References

1. R. Boorstyn, A. Burchard, J. Liebeherr, and C. Oottamakorn. Statistical service assurances for traffic scheduling algorithms. IEEE Journal on Selected Areas in Communications, Special Issue on Internet QoS, 2000.
2. J. Rexford, S. Sen, J. Dey, W. Feng, J. Kurose, J. Stankovic, and D. Towsley. Online Smoothing of Live Variable-Bit-Rate Video. In 7th Workshop Network and Op. Systems Support for Digital Audio and Video, pp. 249–257, St. Louis, MO, May 1997.
3. H. Zhang and D. Ferrari. Rate-Controlled Static-Priority Queueing. In Proceedings of IEEE INFOCOM'93, page 227–236, San Francisco, CA, March 1993.
4. Jeffrey O. Kephart & David Chess, The Vision of Autonomic Computing, IEEE Computers, January 2003, pp.41–50.
5. Javed I. Khan and Seung. S. Yang, A Framework for Building Complex Netcentric Systems on Active Network, Procs of the DARPA Active Networks Conference and Exposition, DANCE 2002, San Jose, CA May 21–24, 2002, IEEE Computer Society Press.
6. Donald L. Stone and Kevin Jeffay, An Empirical Study of Delay Jitter Management Policies, Multimedia Systems Journal, volume 2, number 6, pp. 267–279, January 1995.

7. N. Argiriou and L. Georgiadis, Channel Sharing by Rate-Adaptive Streaming Applications, IEEE INFOCOM'02, New York, June 2002.
8. Jon C. R. Bennett, Kent Benson, Anna Charny, William F. Courtney, Jean-Yves LeBoudec, Delay Jitter Bounds and Packet Scale Rate Guarantee for Expedited Forwarding, IEEE INFOCOM'01, Anchorage, Alaska, April 2001.
9. Seung Yang and Javed I. Khan, Delay and Jitter Minimization in Active Diffusion Computing, IEEE Int. Symp. on Applications and the Internet, SAINT 2003, Orlando, Florida, January 2003.
10. D. Reed, Grids, the Teragrid, and Beyond, IEEE Computers, January 2003, pp.62–68.

An Efficient Heuristic Search for Optimal Wavelength Requirement in Static WDM Optical Networks

Swarup Mandal and Debasish Saha

Indian Institute of Management Calcutta, Kolkata:700104, WB, India

Tel: +91 33 24678300; Fax: +91 33 24678307

{swarup, ds}@iimcal.ac.in

Abstract. This paper presents a heuristic search technique for finding an optimal requirement of wavelengths to satisfy a static lightpath demand in wavelength division multiplexed (WDM) optical networks. We have formulated the problem as a combinatorial optimization problem and used a state space search algorithm based on best first search strategy to solve it. We have compared the performance of our algorithm with another well-known technique with respect to the number of wavelength requirements and call blocking probability. Simulation runs for a wide range of lightpath demands over different WDM networks show that our proposed technique is better in both the above stated dimensions.

1 Introduction

Fiber optic technology [1] holds out the promise of catering to the ever-increasing demand for future bandwidth intensive end-user applications. Wavelength Division Multiplexing (WDM) [2], due to its efficient use of bandwidth, has emerged as the most promising transmission technology for optical networks. It divides the huge bandwidth of an optical fiber into a number of channels, where each channel corresponds to a different wavelength. A WDM optical network consists of optical wavelength routers interconnected by pairs of point-to-point fiber links. A wavelength router receives a message at some wavelength from an input fiber and redirects it to any one of the output fibers at the same wavelength. Hence, a message can be transmitted from one to another node through a wavelength continuous path by configuring the intermediate routing nodes on that path. Such a wavelength continuous path is known as a *lightpath* [3]. The requirement, that the same wavelength must be used on all the links along the selected path, is known as the *wavelength continuity constraint*. Due to this constraint, unique to WDM networks, wavelength channels may not always be utilized efficiently.

1.1 Lightpath Establishment Problem (LEP)

Usually, a lightpath LP_i is uniquely identified by a tuple $\langle \lambda_i, P_i \rangle$, where λ_i is the wavelength used in the lightpath and P_i represents the physical path corresponding to

LP_i . Two lightpaths $LP_1 <\lambda_1, P_1>$ and $LP_2 <\lambda_2, P_2>$ can share the same fiber, if and only if they use different wavelengths i.e., $\lambda_1 \neq \lambda_2$. The problem of establishing lightpaths, with the objective of minimization of the required number of wavelengths to satisfy a set of lightpath demand or minimization of the lightpath blocking probability for a fixed number of wavelengths, is termed as the *lightpath establishment problem* (LEP) [4]. Usually, lightpath establishment is of two types. One is *static* LEP(SLEP) or fixed [4], where a set of lightpaths and their wavelengths are identified a priori. Another is *dynamic* LEP(DLEP) [4], where lightpath management is on-demand, i.e., they are established and terminated on the fly. A good survey of the existing formulations for LEP can be found in [3].

The static lightpath establishment problem (SLEP) is also known as routing and wavelength assignment (RWA) problem in the literature. The problem is conventionally formulated as a mixed-integer linear program, which is shown to be NP complete [4]. In order to make the problem more tractable, SLEP is normally partitioned into *two* sub-problems viz. *routing*, and *wavelength assignment*. Each subproblem can be solved separately. In the previous works [5],[6],[7],[9] none of the authors tried to find an optimal number of wavelengths per link required to satisfy a given set of static lightpath demand in a given WDM optical network. This motivates us to look for a heuristic search technique, which can solve the problem with an objective of finding optimal number of wavelength while reducing the call blocking probability.

1.2 Proposed Study

In this paper we have tried to solve the problem to find an optimal number of wavelength required to satisfy a set of static lightpath demand by using an efficient routing scheme to be run on a physical WDM network. Any wavelength assignment algorithm may be combined with it to solve the second sub-problem viz. *wavelength assignment* [3],[4],[5],[6],[7] [9],[10]. The network architecture, considered in this work is identical to the architecture as described in [5].

1.3 Organization of Paper

This paper is organized in five sections. Following introduction in Section 1, a mathematical formulation of the problem is presented in Section 2. Section 3 is devoted to the description of heuristic search. Section 4 contains the experimental results, and Section 5 concludes the paper.

2 Mathematical Formulation

Our essential aim is to serve the requested lightpaths with the minimum number of wavelengths. This could be done by an efficient wavelength routing scheme over a physical WDM network. For describing the problem let us consider the following notations:

- s and d denote *source* and *destination* of a lightpath when used as a superscript or subscript.
- i and j denote *originating* and *terminating* nodes of a lightpath
- N= Number of nodes in the physical topology.
- $M = (M_{(p,q),(l,m)})$,
where $M_{(pq),(lm)} = 1$, if connection (p, q) and connection (l, m) use a common link
 $= 0$, otherwise
- W = Number of wavelengths on each fiber.
- $z_{pq}(w) = 1$, if node p and node q are connected through wavelength w
 $= 0$, otherwise
- C = Capacity of each wavelength channel in bits/sec.

Objective:

This objective function minimizes the requirement of wavelength necessity for the requested lightpaths i.e. *Minimize W*

Subject to the following constraints:

The maximum capacity of the optical channel between p and q be $\sum_{w=1}^W z_{pq}(w)C$

If, $\sum_{w=1}^W z_{pq}(w)C = 0$ no optical channel is set up between p and q.

$$\sum_{m=1}^N z_{pm}(w)C \leq 1 \text{ and } \sum_{m=1}^N z_{mq}(w)C \leq 1 \text{ where } p,q = 1,2,\dots,N \quad (1)$$

$$z_{pq}(w) + z_{lm}(w) \leq 1 \quad (2)$$

where w = 1, 2, ..., W for all distinct pairs (p,q) and (l,m)

$$z_{pq}(w) = 0 \text{ or } 1 \text{ w } = 1,2,\dots,W \text{ and } i,j = 1,2,\dots,N \quad (3)$$

Lightpath demand can be satisfied

$$t_{ij} > 0 \text{ where } i,j = 1,2,\dots,N \quad (4)$$

Here, N is number of nodes in the physical topology.

For each wavelength, constraint (1) states that there should be at most one connection starting from node p and at most one connection terminating at q, respectively. Constraint (2) indicates that, if connections (p, q) and (l, m) use a common link, $z_{pq}(w)$ and $z_{lm}(w)$ can not be equal to 1 at the same time, i.e., connections (p, q) and (l, m) can not use the same wavelength w at the same time. Constraint 3 indicates that, if $z_{pq}(w) = 1$, then nodes p and q are connected through the wavelength w. Constraint 4 indicates that lightpath demand will be satisfied if t_{ij} becomes greater than 0.

3 Heuristic Search

Any combinatorial optimization problem can be formulated as a state space search problem, which can be solved by heuristic search techniques. In this work, we have used the proposed search technique based on Best First Search [11]. When an algorithm, developed based on Best First Search and uses admissible heuristic [11] runs to its termination, it terminates with an optimal solution [11].

3.1 State Space Representation

The state space as designed for the SLEP is described below:

Let LD be a set of lightpath demand and given by $\{LD_1, LD_2, \dots, LD_n\}$ and LD_SAT be a set of lightpath established to meet the demand LD such that $LD_SAT \subseteq LD$. A state (node) in the state space is represented by the sets LD and $LD_UNS = LD - LD_SAT$. The start node of the state space tree will have LD equals to LD_UNS and LD_SAT equals to null. The goal state is represented as a state in which LD_UNS equals to null and LD_SAT equals to LD. Each state in the level i of the tree represents a distinct possible route path in physical network to satisfy the lightpath demand LD_i . A node n in level i in the state space tree will have cost with two cost components viz. actual cost, $g(n)$ and a heuristic cost $h(n)$. Here $g(n)$ represents the total number of wavelengths required to satisfy lightpaths corresponding to level 1 to level (i-1) in the tree. $h(n)$ represents the minimum number of wavelengths to satisfy the unsatisfied lightpath demand in state n. Let us consider the following Table-1 and Fig.1 to describe the state space as shown in the Fig. 2.

Table 1. Traffic demand for a network

0	4	13	0
3	0	32	0
1	6	0	0
10	0	6	0

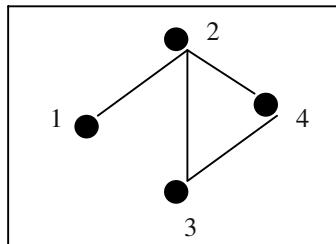


Fig. 1. Physical connection for a WDM network

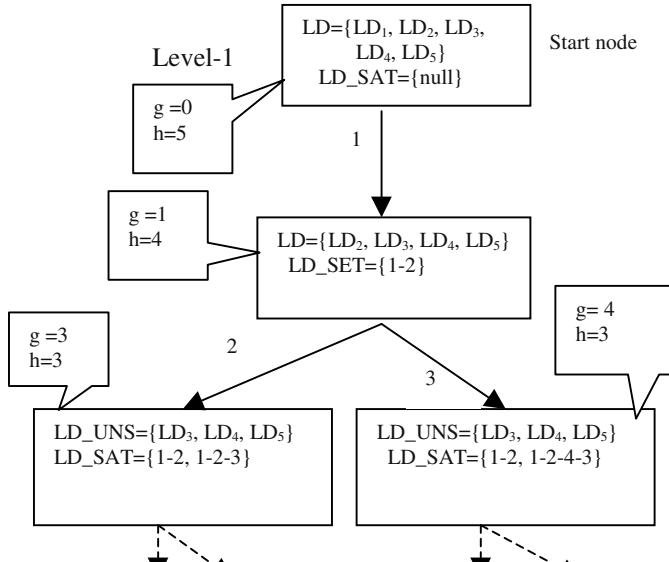


Fig. 2. State space

From the above formulation it is clear that maximum depth of the tree is equal to the number of lightpath demand and each leaf node of the tree is a goal node. Tracing the lightpath along the optimal path (a path in the tree from start node to an optimal goal node), we can find out the number of wavelengths required in each link of a physical WDM network. The maximum number of wavelength in any link in the physical network is taken as the requirement of wavelength per link. Historical traffic demand is assumed as indicated in Table 1 for a physical network as indicated in Fig. 1. A lightpath between a pair of nodes will be set up if there is non zero traffic demand between that pair of nodes.

3.2 Heuristic

Let us consider a state where LD_UNS has m number of unsatisfied lightpath demand. Heuristic value of a state is defined as the minimum total number of wavelengths required to satisfy the unsatisfied lightpath demand. Here, total number of wavelength is represented as total number of hops in the route paths in the physical network corresponding to the unsatisfied lightpath demands (Since, each link in the physical network within a light path should be provided with a wavelength to set up the lightpath). The heuristic as defined above will give a lower bound value of the total number of wavelengths required. As for example the heuristic value of the node in level 2 in Fig. 2 is derived as 4. This is due to the fact that there are four unsatisfied lightpath demand in that state. To satisfy those unsatisfied demand, we will have to find four lightpaths and number of links in each lightpath will be at least one. Thus, minimum total number of wavelengths will be 4 (one wavelength in each link corresponding to 4 lightpaths will give $4 \times 1 = 4$ wavelengths).

3.3 Description of the Algorithm, D3S

The algorithm is a suitable adaptation of Best-First Search [11]. It performs Best-First Search in the first phase and performs Depth-First Search [11] in the second phase. We are now presenting the algorithm in this subsection. In the algorithm a linear LIST is used to store the generated nodes.

```

Procedure D3S ( )
{
    Initialize the LIST with start node
    current_solution:= NULL
    current_solution_cost:= 0
    While(LIST is not empty OR current_solution≠NULL)
        Begin
            Remove the first node n from LIST.
            If n is goal node
                then set: current_solution:= n
                        current_solution_cost:=
                            0
            Else
                Expand n by generating all its children
                and sort the children nodes in ascending
                order of total cost (=g+h) before
                inserting them at the beginning of LIST
        End
    While(LIST is not empty)
        Begin
            Remove nodes from the front of LIST one by
            one and perform Depth First Search
            starting at the removed node, cutting off
            each generated path when its cost equals
            or exceeds current_solution_cost. If a
            solution is found with cost less than the
            current_solution_cost, then update
            current_solution_cost and current_solution
        End
    Output current_solution and current_solution_cost
}

```

4 Results

We have compared the performance of our algorithm with the performance of the technique proposed by Zhang and Acampora [5] (now onwards it is indicated as Z-A algorithm in this paper) with respect to: a) *wavelength requirement* to satisfy a set of static lightpath demands in a given WDM network, and b) *call blocking probability*

for randomly generated calls (when routed on a virtual topology) as found out by respective algorithms, while satisfying the above set of static lightpath demand.

For comparison, we have generated the data set considering the following traffic model:

Calls at node i are generated independently of calls originated at other nodes according to a Poisson process with rate $\lambda \sum_{i=1}^N t_{ij}$. A call destined to node j has the probability given by

$$\text{Prob} = \frac{t_{ij}}{\lambda \sum_{i=1}^N t_{ij}} \quad (5)$$

where λ denotes call arrival rate. We assume that call durations are independent and identically governed by an exponential distribution with mean $1/\mu$. All the calls are assumed to have the same bandwidth b , measured in the same units as the optical channel capacity.

Let us define the call granularity as the number of calls supported on one wavelength (optical channel) which is given by g .

$$g = \frac{\text{capacity per wavelength}}{\text{Bandwidth required per call}} = \frac{C}{b} \quad (6)$$

Let n_{ij} denote the number of ongoing calls between nodes i and j at the moment when a new call arrives. Let n_c denote the access node capacity in units of C , where $1 \leq n_c \leq P$ (P = total available number of wavelengths).

A call is admitted if (7) is satisfied and a path with sufficient bandwidth (found through either routing scheme)

$$1 + \sum_{i=1}^N n_{ij} \leq n_c g \quad \text{and} \quad 1 + \sum_{j=1}^N n_{ij} \leq n_c g \quad (7)$$

Here, the offered load is calculated by

$$\rho = \lambda \max_i \left(\sum_{j=1}^N t_{ij} \right) / \mu \quad (8)$$

Thus the average call blocking probability is defined as:

$$P_{B\text{ avg}} = \frac{\text{Total number of calls blocked}}{\text{Total number of calls generated}} \quad (9)$$

For comparison, we consider the following ideal centralized switch. There are N input ports and N output ports in the switch. The capacity of each input/output port is $n_c C$. A new call arriving on input link i and destined to output link j is admitted only if (7) is satisfied. The centralized switch gives the lowest possible call blocking, caused exclusively by congestion on user/node input/output links, never by the switch fabric itself. The call blocking probability of the centralized switch for the same traffic model is obtained through simulation. We also assume that all optical channels run at some common data rate C , and that the capacity of each access node is an integer

multiple of C (that is the access node's capacity is some integral multiple of the basic optical channel rate; if the optical channel rate is 2.4Gb/s then the node access rate may be 2.4Gb/s, 4.8Gb/s, 7.2Gb/s etc.) The node capacity (normalized by the optical channel's capacity) should be less than or equal to the number of wavelengths available. We impose an access node capacity to determine if the blocking occurs at the access node or inside the network. In all the cases call granularity was taken as 10.

4.1 Wavelength Requirement

Here we have compared the performance of D3S algorithm with the performance of Z-A algorithm for varying static lightpath demand in the network considered in [5]. To find the wavelength requirement (WR) for varying static lightpath demand ranging from 75 to 200 simulation runs are taken and results are plotted as shown in Fig.3. It is seen that the wavelength requirement varies from 9 to 16 and 9 to 25 for D3S algorithm and Z-A algorithm respectively. The percentage of savings on wavelength requirement by D3S over Z-A is 33% (approx.) for a lightpath demand of 200.

From the plot it is found that the percentage of saving is increasing with the increase of lightpath demand. This could be explained as follows. Since Z-A algorithm finds a virtual topology using shortest path algorithm while giving priority to the pair of node having maximum traffic demand t in [5], it simulates greedy search technique and end up getting a virtual topology for which wavelength requirement per link is not the optimal one. On the other hand, D3S checks all possible combinations of paths in the physical network to satisfy the lightpath demand for all pair of nodes and it finds the virtual topology for which the number of wavelengths per link is an optimal one. Thus, D3S reduces wavelength requirement and increases wavelength utilization (as same lightpath demand is catered by reduced number of wavelengths) when it is to meet a set of static lightpath demand.

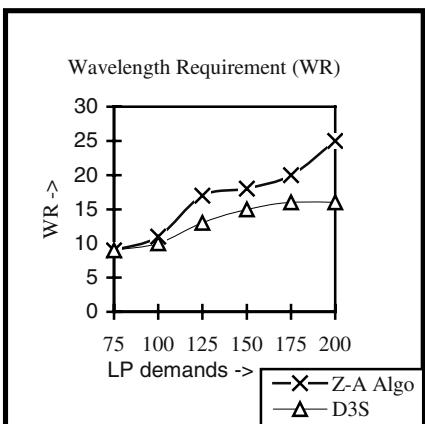


Fig. 3. Plot for WR versus LP demands

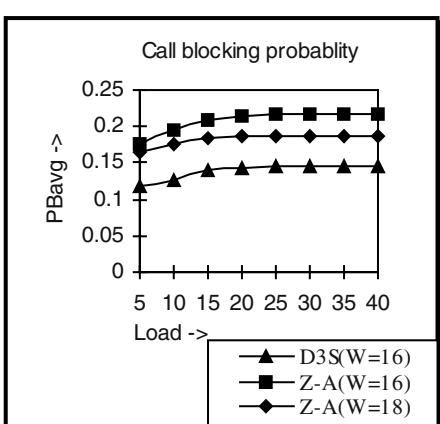


Fig. 4. Blocking probability versus offered load

4.2 Call Blocking Probability

Here we have considered the call blocking probability for a real network [5] with traffic demand matrix for this network [5]. Virtual topology is obtained by providing connection between nodes i and node j if t_{ij} is non-zero. For call blocking analysis the routing strategy as indicated in [5] is used over the virtual topology formed by the respective techniques viz. D3S and Z-A.

We have generated total 20995 calls according to the method described in traffic model for the network [5] and have taken a value of g equal to 2. For our proposed algorithm we have taken optimum number of wavelengths as found by it while satisfying the given set of lightpath demands (Corresponding to the matrix t). Since Z-A algorithm does not find the number of wavelength per link, we have taken the wavelength per link value as equal or higher than what our proposed algorithm, D3S found. In each of the above cases call blockings are found for varying loads (8).

The results as found in simulation is plotted and the graph is presented in Fig. 4. It is seen that when number of wavelength per link is 16 then for the loads ranging from 5 to 40, the call blocking probability (PB_{avg}) varies from 0.17 to 0.21 and 0.12 to 0.14 for Z-A and D3S respectively. Even if the number of wavelengths per link is increased to 18, the PB_{avg} of Z-A does not match with that produced by D3S. It is also apparent that the call blocking probability increases almost linearly with load until load equals to 40 and for loads higher than 40, it is almost constant.

5 Conclusion

In this paper, we have presented an efficient heuristic search technique, D3S, which can be employed for routing on physical network to satisfy a set of static lightpath demand in WDM optical networks. Any static wavelength assignment technique may be used on the virtual topology found by D3S to route a call. D3S is capable of finding an optimal number of wavelengths, required to satisfy a set of static light path demands. For the network and lightpaths demand D3S finds WR which 64% of WR found by Z-A. Again, D3S algorithm shows a call blocking probability, which is 67% that (for a load of 40) Z-A algorithm. Thus, the experimental results show that D3S is better than Z-A with respect to both the wavelength requirement per link and the call blocking probability when they are deployed to solve SLEP.

References

1. P. E. Green, "Progress in Optical Networking", IEEE Communications Magazine, pp. 54–61, January 2001.
2. B. Mukherjee, "WDM Optical Communication Networks: Progress and Challenges", IEEE Journal on Selected Areas in Communications, Vol. 18, No. 10, pp. 1810–1824, October 2000.
3. R. Dutta, G. N. Rouskas, "A Survey of Virtual Topology Design Algorithms for Wavelength Routed Optical Networks", Optical Networks Magazine, pp. 73–89, January 2000.

4. H. Zang, J. P. Jue and B. Mukherjee, "A Review of Routing and Wavelength Assignment Approaches for Wavelength Routed Optical WDM Networks", Optical Networks Magazine, pp.47–60, January 2000.
5. Z. Zhang and A. S. Acampora, "A Heuristic Wavelength Assignment Algorithm for Multihop WDM Networks with Wavelength Routing and Wavelength Re-Use", IEEE/ACM Transactions on Networking, Vol.3, No.3, pp. 281–288, June 1995.
6. S. Banerjee, J. Yoo and C. Chen, "Design of Wavelength – Routed Optical Networks for Packet Switched Traffic", Journal of Lightwave Technology, Vol.15, No. 9, pp. 1636–1646, September 1997.
7. R. Ramaswami and K. N. Sivarajan, "Routing and Wavelength Assignment in All-Optical Networks", IEEE/ACM Transactions on Networking, Vol.3, No.5, pp. 489–500, October 1995.
8. M. Pollack "The K'th best route through a network," Opel'. Res., vol. 9, no. 4, pp. 578–579, 1961.
9. S. Jana, D. Saha, A. Mukherjee and P. Chowdhury," A Fast Approach for Assigning Wavelengths in WDM All Optical Networks(AON) ", IWDC 2002 Conference, Kolkata, India, Dec., 2002
10. S. Jana, S. Chowdhury, D. Saha, A. Mukherjee and P. Chowdhury," Assigning Wavelengths in WDM Optical Networks: A Novel Dynamic Approach" Proc. 6th International Conference HPC Asia, Bangalore, India, Dec., 2002.
11. Nils J. Nilsson, "Artificial Intelligence: A New Synthesis", Morgan Kaufmann Publisher, 1998.

Slot Allocation Schemes for Delay Sensitive Traffic Support in Asynchronous Wireless Mesh Networks*

V. Vidhyashankar, B.S. Manoj, and C. Siva Ram Murthy

Department of Computer Science and Engineering
IIT Madras 600036, INDIA

vidya@dcs.cs.iitm.ernet.in, bsmanoj@cs.iitm.ernet.in, murthy@iitm.ernet.in

Abstract. In this paper, we propose an on-demand QoS routing protocol and heuristics for the slot allocation process in asynchronous single channel wireless mesh networks in the presence of hidden terminals. The heuristics we propose are the Early Fit Reservation (EFR), Minimum Bandwidth Reservation (MBR), and the Position-based Hybrid Reservation (PHR). The heuristic EFR has been found to give the best performance in terms of delay while the PHR and the MBR provide a better throughput. The heuristics proposed above have been adapted to provide an extended battery life for the power constrained mobile nodes and hence reduce the number of battery recharges. The parent and the adapted heuristics are compared in terms of delay and number of battery recharges. Simulation studies have shown that the parent heuristics show better results with respect to delay while the adapted heuristics perform better in terms of the number of *deaths* of mobile nodes.

1 Introduction

One of the major challenges in multihop wireless networks that exists today is that of providing Quality of Service (QoS) guarantee for real-time services. The complexity of the problem arises due to the unpredictability of the network topology caused by the mobility of the nodes. A path between any two nodes in a multihop wireless network can be obtained either by a static table-driven routing approach such as the Destination Sequenced Distance Vector (DSDV) [1] protocol, or by a dynamic on-demand routing approach such as the Dynamic Source Routing (DSR) [2] protocol. In a multihop environment that supports time-bound services, one has to not only set up a path with the minimum length but also reserve bandwidth so as to ensure delivery of data within the prescribed deadline. Hence reservation of bandwidth resources for real-time communications will aid a QoS-guaranteed packet delivery.

A heterogeneous multihop wireless network (wireless mesh network) is a kind of Ad hoc wireless networks that operates with partial infrastructure. Wireless

* This work was supported by Infosys Technologies Ltd., Bangalore, India and Department of Science and Technology, New Delhi, India.

mesh networks provide an alternative communication infrastructure to existing cellular networks. Wireless mesh networks consist of a set of resource-constrained mobile nodes that want to communicate with each other and a set of fixed relay nodes that are equipped with more resources and are dedicated to forward data and possibly serve as gateways to the Internet.

Our work in this paper, involves development of a QoS routing protocol and implementation of slot allocation schemes for mesh networks in an asynchronous environment. By asynchronous environment, we mean that the entire network is not synchronized to a global clock as required in a TDMA system, for synchronizing the super-frame. A node uses relative time information to convey the exact positions of the reservation slots to a neighbor node, much similar to RT-MAC [3]. To distinguish the mobile nodes from the fixed relay nodes, we assume that the former have a finite battery life and may have to recharge their batteries whenever they get discharged, while the latter are assumed to have enough power to sustain for a long period of time. We have modified the DSR protocol [2] to enable it to support QoS routing. We have proposed three heuristics for the slot allocation algorithm and studied their performance using simulations.

This paper is organized as follows. Section 2 describes the related work in this area. Section 3 describes the slot allocation framework and the heuristics. Section 4 presents results of the simulation of our scheme while Section 5 summarizes the paper.

2 Related Work

QoS issues in Ad hoc wireless networks have been studied in several works such as [4], [5], [6], and [7]. But many of these works have assumed the existence of either a multi-antenna model as in [8], which counters the effect of interference, or a CDMA-over-TDMA model as in [5], [6], and [7]. In the CDMA-over-TDMA infrastructure, multiple sessions can share the TDMA slot using different spreading codes. In many of these works, a transmitter-based assignment scheme is used to assign a code to each transmitter for data transmission. Different spreading codes are assigned to those nodes within two hops so as to reduce the hidden terminal problem.

C. R. Lin proposed a table-driven QoS routing mechanism in [5] and an on-demand call admission control scheme in [6] for a TDMA environment. The scheme in [5] reserves slots for real-time traffic using a QoS extension of the DSDV protocol. The bandwidth reservation problem in a TDMA-based scheme has been shown to be equivalent to the satisfiability (SAT) problem which is known to be NP-complete [9]. Hence a heuristic has been proposed in [5] for assigning slots to each node. This protocol suffers from a high blocking probability. The work in [6] proposes an extension of the strategy to an on-demand routing protocol. This yields a better performance than the protocol proposed in [5] as several routes are attempted in parallel. The work in [7] describes a bandwidth allocation algorithm performed at the destination node for every link in the path implemented with an on-demand routing protocol in a CDMA-over-

TDMA based synchronous environment. The algorithm in [7] suffers from the way the links are chosen at every iteration of the algorithm, *i.e.*, when links have the same minimum current bandwidth, the tie is broken randomly. All these protocols, as mentioned before, did not have the need to address the issue of the hidden terminal problem as it had already been taken care of by the hardware mechanism. Moreover the underlying assumption of synchronization exerts pressure on the scarce resources like bandwidth and battery power.

Protocols Used in Our Work: The algorithms proposed in this work are applicable to both synchronous TDMA-based and asynchronous TDMA-based Ad hoc wireless networks. We have chosen an asynchronous environment owing to two major reasons. One is the fact that synchronous environments like the TDMA are expensive in terms of both battery power and bandwidth unlike asynchronous ones. The other is the problem of frequent partitioning and merging that occurs often in Ad hoc networks. In the synchronous case, apart from the synchronization overhead, there is bound to be some calls getting dropped due to collisions or synchronization differences during the synchronization period.

In order to provide call admission control and reservation of bandwidth in the asynchronous environment, we require a MAC protocol which has the respective capabilities. This is the reason for selecting the Real Time MAC (RTMAC) protocol proposed in [3], a bandwidth efficient, flexible and asynchronous MAC protocol that supports both real-time (RT) and best-effort (BE) traffic. In RT-MAC, bandwidth is provided by dividing the transmission time into successive super-frames. The bandwidth reservations for RT traffic are made by reserving variable-length time slots (*conn-slots*) on the super-frames. The slot allocation differs from the TDMA scheme since no time synchronization has been assumed by the authors of [3] and all reservations are done with respect to the relative time period by which the RT session starts. The reservation is performed through a three-way ResvRTS-ResvCTS-ResvACK handshake. In our work, QoS routing has been implemented as an extension of the DSR protocol [2]. We have modified the protocol by piggybacking on the *Route Request* packets, the reservation information along with the relative time information for calculating the path bandwidth.

3 Slot Allocation Framework

The wireless mesh network has been implemented in an asynchronous environment with QoS-DSR as the routing protocol and RTMAC as the MAC protocol. The QoS routing protocol has three phases of operation *viz.*, the *Bandwidth Feasibility Test Phase*, the *Bandwidth Allocation Phase*, and the *Bandwidth Reservation Phase*. These phases will be discussed later in this section.

Power Management

The mobile nodes in the mesh network have finite battery power. Since transmission involves maximum power consumption, we assume that the battery discharges only during transmission. The battery of the node may experience

a charge recovery [10] when the node is idle and recharges whenever it is discharged fully. Once it is fully discharged, we assume that there is a short time period during which the node is unable to operate. The node is then said to have attained a *dormant state* or a *death* is said to have occurred. The relay nodes, on the other hand are assumed to have infinite power as they have a supportive infrastructure. Recent studies on storage cell characteristics [10] show that a pulsed discharge can extend the battery life more than a continuous discharge. We aim to reap the benefits of the pulsed discharge model by strategically placing reservations so as to (possibly) enable recovery during the resulting idle periods. An exponentially decreasing function ([10] and [11]) depending on the current state (the discharge potential and the theoretical capacity) is used to model the recovery. According to the recovery model, the battery may recover when the node is idle. The battery reaches a *dormant state* either when its battery voltage has reached the cut-off voltage (V_{cut}) or when the theoretical capacity has been exhausted.

The QoS-DSR Protocol

In this section, we present a discussion on the routing protocol and the slot allocation schemes.

The Bandwidth Feasibility Test Phase: During this phase, the selection of paths with the required bandwidth is performed. This is achieved during the *Route Request* propagation. Whenever a particular node receives a *Route Request* packet, it checks for bandwidth availability in the link through which it is received. If sufficient bandwidth is available, then the *Route Request* is forwarded, hence avoiding the paths which cannot support the bandwidth requirement. Before forwarding, the node synchronizes its reservation information with the already present reservation information (of the path traversed so far) in the *Route Request* packet that it received. The updated synchronization information is piggybacked on the new *Route Request* packet which is then broadcast. The reservation information is in the form of reservation frames containing time durations. Each of these time durations associated with a state variable indicates whether a slot is reservable during that duration or unreservable because the node and (or) at least one of its neighbors are involved in a call. Of the *Route Request* packets collected within a time interval (called *RouteRequestWindow*), the destination node chooses the one with the maximum number of fixed relay nodes in the path. Other *Route Request* packets that arrive beyond the *RouteRequestWindow* are discarded.

The data structure used for the bandwidth allocation algorithm is called the QoS frame. The reservation information of each link provided in the *Route Request* packet, is used for constructing the QoS frame. It is a linear array of blocks, each block containing a time duration and the state to which it belongs. The state is of the form ASYNC_X.Y where X and Y correspond to the states of the sender and receiver node of the link respectively (refer to Figure 1). X and Y can be one of FREE (the medium is free in its vicinity), UNRESV (when the medium is used by the node either as a sender or as a receiver) or HT (when

the node serves as a hidden terminal to a transmission involving one or more of its neighbors).

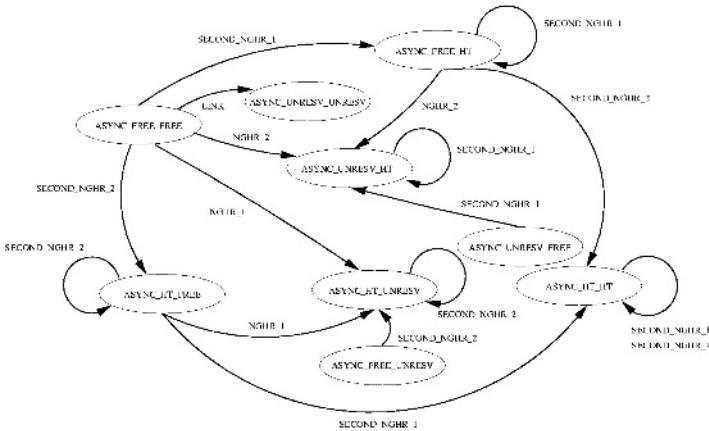


Fig. 1. State diagram describing the *UpdateState* function. The transitions are labeled, each label referring to the nature of the link *viz.*, LINK (the link for which the reservation is being made), NGHR_1 and NGHR_2 (the first neighboring links), and SECOND_NGHR_1 and SECOND_NGHR_2 (the second neighboring links).

The Bandwidth Allocation Phase: During this phase, the destination node performs a link-wise *conn-slot* allocation algorithm to assign the *conn-slots* at every link. The input to the algorithm is the set of reservation frames constructed from the reservation information in the *Route Request* packet. These frames correspond to those of the links of the path. The underlying principle in the algorithm is that the bandwidth of a link is reserved by considering its frame along with the two levels of neighboring links on both the sides of this link. This is done for every link. The algorithm tries to avoid overlapping of reservation in neighboring frames. By doing this, we try to alleviate the hidden terminal problem and allot the necessary bandwidth at the same time. The heuristics, in general, vary with respect to the selection of the link at every iteration of the algorithm. A guard band interval is included on both the sides of the *conn-slot* to cushion the error incurred in synchronization. The algorithm returns the time durations allocated for each link if it is successful, otherwise returns NULL. The function *Resv_Bandwidth* presented in Figure 2 contains the generic bandwidth allocation algorithm. A number of (equal to the number of nodes in the path) invocations of the *GetAvailable_Bandwidth_Info* function which is presented in Figure 2 finally produces the frame *ResultantFrame* which contains the free time durations during which a *conn-slot* may be reserved. The *UpdateState* function used in the algorithm shown in Figure 2 can be understood from the state diagram in Figure 1. The states correspond to those of each time instant in the QoS frame of each link. Hence, all states shown in the diagram qualify to be

initial states (however, the function cannot be called when the state for the time instant is ASYNC_UNRESV_UNRESV). A state transition occurs only for those time instants for which the UpdateState function has been called, also depending on the current nature of the link specified as a parameter of the function. The heuristics proposed are described as follows.

- (i) **Early Fit Reservation (EFR):** The EFR heuristic assigns bandwidth link-by-link starting from the sender to the receiver. At every link, the EFR tries to allocate the first available free *conn-slot* after the *conn-slot* reserved for the previous link. The iterations are performed starting from the first link (that involves the source node), then proceeding with the successive links of the path. This heuristic tries to reduce the end-to-end delay of data delivery.
- (ii) **Minimum Bandwidth-based Reservation (MBR):** The MBR heuristic allocates bandwidth to the links in the increasing order of free *conn-slots* i.e., the link with the least free bandwidth is considered first. At every link, MBR allocates the first free *conn-slot* available. At each iteration, the frame that has the minimum reservable bandwidth will be taken into consideration. Thus we are trying to improve the probability with which the algorithm is successful. The reservation for each link is done by traversing from the beginning of the frame.
- (iii) **Position-based Hybrid Reservation (PHR):** The PHR heuristic assigns bandwidth for every link with the slot placement proportional to its position in the path. The operation of this heuristic is similar to the previous one. The difference lies in the selection of the *conn-slot* for reservation at each link. The link with the least free bandwidth is selected for each iteration. We observe that this heuristic tries to reap the benefits of both the EFR and the MBR heuristics. Thus, the earlier links in the path are reserved (at a greater probability) in the beginning of the frame while the later links in the path are reserved towards the end. Hence this type of reservation yields a lesser delay than the MBR heuristic.

To adapt these heuristics to mesh networks, we have modified them to EFRMesh, MBRMesh, and PHRMesh. The idea of the adapted heuristics is to allocate a free time duration (hole) in a link just before a *conn-slot* during the bandwidth allocation algorithm, whenever the sender node of the link is a mobile node. In other words, when a *conn-slot* has to be allocated to a link, the nature of the sender node is verified. If it is a mobile node, a *conn-slot* is assigned such that there is a hole (*recovery-hole*) just before it. However, if the sender node of the link is a fixed relay node, the *recovery-hole* is not allocated. The *recovery-hole* can possibly be used by the mobile node for the battery to recover. Only the sender node is checked since the power management has been addressed only with respect to transmission in our model. Hence, the adaptive heuristics try to reduce the possibility of the node reaching a *dormant state* at the cost of fragmenting the reservation frame.

The Bandwidth Reservation Phase: If the bandwidth allocation algorithm is successful in its reservation, information about the bandwidth allocation will

GetAvailable_Bandwidth_Info(QoSFrame P, QoSFrame Q, States typeOfIntersection)

1. Let $t_k \leftarrow 0$, $t_{prev} \leftarrow 0$.
 //Let $State(Q_i(t_k, t_l))$ return the state of the frame Q in the time interval (t_k, t_l) .
 //Let SF denote the total super frame time and R denote the output frame.
2. Find the minimum interval (t_k, t_l) such that in none of the frames P and Q is there a state transition.
3. switch(typeOfIntersection)
 - a) case FIRST-TWO:
 If ((State(P, (t_k, t_1)) IN { ASYNC_FREE_FREE, ASYNC_FREE_HT, ASYNC_HT_FREE, ASYNC_HT_HT, ASYNC_UNRESV_FREE, ASYNC_UNRESV_HT }) ^
 (State(Q, (t_k, t_1)) IN { ASYNC_FREE_FREE, ASYNC_HT_FREE })) then
 SetState(R, (t_k, t_1) , ASYNC_FREE_FREE)
 else SetState(R, (t_k, t_1) , ASYNC_UNRESV_UNRESV).
 - b) case LAST-TWO:
 If ((State(Q, (t_k, t_1)) IN { ASYNC_FREE_FREE, ASYNC_FREE_HT, ASYNC_HT_FREE, ASYNC_HT_HT, ASYNC_FREE_UNRESV, ASYNC_HT_UNRESV }) ^
 (State(P, (t_k, t_1)) IN { ASYNC_FREE_FREE, ASYNC_FREE_HT }) then
 SetState(R, (t_k, t_1) , ASYNC_FREE_FREE)
 else SetState(R, (t_k, t_1) , ASYNC_UNRESV_UNRESV).
 - c) case MIDDLE-TWO:
 If ((State(Q, (t_k, t_1)) = ASYNC_FREE_FREE) ^ (State(P, (t_k, t_1)) = ASYNC_FREE_FREE))
 then SetState(R, (t_k, t_1) , ASYNC_FREE_FREE)
 else SetState(R, (t_k, t_1) , ASYNC_UNRESV_UNRESV).
4. If $State(R, (t_k, t_1)) = State(R, (t_{prev}, t_k))$ then Merge(R, (t_{prev}, t_k) , (t_k, t_l))
 else $t_{prev} \leftarrow t_k$.
 //The Merge function merges the previous duration (t_{prev}, t_k) and the current duration (t_k, t_l) into one (t_{prev}, t_k) with the same state.
5. $t_k \leftarrow t_l$.
6. If $t_l < SF$ then goto Step 2.
7. Return R.

The Generic Algorithm**Resv_Bandwidth(QoSFrames, pathLength, t_{RT})**

- 1) $Q_i \leftarrow SelectQoSFrame(QoSFrames, pathLength)$.
 //the i^{th} frame in the path. This selection varies depending on the heuristic employed.
- 2) While ($Q_i \neq \phi \wedge Q_i$ not reserved)
 - a) $Q_{i-2} \leftarrow SelectSecondPrevFrame(i, QoSFrames)$.
 //If there does not exist such a frame, then the function returns
 //a QoS frame with the entire time duration having the state
 //ASYNC_FREE_FREE.
 - b) $Q_{i-1} \leftarrow SelectPrevFrame(i, QoSFrames)$.
 - c) $Q_{i+1} \leftarrow SelectNextFrame(i, QoSFrames)$.
 - d) $Q_{i+2} \leftarrow SelectSecondNextFrame(i, QoSFrames)$.
 - e) $FirstTwo \leftarrow GetAvailable_Bandwidth_Info(Q_{i-2}, Q_{i-1}, FIRST_TWO)$.
 - f) $LastTwo \leftarrow GetAvailable_Bandwidth_Info(Q_{i+1}, Q_{i+2}, LAST_TWO)$.
 - g) $FirstAndLastTwo \leftarrow GetAvailable_Bandwidth_Info(FirstTwo, LastTwo, MIDDLE_TWO)$.
 - h) $ResultantFrame \leftarrow GetAvailable_Bandwidth_Info(Q_i, FirstAndLastTwo, MIDDLE_TWO)$.
 - i) Select time interval $(t_l, t_l + t_{RT})$ having the state ASYNC_FREE_FREE in the frame ResultantFrame. Again this selection varies based on the heuristic.
 - j) If such an interval exists,
 - (i) $Resu[i] \leftarrow t_l$.
 - (ii) $UpdateState(Q_i, t_l, t_l + t_{RT}, SECOND_NGHR_1)$.
 //The UpdateState function updates the state for the given duration as specified in the state diagram shown in Figure 1.
 $UpdateState(Q_{i+1}, t_l, t_l + t_{RT}, NGHR_1)$.
 $UpdateState(Q_{i-1}, t_l, t_l + t_{RT}, LINK)$.
 $UpdateState(Q_{i+2}, t_l, t_l + t_{RT}, NGHR_2)$.
 $UpdateState(Q_{i-2}, t_l, t_l + t_{RT}, SECOND_NGHR_2)$.
 - (iii) Select the next link depending on the heuristic.
 - else
 - (i) Break.
 - (ii) Return ϕ . //Unsuccessful Reservation.
- 3) End of while loop.
- 4) Return Resu.

Fig. 2. Algorithm to obtain information on free bandwidth and the generic algorithm. Steps 1, 2.i, 2.j.(iii) in the generic algorithm depend on the heuristic (EFR, MBR, and PHR) employed.

be attached to the *Route Reply* packet and sent to the previous node in the path. When a node receives a *Route Reply* packet, it then checks whether it is in the path mentioned in the packet. If so, it reads the time allotted for the link between itself and the node which had sent the *Route Reply*. The node performs

call setup using the RTMAC. If it is successful, the *Route Reply* packet is then sent to the next node. This goes on till the entire path reservation is completed.

4 Simulation Results

The proposed QoS-DSR and the heuristics for the asynchronous environment were simulated using GloMoSim. The heuristics were compared under identical environments and loads. We have used Constant Bit Rate (CBR) sessions which generate datagram packets of size 216 bytes, every 60 milliseconds, for both RT and BE connections. Simulation time was taken as 600 seconds and CBR sessions of 200 seconds were generated randomly between 0 and 400 seconds of the simulation. CBR sessions were classified into two classes, namely RT and BE sessions. Terrain range used for the simulations was 1000m × 1000m. Transmission range of a node was taken to be 300m. The simulated network has 30 nodes which were uniformly distributed in the terrain area. Random way-point mobility model is used. The average end-to-end delay in RT communications is a crucial parameter in deciding the performance of the protocol. In these simulations, we find that the average end-to-end RT delay is the least for the EFR heuristic. The reason can be attributed to the manner in which *conn-slots* are assigned. In the EFR, we find that successive links are assigned their slots in order. The session intervals are assigned in an early-fit manner. Hence it is bound to give the least delay when compared to the other heuristics. The PHR gives a delay lesser than its parent heuristic. This can be explained by the slot selection method followed in the heuristic. The first experiment is a comparison of the three parent heuristics on increasing network load ([12]). Figure 3 compares the average end-to-end delay of the three heuristics. The EFR gives the least delay while the MBR yields the maximum delay and the PHR provides a delay in between the two. Figure 4, comparing the throughput of the three heuristics, shows that the MBR and the PHR give a greater throughput than the EFR.

The next set of simulation results compare a pair of heuristics X and XMesh (X = EFR or PHR). An important parameter that has been introduced in this set of results is the number of *deaths* of a mobile node. This value is equal to the number of times a node reaches the *dormant state*. Figure 5, comparing the average RT delay between EFR and EFRMesh under increasing network load with no BE calls, shows that the EFRMesh suffers due to the allocation of *recovery-holes*. Figure 6, that compares the average number of *deaths* between PHR and PHRMesh under increasing network load with a heavy BE load, shows that, on an average, PHRMesh causes a lower number of *deaths* than PHR. Figure 7 shows that the call dropping ratios in PHR and PHRMesh increase with increasing mobility, with the PHRMesh heuristic exhibiting better results on an average. Figure 8 shows that the EFRMesh heuristic displays a lower number of *deaths* of mobile nodes than the EFR heuristic, on increasing mobility.

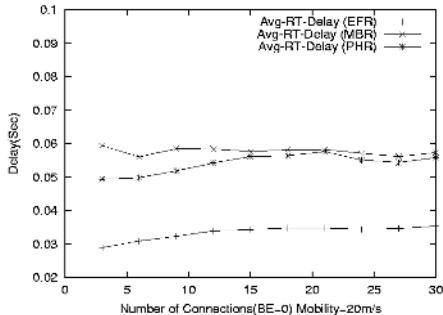


Fig. 3. Average end-to-end delay vs. network load (mobility = 20m/s).

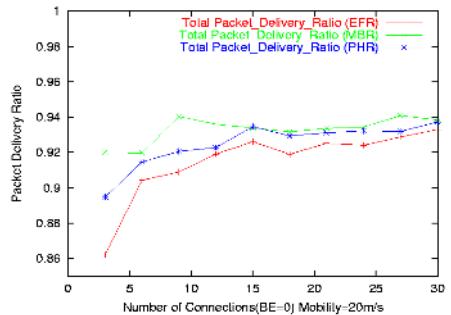


Fig. 4. Packet delivery ratio vs. network load (mobility = 20m/s).

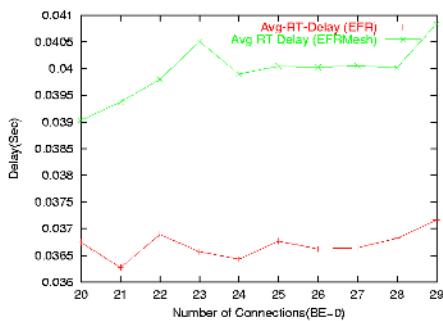


Fig. 5. Average end-to-end delay vs. network load (mobility = 20m/s).

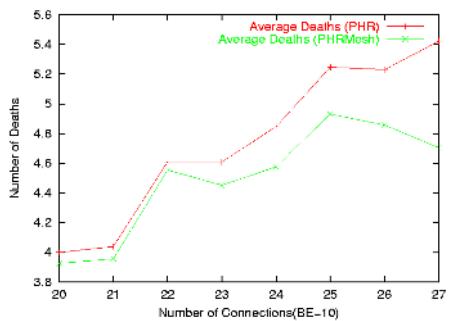


Fig. 6. Average number of *deaths* vs. network load (mobility = 20m/s).

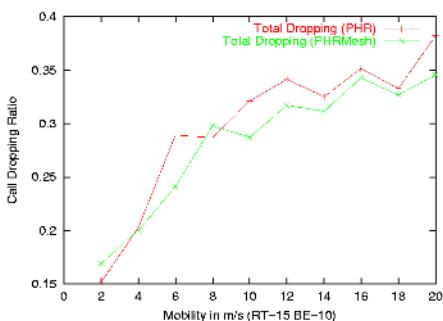


Fig. 7. Average call dropping ratio vs. mobility.

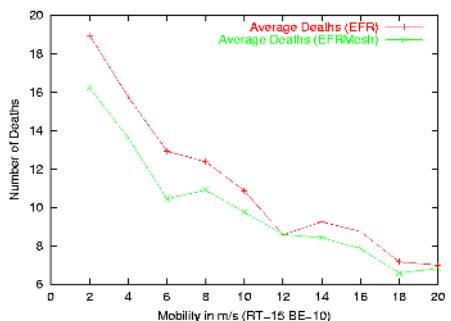


Fig. 8. Average number of *deaths* of mobile nodes vs. mobility.

5 Summary

In this paper, we have proposed a QoS extension of the DSR protocol with slot allocation heuristics for the asynchronous multihop wireless networks. The heuristic EFR has been found to give the best performance in terms of delay while the MBR provides a better throughput. The PHR provides an intermediary performance. The heuristics have been adapted to the mesh environment to provide an increased lifetime for the power-constrained mobile nodes. We have adopted a pulsed discharge scheme for the mobile nodes according to which the battery of a mobile node recovers during idle time and reaches a *dormant state* whenever it gets discharged completely. The fixed infrastructure-based nodes, on the other hand, are assumed to have infinite power. Simulation studies have shown that the adapted heuristics perform better in terms of the number of *deaths* of mobile nodes.

References

1. C. E. Perkins and P. Bhagwat, "Highly Dynamic Destination-Sequenced Distance Vector Routing (DSDV) for Mobile Computers," *Proc. of ACM SIGCOMM 1994*, pp. 234-244, September 1994.
2. D. B. Johnson and D. A. Maltz, "Dynamic Source Routing in Ad hoc Wireless Networks", *Mobile Computing*, Edited by T. Imielinski and H. Korth, Kluwer, pp. 153-181, 1996.
3. B. S. Manoj and C. Siva Ram Murthy, "Real-time Traffic Support for Ad Hoc Wireless Networks", *Proc. of IEEE ICON 2002*, August 2002.
4. C. R. Lin and M. Gerla, "MACA/PR: An Asynchronous Multimedia Multihop Wireless Network", *Proc. of IEEE INFOCOM '97*, March 1997.
5. C. R. Lin and J. S. Liu, "QoS Routing in Ad hoc Wireless Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 8, pp. 1426-1438, August 1999.
6. C. R. Lin, "Admission Control in Time-Slotted Multihop Mobile Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 19, No. 10, pp. 1974-1983, October 2001.
7. H. C. Lin and P. C. Fung, "Finding Available Bandwidth in Multihop Mobile Wireless Networks", *Proc. of IEEE VTC*, Tokyo, Japan, May 2000.
8. S. Chen and K. Nahrstedt, "Distributed Quality-of-Service Routing in Ad Hoc Networks", *IEEE Journal on Selected Areas in Communications*, Vol. 17, No. 8, pp. 1594-1603, August 1999.
9. M. R. Garry and D. S. Johnson, *Computers and Intractability*, San Francisco, CA Freeman 1979.
10. C. F. Chiasserini and R. R. Rao, "Importance of a Pulsed Battery Discharge in Portable Radio Devices", *Proc. of ACM MOBICOM 1999*, August 1999.
11. D. Panigrahi, C. F. Chiasserini, S. Dey, R. R. Rao, A. Raghunathan, and K. Lahiri, "Battery Life Estimation for Mobile Embedded Systems", *Proc. Int. Conf. VLSI Design*, pp. 55-63, January 2001.
12. V. Vidhyashankar, B. S. Manoj, and C. Siva Ram Murthy, "Slot Allocation Strategies for Delay Sensitive Traffic in Multihop Wireless Networks", *Technical Report, HPCN Lab, Department of Computer Science and Engineering, IIT Madras*, April 2003.

Multicriteria Network Design Using Distributed Evolutionary Algorithm

Rajeev Kumar

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur - 721 302, India
rkumar@cse.iitkgp.ernet.in

Abstract. In this paper, we revisit a general class of multicriteria multiconstrained network design problems and attempt to solve, in a novel way, with a Distributed Evolutionary Algorithm (EA). A major challenge to solving such problems is to capture possibly all the (representative) equivalent and diverse solutions at convergence. In this work, we formulate, without loss of generality, a bi-criteria bi-constrained communication network topological design problem. Two of the primary objectives to be optimized are network delay and cost subject to satisfaction of reliability and flow-constraints. This is a *NP-hard* problem and the two-objective optimal solution front is not known *a priori*. Therefore, we adopt randomized search method and use multiobjective EA that produces diverse solution-space. We employ a distributed version of the algorithm and generate solutions from multiple tribes to ensure convergence. We tested this approach for designing networks of different sizes and found that the approach scales well with larger networks. Results are compared with those obtained by two traditional approaches namely, the exhaustive search and a heuristic search.

1 Introduction

Network design problems where even a single cost function or objective value (e.g., minimal spanning tree or shortest path problem) is optimized, are often NP-hard [1]. Many such uni-criterion network design problems are well studied and many heuristics/methods exist for obtaining exact/approximate solutions in polynomial-time [2]. But, in most real-life applications, network design problems generally require simultaneous optimization of multiple and often conflicting objectives, subject to satisfaction of some constraints. For example, topological design of communication networks, particularly mesh/wide area networks is a typical multiobjective problem involving simultaneous optimization of cost of the network and various performance criteria such as average delay of the network, throughput subject to some reliability measures and bandwidth/flow-constraints. This requires optimization of conflicting factors, subject to various constraints. Exploring the whole solution space for such a design problem is an NP hard problem [3]. Similar design problems exist for multicast routing of multi-media communication in constructing a minimal cost spanning/Steiner tree with given constraints on diameters [4]. Such multicriteria network design problems occur in many other engineering applications too. The VLSI circuit design aims at finding minimum

cost spanning/Steiner tree given delay bound constraints on source-sink connections [5]. Analogously, there exists the problem of degree/diameter- constrained minimum cost networks [6].

Many NP-hard bicriteria network design problems have been attempted and approximate solutions obtained using heuristics/methods, and verified in polynomial time, see - [6], [7] and [8]. For example, Deo et al. [6] and Ravi et al. [8] presented approximation algorithm by optimizing one criterion subject to a budget on the other. We argue that the use of heuristics may yield *single* optimized solutions in each objective-space, and may not yield many other equivalent solutions. Secondly, extending this approach to multi-criteria problems (involving more than two objectives/constraints) the techniques require improving upon more than one constraints. Thirdly and more importantly, such approaches may not yield all the *representative* optimal solutions. For example, most conventional approaches to solve network design problems start with a Minimum Spanning Tree (MST), and thus effectively minimizes the cost.

In this work, we try to overcome the disadvantages of conventional techniques and single objective EAs. We use multiobjective EA to obtain a Pareto-front. For a wide-ranging review and a critical analysis of evolutionary approaches to multiobjective optimization - see [9] and [10] for details. There are many implementation of multiobjective EAs, for example, MOGA [11], NSGA [12], SPEA [13]) and PEAS [14]. These implementations achieve diverse and equivalent solutions by some diversity preserving mechanism, they do not talk about convergence. Any explicit diversity preserving method needs prior knowledge of many parameters and the efficacy of such a mechanism depends on successful fine-tuning of these parameters. In a recent study, Purshouse & Fleming [17] extensively studied the effect of sharing, along with elitism and ranking, and concluded that while sharing can be beneficial, it can also prove surprisingly *ineffective* if the parameters are not carefully tuned.

Some other recent studies have been done on combining convergence with diversity. Laumanns et al. [15] proposed an ϵ -dominance for getting an ϵ -approximate Pareto-front for problems whose optimal Pareto-set is *known*. Kumar & Rockett [16] proposed use of Rank-histograms for monitoring convergence of Pareto-front while maintaining diversity without any *explicit* diversity preserving operator. Their algorithm is demonstrated to work for problems of *unknown* nature. Secondly, assessing convergence does not need any *a priori* knowledge for monitoring movement of Pareto-front using rank-histograms.

In this work, we use their Pareto Converging Genetic Algorithm (PCGA) [16] which has been demonstrated to work effectively across complex problems and achieves diversity without needing *a priori* knowledge of the solution space. PCGA excludes any explicit mechanism to preserve diversity and allows a natural selection process to maintain diversity. Thus multiple, equally good solutions to the problem, are provided. Another major challenge to solving *unknown* problems is how to ensure convergence. In this work, we use a distributed version of PCGA algorithm and generate solutions using multiple tribes and then merge to ensure convergence. PCGA assesses convergence to the Pareto-front which, by definition, is unknown in most real search problems, by use of rank-histograms.

We select topological design of communication network as a sample network problem domain. We present a novel approach to design a network with two minimization

objectives of cost and delay subject to satisfaction of reliability and flow constraints. (In the past, EAs have been extensively used in *single* objective optimization for various communication network related design problems - we give a brief survey of such work in the next section.) The remainder of the paper is organized as follows. In section 2, we include an overview of multiobjective evolutionary algorithm (MEA) and the related work done for communication network design problem. We describe, in section 3, a suitable model for the representation of a communication network and its implementation using the distributed PCGA. Then, we present results in section 4 along with a comparison with the conventional methods. Finally, we draw conclusions in section 5.

2 An Overview

2.1 Multiobjective Evolutionary Algorithms

Evolutionary/Genetic Algorithms (EAs/GAs) are randomized search techniques that work by mimicking the principles of genetics and natural selection. (In this paper, we use the term EA and GA interchangeably.) EAs are different from traditional search and optimization methods used in engineering design problems. Most traditional optimization techniques used in science and engineering application can be divided into two broad classes: direct search algorithms requiring only the objective function; and gradient search methods requiring gradient information either exactly or numerically. One common characteristic of most of these methods is that they all work on a point-by-point basis. An algorithm begins with an initial solution (usually supplied by the user) and a new solution is calculated according to the steps of the algorithm. These traditional techniques are apt for well-behaved, simple objective functions, and tend to get stuck at sub-optimal solutions. Moreover, such approaches yield a *single* solution. In order to solve complex, non-linear, multimodal, discrete or discontinuous problems, probabilistic search heuristics are needed which may work with a set of points/initial solutions, especially for multiobjective optimization which yields a set of optimal points instead of a single point.

Goldberg used the notion of Pareto-optimality for assigning equal probabilities of regeneration to all the non-dominated individuals in a population - see [9], [10] and [11]. The major achievement of the Pareto rank-based research is that a multiobjective vector is reduced to a scalar fitness without combining the objectives in any way.

Further, the use of fitness based on Pareto ranking permits non-dominated individuals to be sampled at the same rate thus according equal preference to all non-dominated solutions in evolving the next generation. The mapping from ranks to fitness values however is an influential factor in selecting mates for reproduction. Since then, many rank-based multiobjective EAs have been developed - see [9] and [10] for details of the algorithms and their implementations. Almost all the multiobjective genetic algorithms/implementations have ignored the issue of convergence and are thus, unsuitable for solving *unknown* problems.

In this work, we use a Pareto Converging Genetic Algorithm(PCGA) [16] which is centered around Goldberg's notion of Pareto-optimality [11]. PCGA excludes any explicit mechanism to preserve diversity and allows a natural selection process to maintain diversity. The baseline PCGA assesses convergence to the Pareto-front which, by

definition, is unknown in most real search problems, by use of rank-histograms within a single tribe. We employ the distributed version of the algorithm using a multi-tribal approach; the distributed algorithm is particularly suited for such complex problems of unknown nature and assess convergence among multiple tribes.

2.2 Related Work on Communication Network Optimization

Since Network Design Optimization is an NP-hard problem, heuristic techniques have been used widely for such design. Heuristic methods that have been used include techniques, such as branch exchange, cut saturation etc. For example Jan et al. developed a branch and bound based technique to optimize network cost subject to a reliability constraint [18]. Ersoy and Panwar developed a technique for the design of interconnected LAN and MAN networks to optimize average network delay [19]. Clarke and Anandalingam used a heuristic to design minimal cost and reliable network [20]. However, these being heuristics, they do not ensure that the solutions obtained are optimal. Some of these heuristics evaluate trees and thus a large number of possible solutions are left unexplored.

Linear and Integer Programming have been used to a limited extent for network optimization since the number of equations increases exponentially with the number of nodes [21]. Also, greedy randomized search procedures [22] and other meta heuristics have been used for combinatorial optimization. Ribeiro and Rossetti [23] proposed a parallel Greedy Randomized Adaptive Search Procedures (GRASP) heuristic with path-relinking for a 2-path network design problem.

EAs have been extensively used in single objective optimization for many communication network related optimization problems. For example, Baran and Laufer [24] presented an Asynchronous Team Algorithms (A-Team) implementation, in a parallel heterogeneous asynchronous environment, to optimize the design of reliable communication networks given the set of nodes and possible links. The proposed Team combines parallel GAs, with different reliability calculation approaches in a network of personal computers. Abuali et al. assigned terminal nodes to concentrator sites to minimize costs while considering maximum capacity [25]. Ko et al. used EA for design of mesh networks but the optimization was limited to optimizing the single objective of cost while keeping minimum network delay as a constraint [26]. However, this has been extended to multiobjective optimization, in their recent work, for design of self-healing networks [27]. Elbaum and Sidi used EA to design a LAN with the single objective of minimizing network delay [28]. Kumar et al. used EA for the expansion of computer networks while optimizing the single objective of reliability [29]. White et al. used EA to design Ring Networks optimizing the single objective of network cost [30]. Dengiz et. al [31] presented a EA with specialized encoding, initialization, and local search operators to optimize the design of communication network topologies.

Most approaches attempted to optimize just one objective. For some approaches, the problem is broken down into a number of subproblems, solved in sequence using some heuristics thereby possibly leading to locally optimal design. Ravi et al. [8] and Deo et al. [6] presented approximation algorithm by optimizing one criterion subject to a budget on the other. Since then, many polynomial-time algorithm have been developed for several NP-hard optimization problems arising in network design. Different connectivity

requirements such as spanning trees, Steiner trees, generalized Steiner forests, and 2-connected networks have been considered.

However, a practical multiobjective optimization approach should *simultaneously* optimize multiple objectives subject to satisfiability of multiple constraints. In this work, we present a framework using EAs that simultaneously optimize multiple objectives and produces a set of non-dominated *equivalent* solutions that lie on (near-) optimal Pareto-front.

3 Design and Implementation

Topological design of WANs involves determining the layout of links between nodes given the mean/peak inter node traffic such that certain parameters of the network are optimized. In the solution developed, the total network cost and average delay on links is minimized simultaneously to obtain a Pareto front of optimal non-dominated solutions. For such a design, we use the following network parameters: the total number of nodes in the network N , the distance matrix D_{ij} which gives the physical distance between nodes i and j in kms, the traffic matrix T_{ij} which gives the expected peak network traffic between nodes i and j in packets per second, the number of types of network equipment slabs available K , and the number of types of link slabs available M along with the link cost per unit distance and link capacity. We use two objective functions - cost and delay - each of which is approximated by the following formulation:

1. Cost:

$$Cost = CostNodes + CostLinks + CostAmps$$

where,

$$CostNodes = \sum_i C_i; \quad C_i = \text{cost of the network equipment placed at node } i$$

$$CostLinks = \sum_i \sum_j C_{ij}; \quad C_{ij} = \text{cost of the link between node } i \text{ and node } j$$

$$CostAmp = \frac{\sum_i \sum_j D_{ij} \times A}{L}; \quad L = \text{maximum distance for which the signal is sustained without amplification, and } A = \text{cost of each amplifier unit.}$$

2. Average Delay:

$$AvgDelay = \frac{\sum_i \sum_j (Delay_{ij} \times LinkFlow_{ij})}{\sum_i \sum_j LinkFlow_{ij}}$$

$LinkFlow_{ij} = \sum_k \sum_l Traffic_{kl}$ $\forall k, l$ nodes in the network such that the route from node k to node l includes the link (i, j) . From queuing theory,

$$Delay_{ij} = \frac{1}{Cap_{ij} - LinkFlow_{ij}}$$

$Delay_{ij}$ is the link delay for packets flowing through link (i, j) , and Cap_{ij} is the capacity of link (i, j) . $LinkFlow_{ij}$ and $Delay_{ij}$ are 0 if there is no link between nodes i and j . $AvgDelay$ is ∞ if the network cannot handle the required traffic pattern with the existing capacities of the links and the routing policy adopted.

Here, we have assumed a traditional M/M/1 queuing delay. However, same design can easily be adopted for other type of traffic, e.g., self-similar/multimedia traffic using M/G/ ∞ or other models. Similarly, the two-objective formulation can be extended to higher-dimensional objective space for handling delay-jitter and other quantitative Quality of Service (QoS) parameters involving delay-constraint networks and other communication schemes including optical/VPN/mobile networks.

In this work, optimization of cost and delay functions are done subject to the following two constraints. Flow along a link (i, j) should not exceed the capacity of the link. Checking whether the total traffic along a link exceeds the capacity imposes this constraint. If it does, then the network is penalized. The network generated has to be reliable. The number of articulation points is a measure of the unreliability of the network. An articulation point of a graph is a vertex whose removal disconnects the graph. The number of articulation points is determined, and this constraint is imposed penalizing the network proportional to their number.

To calculate the traffic through a particular link the routes between the nodes have to be known so that by superposition principle the total traffic on a link can be calculated. Routing is dynamic in real life and at any point the delays on the various links calculated from the traffic flowing through them gives the best route to be evaluated from the traffic matrix. For solving the design problem at least a rough static route has to be obtained. Dijkstra's shortest path algorithm is used for routing. The metric used for this purpose is the length of the link.

In the encoding scheme chosen, every chromosome encodes a possible topology for interconnecting the given nodes; i.e., a chromosome represents a network, which is an individual in a set of potential solutions of the problem. This set of potential solutions constitutes a population. A constant length bit string representation was used to represent the chromosome. The chromosome consists of two portions; the first portion containing details of the network equipments at the nodes and the second portion consisting of details of the links.

In this work, we use hybridization of EAs and conventional algorithm in generating the initial population so that the time for exploitation and exploration of the search space is significantly reduced, and the number of lethals produced for large nets is minimized. We use Pareto-rank based EA implementation and Roulette wheel selection for selecting the parents. We use multi-point crossover; the number of crossover points depends on the problem-size. We use a simple bit-flipping mutation to further increase the exploration of the solution space.

We compute *Intra-Island Rank-Histogram* for each epoch of the genetic evolution and monitor the movement of Pareto-front. Since, this is a *hard* problem, it is likely that the problem may get trapped in local optima. To ensure a global (near-) optimal Pareto-front, we use a multi-tribal/island approach and monitor the Pareto-front using *Inter-Island Rank histogram* [16].

4 Results

We collected data of mass communication networks of different cities to carry out the simulation. We used the data which was used by the researchers in their previous work,

e.g., [26], [27]. We tested the algorithm for networks of varying sizes and convergence to an optimal Pareto front was observed. We conducted the experiments on a cluster of four machines with many sets of random populations, and analyzed many sets of results. A few representative results, due to space limitations, are included below. However, we shall present many more results during the conference.

First we computed results through EA for a network of ten nodes. Then, an exhaustive search was done for networks of size $N = 10$ nodes in order to generate and evaluate all the possible topologies satisfying the constraints. We created a Pareto-front of all non-dominated solutions; this front forms the *optimal* Pareto-front for the ten-node network.

The deterministic solution is slightly superior to the results obtained by the single-tribe EA. This is expected because the deterministic algorithm exhaustively searches all possible topologies. However, the gap between the results obtained by exhaustive search and EA is quite close; this gap may be specific to a solution space which was obtained by a random sampling of the initial population or due to premature convergence. Thus, there exists a scope for further improvement for the solutions obtained by EA. So we run EA on a cluster of 4 machines to form 4 tribes and merge the solutions while assessing convergence using rank-histogram, the final Pareto-front almost overlapped with the optimal front obtained by exhaustive search. (We are unable to include the graph/plots due to space limitation).

The exhaustive search is of exponential complexity and is completely unfit for networks with more than 10 nodes. The complexity of the exhaustive search was found to be $O(2^{N^2})$. This is because there are $2^{\binom{N}{2}}$ graphs possible with N nodes. We observed that for $N = 10$ node network the exhaustive search took more than 10 hours on a typical Intel Pentium P-IV, 1.7 MHz machine, whereas, the EA took a couple of minutes only. We could not compute the results for $N > 10$ nodes because of the exponential nature of the problem.

Next, we consider a larger network of 36 nodes. The results are shown in Figure 1. We include the initial population, and the (near-) optimal solutions obtained from two tribes. We obtain solutions from two other randomly generated populations, and merge the results from all the four tribes. The final (near-) optimal Pareto-front obtained from EA is also shown in Figure 1. For comparison, we adopt a Branch Exchange Heuristic. As such this heuristic yields a single solution. We modify this heuristic technique incorporating an ϵ -constraint to yield a cluster of solutions within a radius of the ϵ value. Then we obtained solutions across the whole Pareto-front by varying the constraint values across the total front to extend its use to multiobjective optimization. The Pareto-front obtained by this approximation method is included in Figure 1.

The results obtained by the ϵ -constraint branch exchange algorithm are comparable to a subset of the solutions obtained by EA. The diversity of this method is less compared to that of EA. Importantly, the final front obtained by merging all the four tribes is closer to the actual front than the front obtained by the heuristic. This is due to the fact that the branch exchange method considers only those network topologies that are spanning trees. However, such heuristics are unable to obtain most regions of the front. Moreover, there exists scope for further improvement of the results obtained by EA. This is a distinct advantage of distributed EA in solving such *hard* problem.

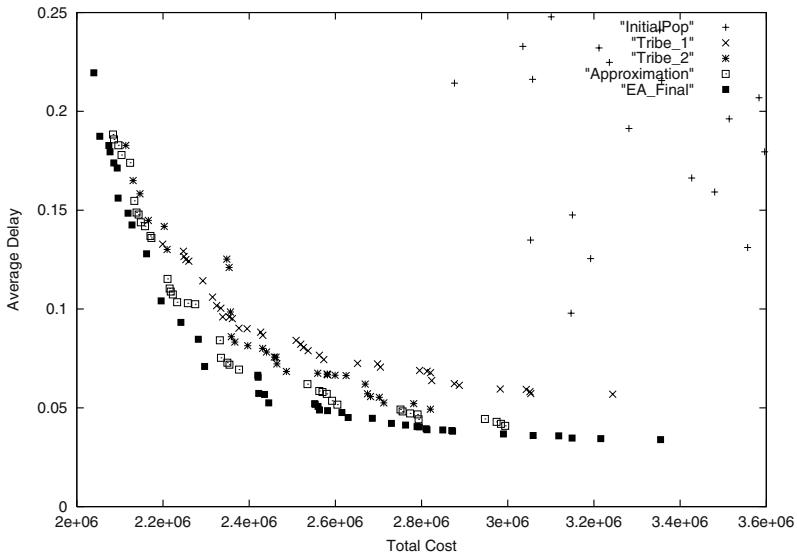


Fig. 1. A 36-node network: Comparisons of solutions obtained by EA and the Approximation scheme. The initial population is also shown here to assess the optimization done by EA. The advantage of distributed EA over single tribes and approximation schemes are clearly seen.

5 Discussion and Conclusions

In this work, we demonstrated the solution of optimizing topologies of communication networks subject to their satisfying the twin objectives of minimum cost and delay along with two constraints using a distributed EA. The solution to the network design problem is a set of optimal network topologies that are non-inferior with respect to each other. The multiple objectives to be optimized have not been combined into one and hence the general nature of the solution is maintained. A network designer having a range of network cost and packet delay in mind, can examine several optimal topologies simultaneously and choose one based on these requirements and other engineering considerations. These topologies are reliable in case of single link failures and it is guaranteed that the maximum packet load on any link will not exceed the link capacity. Thus the network is two edge connected and satisfies the constraints.

The algorithm has been test run on smaller as well as larger networks. Though the initial population used in EA was taken from some hybridization of spanning tree and random topologies, the final obtained Pareto-front was located far apart from the initial population (Figures 1); this indicates that much optimization has been done by EA.

We have demonstrated that the EA is able to achieve diversity across the Pareto-front for smaller and larger networks. The obtained results are very close to the *best* results obtained by exhaustive search (we could test this for smaller networks only). However, with a distributed algorithm, the combined results are superior and almost overlapped with the actual Pareto-front (obtained from exhaustive search) for a smaller network. Nonetheless, this could not be tested for larger networks.

For larger networks, we compared the EA results with those obtained by the approximation algorithm. The approximation algorithm does not guarantee optimal solutions. As such, the approximation algorithm yields a *single* solution; however, we obtained a Pareto-front by varying the constraint-value over the entire range of the objective-value in increments. The (*near*)-optimal Pareto-front obtained by distributed EA is superior than the front obtained by the approximation algorithm in terms of both - diversity and closeness to *real* Pareto-front - which is *unknown* in this case.

Thus, EA achieves greater diversity by spanning the whole Pareto-front with representative solutions in *polynomial time*. The EA works well for both smaller and larger networks, and thus the scheme is scalable. The quality of solutions, in terms of closeness to optimal-values, obtained by distributed EA is comparable to exhaustive search for smaller networks, and superior to those obtained by heuristics/approximation methods for larger networks. In case of form in either of the cases, The compute resources are significantly reduced for the former, and are comparable for the later case.

In this work, though we have considered a special case of network topology design, the framework is *generic* to solve multi-criteria multi-constrained network design problems involving larger number of objectives related to QoS aware networks handling different types of traffic over different communication schemes.

Acknowledgements. This research is supported by Ministry of Human Resource Development (MHRD), Government of India project grant.

References

1. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979. San Francisco, LA: Freeman.
2. D. Hochbaum (Ed.). *Approximation Algorithms for NP-Hard problems*, 1997. Boston, MA: PWS.
3. M. Gerla and L. Kleinrock. On the topological design of distributed computer networks. *IEEE Trans. Communications*, 25(1): 48–60, 1977.
4. V. P. Kompella, J. C. Pasquale, and G. C. Polyzos. Multicast routing for multimedia communication. *IEEE/ACM Trans. Networking*, 286–292, 1993.
5. M. Borah, R. M. Owens, and M. J. Irwin. An edge-based heuristic for Steiner routing. *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, 13(12): 1563–1568, 1995.
6. N. Boldon, N. Deo, and N. Kumar. Minimum-weight degree-constrained spanning tree problem: Heuristics and implementation on an SIMD parallel machine. *Parallel Computing*, 22(3): 369–382, 1996.
7. M. V. Marathe, R. Ravi, R. Sundaram, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt. Bicriteria network design problems. *J. Algorithms*, 28(1): 142–171, 1998.
8. R. Ravi, M. V. Marathe, S. S. Ravi, D. J. Rosenkrantz, and H. B. Hunt. Approximation algorithms for degree-constrained minimum-cost network design problems. *Algorithmica*, 31(1): 58–78, 2001.
9. C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Problems*, 2002. Boston, MA: Kluwer.
10. K. Deb. *Multiobjective Optimization Using Evolutionary Algorithms*, 2001. Chichester, UK: Wiley.

11. C. M. Fonseca and P. J. Fleming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms – Part I: a unified formulation. *IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans*, 28(1): 26–37, 1998. 26–37.
12. K. Deb et al. A fast non-dominated sorting genetic algorithm for multiobjective optimization: NSGA-II. *Parallel Problem Solving from Nature*, PPSN-VI: 849–858, 2000.
13. E. Zitzler, M. Laumanns and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. *Eurogen 2001*.
14. Knowles, J. D. and Corne, D. W. Approximating the non-dominated front using the Pareto Achieved Evolution Strategy. *Evolutionary Computation*, 8(2): 149–172, 2000.
15. M. Laumanns, L. Thiele, K. Deo and E. Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3): 263–182, 2002.
16. R. Kumar and P. I. Rockett. Improved sampling of the Pareto-front in multiobjective genetic optimizations by steady-state evolution : a Pareto converging genetic algorithm. *Evolutionary Computation*, 10(3): 283–314, 2002.
17. R. C. Purshouse and P. J. Fleming. Elitism, sharing and ranking choices in evolutionary multi-criterion optimization. Research Report No. 815, Dept. Automatic Control & Systems Engineering, University of Sheffield, Jan. 2002.
18. R. H. Jan, F. J. Hwang, and S. T. Cheng. Topological optimization of a communication network subject to a reliability constraint. *IEEE Trans. Reliability*, 42(1): 63–69, 1993.
19. C. Ersoy and S. S. Panwar. Topological design of interconnected LAN/MAN Networks. *IEEE J. Select. Areas Communication*, 11(8): 1172–1182, 1993.
20. L. W. Clarke and G. Anandalingam. An integrated system for designing minimum cost survivable telecommunication networks. *IEEE. Trans. Systems, Man and Cybernetics- Part A*, 26(6): 856-862, 1996.
21. A. Atamturk and D. Rajan. Survivable network design: simultaneous routing of flows and slacks. Research Report, IEOR, University of California at Berkeley.
22. T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 1995.
23. C.C. Ribeiro and I. Rossetti. A parallel GRASP heuristic for the 2-path network design problem. Euro-Par 2002 Conference.
24. B. Baran and F. Laufer. Topological optimization of reliable networks using A-Teams. National Computer Center, National University of Asuncion, University Campus of San Lorenzo - Paraguay.
25. F. N. Abuali, D. A. Schnoenefeld, and R. L. Wainwright. Designing telecommunication networks using genetic algorithms and probabilistic minimum spanning Trees. In *Proc. 1994 ACM Symp. Applied Computing*, pp. 242–246, 1994.
26. K. T. Ko, K. S. Tang, C.Y. Chan, K. F. Man and S. Kwong. Using genetic algorithms to design mesh networks. *IEEE Computer*, 30(8): 56–61, 1997.
27. S. Kwong, T. M. Chan, K. F. Man and H. W. Chong. The use of multiple objective genetic algorithm in self-healed network. *Applied Soft Computing*, 2: 104–128, 2002.
28. R. Elbaum and M. Sidi. Topological design of local-area networks using genetic algorithms. *IEEE/ACM Trans. Networking*, 4(5): 766–777, 1996.
29. A. Kumar, R. M. Pathak, and Y.P. Gupta. Genetic-algorithm based reliability optimization for computer network expansion. *IEEE Trans. Reliability*, 44(1): 63–72, 1995.
30. A. R. P White, J. W. Mann, and G. D. Smith. Genetic algorithms and network ring design. *Annals of Operational Research*, 86: 347–371, 1999.
31. B. Dengiz, F. Altiparmak, and A. E. Smith. Local search genetic algorithm for optimal design of reliable networks. *IEEE Trans. Evolutionary Computation*, 1(3): 179–188, 1997.

GridOS: Operating System Services for Grid Architectures

Pradeep Padala and Joseph N. Wilson

Computer & Information Science & Engineering
University of Florida
Gainesville, Florida 32611-6120
{ppadala,jnw}@cise.ufl.edu

Abstract. In this work, we demonstrate the power of providing a common set of operating system services to Grid Architectures, including high-performance I/O, communication, resource management, and process management. In the last few years, a number of exciting projects like Globus, Legion, and UNICORE developed the software infrastructure needed for grid computing. However, operating system support for grid computing is minimal or non-existent. Tool writers are forced to re-invent the wheel by implementing from scratch. This is error prone and often results in sub-optimal solutions. To address these problems, we are developing GridOS, a set of operating system services that facilitate grid computing. The services are designed to make writing middleware easier and make a normal commodity operating system like Linux highly suitable for grid computing. The modules are designed to be policy neutral, exploit commonality in various grid infrastructures and provide high-performance. Experiments with GridOS verify that there is dramatic improvement in performance when compared to the existing grid file transfer protocols like GridFTP. Our proof-of-concept middleware shows that writing middleware is easy using GridOS.

1 Introduction

A Grid[1] enables the sharing, selection, and aggregation of a wide variety of geographically distributed resources including supercomputers, storage systems, data sources and specialized devices owned by different organizations administered with different policies. Grids are typically used for solving large-scale resource and computing intensive problems in science, engineering, and commerce. In the last few years, a number of exciting projects like Globus[2], Legion[3] and UNICORE[4], developed the software infrastructure needed for grid computing. Various distributed computing problems have been solved using these tools and libraries. However, operating system support for grid computing is minimal or non-existent. Though these tools have been developed with different goals, they use a common set of services provided by the existing operating system to achieve different abstractions.

GridOS provides operating system services that support grid computing. It makes writing middleware easier and provides services that make a normal

commodity operating system like Linux more suitable for grid computing. The services are designed as a set of kernel modules that can be inserted and removed with ease. The modules provide mechanisms for high performance I/O (gridos.io), communication (gridos.comm), resource management (gridos.rm), and process management (gridos.pm). These modules are designed to be policy neutral, easy to use, consistent and clean. We have also developed modules on top of the above mentioned primary modules that provide high data transfer rates similar to GridFTP[5].

In this paper, we first review the existing toolkits, their common mechanisms to facilitate grid computing and PODOS[6] a work similar to ours. Then, we describe the modular architecture of GridOS. Next, we explain the current state of implementation. We conclude by showing the performance results obtained using GridOS modules.

2 Previous Work

Primary motivation for this work comes from chapter twenty of the book “*The Anatomy of the Grid: Enabling Scalable Virtual Organizations*”, edited by Ian Foster et al[1], in which the authors discuss the challenges in the operating system interfaces for grid architectures. The book discusses various principles but stops short of implementation details.

While there has been little work on Operating System interfaces, there has been tremendous development in grid middleware. Projects like Globus and Legion provide elaborate software infrastructure for writing grid applications. These tools and libraries have to cope with the existing operating system services that are not designed for high-performance computing. As a result, they are forced to implement some commonly used high-performance optimizations like multiple TCP streams and TCP buffer size negotiation that more suitably should be implemented in the operating system’s kernel. These tools, though quite different, often use the same set of low-level services like resource management, process management, and high-performance I/O. For example, both Globus and Legion have to query the operating system for resources and maintain information for effective resource management. As there is no operating system support for these low-level services, middleware developers must implement them from scratch. This is error prone and often results in sub-optimal solutions.

There have been some attempts to add operating system services that perform high performance computing. WebOS[7] provides operating system services for wide area applications provides services for wide area applications. PODOS[6], a performance oriented distributed operating system, is similar and provides high performance through optimized services in the kernel. This work succeeds in providing high performance. But due to the extensive changes made to the kernel, it is difficult to port this work to newer kernels. It is also difficult to extend the services due to its monolithic structure. GridOS addresses these

problems by providing a modular architecture that requires minimal changes to the kernel and makes it easy to extend the services.

Apart from the above mentioned work, there has been great amount of research done in distributed operating systems. Systems like Amoeba[8] and Sprite[9] had great impact on the distributed programming paradigm. We have incorporated some of the principles used in designing these distributed operating systems in GridOS.

3 GridOS Design

The following principles drive the design of GridOS. These principles derive from the fact that the toolkits like Globus require a common set of services from the underlying operating system.

- *Modularity.* The key principle in GridOS is to provide modularity. The Linux kernel module architecture is exploited to provide a clean modular functionality.
- *Policy Neutrality.* GridOS follows one of the guiding principles of design of operating systems: policy free mechanisms. Instead of providing a “black box” implementation that takes care of all possibilities, the modules provide a policy-free API which can be used to develop high level services like GridFTP.
- *Universality of Infrastructure.* GridOS provides a basic set of services that are common to prevalent grid software infrastructures. It should be easy to build radically different toolkits like Globus (a set of independent services) and Legion (an object-based meta-systems infrastructure).
- *Minimal Core Operating System Changes.* We do not want to make extensive modifications to the core operating system as that would make it difficult to keep up with new OS versions.

These guiding principles led us to develop a layered architecture (Figure 1). The lowest layer contains the primary GridOS modules that provide high-performance grid computing facilities. These modules are orthogonal and provide basic mechanisms. They do not mandate any policies. The upper layers provide specific services similar to GridFTP, GASS[10].

This approach is motivated by the observation that the toolkits usually make policy decisions on behalf of the grid applications. The toolkit, knowing the application requirements and having domain knowledge, can make a better decision about what policy to employ. This also encourages a wide variety of toolkits to be developed on top of GridOS.

4 Module Design

In the following sections we describe GridOS modules. We explain the implementation details in a later section.

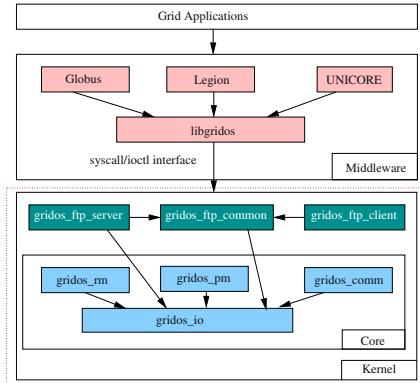


Fig. 1. Major Modules and Structure of GridOS

4.1 Core Modules

High-Performance I/O Module(gridos.io): This module provides High-Performance network I/O capabilities for GridOS. In an increasing number of scientific disciplines, large amounts of data (sometimes of the order of peta bytes) are accessed and analyzed regularly. For transporting these large amounts of data, high-speed WANs are used. These networks have high bandwidth and large round trip times (RTT).

In order to take full advantage of these high speed networks, various operating system parameters must be tuned and optimized. Several techniques for achieving high-performance are outlined below. It is important to realize that some of these techniques are not additive, but rather complementary. Applying a single technique may have little or no effect, because the absence of any one of the techniques can create a bottleneck.

No User-Space Copying. A user-space FTP server requires the kernel to copy the network buffers to user-space buffers to retrieve data from the client. It then requires the kernel to copy the user-space buffer to file system buffers to write data to a file. This incurs a large overhead due to time spent in copying.

Because `gridos.io` and `gridos.ftp` are kernel modules that handle both network and file system I/O, thus double copying can be avoided.

TCP WAN Throughput. TCP (Transmissions Control Protocol) is the standard transport layer used over IP networks. TCP uses the congestion window (CWND), to determine how many packets can be sent before waiting for an acknowledgment. On wide area networks, if CWND is too small, long network delays will cause decreased throughput. This follows directly from Little's Law[11] that $WindowSize = Bandwidth * Delay$.

The TCP “slow start” and “congestion avoidance” algorithms determine the size of the congestion window. The kernel buffers are allocated depending on the maximum size of the congestion window.

To get maximal throughput, it is essential to use optimal TCP send and receive kernel buffer sizes for the WAN link we are using. If the buffers are too small, the TCP congestion window will never fully open up. If the buffers are too large, a fast sender can overrun a slow receiver, and the TCP window will shut down. The optimal buffer size for a link is $\text{bandwidth} * \text{RTT}$ [12]

Apart from the **default** send and receive buffer sizes, **maximum** send and receive buffer sizes must be tuned. The default maximum buffer size for Linux is only 64KB which is quite low for high-speed WANs.

The `gridos.io` module provides various ways of controlling these buffer sizes through `gridos` system call.

Communication Module(`gridos.comm`): This module provides multiple communication methods that support both automatic and programmer assisted method selection. Grid applications are heterogeneous not only in their computational requirements, but also in types of communication. Different communication methods differ in usage of network interfaces, low-level protocols and data encodings and may have different quality of service requirements. This module allows communication operations to be specified independent of methods used to implement them.

The module also provided multi-threaded communication which is used in implementing the FTP module. Using the library various high-level mechanisms like MPI (Message Passing Interface) can be implemented.

Resource Management Module(`gridos.rm`): Grid systems allow applications to assemble and use collections of resources on an as-needed basis, without regard to physical location. Grid middleware and other software architecture that manage resources have to locate and allocate resources according to application requirements. They also have to manage other activities like authentication and process creation that are required to prepare a resource to use. See [13] for details on various issues involved in resource management.

Interestingly, these software infrastructure use a common set of services. `gridos.rm` provides these facilities so that software infrastructure can be developed to address higher-level issues like co-allocation, online-control etc.

The module provides facilities to query and use resources. The module maintains information about current resource usage of processes started by local operating system and GridOS modules. It also provides facilities to reserve resources. The module doesn't provide process migration facility but the basic facilities can be used to develop systems like condor[14].

Process Management Module(`gridos_pm`): This module allows creation and management of processes in GridOS. It provides a global PID (GPID) for every process in GridOS and provides communication primitives which can be used on top of `gridos.comm` for processes to communicate among themselves[15].

The module also provides services for process accounting. This feature is important in accounting for the jobs which are transported to GridOS.

4.2 Additional Modules

gridos_ftp_common: This module provides facilities common to any FTP client and server. This includes parsing of FTP commands, handling of FTP responses etc.

gridos_ftp_server: This module implements a simple FTP server in kernel space. Due to its design, copying of buffers between user and kernel space is avoided. The server does not start a new process for each client as is usually done in typical FTP servers. This incurs low overhead and high-performance. The server also uses `gridos.io` facilities to monitor bandwidth and adjust the file system buffer sizes. The file system buffers are changed depending on the file size to get maximum overlap between network and disk I/O.

gridos_ftp_client: This module implements a simple FTP client in kernel. The main purpose of this module is to decrease the overhead of writing or reading file on the client side. Our experiments indicate that primary overhead on the client side is the time spent in reading and writing files. By carefully optimizing the file system buffers to achieve maximum overlap between network and disk I/O, high-performance is achieved.

5 Implementation

We have implemented a subset of GridOS on Linux using the stable 2.4.20 kernel. The Linux kernel module architecture is exploited to provide ease of installation for users and ease of modification for developers. The code is divided into a small patch to the core kernel and a set of kernel modules. The patch is designed to be minimal and adds the basic `ioctl` and `syscall` interfaces to GridOS.

Currently, we have implemented `gridos.io`, `gridos_ftp_server`, `gridos_ftp_client` and the library wrapper `libgridos`. We have also implemented a simple middleware called `gridos-middleware` as a proof-of-concept showing the ease with which middleware can be developed. The following sections explain the internals of the modules.

5.1 IO Module

The globus IO module implementation is divided into two APIs, one each for the network and the file system. The IO module is designed to minimize copy operations and let the data flow remain within the kernel.

The network API includes functions to read and write data from a gridos managed socket. Both blocking and non-blocking read calls are provided. Gridos FTP modules make extensive use of the non-blocking read for high-performance. It also has functions to set various buffer management options.

- *gridos_io_sync_read*: This function is used to read data from a gridos managed socket in blocking mode.
- *gridos_io_async_read*: This function is used to read data from a gridos managed socket in non-blocking mode
- *gridos_io_write*: This function writes data to the gridos managed socket
- *gridos_io_buffer_setopt*: This function sets options for buffer management. The options include setting of TCP send and receive buffer sizes, maximum TCP buffer size etc.
- *gridos_io_buffer_getopt*: This function returns the current buffer management options.

The file system API has similar functions for reading and writing data to files. These functions call the Linux Kernel's VFS (Virtual File System) functions to achieve this. The API also has a higher-level function *gridos_file_copy* which can be used to copy a file locally. This can be used for high-performance local copying of files.

5.2 FTP Client and Server Modules

The FTP server API allows the user to create an FTP server on a specified port. Various configuration options like *docroot* (top level directory for anonymous FTP) *threads* (number of simultaneous threads) etc can be set to control the behaviors of FTP module. Dynamic configuration can be done via Linux's *sysctl* mechanism. This provides access to FTP module features through */proc* interface which can be used to read and modify various variables in the kernel address space.

6 Performance

We have conducted various experiments to measure the performance of GridOS comparing it to standard OS services and Globus services. The use of GridOS services results in a dramatic increase in throughput in our experiments. First, we compare ftp performance using GridOS and the standard *proftpd* shipped with Mandrake Linux, which showcases the advantages of zero-copy I/O. Then, we compare performance of transporting a file using GridFTP server and GridOS ftp using *globus-url-copy* as the client. This reveals an important bottleneck in high-performance I/O: file writing overhead. Finally, we compare performance of file transfer using GridOS ftp client and server and *globus-url-copy* and GridFTP server.

The experiments confirm that on average GridOS is 4 times faster than ftp and 1.5 times faster than GridFTP.

6.1 Test Environment and Methodology

The tests are conducted in an experimental grid setup in the self-managed network of CISE¹ at University of Florida. Globus toolkit version 2.2 is used for

¹ Computer & Information Science & Engineering

setting up the grid. The two testing machines were connected over 100Mbps Ethernet network. The machines are kept idle while running the tests so as not to allow other processes running on these machines affect the performance numbers.

In each experiment files of sizes varying from 1KB to 512MB are used and the total time taken to transport each of these files is calculated. For each transfer the average of 5 runs for each file is taken. To avoid the affects of serving from the cache, testing machine is rebooted after each run.

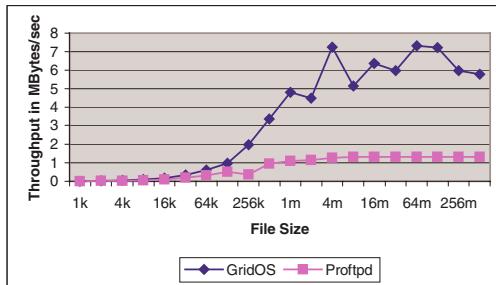


Fig. 2. GridOS vs Proftpd using standard ftp client

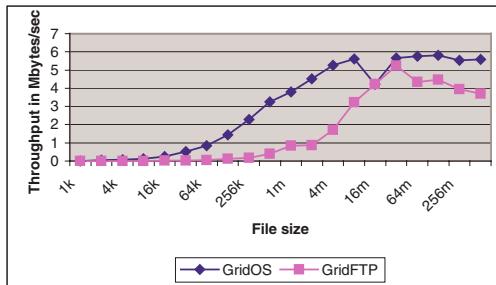


Fig. 3. GridOS ftp vs GridFTP using globus-url-copy as client

6.2 GridOS vs. Proftpd

In our first experiment, we compared the performance of the proftpd and GridOS ftp servers using the standard ftp client shipped with Mandrake Linux as the client. Results are shown in figure 2. This experiment show-cases the advantages of serving the files from within the kernel. GridOS consistently performs better than Proftpd for all file sizes.

6.3 GridFTP vs. GridOS ftp Server Using globus-url-copy Client

This experiment is done using the `globus-url-copy` client for a fair comparison. Results are shown in figure 3. GridFTP performs poorly for small files due to the

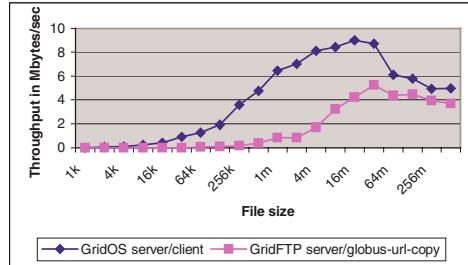


Fig. 4. GridOS ftp server/client vs GridFTP server/globus-url-copy

overhead incurred in the performing security mechanisms. GridOS ftp employs the standard ftp password authentication for security.

We also conducted experiments using standard ftp client instead of globus-url-copy for GridOS. Performance was poor compared to the performance obtained using globus-url-copy. Globus-url-copy is specifically designed for high-performance computing and has better buffer management. This led us to develop an in-kernel ftp client.

6.4 GridFTP Server/Client vs. GridOS ftp Server/Client

In this experiment, *globus-url-copy* and *gridos-ftp-client* are used as clients for GridFTP and GridOS ftp server respectively. GridOS ftp client makes effective buffer management and is designed on the same lines as *globus-url-copy*. Results are shown in figure 4. Based on our experiments we observe that the performance drop for larger files is due to the overhead involved in file writing.

7 Conclusions and Future Work

We have described a set of operating system services for grid architectures. These services make a commodity operating system like Linux highly suitable for high-performance computing. We have identified a common set of services that use grid software infrastructures like Globus, Legion, Condor etc. The services are developed as a set of modules for ease of use and installation. Our performance experiments show that high-performance can be achieved with GridOS modules. We have also described a proof-of-concept middleware that uses GridOS facilities.

The work presented here is a first step towards a grid operating system which provides extensive, flexible services for grid architectures. Our next step is to implement other GridOS modules. We have plans to deploy it in a wide area computing testbed. Our first target is the grid used by GriPhyN (Grid Physics Network) at the University of Florida. This environment will enable us to evaluate GridOS modules more realistically. We also plan to port Globus libraries to GridOS thus providing a complete software infrastructure for grid architectures.

References

1. Foster, I., Kesselman, C., eds.: *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers (1999)
2. Foster, I., Kesselman, C.: Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing* **11** (1997) 115–128
3. Grimshaw, A.S., Wulf, W.A., the Legion team: The legion vision of a worldwide virtual computer. *Communications of the ACM* **40** (1997) 39–45
4. Huber, V.: UNICORE: A Grid computing environment for distributed and parallel computing. *Lecture Notes in Computer Science* **2127** (2001) 258–266
5. Allcock, B., Bester, J., Bresnahan, J., Chervenak, A.L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., Tuecke, S.: Data management and transfer in high-performance computational grid environments. *Parallel Computing* **28** (2002) 749–771
6. Vazhkudai, S., Syed, J., Maginnis, T.: PODOS — the design and implementation of a performance oriented Linux cluster. *Future Generation Computer Systems* **18** (2002) 335–352
7. Vahdat, A., Anderson, T., Dahlin, M., Belani, E., Culler, D., Eastham, P., Yoshikawa, C.: WebOS: Operating system services for wide area applications. In: *Proceedings of the Seventh Symposium on High Performance Distributed Computing*. (1999)
8. Tanenbaum, A.S., Mullender, S.: An overview of the Amoeba distributed operating system. *Operating Systems Review* **15** (1981) 51–64
9. Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B.: The Sprite network operating system. *Computer* **21** (1988) 23–36
10. Bester, J., Foster, I., Kesselman, C., Tedesco, J., Tuecke, S.: GASS: A data movement and access service for wide area computing systems. In: *Proc. IOPADS'99*. ACM Press (1999)
11. Kleinrock, L.: *Queueing Systems: Theory*. Volume 1. John Wiley and Sons (1975)
12. Semke, J., Mathis, M., Mahdavi, J.: Automatic TCP buffer tuning. *SIGCOMM* 98 (1998)
13. Czajkowski, K., Foster, I., Karonis, N., Kesselman, C., Martin, S., Smith, W., Tuecke, S.: A resource management architecture for metacomputing systems. In: *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, Springer-Verlag LNCS 1459 (1998) 62–82
14. Litzkow, M.J., Livny, M., Mutka, M.W.: Condor : A hunter of idle workstations. In: *8th International Conference on Distributed Computing Systems*, Washington, D.C., USA, IEEE Computer Society Press (1988) 104–111
15. Maginnis, P.T.: Design considerations for the transformation of MINIX into a distributed operating system. In ACM, ed.: *Proceedings, focus on software / 1988 ACM Sixteenth Annual Computer Science Conference*, February 23–25, the Westin, Peachtree Plaza, Atlanta, Georgia, New York, NY 10036, USA, ACM Press (1988) 608–615

Hierarchical and Declarative Security for Grid Applications

Isabelle Attali, Denis Caromel, and Arnaud Contes

INRIA Sophia Antipolis, CNRS – I3S – Univ. Nice Sophia Antipolis,
BP 93, 06902 Sophia Antipolis Cedex - France
`First.Last@inria.fr`

Abstract. Grid applications must be able to cope with large variations in deployment: from intra-domain to multiple domains, going over private, to virtually-private, to public networks. As a consequence, the security should not be tied up in the application code, but rather easily configurable in a flexible, and abstract manner. Moreover, any large scale Grid application using hundreds or thousands of nodes will have to cope with migration of computations, for the sake of load balancing, change in resource availability, or just node failures.

To cope with those issues, this article proposes a high-level and declarative security framework for object-oriented Grid applications. In a rather abstract manner, it allows to define a hierarchical policy based on various entities (domain, host, JVM, activity, communication, ...) in a way that is compatible with a given deployment. The framework also accounts for open and collaborative applications, multiple principles with dynamic negotiation of security attributes and mobility of computations. This application-level security relies on a Public Key infrastructure (PKI).

1 Introduction

This paper aims at proposing an infrastructure where Grid security is expressed outside the application code, outside the firewall of security domains, and in both cases in a high-level and flexible language. The security model discussed in this paper is based on *ProActive* [1,2], a Java library for developing concurrent, distributed and mobile applications. Moreover, as the costs of *Authentication*, *Integrity*, *Confidentiality* (A,I,C) are far from negligible, these characteristics have to be set according to the security requirements and to a given deployment (both hosts and networks). Overall, we propose a framework allowing:

- declarative security attributes (A,I,C), outside any source code and away from any API;
- policies defined hierarchically at the level of administrative domain,
- dynamic security policy, taking into account the nature (private or public) of the underlying networks;
- dynamically negotiated policies (for multi-principals applications),
- policies for remote creation and migration of activities.

Our security framework focuses on authentication of users and processes; it supports user-to-process, process-to-user, process-to-process authentication. We provide authentication solutions that allow a user, user's processes, and the resources used by those processes to verify each other's identity and to communicate securely. We assume that processes are running on a trusted environment, our framework is not intended to protect objects from malicious JVM but applications from each other and network attacks.

Section 2 introduces and discusses related work. Section 3 provides the requested background on the research *ProActive* platform. Section 4 introduces the security model.

2 Related Work

There is a large body of previous work on security in distributed systems. Legion [3] is the closest system to ours. It encapsulates functionalities as objects and provide mechanisms for their location, migration, etc. Objects in these systems interacts via remote method invocations and the main security objectives are authenticating the communications parties, protecting traffic, enforcing access control, delegating rights and enforcing site-specific security concerns. Unlike our system, Legion does not support a hierarchical management of security policies. The Globus system [4] is a computational grid providing pervasive, dependable, and consistent access to high-performance computational resources, despite geographical distribution of both resources and users. In [5], authors propose a strategy for addressing security within the Globus' Open Grid Services Architecture (OGSA). It defines a comprehensive Grid security architecture that supports, integrates and unifies popular security models, mechanisms, protocols, platforms and technologies in a way that enables a variety of systems to inter-operate securely. The use of a user proxy addresses single sign-on and avoids the need to communicate user credentials. The architecture also supports an inter-domain and intra-domain security interoperability. The notion of virtual organization is defined as a set of individuals and/or institutions sharing resources and services under a set of rules and policies governing the extent and conditions for that sharing.

The .NET [6] framework provides security features allowing protection of a host against malicious code. Security system is based on user- and code-identity using public/private keys, hash functions, digital signatures and security policies. There are four policy levels Enterprise, Machine, User and Application Domain.

However, all these security architectures are not aware of object migration. Mobile security concepts are widely studied in agent systems. There are numerous agent systems like MAP [7], Ajanta [8], Aglets [9], Mole [10]. In an agent system, a user creates an agent which is a mobile object and migrates its agent to hosts where the agent can perform some tasks. Security in such systems involves the protection of a host from the agents that reach it and run on it, protection of an agent from other agents, from its host, from the outside environment. The security model considers the issue of authentication and authorization to protect

the hosts from the agent and the agents from others. Security is based on public key authentication.

Nevertheless, agent systems' security model considers the existence of a fixed base. An agent starts from the base, goes to remote sites and comes back to its base. In our model, applications are fully mobile.

3 Background on Distributed Mobile Objects

ProActive is a Java library for distributed and mobile, and Grid computing, implemented on top of RMI [11] as the transport layer. *ProActive* features remote active objects, asynchronous two-way communications with transparent futures, high-level synchronization mechanisms, and migration of active objects. As *ProActive* is built on top of standard Java APIs, it does not require any modification to the standard Java execution environment.

3.1 Base Model

A distributed or concurrent application built using *ProActive* is composed of a number of medium-grained entities called *active objects*. Each active object has one distinguished element, the *root*, which is the only entry point to the active object; all other objects inside the active object are called *passive objects* and cannot be referenced directly from objects which are outside this active object (see Figure 1); the absence of sharing is important with respect to security.

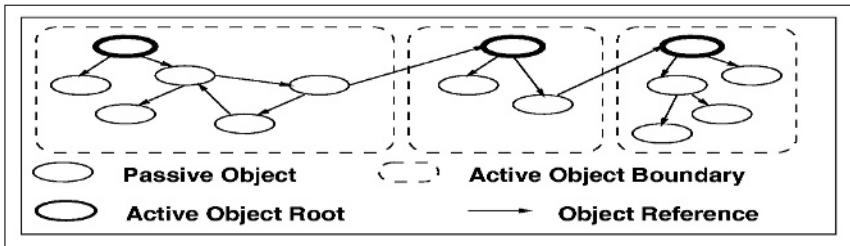


Fig. 1. A typical object graph with active objects

3.2 Deployment Descriptors: Nodes and Virtual Nodes

The security architecture presented in this article relies on two related abstractions for deploying Grid applications: *Node* and *Virtual Node*. We provide here a very brief introduction to those concepts, see [2] for further details.

A *Node* gathers several active objects in a logical entity. It provides an abstraction for the physical location of a set of activities. At any time, a JVM hosts one

or several nodes. Active objects are bound to a node at creation or after migration. In order to have a flexible deployment (eliminating from the source code machine names, creation protocols, registry and lookup protocols), the system relies on *Virtual Nodes* (VNs). A VN is identified as a name (a simple string), used in a program source, defined and configured in an XML descriptor file. The correspondence between Virtual Nodes and Nodes, the mapping, is created by a deployment descriptor.

4 Declarative Grid Security

Grid applications usually involve hundreds of objects on worldwide grids. Grid's resources are dynamically acquired and often differ at each execution. Grid security needs to be defined at a high level of abstraction to address theses issues. One must be able to express policies in a rather declarative manner. The general syntax to provide security rules is the following:

```
Entity[Subject] -> Entity [Subject]
                      : Interaction # [SecurityAttributes]
```

Being in a PKI infrastructure, the subject is a certificate, or credential. Other “elements” (Domain, Virtual Node, Object) are rather specific to Grid applications and, in some cases, to the object-oriented framework. An “entity” is an element on which one can define a security policy. “Interaction” is a list of actions that will be impacted by the rule. Finally, security attributes specify how, if authorized, those interactions have to be achieved.

In order to provide a flavor of the system, we consider the following example.

```
Domain[inria.fr] -> Domain[ll.cnrs.fr] : Q,P # [+A,+I,?C]
```

The rule specifies that between the domain *inria.fr* (identified by a specific certificate) and the parallel machine *ll.cnrs.fr*, all communications (reQuests, and rePlies) are authorized, they are done with *authentication* and *integrity*, *confidentiality* being accepted but not required.

Figure 2 provides the general syntax of policy descriptors. The keywords *Accept* and *Deny* allow to define policies in explicit acceptance or denial form. For instance, to operate in mode “*everything that is not authorized is precluded*”, potentially to a specific domain, a policy should include a rule:

```
Deny: Domain[*] -> Domain[inria.fr] : *
```

The remainder of this section defines and details the different elements. We first identify the participating entities (Hosts, Domains, etc.), the security subjects (e.g. users), security objects (e.g. objects, resources, communication, ...), and of course the security relationships (i.e. attributes) between them.

4.1 Hierarchical Security Entities

Grid programming is about deploying processes (activities) on various machines. In the end, the security policy that must be ensured for those processes depends upon many factors: first of all, the application policy that is needed, but also the machine locations, the security policies of their administrative domain, and the network being used to reach those machines.

Section 3.2 defined the notions of *Virtual Nodes*, and *Nodes*. Virtual Nodes are application abstractions, and nodes are only a run-time entity resulting from the deployment: a mapping of Virtual Nodes to processes and hosts. A first decisive feature allows to define application-level security on those application-level abstractions:

Definition 1. *Virtual Node Security*

Security policies can be defined at the level of Virtual Nodes. At execution, that security will be imposed on the Nodes resulting from the mapping of Virtual Nodes to JVMs, and Hosts.

As such, virtual nodes are the support for intrinsic application level security. If, at design time, it appears that a process always requires a specific level of security (e.g. authenticated and encrypted communications at all time), then that process should be attached to a virtual node on which those security features are imposed. It is the designer responsibility to structure his/her application or components into virtual node abstractions compatible with the required security. Whatever deployment occurs, those security features will be maintained. We expect this usage to be rather occasional, for instance in very sensitive applications where even an intranet deployment calls for encrypted communications.

The second decisive feature deals with a major Grid aspect: deployment-specific security. The issue is actually twofold:

1. allowing organizations (security domains) to specify general security policies,
2. allowing application security to be specifically adapted to a given deployment environment.

Domains are a standard way to structure (virtual) organizations involved in a Grid infrastructure; they are organized in a hierarchical manner. They are the logical concept allowing to express security policies in a hierarchical way.

Definition 2. *Declarative Domain Security*

Fine grain and declarative security policies can be defined at the level of Domains. A Security Domain is a domain to which a certificate and a set of rules are associated.

This principle allows to deal with the two issues mentioned above:

(1) the administrator of a domain can define specific policy rules that must be obeyed by the applications running within the domain. However, a general rule expressed inside a domain may prevent the deployment of a specific application. To solve this issue, a policy rule can allow a well-defined entity to weaken it. As we are in a hierarchical organization, allowing an entity to weaken a rule means

allowing all entities included to weaken the rule. The entity can be identified by its certificate;

(2) a Grid user can, at the time he runs an application, specify additional security based on the domains being deployed onto.

The Grid user can specify additional rules directly in his deployment descriptor for the domains he deploys onto. Note that those domains are actually dynamic as they can be obtained through external allocators, or even Web Services in an OGSA infrastructure [5]. Joker rules might be important in that case to cover all cases, and to provide a conservative security strategy for un-forecasted deployments.

Finally, as active objects are active and mobile entities, there is a need to specify security at the level of such entities.

Definition 3. Active Object Security

Security policies can be defined at the level of Active Object. Upon migration of an activity, the security policy attached to that object follows.

In open applications, e.g. several principals interacting in a collaborative Grid application, a JVM (a process) launched by a given principal can actually host an activity executing under another principal. The principle above allows to keep specific security privileges in such case. Moreover, it can also serve as a basis to offer, in a secure manner, hosting environments for mobile agents.

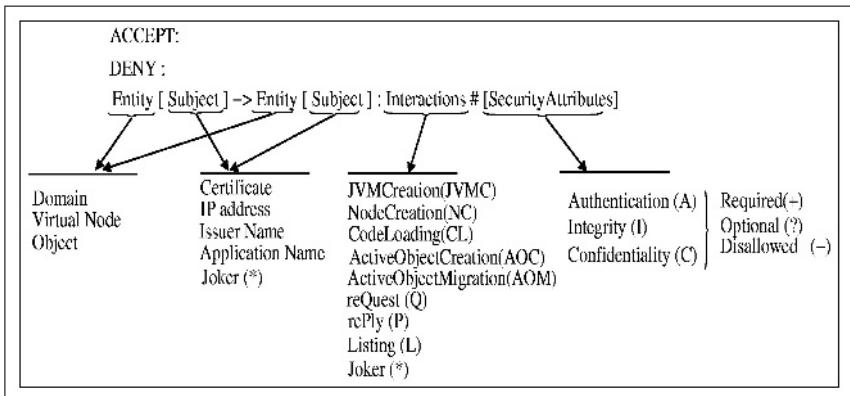


Fig. 2. Syntax and attributes for policy rules

4.2 Interactions, Security Attributes

Security policies are able to control all the *interactions* that can occur when deploying and executing a multi-principals Grid application. With this goal in mind, interactions span over the creation of processes (JVM in our case), to the monitoring of activities (ActiveObjects) within processes, including of course

the communications. Here is a brief description of those interactions:

- JVMCreation (JVMC): creation of a new JVM process
- NodeCreation (NC): creation of a new Node within a JVM (as the result of Virtual Node mapping)
- CodeLoading (CL): loading of bytecode within a JVM
- ActiveObjectCreation (AOC): creation of a new activity (active object) within a Node
- ActiveObjectMigration (AOM): migration of an existing activity object to a Node
- Request (Q), Reply (P): communications, method calls and replies to method calls
- Listing (L): list the content of an entity; for Domain/Node provides the list of Node/Active Objects, for Active Object allows to monitor its activity.

For instance, a domain is able to specify that it accepts downloading of code from a given set of domains, provided the transfers are authenticated and guaranteed not to be tampered with. As a policy might leave open the integrity of communications, and also because not allowing confidentiality can be a domain (or even a country) policy, those 3 security attributes can be specified in 3 modes: Required (+), Optional (?), Disallowed (-)

For a given interaction, a tuple $[+A,?I,-C]$ means that authentication is required, integrity is accepted but not required, and confidentiality is not allowed.

As a Grid operates in decentralized mode, without a central administrator controlling the correctness of all security policies, these policies must be *combined*, *checked*, and *negotiated* dynamically. The next two sections present that aspect.

4.3 Combining Policies

As the proposed infrastructure takes into account different actors of the Grid (domain administrator, Grid user), even for a single-principal single-domain application, there are potentially several security policies activated. This section deals with the combination of those policies to obtain the final security tuples of a single entity. An important principle being that a sub-domain cannot weaken a super-domain's rule. During execution, each activity (Active Object) is always included in a *Node* (due to the Virtual Node mapping) and at least in one *Domain*, the one used to launch a JVM (D_0). Figure 3 hierarchically represents the security rules that can be activated at execution: from the top, hierarchical domains (D_n to D_0), the virtual node policy (VN), and the Active Object (AO) policy. Of course, such policies can be inconsistent, and there must be clear principles to combine the various sets of rules.

There are three main principles: (1) choosing the *most specific rules* within a given domain (as a single Grid actor is responsible for it), (2) an interaction is valid only if all levels accept it (absence of weakening of authorizations), (3) the security attributes retained are the most constrained based on a partial order (absence of weakening of security).

The order on domains being inclusion ($d_1 < d_2$ iff d_1 included in d_2), the *Most Specific Rules* are defined as the set of *minimal elements* in the partial

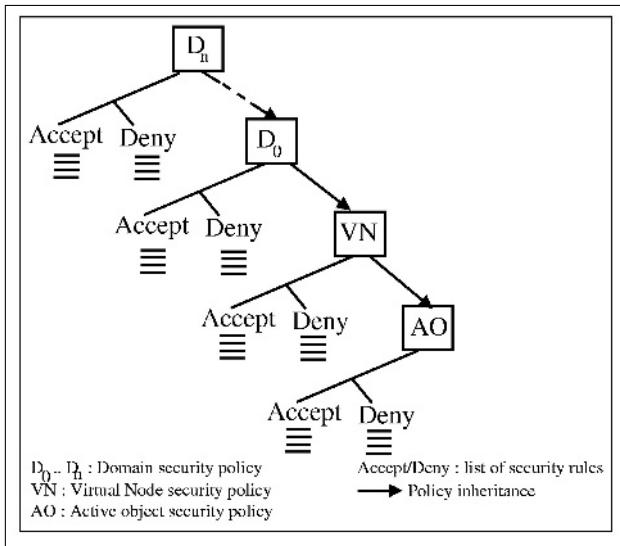


Fig. 3. Hierarchical security levels

order on rules defined as:

$$D_1 \rightarrow D_2 \leq D_3 \rightarrow D_4 \Leftrightarrow D_1 \leq D_3 \text{ and } D_2 \leq D_4$$

It constitutes a *semi-lattice* with the top element being $* \rightarrow *$. As an example, if a security policy is written:

```
Domain[*] -> Domain[*] : Q,P : [+A,+I,+C]
Domain[CardPlus] -> Domain[CardPlus] : Q,P : [+A,?I,?C]
```

within the CardPlus domain, the second rule will be chosen (integrity and confidentiality will be optional). Of course, comparison of rules is only a partial order, and several incompatible most specific rules can exist within a single level (e.g. both ACCEPT and DENY most specific rules for the same interaction, or both +A and -A).

Between levels, an incompatibility can also occur, especially if a sub-level attempts to weaken the policy on a given interaction (e.g. a domain prohibits confidentiality [-C] while a sub-domain or the Virtual Node requires it [+C], a domain D_i prohibits loading of code while D_j ($j \leq i$) authorizes it). In all incompatible cases, the interaction is not authorized and an error is reported.

4.4 Dynamic Policy Negotiation

During execution, entities interact by pair with each other. Each entity, for each interaction (JVM creation, communication, migration, ...), will want to apply a security policy based on the resolution presented in the previous section. Before starting an interaction, a *negotiation* occurs between the two entities involved. Figure 4 shows the result of such negotiation. For example, if for a

given interaction, entity A's policy is $[+A,?I,?C]$, and B's policy is $[+A,?I,-C]$, the negotiated policy will be $[A,?I,-C]$. If, for a result rule, one of the communication attributes is optional, the attribute is not activated.

Besides the interactions not being accepted by an entity, two other cases lead to an error: when an attribute is required by one, and disallowed by the other. In such cases, the interaction is not authorized and an error is reported. If a valid security policy is found between two entities, the interaction can occur. In the case that the agreed security policy includes confidentiality, the two entities negotiate a session key.

		ENTITY A		
		Required (+)	Optional (?)	Disallowed (-)
E N T I T Y	Required (+)	+	+	Error
	Optional (?)	+	?	-
	Disallowed (-)	Error	-	-

Fig. 4. Result of security negotiations

4.5 Migration and Negotiation

In large scale Grid applications, migration of activities is an important issue. The migration of Active Objects must not weaken the security policy being applied. When an active object migrates to a new location, three cases may happen :

- the object migrates to a node belonging to the same virtual node and included inside the same domain. In this case, all already negotiated sessions remain valid.
- the object migrates to a known node (created during the deployment step) but which belongs to another virtual node. In this case, all already negotiated sessions can be invalid. This kind of migration imposes re-establishing the object policy, and upon a change, re-negotiating with interacting entities.
- The object migrates to an unknown node (not known at the deployment step). In this case, the object migrates with a copy of the application security policy. When a secured interaction will take place, the security system retrieves not only the object's application policy but also policies rules attached to the node on which the object is to compute the policy.

5 Conclusion

We have proposed a decentralized, declarative security mechanism for distributed systems that features interoperability with local policies, dynamically negotiated security policies, and multi-users systems. The use of a logical representation of the application (Virtual Nodes) allows to have a security policy adaptable to the deployment, a crucial feature for Grid applications. The security mechanism allows to express all security-related configurations within domain policy files, and application deployment descriptor, outside the source code of the application. About security overhead, a full process (policy and session key negotiations) is achieved in 50 ms against 10 ms in unsecured mode.

As a very short term perspective, the implementation of group communication security is in the process of being finalized; a single session key being used for all group members. We do not advocate to have solved all Grid security problems. Rather, this work is an attempt to contribute to the construction of flexible solutions that are very much needed for Grid deployment.

References

1. Caromel, D., Klauser, W., Vayssi  re, J.: Towards Seamless Computing and Meta-computing in Java. *Concurrency Practice and Experience* **10** (1998) 1043–1061
2. Baude, F., Caromel, D., Mestre, L., Huet, F., Vayssi  re, J.: Interactive and descriptor-based deployment of object-oriented grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, IEEE Computer Society (2002) 93–102
3. Grimshaw, A., et al., W.W.: The Legion Vision of a World-wide Virtual Computer. *Communications of the ACM* **40** (1997)
4. Foster, I., Kesselman, C.: The Globus project: a status report. *Future Generation Computer Systems* **15** (1999) 607–621
5. Foster, I.T., Kesselman, C., Tsudik, G., Tuecke, S.: A Security Architecture for Computational Grids. In: ACM Conference on Computer and Communications Security. (1998) 83–92
6. Wesley, A., ed.: .NET Framework Security. Addison Wesley Professional (2002)
7. Puliafito, A., Tomarchio, O.: Security Mechanisms for the MAP Agent System (2000) In 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), 2000.
8. Karnik, N.M., Tripathi, A.R.: Security in the Ajanta Mobile Agent System. *Software, Practice and Experience* **31** (2001) 301–329
9. Karjoh, G., Lange, D., Oshima, M.: A Security Model for Aglets. *IEEE Internet Computing* **1** (1997) 68–77
10. Baumann, J., Hohl, F., Rothermel, K.: Mole - Concepts of a Mobile Agent System. Technical Report TR-1997-15, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, Distributed Systems (1997)
11. Sun Microsystems: Remote methode invocation
<http://java.sun.com/products/jdk/rmi> (2000)

A Middleware Substrate for Integrating Services on the Grid*

Viraj Bhat and Manish Parashar

The Applied Software Systems Laboratory,
Department of Electrical and Computer Engineering,
Rutgers, The State University of New Jersey,
94 Brett Road, Piscataway, NJ 08854
`{virajb, parashar}@caip.rutgers.edu`
<http://www.caip.rutgers.edu/TASSL>

Abstract. In this paper we present the design, implementation and evaluation of the Grid-enabled Discover middleware substrate. The middleware substrate enables Grid infrastructure services provided by the Globus Toolkit (security, information, resource management, storage) to interoperate with collaborative services provided by Discover (collaborative application access, monitoring, and steering). Furthermore, it enables users to seamlessly access and integrate local and remote services to synthesize customized middleware configurations on demand.

1 Introduction

Grid computing [1,2] is rapidly emerging as the dominant paradigm of wide area distributed computing. Its goal is to realize a persistent, standards-based service infrastructure that enables coordinated sharing of autonomous and geographically distributed hardware, software, and information resources. The emergence of such Grid environments has made it possible to conceive a new generation of applications based on seamless aggregations, integrations and interactions of resources, services/components and data. These Grid applications will be built on a range of services including multipurpose domain services for authentication, authorization, discovery, messaging, data input/output, and application/domain specific services such as application monitoring and steering, application adaptation, visualization, and collaboration. Recent years have also seen the development and deployment of a number of application/domain specific problem solving environments (PSEs) and collaboratories (e.g. Upper Atmospheric Research Collaboratory (UARC) [3], Discover [4], Astrophysics Simulation Collaboratory (ASC) [5], Diesel Combustion Collaboratory (DCC) [6], and Narrative-based, Immersive, Constructionist/Collaborative Environments for children (NICE) [7]). These systems provide specialized services to their user

* Support for this work was provided by the NSF via grant numbers ACI 9984357 (CA-REERS), EIA 0103674 (NGS) and EIA-0120934 (ITR), DOE ASCI/ASAP (Caltech) via grant numbers PC295251 and 1052856.

communities and/or address specific issues in wide area resource sharing and Grid computing. However, emerging Grid applications require combining these services in a seamless manner. For example, the execution of an application on the Grid requires security services to authenticate users and the application, information services for resource discovery, resource management services for resource allocation, data transfer services for staging, and scheduling services for application execution. Once the application is executing on the Grid, interaction, steering, and collaboration services allow geographically distributed users to collectively monitor and control the application allowing the application to be a true research or instructional modality. Once the application terminates data storage and clean up services come into play. While enabling laboratories/PSEs to share services and capabilities has many advantages, enabling such interoperability presents many challenges. The PSEs have evolved in parallel with the Grid computing effort and have been developed to meet unique requirements and support specific user communities. As a result, these systems have customized architectures and implementations, and build on specialized enabling technologies. Furthermore, there are organizational constraints that may prevent such interaction as it involves modifying existing software. A key challenge then, is the design and development of a robust and scalable middleware that addresses interoperability, and provides essential enabling services such as security and access control, discovery, and interaction and collaboration management. Such a middleware should provide loose coupling among systems to accommodate organizational constraints and an option to join or leave this interaction at any time. It should define a minimal set of interfaces and protocols to enable the PSEs to share resources, services, data and applications on the Grid while being able to maintain their architectures and implementations of choice. A key goal of the Global Grid Forum and the Open Grid Services Architecture (OGSA) [1] is to address these challenges by defining community standards and protocols.

The primary objective of this paper is to investigate the design of a prototype middleware that will enable interoperability between PSE/collaboratory and Grid services to support the overall execution of computational applications on the Grid. In this paper we present the design, implementation and evaluation of the Grid-enabled Discover middleware substrate that enables Grid infrastructure services provided by the Globus Toolkit [2] to interoperate with collaborative services provided by the Discover computational collaboratory, and enables users to seamlessly access and integrate local and remote services to synthesize customized middleware configurations on demand. This work builds on our previous work on the CORBA Community Grid (CoG) Kit [8] and Discover [4].

2 The Grid-Enabled Middleware Architecture

The overall goal of the Grid-enabled Discover middleware substrate is to define interfaces and mechanisms for integration and interoperation of the services provided by Discover and the Globus Toolkit. A schematic overview of the middleware substrate is presented in Figure 1, and consists of a network of peer

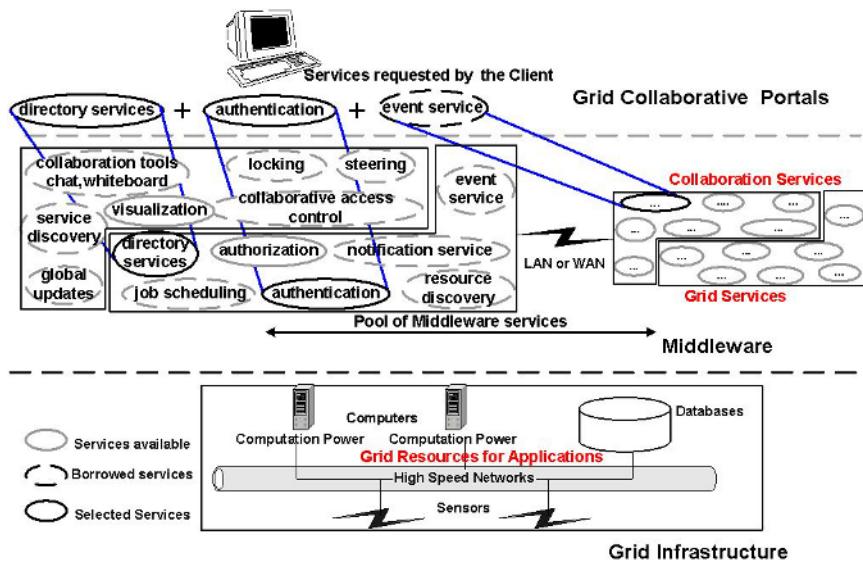


Fig. 1. Discover Grid-enabled middleware for interoperable collaboratories

hosts that export a selection of services. The middleware essentially provides a “repository of services” view to clients and controlled access to local and remote services. It can be thought of as consisting of two service layers distributed across on the Grid (see Figure 1): the ***Grid Service Layer*** consisting of Globus Grid services, and the ***Collaboratory Service Layer*** consisting of Discover collaborative services.

3 The Grid-Enabled Middleware Architecture

The Grid-enabled Discover middleware architecture consists of collaborative client portals at the front end, computational resources, services and applications at the backend and a network of peer hosts (servers) providing services in the middle. These components are described below. The prototype implementation of the middleware substrate builds on CORBA/IOP and provides peer-to-peer connectivity between hosts within and across domains. Server/service discovery mechanisms are built using the CORBA Naming and Trader services, which allows a server to locate remote servers and to access applications/services connected to the remote servers.

3.1 Discover Middleware Host (Server)

Discover interaction/collaboration servers build on commodity web servers, and extend their functionality (using Java Servlets) to provide specialized services

for Grid resource access, application deployment, management, real-time application interaction and steering for collaboration between client groups. Clients are Java applets and communicate with the server over HTTP. Application-to-server communication uses a standard distributed object protocols such as CORBA or a more optimized, custom protocol over TCP sockets. An *ApplicationProxy* object is created for each active application/service at the server, and is given a unique identifier. This object encapsulates the entire context for the application. Three communication channels are established between a server and an application for application registration and updates, client interaction requests and application responses respectively. Core service handlers provided by each server include the *MasterHandler*, *CollaborationHandler*, *CommandHandler*, Security/Authentication Handler, Grid Service Handlers (GSI, MDS, GRAM, GASS) and the Daemon servlet that listens for application connections. Details about the Discover servers can be found in [4].

3.2 Discover Middleware Services

The Discover Grid enabled middleware substrate defines interfaces for three classes of services. The first is the *DiscoverCorbaServer* service interface, which can be generally termed as the service discovery service. This service inherits from the CORBA Trader service and allows hosts to locate services on demand. The second is the *DiscoverCollab* service interface, which provides uniform access to local or remote collaborative services. Finally, the third class consists of interfaces to the Grid infrastructure services and provides uniform access to underlying Grid resources. This class includes the *DiscoverGSI*, *DiscoverMDS*, *DiscoverGRAM*, *DiscoverGASS* and *DiscoverEvent* service interfaces. Each host that is a part of the middleware substrate instantiates CORBA objects that implement these interfaces and are essentially wrappers around the corresponding services. Each host implements the *DiscoverCorbaServer* interface and may implement one or more of the other interfaces.

3.3 The Discover Portal

The Discover portal consists of a virtual desktop with local and shared areas. The shared areas implement a replicated shared workspace and enable collaboration among dynamically formed user groups. Locking mechanisms are used to maintain consistency. The base portal is presented to the user after authentication and access verification using Grid credentials. This provides the user with a list of available Grid and Collaboratory services that the user is authorized to access and allows the user to select the set of local or remote services to be used during the session. The application interaction desktop consists of (1) a list of interaction objects and their exported interaction interfaces (views and/or commands), (2) an information pane that displays global updates (current time step of a simulation) from the application, and (3) a status bar that displays the current mode of the application (computing, interacting) and the status of

issued command/view requests. The list of interaction objects is once again customized to match the client's access privileges. Chat and whiteboard tools can be launched from the desktop to support collaboration. View requests generate separate (possibly shared) panes using the corresponding view plug-in.

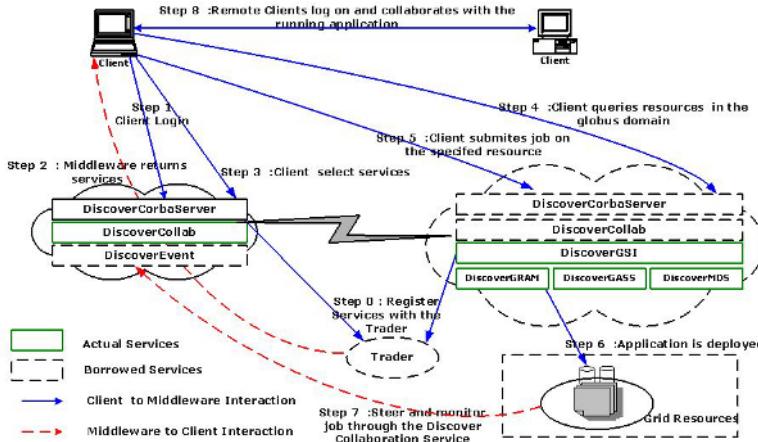


Fig. 2. Operation of the Discover Grid-enabled middleware.

4 Operation of the Discover Grid Enabled Middleware

This overall operation of the Grid enabled middleware is illustrated in Figure 2. Each host joins the middleware and registers its services with the CORBA Trader service (through the local *DiscoverCorbaServer* service). Each service is uniquely identified by trader by its name and the machine address of its host. A client logging onto the middleware through the Discover portal first authenticates with the *DiscoverCollab* service. The client is then presented with a list of all services and applications, local and remote, to which the client has access privileges. The client can now interactively compose and configure its middleware stack using these services, and can use this customized stack and associated local and remote Grid as well as Collaboratory services to acquire resources, configure and launch applications, connect to, monitor and steer the applications, terminate applications and collaborate with other users. Note the client has to perform a second level of authentication with the *DiscoverGSI* service before accessing available resources, services or applications. The credentials presented by the client during this authentication are used to delegate the required client proxies. Through these proxies, clients can discover local and remote resources using the *DiscoverMDS* service, allocate resources and run applications using *DiscoverGRAM* service, monitor the status of applications and resources using the *DiscoverEvent* service and perform data/file transfer using the *DiscoverGASS*

service. *DiscoverGRAM* also allows authorized users to terminate an application. The *DiscoverCollab* services enable the client to monitor, interact with and steer (local and remote) applications and to collaborate with other users connected to the middleware. Key operations are briefly described below.

Security/Authentication: The Discover security model is based on the Globus GSI protocol and builds on the CORBA Security Service. The GSI delegation model is used to create and delegate an intermediary object (CORBA GSI Server Object) between the client and the service. Each Discover server supports a two-level access control for collaborative services, the first level manages access to the server while the second level manages access to a particular application. Applications are required to be registered and provide a list of users and their access privileges. This information is used for customized access control.

Discovery of servers, applications and resources: Peer Discover servers locate each other using the CORBA trader services. The CORBA trader service maintains server references as service-offer pairs. All Discover servers are identified by the service-id “Discover”. The service offer contains the CORBA object reference and a list of properties defined as name-value pairs. Thus the object can be identified based on the service it provides or its properties. Applications are located using their globally unique identifiers, which are dynamically assigned by the Discover server and are a combination of the server’s IP address and a local count at the server. Resources are discovered using the Globus MDS Grid information service, which is accessed via the *MDSHandler* servlet and the *DiscoverMDS* service interface.

Accessing Globus Grid services - Job submission and remote data access: Discover middleware allows users to launch applications on remote resources using the *DiscoverGRAM* service. Clients invoke the *GRAMHandler* servlet to submit jobs. The *DiscoverGRAM* service submits jobs to the Globus gatekeeper after authenticating using the *DiscoverGSI* service. The user can monitor jobs using the *DiscoverEvent* service. Similarly, clients can store and access remote data using the *DiscoverGASS* service. The *GASSHandler* servlet invokes the delegated *DiscoverGASS* service to transfer files using the specified protocol.

Distributed Collaboration: The Discover collaborative enables multiple clients to collaboratively interact with and steer local and remote applications. The Collaboration Handler servlet at each middleware host handles the collaboration on its side, while a dedicated polling thread is used on the client side. All clients connected to an application instance form a collaboration group by default. However, as clients may connect to an application through a remote host, collaboration groups can span multiple hosts.

Distributed locking and logging for interactive steering and collaboration: Session management and concurrency control is based on capabilities granted by

the middleware. A simple locking mechanism is used to ensure that the application remains in a consistent state during collaborative interactions. This ensures that only one client “drives” (issues commands) to the application at any time. In the distributed middleware case, locking information is only maintained at the application’s middleware host i.e. the Discover middleware to which the application connects directly. The session archival handler maintains two types of logs. The first log maintains all interactions between a client and an application. For remote applications, the client logs are maintained at the middleware host where the clients are connected. The second log maintains all requests, responses, and status messages for each application throughout its execution. This log is maintained at the application’s middleware host (the middleware to which the application is directly connected). The Discover Grid-enabled middleware enables local and remote services to be combined in an ad hoc way and collectively used to get achieved desired behaviors. For example, consider the scenario as illustrated in Figure 3. In this example, a client copies log files generated by the application during a run using a remote *DiscoverGASS* service. The client logs on to the middleware (step 1) and access the logging collaboratory service (part of DiscoverCollab). The logging service uses the client’s credentials and the *DiscoverGSI* service (step 2) to create and delegate a proxy logging service (step 3). This proxy logging service interacts with the *DiscoverGASS* service (step 4) to transfer the log files to the local host (step 5). Note that these interactions are over a secure IIOP channel.

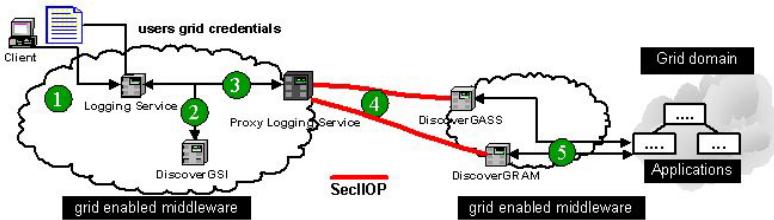
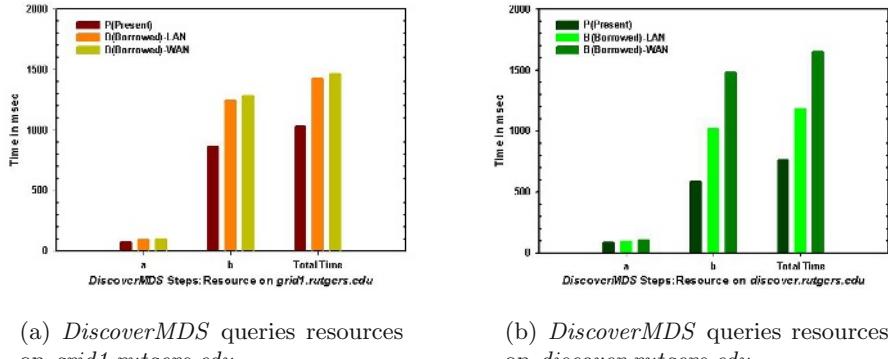


Fig. 3. Delegation model across services.

5 Experimental Evaluation

This section presents an experimental evaluation of the Discover middleware. It consisted of deployments at *grid1.rutgers.edu*, *discover.rutgers.edu* and *tassl-pc-2.rutgers.edu* at Rutgers University and *ajax.ices.utexas.edu* at University of Texas. Deployments at *grid1.rutgers.edu* and *ajax.ices.utexas.edu* had complete installations (Grid and Collaboratory services) while *discover.rutgers.edu* had only Grid services and *tassl-pc-2.rutgers.edu* had only Collaboratory services. We used the transport equation application kernel with adaptive mesh refinement (*tportamr*) for our experiments. The application was run on Beowulf clusters at Rutgers. The evaluations consisted of evaluating the latencies in accessing local

and remote services over local and wide area networks and are presented below. Note that an evaluation of collaborative services was presented in [4].



(a) *DiscoverMDS* queries resources on *grid1.rutgers.edu*

(b) *DiscoverMDS* queries resources on *discover.rutgers.edu*

Fig. 4. *DiscoverMDS* service discovers and queries resources on local machine and machine in the LAN **a** is the time to locate the *DiscoverMDS* service and **b** is the time to query the resources on the selected host.

Evaluation of the *DiscoverMDS* service: The evaluation of the *DiscoverMDS* service is divided into three cases. In the first case the *DiscoverMDS* service is locally present (case P). In the second case the *DiscoverMDS* service is borrowed from a remote host over LAN (case B-LAN). In the third case the *DiscoverMDS* service is borrowed from a remote host over WAN (case B-WAN). In all three cases clients used the *DiscoverMDS* service to discover resources at Rutgers. In each case, the experiment consists of two steps: (a) discovering the *DiscoverMDS* service using the CORBA Trader service and (b) invoking the service to discover resources. The times for steps (a) and (b) for discovering resources on *grid1.rutgers.edu* and *discover.rutgers.edu* are plotted in Figure 4(a) and 4(b) respectively. As seen in the plots, the time for discovering the service (step a) is small compared to the time for querying for resources (step b). This is primarily because of the overheads of querying MDS and packing, transporting and unpacking the large amount of returned resource information. Note that the average time for querying resources on *discover.rutgers.edu* is larger than that for *grid1.rutgers.edu* as *discover.rutgers.edu* is a 16 node cluster while *grid1.rutgers.edu* is a single processor machine.

Evaluation of the *DiscoverGRAM* service: The evaluation of *DiscoverGRAM* consisted of using the service to launch and terminate the *tportamr* application on *grid1.rutgers.edu*. Application deployment consisted of the following steps: (a) discovering the *DiscoverGRAM* service, (b) using *DiscoverGSI* to delegate a service proxy, (c) create an event channel for application monitoring, and (d) launch the application on the selected host i.e. *grid1.rutgers.edu*.

Application termination similarly consisted of the following steps: (f) discovering the *DiscoverGRAM* service, (g) using *DiscoverGSI* to delegate a service proxy, (h) creating an event channel for application monitoring, and (i) terminate the application selected. Note that the resource for launching the application and the application to be terminated are discovered and selected using the *DiscoverMDS* service. The times required for each step are plotted in Figure 5. As in the previous experiment, we consider three cases: in case P, the required services are local, in case B-LAN, the required services are borrowed over LAN, and in case B-WAN, the required services are borrowed over WAN. Note that the times for launching and terminating the application are quite comparable for the three cases. The large termination time is due to the cleanup performed by GRAM.

a	Resolving services: <i>DiscoverGRAM</i>
b	Delegation : <i>DiscoverGSI</i>
c	Event channel creation: <i>DiscoverEvent</i>
d	Job start time on <i>grid1.rutgers.edu</i>
e	Total time to start job - i.e. $\approx (a+b+c+d)$
f	Resolving services : <i>DiscoverGRAM</i>
g	Delegation with the <i>DiscoverGSI</i>
h	Event channel creation <i>DiscoverEvent</i>
i	Job cancellation time: <i>DiscoverGRAM</i>
j	Total time to cancel job - i.e $\approx (f+g+h+i)$

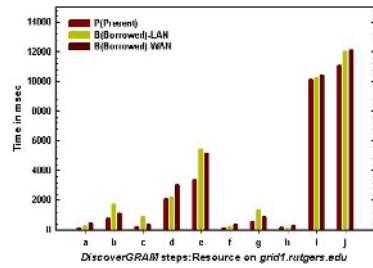


Fig. 5. *DiscoverGRAM* service launches the *tportamr* application using steps a, b, c, and d, and terminates the *tportamr* application using steps f, g, h, i on *grid1.rutgers.edu*.

Evaluation of the *DiscoverGASS* service: The evaluation of the *DiscoverGASS* service consisted of using the service to transfer files of different sizes. We measured the time required to transfer files between *grid1.rutgers.edu* and *discover.rutgers.edu*. This experiment considers the case P where the *DiscoverGASS* service was locally present. The file transfer times and the file sizes in bytes are plotted in Figure 6 using a log-log scale. The file sizes varied exponentially from 2 bytes to 10 MB. The equivalent file transfer times ranged from 9 msec. to 637 msec. It is seen that the *DiscoverGASS* performed well for small and medium file sizes (9 msec. for ≈ 2 bytes and 47 msec. for ≈ 1 MB). However the performance deteriorated (637 msec.) as file sizes approached 10 MB. The size of log files generated during the course of the *DiscoverGRAM* experiment was around 100 KB. We are currently evaluating cases where the service is borrowed over LAN (B-LAN) and over WAN (B-WAN).

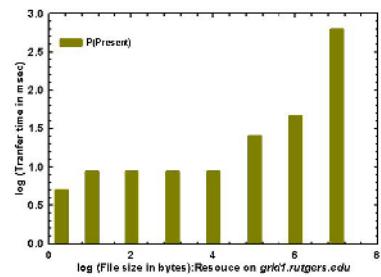


Fig. 6. Log-Log plot of transfer times for various file sizes using *DiscoverGASS* service (P case).

6 Conclusions

This paper presented the design, implementation, operation and evaluation of the Discover Grid-enabled middleware substrate. The middleware substrate enables Grid infrastructure services provided by the Globus Toolkit (security, information, resource management, storage) to interoperate with collaborative services provided by Discover (collaborative application access, monitoring, and steering). Furthermore, it enables users to seamlessly access and integrate local and remote services to synthesize customized middleware configurations on demand. Clients can use the Grid as well as Collaboratory services integrated by the middleware to acquire resources, configure and launch applications, connect to monitor and steer the applications, terminate applications and collaborate with other users.

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid:An Open Grid Services Architecture for Distributed Systems Integration. In: Proceedings of the Open Grid Service Infrastructure WG, Global Grid Forum. (2002)
2. Foster, I., Kesselman, C.: Globus:A Metcomputing Infrastructure Toolkit. International Journal of Supercomputer Applications **11** (1997) 115–128
3. Olson, G., Atkins, D.E., Finholt, T., Clauer, R.: The Upper Atmospheric Research Collaboratory. ACM Interactions **5** (1998) 48–55
4. Mann, V., Parashar, M.: Middleware Support for Global Access to Integrated Computational Collaboratories. In: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing,IEEE Computer Society Press, San Francisco, CA (2001) 35–46
5. Russell, M., Allen, G., Daues, G., von Laszewski, G.: The Astrophysics Simulation Collaboratory A Science Portal for Enabling Community Software Development. In: Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing, San Francisco, CA (2001) 207–215
6. Diachin, D., Freitag, L., Heath, D., Herzog, J., Michels, W., Plassmann, P.: Remote Engineering Tools for the Design of Pollution Control Systems for Commercial Boilers. International Journal of Supercomputer Applications **10** (1996) 208–218
7. Roussos, M., Johnson, A., J. Leigh, C.B., Vasilakis, C., Moher, T.: The NICE Project: Narrative, Immersive, Constructionist/Collaborative Environments for Learning in Virtual Reality. In: Proceedings of ED-MEDIA/ED-TELECOM 97, Calgary, Canada (1997) 917–922
8. Verma, S., Parashar, M., Gawor, J., von Laszewski, G.: Design and Implementation of a CORBA Community Grid Kit. In: Proceedings of the 2nd International Workshop on Grid Computing,Lecture Notes in Computer Science,Editors:C. A. Lee, Springer-Verlag, Denver, CO (2001) 2–13

Performance Analysis of a Hybrid Overset Multi-block Application on Multiple Architectures^{*}

M. Jahed Djomehri¹ and Rupak Biswas²

¹ CSC, NASA Ames Research Center, Moffett Field, CA 94035;
`djomehri@nas.nasa.gov`

² NAS Division, NASA Ames Research Center, Moffett Field, CA 94035;
`Rupak.Biswas@nasa.gov`

Abstract. This paper presents a detailed performance analysis of a multi-block overset grid CFD application on multiple state-of-the-art computer architectures. The application is implemented using a hybrid MPI+OpenMP programming paradigm that exploits both coarse and fine-grain parallelism. The hybrid model also extends the applicability of multi-block programs to large clusters of SMP nodes. Extensive investigations were conducted on Cray SX-6, IBM Power3 and Power4, and SGI Origin3000 platforms. Overall results for complex vortex dynamics simulations demonstrate that the SX-6 achieves the highest performance and outperforms the RISC-based architectures; however, the best scaling performance was achieved on the Power3.

1 Introduction

A large fraction of high performance computing (HPC) platforms today use cache-based microprocessors, which are assembled as systems of symmetric multi-processing (SMP) nodes. Additionally, in order for many important scientific “legacy codes” originally developed for vector machines to perform efficiently on cache-based architectures, some significant changes in their algorithmic structures must be made. Recent development of vector-based SMP systems [4] offers a new HPC alternative for many of these legacy codes. To evaluate and compare the performance of state-of-the-art cache- and vector-based architectures in conjunction with practical scientific problems, we have selected a high-fidelity NASA production Navier-Stokes CFD application, called OVERFLOW-D [5], that is based on multi-block overset grid methodology. A brief overview of the application is given in Section 2.

This paper describes our performance analysis of a hybrid MPI+OpenMP programming paradigm implementation of OVERFLOW-D when executed on different computer architectures. The approach consists of two levels of parallelism: the first is coarse-grained based on MPI message passing while the

* First author supported by NASA under contract DTTS59-99-D-00437/A61812D.

second is fine-grained based on OpenMP directives [3]. One major advantage of the combined paradigm is that it extends the applicability of multi-block applications to large clusters of SMP nodes. Details of the hybrid model as applied to OVERFLOW-D is presented in Section 3. Furthermore, we describe the modifications that were made to a key numerical algorithm of the application, namely the LU-SGS linear solver, in order to enhance the parallel performance of the hybrid implementation. These modifications are reported in Section 4.

All performance evaluations were conducted on four parallel machines: the Cray SX-6 vector system, the cache-based IBM Power3 and Power4 machines, and the shared memory SGI Origin3000 platform. A brief description of these architectures is given in Section 5. Performance results obtained using complex vortex dynamics simulations of a realistic problem are presented in Section 6. Overall results demonstrate that the SX-6 outperforms the RISC architectures; however, the Power3 demonstrates the best scalability and the Origin3000 achieves the highest sustained floating-point performance relative to peak.

2 Overset Methodology

In this section, we provide a brief overview of the high-fidelity multi-block overset grid application for Navier-Stokes simulations, called OVERFLOW-D [5].

2.1 Flow Solver

The overset grid application is popular within the aerodynamics community due to its ability to handle complex designs with multiple geometric components. OVERFLOW-D is explicitly designed to simplify the modeling of problems when components are in relative motion. At each time step, the flowfield equations are solved independently on each grid (also known as blocks or zones) in a sequential manner. Overlapping boundary points or inter-grid data is updated from previous solutions prior to the start of the current time step using a Chimera interpolation procedure [6]. The code uses finite differences in space, with a variety of implicit/explicit time stepping. A domain connectivity program is used to determine the inter-grid boundary data.

The main computational logic at the top level of the sequential code consists of a “time-loop” and a “grid-loop”, where the grid-loop is nested inside the time-loop. Within the grid-loop, solutions are obtained on the individual grids with imposed boundary conditions, where the Chimera interpolation procedure successively updates inter-grid boundaries after computing the numerical solution on each grid. Upon completion of the grid-loop, the solution is automatically advanced to the next time step by the time-loop. The overall procedure may be thought of as a Gauss-Seidel iteration.

2.2 Grid Connectivity

The Chimera interpolation procedure [6] determines the proper connectivity of the individual grids. Adjacent grids are expected to have at least a one-cell over-

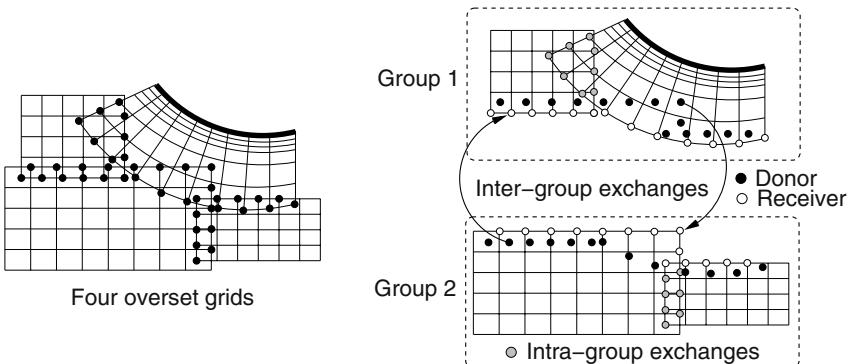


Fig. 1. Overset grid intra-group and inter-group communication.

lap to ensure solution continuity. A program named Domain Connectivity Function (DCF) [5] computes the inter-grid donor points that have to be supplied to other grids (see Fig. 1). The DCF procedure is incorporated into OVERFLOW-D and fully coupled with the flow solver. All boundary exchanges are conducted at the beginning of every time step based on the interpolatory updates from the previous time step. In addition, for dynamic grid systems, DCF has to be invoked at every time step to create new inter-grid boundary data.

3 Hybrid Programming Model

In the following subsections, we briefly describe a two-level parallel model [3] based on MPI+OpenMP that has been incorporated into our CFD application.

3.1 MPI Parallelization

The first level of parallelization is based on the MPI library using the single program multiple data (SPMD) paradigm. The MPI model has been developed around the multi-block feature of the sequential code, which offers a natural coarse-grain parallelism [8]. To facilitate parallel execution, a grouping strategy is required to assign each grid to an MPI process, and concurrently distribute the workload in a balanced fashion. The total number of groups, G , is equal to the number of MPI processes, M . Since a grid can only belong in one group, the total number of grids, Z , must be at least equal to M . A number of different grouping strategies [2] are available for overset grid applications. In this paper, the bin-packing approach was used and is described later. The assignment of groups to processors is somewhat random, and is taken care of by the operating system, usually based on a first-touch strategy at the time of the run.

The logic in the MPI model differs slightly from that of the sequential case mentioned in 2.1. Here the grid-loop is subdivided into two procedures, a loop over groups (“group-loop”) and a loop over the grids within each group. Since each MPI process is assigned to only one group, the group-loop is performed in

parallel, with each group performing its own sequential grid-loop. The inter-grid boundary updates among the grids within each group (called intra-group updates) are performed as in the serial case. Chimera updates are also necessary for overlapping grids that are in different groups, and are known as inter-group exchanges (see Fig. 1). These inter-group exchanges are transmitted at the beginning of every time step based on interpolations from the previous time step. Message passing is via MPI asynchronous communication calls.

Grouping Algorithm. The original parallel version of OVERFLOW-D uses a grid grouping strategy based on a bin-packing algorithm [8]. It is one of the simplest clustering techniques that strives to maintain a uniform number of “weighted” grid points per group while retaining some degree of connectivity among the grids within each group. Prior to the grouping procedure, each grid is weighted depending on the physics of the solution sought. The goal is to ensure that each weighted grid point requires the same amount of computational work. The bin-packing algorithm then sorts the grids by size in descending order, and assigns a grid to every empty group. Therefore, at this point, the G largest grids are each in a group by themselves. The remaining $Z - G$ grids are then handled one at a time: each is assigned to the smallest group that satisfies the connectivity test with other grids in that group. The connectivity test only inspects for an overlap between a pair of grids, regardless of the size of the boundary data or their connectivity to other neighboring grids. The process terminates when all grids are assigned to groups.

3.2 OpenMP Parallelization

The second level of parallelism in the hybrid approach is based on the OpenMP programming model, where explicit compiler directives are inserted into the code at the loop level. The logic is the same as in the pure MPI case, only the computationally intensive portion of the code (i.e. the grid-loop) is multi-threaded via OpenMP. In our current implementation, an equal number of OpenMP threads are spawned for each MPI task. The total number of processors used is the product of the number of MPI tasks and OpenMP threads. The OpenMP thread initialization follows a fork/join procedure. Whenever a parallel region is encountered, one of threads acts as the master while the others behave as team members; otherwise the master executes alone while the others remain idle. Message passing is performed by the master thread only; in other words, there is no inter-group communication among the threads. Master threads within each MPI task exchange inter-group boundary data in OVERFLOW-D.

4 LU-SGS Reorganization

Both the pure MPI and hybrid programming models discussed above were developed based on the sequential version of OVERFLOW-D, the organization of which was designed to exploit vector machines. The same basic code structure

is used on all our target machines, except for the LU-SGS linear solver that required significant modifications to enhance efficiency. LU-SGS combines the advantages of LU factorization and Gauss-Siedel relaxation to improve numerical convergence. Unfortunately, the inherited data dependencies in the scheme require the availability of the solution on the previous diagonal line before advancing. This “hyper-line” algorithm, similar to the hyper-plane scheme [1], was used in the original code to achieve reasonable parallel performance on vector machines. However, for cache-based architectures, there are two main deficiencies: poor cache utilization and small communication granularity. In fact, a naive version of the OpenMP LU-SGS code performed very poorly on an Origin3000 due to the insertion of OpenMP directives into some of the inner loops.

A smarter approach to parallelize the LU-SGS scheme is the pipeline algorithm described in [9] that has better cache performance and less communication than the hyper-plane method. The new parallel version improved hybrid performance significantly and was implemented on the Power3, Power4, and Origin3000 machines, while a vector strategy was executed on the SX-6. Except for a few minor changes in some subroutines to meet the specific MPI/OpenMP compiler requirements on each platform, the LU-SGS program has been the only OVERFLOW-D module to be reorganized to enhance efficiency.

5 Target Architectures

All experiments were performed on four state-of-the-art parallel machines: Cray SX-6, IBM Power3 and Power4, and SGI Origin3000. The cacheless SX-6 uses vectorization to exploit regularities in the computational structure, thereby expediting uniform operations on independent data sets. The system at Arctic Region Supercomputing Center is a single SMP node consisting of eight 500 MHz processors, each with a peak performance of 8 Gflops/s. Since the SX-6 vector unit is significantly more powerful than the scalar processor, it is critical to achieve high vector operation ratios, either via compiler discovery or explicitly through code (re-)organization.

The Power3 system at Lawrence Berkeley National Laboratory has 380 SMP compute nodes, each with 16 375 MHz processors and a peak performance of 1.5 GFlops/s per processor. Multi-node configurations are networked via the IBM Colony switch using an omega-type topology. The Power4 pSeries 690 is the latest generation of IBM’s RS/6000 product line. The temporary system at NASA Ames Research Center (ARC) was composed of two 32-way SMP nodes, coupled together via the Colony switch. Each SMP consists of 16 Power4 chips, where a chip contains two 1.3 GHz processor cores. Each core has a peak performance of 5.2 Gflops/s.

The SGI Origin3000 is a scalable, hardware-supported cache-coherent nonuniform memory access (CC-NUMA) system, with an aggressive communication architecture. The hardware makes all memory equally accessible from a software perspective by sending memory requests through routers located on the nodes. Memory access time is nonuniform, depending on how far away the

word resides from the processor. Results presented in this paper were obtained on the 128-node system at ARC. Each Origin3000 node contains four 400 MHz processors and a peak performance of 0.8 GFlops/s per processor.

6 Performance Results

The CFD problem used for the experiments in this paper is a Navier-Stokes simulation of vortex dynamics in the complex wake flow region for hovering rotors. Our test case consists of 41 blocks and approximately eight million grid points. Figure 2 shows a sectional view of the grid system and a cut plane through the computed vortex wake. The Cartesian off-body wake grids surround the curvilinear near-body grids with uniform resolution, but become gradually coarser upon approaching the outer boundary of the computational domain. A complete description and analysis of this application can be found in [7].

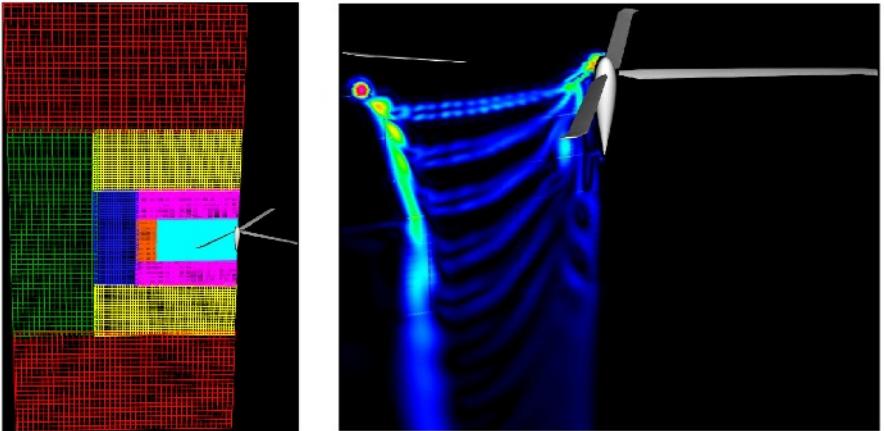


Fig. 2. Sectional view of grid system and computed vorticity magnitude contours.

Tables 1, 2, and 3 show total execution timings, T_{exec} , on the Cray SX-6, IBM Power3 and Power4, and SGI Origin3000 systems, respectively. T_{exec} is the time required to solve every iteration of the application (averaged over 20 iterations), and includes the computation, communication, Chimera interpolation, and processor idle times. T_{exec} is reported for both the MPI and hybrid paradigms to demonstrate the impact of the second level of parallelism introduced by OpenMP. The hybrid runs with M MPI tasks and one OpenMP thread are conceptually equivalent to pure MPI runs with M tasks; however, due to procedural differences, the timings may be different.

Performance results on the SX-6 are presented in Table 1. The MPI and hybrid paradigms are appended with a $-V$ or $-NV$ to indicate whether or not the code was vectorized, with respect to the LU-SGS linear solver. The

Table 1. Performance results on the Cray SX-6 system

P	MPI Tasks	OpenMP Threads	Paradigm	T_{exec} (sec)	Mflops/s	VOR (%)
2	2	—	MPI-NV	16.4	760	77
2	2	—	MPI-V	5.5	2265	80
2	2	1	Hybrid-NV	16.7	746	77
2	2	1	Hybrid-V	5.6	2492	77
4	4	—	MPI-NV	9.1	1369	68
4	4	—	MPI-V	2.8	4450	76
4	4	1	Hybrid-NV	9.1	1369	68
4	4	1	Hybrid-V	2.8	4450	71
4	2	2	Hybrid-V	3.6	3461	71
6	6	—	MPI-NV	5.7	2185	67
6	6	—	MPI-V	2.0	6230	73
6	6	1	Hybrid-NV	5.9	2111	67
6	6	1	Hybrid-V	2.1	5934	68
6	2	3	Hybrid-V	3.0	4153	66
8	8	—	MPI-NV	5.9	2111	61
8	8	—	MPI-V	1.6	7787	69
8	8	1	Hybrid-NV	6.1	2042	60
8	8	1	Hybrid-V	1.6	7787	69
8	2	4	Hybrid-V	2.5	4984	67
8	4	2	Hybrid-V	1.8	6922	68

table includes data regarding floating point operations per second (Mflops/s) and vector operation ratio (VOR). Observe that T_{exec} for runs with the vectorized version of LU-SGS are smaller than the non-vectorized ones by at least a factor of three, signifying the performance improvement gained by vectorizing the solver. The relatively limited VOR explains why the code achieves a maximum of only 7.8 Gflops/s on eight processors (12% of peak). Reorganizing OVERFLOW-D would achieve higher vector performance; however, extensive effort would be required to modify this production code.

Except for $P = 8$, the hybrid paradigm slightly underperforms MPI due to the overhead associated with OpenMP thread management. For a given total number of processors, runs with larger numbers of OpenMP threads appear to be less efficient than those with fewer threads. This is also due to OpenMP overheads. However, the primary advantage of using the hybrid paradigm for overset grid applications is that it allows execution on larger processor counts. The performance scalability for both paradigms is almost identical but is expected to suffer for large numbers of MPI tasks due to workload imbalance.

Timing results and Mflops/s on the IBM Power3 and Power4 systems are shown in Table 2. As expected, the Power4 outperforms the Power3 over the entire range of processors. Note that for $P = 32$, the Power3 runs were split across two SMP nodes communicating via the Colony switch (indicated by 2N in the table); whereas all runs on the Power4 were executed on one SMP node enjoying fast intra-cabinet interconnects. Nevertheless, the Power3 results are

Table 2. Performance results on the IBM Power3 and Power4 systems

P	MPI Tasks	OpenMP Threads	Paradigm	Power3		Power4	
				T_{exec} (sec)	Mflops/s	T_{exec} (sec)	Mflops/s
2	2	—	MPI	46.7	266	15.8	788
2	2	1	Hybrid	28.9	431	18.2	684
4	4	—	MPI	26.6	468	8.5	1465
4	4	1	Hybrid	14.9	838	10.1	1233
4	2	2	Hybrid	15.2	819	10.5	1186
8	8	—	MPI	13.2	943	4.3	2897
8	8	1	Hybrid	7.4	1683	6.0	2076
8	2	4	Hybrid	9.2	1354	5.9	2111
8	4	2	Hybrid	8.0	1557	6.2	2009
16	16	—	MPI	8.0	1557	3.7	3367
16	16	1	Hybrid	4.6	2708	4.5	2768
16	8	2	Hybrid	4.1	3039	3.9	3194
16	4	4	Hybrid	4.8	2595	3.7	3367
16	2	8	Hybrid	7.6	1639	4.0	3115
32	32	—	MPI	4.5 (2N)	2768	3.4	3664
32	32	1	Hybrid	4.7 (2N)	2651	2.8	4450
32	16	2	Hybrid	2.4 (2N)	5191	3.6	3461
32	8	4	Hybrid	2.6 (2N)	4792	2.7	4614
32	4	8	Hybrid	3.8 (2N)	3461	2.8	4450
32	2	16	Hybrid	14.1 (2N)	883	3.4	3664

impressive. For small numbers of processors ($P = 2$ and $P = 4$), the Power4 timings are significantly better than those for the Power3; this is due to the Power4's faster clock and complex but effective data locality system implemented via the architectural association of the L1, L2, and L3 caches.

For $P = 8$, both systems achieve about 7% of peak performance for the pure MPI runs; however, the hybrid paradigm on the Power3 runs at more than 14% of peak. For $P = 32$, the Power3 remains scalable (achieving 11% of peak), whereas the Power4 performance deteriorates significantly. This is probably because of the complex architecture of the Power4 which we were unable to fully exploit. Timing comparisons between the pure MPI and hybrid paradigms on the Power3 show that the latter outperforms the former, and for some cases, by nearly a factor of two. Rather surprisingly, the hybrid runs with one thread are also significantly better than MPI for the same value of P . This is probably due to OpenMP compiler optimizations in the hybrid case. On the Power4, MPI performs better than the hybrid strategy for $P < 16$; however, the reverse is true for $P = 32$. As on the SX-6, runs with larger numbers of OpenMP threads beyond an optimal value is less efficient, for a fixed value of P . On the Power3, the timing for $P = 32$ with two MPI tasks and 16 OpenMP threads is extremely poor; the reasons being a lack of data locality and that the job is split across two SMP nodes.

Timing results and floating point operations per second on the SGI Origin3000 are shown in Table 3. The overall performance trend is similar to that

Table 3. Performance results on the SGI Origin3000 system

P	MPI Tasks	OpenMP Threads	Paradigm	T_{exec} (sec)	Mflops/
2	2	—	MPI	39.7	313
2	2	1	Hybrid	39.9	312
4	4	—	MPI	21.8	571
4	4	1	Hybrid	22.0	566
4	2	2	Hybrid	21.5	579
8	8	—	MPI	11.2	1112
8	8	1	Hybrid	11.3	1102
8	2	4	Hybrid	14.1	883
8	4	2	Hybrid	13.7	909
16	16	—	MPI	6.1	2042
16	16	1	Hybrid	6.3	1977
16	8	2	Hybrid	7.6	1639
16	4	4	Hybrid	6.6	1887
16	2	8	Hybrid	8.3	1501
32	32	—	MPI	3.8	3278
32	32	1	Hybrid	3.8	3278
32	8	4	Hybrid	3.4	3664
32	4	8	Hybrid	4.7	2651
32	2	16	Hybrid	8.8	1416

of the Power4. Surprisingly, when using two tasks and 16 threads, performance is extremely poor. Similar observations could also be made for the Power3 from Table 2. There are at least two possible reasons: lack of data locality and the overhead associated with OpenMP procedures (such as fork/join and synchronization). Note that performance on the Power4 for this case is not adversely affected because of its better cache structure and larger node size. A maximum of 3.6 Gflops/s is achieved on 32 processors (14% of peak performance). As on the SX-6 and the Power4, MPI results are slightly better than those with the hybrid scheme. Also, increasing the number of OpenMP threads does not help.

In terms of absolute timings (T_{exec}), the SX-6 (when running the vectorized solver) outperforms the other three architectures. Results show that the best run time for eight processors on the SX-6 (1.6 secs) is more than 40% less than the best 32-processor Power4 number (2.7 secs). Scalability on the Power3 exceeds all others; the Origin3000 ranks second for our test application. The Origin3000 also demonstrated the highest sustained performance (14% of peak on 32 processors).

The hybrid programming paradigm is the most complex as it combines two layers of coarse- and fine-grain parallelism. In general, it therefore requires more programmer effort; however, our results show that for the same total number of processors, the best hybrid run performs comparably to that of the pure MPI implementation. On the Power3 though, the hybrid results were significantly (and rather surprisingly) better than MPI. Adding more OpenMP threads beyond an optimal number, depending on the number of MPI tasks, did not improve performance. However, the primary advantage of the hybrid paradigm for overset

grid applications is that it extends their applicability to large clusters of SMP nodes. In other words, hybrid programming is particularly appropriate when the number of overset grids is less than the number of processors to be used, or when load balancing becomes difficult due to the wide disparity in grid sizes [2].

7 Summary and Conclusions

In this paper, we presented a detailed performance analysis of an MPI+OpenMP implementation of a high-fidelity multi-block CFD application on multiple computer architectures. We conducted our experiments with a realistic vortex dynamics problem on Cray SX-6, IBM Power3 and Power4, and SGI Origin3000 machines. We demonstrated the important role of restructuring a key kernel of the application, namely the LU-SGS linear solver, to improve performance on these platforms. We showed that in terms of execution timings, the SX-6 outperforms the other three architectures; in fact, the best run time for eight processors on the SX-6 is more than 40% less than the best 32-processor run on the Power4. However, Power3 scalability exceeds all others, and the Origin3000 demonstrated the highest sustained performance: 14% of peak on 32 processors. We conclude that even though the pure MPI approach demonstrated a slight edge over the hybrid method, both paradigms still perform similarly for the same total number of processors, except for the Power3 where the hybrid results were significantly better.

References

1. E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga, Solution of regular, sparse triangular linear systems on vector and distributed-memory multiprocessors, *Tech. Rep. RNR-93-007*, NASA Ames Research Center, 1993.
2. M.J. Djomehri, R. Biswas, and N. Lopez-Benitez, Load balancing strategies for multi-block overset grid applications, in: *Proc. 18th Intl. Conf. on Computers and Their Applications* (Honolulu, HI, 2003) 373–378.
3. M.J. Djomehri and H. Jin, Hybrid MPI+OpenMP programming of an overset CFD solver and performance investigations, *Technical Report NAS-02-002*, NASA Ames Research Center, 2002.
4. Earth Simulator Center, See URL <http://www.jamstec.go.jp>.
5. R. Meakin, On adaptive refinement and overset structured grids, in: *Proc. 13th AIAA Computational Fluid Dynamics Conf.* (Snowmass, CO, 1997) Paper 97-1858.
6. J.L. Steger, F.C. Dougherty, and J.A. Benek, A Chimera grid scheme, *Advances in Grid Generation*, ASME FED-5 (1983).
7. R.C. Strawn and M.J. Djomehri, Computational modeling of hovering rotor and wake aerodynamics, *Journal of Aircraft* 39 (2002) 786–793.
8. A.M. Wissink and R. Meakin, Computational fluid dynamics with adaptive overset grids on parallel and distributed computer platforms, in: *Proc. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, NV, 1998) 1628–1634.
9. M. Yarrow and R. Van der Wijngaart, Communication improvement for the LU NAS Parallel Benchmark: A model for efficient parallel relaxation schemes, *Technical Report NAS-97-032*, NASA Ames Research Center, 1997.

Complexity Analysis of a Cache Controller for Speculative Multithreading Chip Multiprocessors

Yoshimitsu Yanagawa¹, Luong Dinh Hung², Chitaka Iwama²,
Niko Demus Barli², Shuichi Sakai², and Hidehiko Tanaka²

¹ The Institute of Space and Astronautical Science,
3-1-1 Yoshinodai, Sagamihara, Kanagawa 229-8510, Japan
yanagawa@pub.isas.ac.jp

² Graduate School of Information Science and Technology, The University of Tokyo,
7-3-1 Hongo Bunkyo-ku, Tokyo 113-8654, Japan
{hung,chitaka,niko,sakai,tanaka}@mtl.t.u-tokyo.ac.jp

Abstract. Although many performance studies of memory speculation mechanisms in speculative multithreading chip multiprocessors have been reported, it is still unclear whether the mechanisms are complexity effective and worth implementing. In this paper, we perform a complexity analysis of a cache controller designed by extending an MSI controller to support thread-level memory speculation. We model and estimate the delay of the control logic on critical paths and the area overhead to hold additional control bits in the cache directory. Our analysis shows that for many protocol operations, the directory access time occupies more than half of the total delay. The total overhead is however smaller than the delay for accessing the cache tags. Since the protocol operations can be performed in parallel with the tag access, the resulting critical path latency is only slightly increased.

1 Introduction

Chip Multiprocessor (CMP) architecture is becoming increasingly attractive because of its capability to exploit Thread Level Parallelism (TLP) in multi-threaded or multiprogram workloads. However, to be fully accepted as a general purpose platform, it must also deliver high performance when executing sequential applications. In order to extract TLP from sequential applications, a technique called speculative multithreading has been proposed [8,3,9,5,11,6]. In speculative multithreading execution, a sequential program is partitioned into threads and speculatively executed in parallel.

Speculative multithreading architectures usually use a combination of control and data speculation to exploit TLP. One of useful speculation techniques is thread-level memory speculation, in which a thread executes memory *load* speculatively regardless the possibility that a predecessor thread may *store* into

the same memory location. A number of hardware based mechanisms to support memory speculation have been proposed. These mechanisms usually extend the functionality of cache to buffer speculative state of memory and detect memory dependency violations [2, 3, 9, 5].

Although many performance studies of the memory speculation mechanisms have been reported, it is still unclear whether the mechanisms are complexity effective and worth implementing. This research is an effort to answer this question. Here, we first modify the MSI (Modified-Shared-Invalid) cache coherence protocol to support thread-level memory speculation. The hardware organization of the cache controller necessary to implement the protocol is then described and complexity analysis is performed. We model and estimate the delay of the control logic on critical paths and the area overhead to hold additional control bits in the cache directory. For the control logic circuits, we assume an implementation that uses static CMOS circuit style. The delays of the circuits are then estimated using the method of logical effort [10]. For the cache directory, the delay is estimated using the CACTI tool [7].

We found that for a 32-kB cache with 64-byte lines, the increase in area of the cache directory over the original MSI cache directory is significant. For many protocol operations, the directory access time occupies more than half of the total delay. This delay overhead is however smaller than the delay for accessing the cache tags. Since cache directory access and protocol logic operation can be performed in parallel with the cache tag access, significant increase in the critical path delay can be avoided. In such a case, our analysis shows that the critical path latency is increased slightly by 9%. This can be further reduced or eliminated if we employ a faster circuit style such as domino logic circuits.

The rest of this paper is organized as follows. Section 2 describes the thread-level memory speculation model assumed in this research and related work. Section 3 explains cache controller supports for the memory speculation. Methodology of the complexity analysis is described in Section 4 and the results are discussed in Section 5. Finally Section 6 concludes the paper.

2 Speculative Multithreading Architecture

2.1 Baseline Model

The CMP architecture assumed in this paper consists of four Processing Units (PUs) with private register file, L1 I-Cache, and L1 D-Cache. A unified L2 cache is shared by all the PUs. The CMP also has a centralized *thread control unit* which predicts threads following the program order and schedules them to the PUs in a round-robin fashion.

A sequential program is partitioned into threads at compile time. A thread is defined as a connected subgraph of a static control flow graph with a single entry point similar to Multiscalar model [8]. Thread boundaries are put at function invocations, innermost loop iterations, and at remaining points necessary to satisfy the thread definition.

Data dependences through register are synchronized using a combination of compiler and hardware supports. On the other hand, memory dependences are handled speculatively. A load instruction in a speculative thread is allowed to execute before all possible dependences resolve. Violation occurs when a predecessor thread stores to the same address as the load later in time. In this case, the violating thread and all its successor must be squashed and restarted. The design of the mechanism is discussed in detail in Section 3.

2.2 Related Work

Many models and hardware/software supports for thread level memory speculation have been proposed [1, 2, 3, 5, 9, 11]. Among them, [3, 5, 9] provide details on how to implement the mechanism on CMP caches.

Hydra [3] proposed an extension of cache directory to handle speculative state of the cache line. The speculation state is managed on per-line basis. A write-through policy is employed and a speculation buffer is attached to each cache to buffer speculatively stored value. STAMPede [9] extends the traditional MESI protocol with additional states to support memory speculation. Similar to Hydra, the management of speculative state is performed on per-line basis. However, it differs in that it does not require special buffer to hold speculative memory values. In [5], memory speculation is performed using a centralized table called Memory Disambiguation Table (MDT). MDT is located between the private L1 caches and the shared L2 cache. It records loads and stores executed on L1 caches. The state of memory is managed on per-word basis. Since the entry of MDT is limited, memory operation of speculative threads need to be stalled if the table is full.

Although performance evaluations of the mechanisms mentioned above were also presented along with the proposals, to our extent of knowledge, there is no published work that analyzes the complexity involved in term of area and delay overhead. This paper is a first attempt to quantitatively measure the complexity. For coherence protocol, we extend the MSI protocol to support memory speculation. Furthermore, we choose to manage the memory state on a per-word basis, since it has been shown that maintaining the state on a per-line basis results in poor performance [5].

3 Cache Controller Support for Memory Speculation

3.1 Organization

We integrated a hardware mechanism into each PU's cache controller to support thread-level memory speculation. The organization of the controller is shown in Figure 1. The controller mainly consists of four units: *cache directory* that holds the state of each word in the cache, *state controller* that manages the state transition of the word, *violation detector* that detects memory dependence violations, and *data forwarder* that controls data forwarding between PUs.

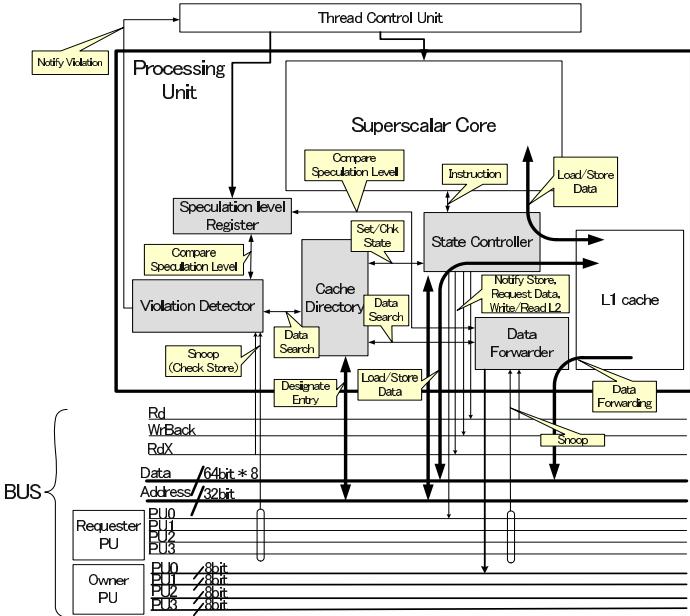


Fig. 1. Cache controller organization

The controller manages the state by snooping memory events broadcasted on a shared memory bus. This bus includes a pair of data bus and address bus, and a number of control lines necessary to maintain the consistency of memory. *Rd*, *WrBack*, and *RdX* lines perform similar function as in the original MSI protocol. When asserted they indicate a bus read operation, write back operation, and read exclusive (initiated by a write) respectively. The *Requester PU* and *Owner PU* lines are used to support thread-level memory speculation described shortly below.

3.2 Thread-Level Memory Speculation Support

To support thread-level memory speculation, the coherence protocol must provide mechanisms to (1) buffer data speculatively stored by a thread, (2) forward the correct version of data to a consumer thread, and (3) detect memory dependency violations.

The buffering support is implemented by only allowing any data speculatively stored by a thread to be committed into L2 cache after the thread becomes a non-speculative thread.

To support the data forwarding, we need to identify the *speculation level* of currently running threads. When a thread is less speculative than another thread, then we say that the first thread has a lower speculation level than the second thread. For a given consumer thread, we need to identify set of data

created by threads with lower speculation levels than the consumer, and forward the data created by the highest speculative thread in the set. The *Requester PU* and *Owner PU* lines are used for this purpose. When a PU starts a read transaction request by asserting the *Rd* line and its corresponding *Requester PU* line, other caches see the request, and assert the *Owner PU* line when they have the corresponding word. These signals are then used to arbitrate which one of the owners is responsible for forwarding the requested word.

Memory dependence violation detection mechanism is provided as follows. All words referred by speculative loads are recorded in the cache directory. When a store is executed, its destination address is broadcasted on the bus. Receiving this address, each PU that executes a thread with higher speculation level checks its corresponding cache entry. Violation is detected if the PU has previously loaded a value from the same location. In this case, the cache controller notifies thread control unit to squash and restart the execution of the thread.

3.3 State Transition

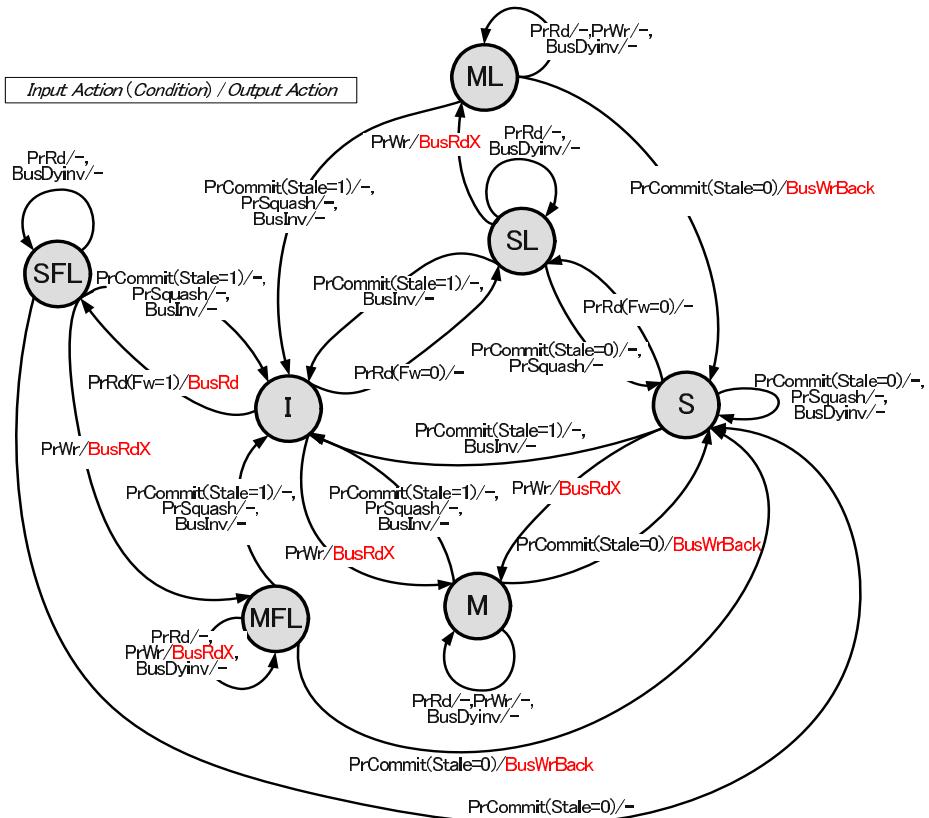
The cache directory contains an entry of eight control bits for each word: *Modified*, *Invalid*, *Forwarded*, *Load*, *Stale*, *Store 0*, *Store 1*, and *Store 2* bits. The first four bits are used to encode the state of the corresponding word. A word can be in one of the following seven states: *Modified (M)*, *Shared (S)*, *Modified-Loaded (ML)*, *Shared-Loaded (SL)*, *Modified-Forwarded-Loaded (MFL)*, *Shared-Forwarded-Loaded (SFL)* and *Invalid (I)*. The encoding of these states is shown in Table 1 and the state transitions possible are illustrated in Figure 2. The list of events as an input to the state diagram is listed in Table 2.

Forwarded bit is set when the word was forwarded from a predecessor thread. When any of the predecessor threads is squashed, memory words with the Forwarded bit set are invalidated. It is because the forwarded data may be incorrect due to misspeculation.

Load bit is set when the first access to the memory word is a load, and is used to detect dependency violations. Store bits identify which of the PUs have stored to the corresponding address. Each Store bit corresponds to one of the three other PUs. Store bits are used to avoid unnecessary violation detections. When a store is executed by another PU, its destination address is broadcasted

Table 1. List of states of a word in the cache

State	Forwarded	Load	Modified	Invalid
M	0	0	1	0
S	0	0	0	0
ML	0	1	1	0
SL	0	1	0	0
MFL	1	1	1	0
SFL	1	1	0	0
I	X	X	X	1

**Fig. 2.** State transition diagram

on the bus. If the store comes from a predecessor thread, cache controller checks both the Load bit and the Store bits. Violation is detected if the Load bit is set, and if none of the Store bits corresponding to predecessor threads between the current thread and the storing thread, is set.

Stale bit of a cache word in a PU is set at arrival of *Delayed Invalidation (Dynv)* message. This message is generated by the bus interface when a successor thread performs a memory store to the location of the cache word. It indicates that the corresponding value of the cache word can no longer be used by any future thread that will be assigned to this PU. When the current thread of the PU commits, all words with Stale bit set are invalidated.

4 Methodology

The complexity of our cache model is analyzed in terms of delay overhead incurred by additional memory state bits and control logic. To quantify delay over-

Table 2. List of cache coherence events

Input	Description
PrRd (Fw = 0)	The PU issues a memory load and the data is not forwarded by another thread
PrRd (Fw = 1)	The PU issues a memory load and the data is forwarded by another thread
PrWr	The PU issues a memory store
PrSquash	The thread is squashed
PrCommit (Stale=0)	The thread is committed and the Stale bit is set
PrCommit (Stale=1)	The thread is committed and the Stale bit is not set
BusInv	A predecessor thread issues a memory store on the bus
BusDyinv	A successor thread issues a memory store on the bus

head of the state bits, we estimate cache access time using the CACTI tool [7]. Assuming that the state bits are kept in a separate cache directory, we compare the access time for the directory and for the cache itself, and discuss possible impact on cache coherence operations. Delay of the control logic is estimated using the method of logical effort [10]. Using this method, the delay incurred by a logic gate is calculated as the sum of parasitic delay p and effort delay f . The effort delay is further expressed as the product of logical effort g , which describes how much bigger than an inverter a gate must be to drive loads as well as the inverter can, and electrical effort h , which is the ratio of output to input capacitance of the gate.

$$d = f + p = gh + p \quad (1)$$

Delay along N -stage logic path D is given by the sum of delay through each stage:

$$D = \sum_{i=1}^N f_i + \sum_{i=1}^N p_i \quad (2)$$

It is known that D is minimum when effort delay through each stage is equal to an optimal effort delay \hat{f} :

$$\hat{D} = N\hat{f} + \sum_{i=1}^N p_i \quad (3)$$

where \hat{f} is given by

$$\hat{f} = F^{1/N} = \left(\prod_{i=1}^N g_i \prod_{i=1}^N b_i \prod_{i=1}^N h_i \right)^{1/N} = (GBH)^{1/N} \quad (4)$$

Here, b_i is branching effort of stage i , which estimates the fanout effect of the logic gate in the stage. G , B , and H are the products of g_i , b_i , and h_i respectively.

To estimate delay overhead of control logic, we model the critical path of the logic and calculate the optimal delay \hat{D} along the path. We also optimize the path

Table 3. L1 cache parameters

Size	32 kB
Line Size	64 bytes
Associativity	2
Tag	18 bits

Table 4. Cache access time

	Delay [FO4]	Delay @90nm [ps]
Data	17.3	623
Tag	19.1	688
Directory	11.4	410

by adjusting the number of logic stages so that the stage effort becomes optimal. Interested readers should refer to [10] for the details of this optimization.

We assume static CMOS logic style for the circuits. The delay through a fanout-of-four (FO4) inverter is used as the delay metric. Using FO4 metric is preferable since the metric holds constant over a wide range of process technologies. However, to provide a concrete example, absolute delays at 90 nm technology will also be shown, approximating that $1 \text{ [FO4]} = 36 \text{ ps}$.

5 Complexity Analysis

5.1 Cache Directory Access Time

This section discusses the overhead incurred by the cache directory that holds the cache state bits. Our coherence protocol requires eight state bits per word. For a 32-kB cache described in Table 3, the state bits amount to 4 kB. The directory size is much larger than the one for conventional coherence protocol, because additional state bits are required to maintain the speculative states and the states are kept on a per word basis. For example, the simplest MSI protocol only needs three state bits per line, so the directory would be only 192 bytes in size.

We estimated access latency of the cache using CACTI, assuming that the cache tag, data, and the directory are each kept in a separate memory structure. Table 4 shows the result. Access time for the directory is shorter than that for the data array because the directory is smaller in size. Tag access time is estimated to be longer than data access time because of the delay in the comparator. Therefore, the critical path in this model is in tag access operation. Since the directory access can be performed in parallel to the tag access, it is expected that the directory would not affect the cache access latency. Note that this cache configuration assumes a 32-bit address space. In a larger address space, the size of the tag array would be larger, so the directory is less likely to come on the critical path.

5.2 Control Logic Delay

We estimate complexity of control logic required for four main operations of the cache controller described below:

- **Forward – Owner Probing:** On receiving a forward request, each PU checks if it has the requested line in the cache. Then it checks modified bit

for each word in the line, and if the bit is set, asserts *owner PU line* to claim its ownership.

- **Forward – Data Transfer:** Each PU examines which PUs have claimed ownership for the requested words. It compares speculation level of all the owner PUs, and checks if it is qualified to forward the word. If it is, it puts the word on the bus, and if not, send a request to L2 cache.
- **Violation Detection :** On receiving a read exclusive (store) notification broadcast, each PU first identifies which of the PUs have previously stored to the corresponding word by checking the store bits. Violation detection is performed by comparing the speculation levels of the storing PUs and by checking the load bit of the corresponding word.
- **State Transition:** The state bits in the cache directory are managed according to the state transition described in Figure 2 and Table 2.

Figure 3~6 shows the critical paths of each logic block along with their estimated delay. For example, Figure 3 shows the critical path of the Owner Probing circuit. The total delay is calculated as follows. First, we determined the logical effort g and the parasitic delay p of each stage in the circuit. Then, we calculated the path logical effort G and the path electrical effort H . Here, we assume the load capacitance of the bus is 100 times larger than the input capacitance of the first stage logic, so that $H = 100$. Since there is no branch on this path, the path branching effort B is 1. Optimal stage effort ρ is precalculated from the process parameters and is set to 3.17 in our case. By using these results, we can estimate the best number of stages $\hat{N} = 4.49$. We use the nearest integer as the actual number of stages $N = 4$. Then, by applying Equation 3 and 4, we can estimate the optimal logic delay of this circuit $\hat{D} = 3.91[FO4]$. Combining the logic delay and the delay of cache directory access, we found that the total delay is approximately 15.3[FO4].

Table 5 summarizes the total delay incurred by each operation. For operations that require access to the cache directory, the total delays are given by the sum of the logic delay and the directory access time, estimated as 11.4 [FO4] in Table 4. It can be seen that the directory overhead occupies a large part of the total delay. The largest delay of 20.8 [FO4] is required for state transition. However, if the tag access time is longer than the directory access time, this overhead can be partially hidden by overlapping the control operations with the tag access. In this

Table 5. Control logic delay and total time required for each coherence protocol operation

	Logic Delay [FO4]	Directory Delay [FO4]	Total Delay [FO4]	Total Delay @90nm [ps]
Owner Probing	3.91	11.4	15.3	550
Data Transfer	7.51	0	7.51	270
Violation Detection	8.98	11.4	20.4	734
State Transition	9.35	11.4	20.8	749

case, the control logic extends the cache access time only by 9%, from 19.1 [FO4] (tag access) to 20.8 [FO4] (state transition). It should be noted, however, that the control logic must be duplicated for each way in the cache.

To summarize, it is estimated that the cache controller can operate at a cycle time of approximately 21 [FO4]. It should be noted that the cycle time shown here is pessimistic since we assume a static CMOS circuit throughout the analysis. A shorter cycle time is possible by applying a faster circuit style such as domino logic. Using the domino circuit style typically enables a logic to be implemented 1.5 to 2 times faster [4]. In this case, the delay of the state transition operation can be made lower than the cache access delay. The directory access time may also be shorter in the real design than the time estimated by CACTI, since CACTI assumes only typical optimization techniques. Finally, the cycle time can also be shortened by pipelining the protocol operations into several stages, at the cost of increased hardware complexity.

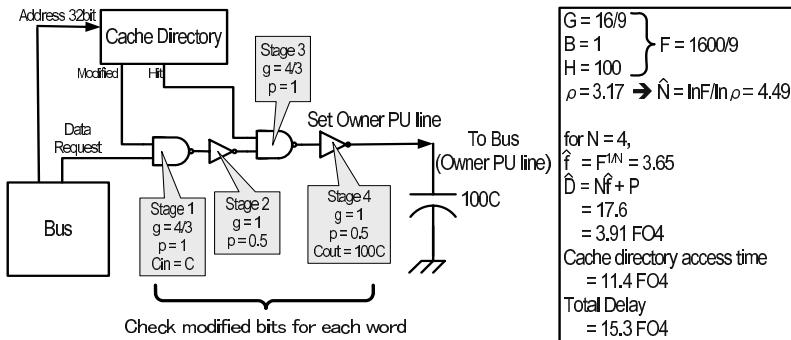


Fig. 3. Forward - Owner Probing

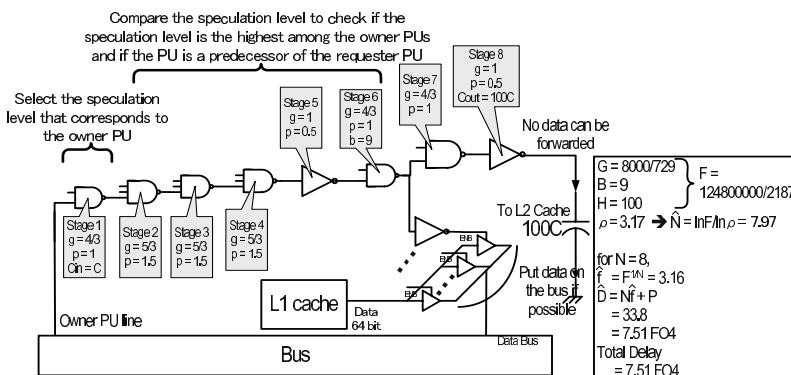


Fig. 4. Forward - Data Transfer

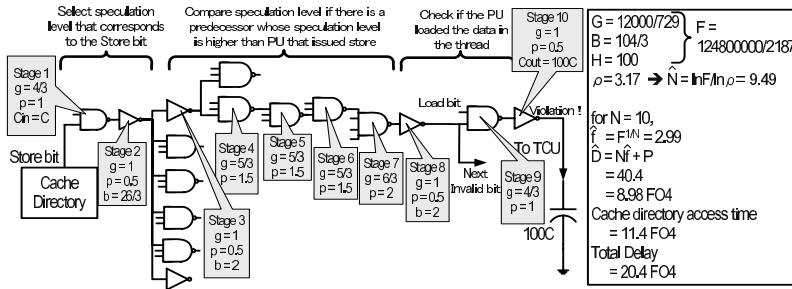


Fig. 5. Violation Detection

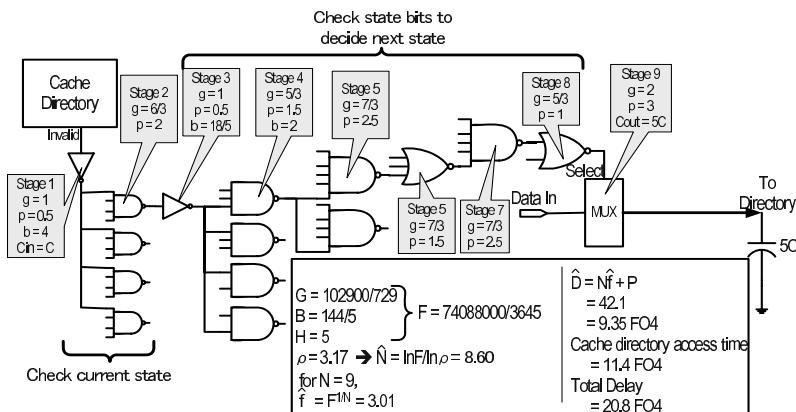


Fig. 6. State Transition

6 Conclusion

In this research, we performed a complexity analysis of a cache controller designed by extending an MSI controller to support thread-level memory speculation. We modeled and estimated the delay of the cache control logic on critical paths and the area overhead to hold additional control bits in the cache directory. We found that the increase in area of cache directory over the original MSI cache directory is significant. This is a consequence of the requirements to maintain speculative state of memory on a per-word basis rather than per-line basis. For many protocol operations, the directory access time occupies more than half of the total delay. This delay overhead is however smaller than the delay for accessing and comparing cache tags. When the protocol operations are performed in parallel with tag comparison, the critical path latency is increased only by 9%. Overall, for a 32-kB cache with 64-byte lines, our results showed that the

cache controller can be implemented with a cycle time of less than 21 [FO4]. The cycle time can be made shorter by applying a faster circuit style or by pipelining the protocol operations into more stages. Quantitative evaluations of these optimizations are the future work of this research.

Acknowledgements. This research is partially supported by Grant-in-Aid for Fundamental Scientific Research B(2) #13480077 from Ministry of Education, Culture, Sports, Science and Technology Japan, Semiconductor Technology Academic Research Center (STARC) Japan, CREST project of Japan Science and Technology Corporation, and by 21st century COE project of Japan Society for the Promotion of Science. We are also thankful for anonymous reviewers for their constructive critics and suggestions.

References

1. M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–557, 1996.
2. S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. In *Proc. of the 4th HPCA*, pages 195–205, 1998.
3. L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *Proc. of the 8th ASPLOS*, pages 58–69, 1998.
4. D. Harris and M. A. Horowitz. Skew-Tolerant Domino Circuits. *IEEE Journal of Solid State Circuits*, 31(11):1687–1696, 1997.
5. V. Krishnam and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.
6. P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proc. of the 12th ICS*, pages 77–84, 1998.
7. P. Shivakumar and N. P. Jouppi. CACTI 3.0: An Integrated Cache Timing, Power, and Area Model. Technical Report WRL-2001-2 HP Labs Technical Reports, 2001.
8. G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proc. of the 22nd ISCA*, pages 414–425, 1995.
9. J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. of the 27th ISCA*, pages 1–12, 2000.
10. I. E. Sutherland, R. F. Sproull, and D. Harris. *Logical Effort: Designing Fast Cmos Circuits*. Morgan Kaufmann, 1999.
11. J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

One Chip, One Server: How Do We Exploit Its Power?

Per Stenstrom

Chalmers University of Technology, Sweden

Abstract. Application and technology trends have made integration of a multiprocessor on a single chip a reality. While its performance potential is truly amazing, technology as well as software trends introduce important system challenges. These range from how to transparently exploit thread-level parallelism to how to cope with the increasing processor/memory speedgap and power consumption. Apart from summarizing what has been done towards these challenges, I will point out what I believe are interesting avenues to pursue in the future.

Data Locality Optimization for Synthesis of Efficient Out-of-Core Algorithms

Sandhya Krishnan¹, Sriram Krishnamoorthy¹, Gerald Baumgartner¹, Daniel Cociorva¹, Chi-Chung Lam¹, P. Sadayappan¹, J. Ramanujam², David E. Bernholdt³, and Venkatesh Choppella³

¹ Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210, USA.

{krishnas,krishnsr,gb,cociorva,clam,saday}@cis.ohio-state.edu

² Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803, USA.
jxr@ece.lsu.edu

³ Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA.
{bernholdtde,choppellav}@ornl.gov

Abstract. This paper describes an approach to synthesis of efficient out-of-core code for a class of imperfectly nested loops that represent tensor contraction computations. Tensor contraction expressions arise in many accurate computational models of electronic structure. The developed approach combines loop fusion with loop tiling and uses a performance-model driven approach to loop tiling for the generation of out-of-core code. Experimental measurements are provided that show a good match with model-based predictions and demonstrate the effectiveness of the proposed algorithm.

1 Introduction

Many scientific and engineering applications need to operate on data sets that are too large to fit in the physical memory of the machine. Some applications process data by *streaming*: each input data item is only brought into memory once, processed, and then over-written by other data. Other applications, like Fast Fourier Transform calculations and those modeling electronic structure using Tensor Contractions, like coupled cluster and configuration interaction methods in quantum chemistry, employ algorithms that require more elaborate interactions between data elements; data cannot be simply streamed into processor memory from disk. In such contexts, it is necessary to develop so called *out-of-core* algorithms that explicitly orchestrate the movement of subsets of the data between main memory and secondary disk storage. These algorithms ensure that data is operated in chunks small enough to fit within the system's physical memory but large enough to minimize the cost of moving data between disk and main memory.

This paper presents an approach to automatically synthesize efficient out-of-core algorithms in the context of the Tensor Contraction Engine (TCE) program synthesis tool [1,3,2,4]. The TCE targets a class of electronic structure calculations which involve many computationally intensive components expressed as tensor contractions. Although the current implementation addresses tensor contractions arising in quantum chemistry,

the approach developed here is more general. It can be used to automatically generate efficient out-of-core code for a broad range of computations expressible as imperfectly nested loop structures and operating on arrays potentially larger than the physical memory size.

The paper is organized as follows. In the next section, we elaborate on the out-of-core code synthesis problem in the computational context of interest. Sec. 3 presents an empirically derived model of disk I/O costs, that drives the out-of-core optimization algorithm presented in Sec. 4. Sec. 5 presents experimental performance data on the application of the new algorithm, and conclusions are provided in Sec. 6.

2 The Computational Context

In this paper we address the data locality optimization problem in the context of disk-to-memory traffic. Our proposed optimizations are part of the Tensor Contraction Engine, which takes as input a high-level specification of a computation expressed as a set of tensor contraction expressions, and transforms it into efficient parallel code. The TCE incorporates several compile-time optimizations, including algebraic transformations to minimize operation counts [9,10], loop fusion to reduce memory requirements [6,8,7], space-time trade-off optimization [2], communication minimization [3] and data locality optimization [5,4] of memory-to-cache traffic. Although there are many similarities between the two varieties of data locality optimizations, our previous approach is not effective for the disk-to-memory context due to the constraint that disk I/O must be in large contiguous blocks in order to be efficient.

In the class of computations considered, the final result to be computed can be expressed using a collection of multi-dimensional summations of the product of several input arrays.

As an example, we consider a transformation often used in quantum chemistry codes to transform a set of two-electron integrals from an atomic orbital (AO) basis to a molecular orbital (MO) basis:

$$B(a, b, c, d) = \sum_{p, q, r, s} C1(d, s) \times C2(c, r) \times C3(b, q) \times C4(a, p) \times A(p, q, r, s)$$

Here, $A(p, q, r, s)$ is an input four-dimensional array (assumed to be initially stored on disk), and $B(a, b, c, d)$ is the output transformed array, which needs to be placed on disk at the end of the calculation. The arrays $C1$ through $C4$ are called transformation matrices. In reality, these four arrays are identical; we identify them by different names in our example in order to be able to distinguish them in the text.

The indices p, q, r , and s have the same range N , denoting the total number of orbitals, and equal to $O + V$, where O is the number of occupied orbitals in the chemistry problem, V is the number of unoccupied (virtual) orbitals. Likewise, the index ranges for a, b, c , and d are the same, and equal to V . Typical values for O range from 10 to 300; the number of virtual orbitals V is usually between 50 and 1000.

The calculation of B is done in four steps to reduce the number of floating point operations from the order of $V^4 N^4$ in the initial formula (8 nested loops, for p, q, r, s, a, b, c , and d) to the order of $V N^4$:

$$B(a, b, c, d) = \sum_s C1(d, s) \times \\ \left(\sum_r C2(c, r) \times \left(\sum_q C3(b, q) \times \left(\sum_p C4(a, p) \times A(p, q, r, s) \right) \right) \right)$$

This operation-minimal approach results in the creation of three temporary intermediate arrays $T1$, $T2$, and $T3$: $T1(a, q, r, s) = \sum_p C4(a, p)A(p, q, r, s)$, $T2(a, b, r, s) = \sum_q C3(b, q)T1(a, q, r, s)$, and $T3(a, b, c, s) = \sum_r C2(c, r)T2(a, b, r, s)$. Assuming that the available memory limit on the machine running this calculation is less than V^4 (which is 3TB for $V = 800$), any of the logical arrays A , $T1$, $T2$, $T3$, and B is too large to entirely fit in memory. Therefore, if the computation is implemented as a succession of four independent steps, the intermediates $T1$, $T2$, and $T3$ have to be written to disk once they are produced, and read from disk before they are used in the next step. Furthermore, the amount of disk access volume could be much larger than the total volume of the data on disk containing A , $T1$, $T2$, $T3$, and B . Since none of these array can be fully stored in memory, it may not be possible to perform all multiplication operations by reading each element of the input arrays from disk only once.

We use loop fusion and loop tiling to reduce memory requirements. To illustrate the benefit of loop fusion, consider the first two steps in the AO-to-MO transformation: $T1(a, q, r, s) = \sum_p C4(a, p)A(p, q, r, s)$; $T2(a, b, r, s) = \sum_q C3(b, q)T1(a, q, r, s)$. Fig. 1(a) shows the loop structure for the direct computation as a two-step sequence: first produce the intermediate $T1(1 : Na, 1 : Nq, 1 : Nr, 1 : Ns)$ and then use $T1$ to produce $T2(1 : Na, 1 : Nb, 1 : Nr, 1 : Ns)$. We refer to this as an *abstract* form of a specification of the computation, because it cannot be executed in this form if the sizes of arrays are larger than limits due to the physical memory size. We later address the transformation of abstract forms into concrete forms that can be executed — the concrete forms have explicit disk I/O statements between disk-resident arrays and their in-memory counterparts.

Since all loops in either of the loop nests are fully permutable, and since there are no fusion-preventing dependences, the common loops a , q , r , and s can be fused. Once fused, the storage requirements for $T1$ can be reduced by contracting it to a scalar as shown in Fig. 1(b). Although the total number of arithmetic operations remains unchanged, the dramatic reduction in size of the intermediate array $T1$ implies that it can be completely stored in memory, without the need for any disk I/O for it. In contrast, if $Na \times Nq \times Nr \times Ns$ is larger than available memory, the unfused version will require that $T1$ be written out to disk after it is produced in the first loop, and then read in from disk for the second loop.

The code synthesis process in the Tensor Contraction Engine [1] involves multiple steps, including algebraic transformation, loop fusion and loop tiling. The loop fusion and loop tiling steps are coupled together. The fusion step provides the tiling step with a set of candidate fused loop structures with desirable properties; the tiling step seeks to find the best tile sizes for each of the fused loop structures supplied, and chooses the one that permits the lowest data movement cost overall. Our previous work had focused on tiling to minimize memory-to-cache traffic. In this paper, we describe an approach to minimize disk I/O time for situations where out-of-core algorithms are needed.

```

double T1(Na,Nq,Nr,Ns)
double T2(Na,Nb,Nr,Ns)
T1(*,*,*,*) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
  FOR q = 1, Nq
    FOR r = 1, Nr
      FOR s = 1, Ns
        FOR p = 1, Np
          T1(a,q,r,s)
          += C4(a,p) * A(p,q,r,s)
        END FOR p,s,r,q,a
      FOR a = 1, Na
        FOR b = 1, Nb
          FOR r = 1, Nr
            FOR s = 1, Ns
              FOR q = 1, Nq
                T2(a,b,r,s)
                += C3(b,q) * T1(a,q,r,s)
              END FOR q,s,r,b,a
            END FOR r,s,q,b
          END FOR b,r,s
        END FOR a,b,r,s
      END FOR a,b,r,s
    END FOR r,s,q,a
  END FOR q,r,s,a
END FOR a,r,s,q,a

```

(a) Unfused form

```

double T1(1,1,1,1)
double T2(Na,Nb,Nr,Ns)
T1(1,1,1,1) = 0
T2(*,*,*,*) = 0
FOR a = 1, Na
  FOR q = 1, Nq
    FOR r = 1, Nr
      FOR s = 1, Ns
        FOR p = 1, Np
          T1(1,1,1,1)
          += C4(a,p) * A(p,q,r,s)
        END FOR p
      FOR b = 1, Nb
        T2(a,b,r,s)
        += C3(b,q) * T1(1,1,1,1)
      END FOR b
    END FOR s,r,q,a
  END FOR r,s,q,a
END FOR a,s,r,q,a

```

(b) Fused code

Fig. 1. Example of the use of loop fusion to reduce memory

We have previously addressed the issue of the data locality optimization problem arising in this synthesis context, focusing primarily on minimizing memory-to-cache data movement [5,4]. In [5], we developed an integrated approach to fusion and tiling transformations for the class of loops arising in the context of our program synthesis system. However, that algorithm was only applicable when the sum-of-products expression satisfied certain constraints on the relationship between the array indices in the expression. The algorithm developed in [4] removed the restrictions assumed in [5]. Its cost model was based on an idealized fully associative cache with a line size of one. A tile size search procedure estimated the total capacity miss cost for a large number of combinations of tile sizes for the various loops of an imperfectly nested loop set. After the best combination of tile sizes was found, tile sizes were adjusted to address spatial locality considerations. This was done by adjusting the tile sizes for any loop indexing the fastest varying dimension of any array to have a minimum value of the cache linesize. The approach cannot be effectively used for the out-of-core context because the “line-size” that must be used in the adjustment procedure would be huge — corresponding to a minimum disk read chunk-size to ensure good I/O bandwidth.

3 Modeling Performance of Disk I/O

The tile sizes for each array on disk are chosen so as to minimize the cost of I/O for that array within the memory constraint. The I/O cost is modeled based on empirically derived I/O characteristics of the underlying system. In this section, the I/O characteristics of the system are discussed and the I/O cost model is derived.

We evaluated the I/O characteristics of a Pentium II system that was available for exclusive use, without any disk interference from other users. The details of the system are shown in Table 1. Reads and writes were done for different block sizes at different

Table 1. Configuration of the system whose I/O characteristics were studied.

Processor	OS	Compiler	Memory	Hard disk
Pentium II 300 MHz	Linux 2.4.18-3	gcc version 2.96	128MB	Maxtor 6L080J4

strides and the per-byte transfer time was measured. The block size was varied from 16KB to 2MB. The strides was set to be multiples of the block size used. The read and write characteristics are shown in Fig. 2(a) and Fig. 2(b).

The graphs show that reads and writes exhibit different characteristics. This is due to the difference in their semantics. The cost of reads, which are synchronous, includes the disk access time (seek time + latency). On the other hand, writes return after copying the data to a system buffer. The I/O subsystem subsequently initiates writes of the data from the system buffer to disk. The I/O subsystem reorders the writes, reducing the average disk access time. The reordering of write requests make the cost of writing data less dependent on the actual layout of data on disk. For reads, the data transfer for a requested block can happen before the actual request arrives, due to read ahead, also called prefetching. Most file systems read ahead to exploit access locality. Read-ahead leads to lower read costs than write costs.

The read characteristics show a clear difference between sequential access and strided access even at large block sizes. This effect can be attributed to the presence of disk access time in the critical path of reads. Also, since the block sizes requested are large, the disk access time approaches the average disk access time. This additional cost, incurred for every read request, approximately doubles with halving the block size.

Below a certain block size, the reads do not completely take advantage of read ahead. For example, with a stride of two, small block sizes lead to only alternate blocks being used, though contiguous blocks are read into memory by the I/O subsystem. This has the effect of increasing the total data read. Hence, for smaller block sizes, the cost per byte increases proportionally with stride.

We model the per-byte cost of I/O as a linear function of block size and disk access time. Based on the observed trends in disk read time as a function of blocksize and stride, we develop a model below, for the time to access an arbitrary multi-dimensional “brick” of data from a multi-dimensional array with a linearized column-major layout on disk. Access of a brick of data will require a number of disk reads, where each read can only access a contiguous set of elements on disk.

Let T_1, \dots, T_4 be the tile sizes for the four dimensions N_1, \dots, N_4 , respectively. The block size BS and stride S of access are determined as follows:

$$BS = \begin{cases} T_1 & \text{if } T_1 < N_1 \\ T_1 * T_2 & \text{if } T_1 = N_1 \text{ and } T_2 < N_2 \\ T_1 * T_2 * T_3 & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 < N_3 \\ T_1 * T_2 * T_3 * T_4 & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 = N_3 \end{cases}$$

$$S = \begin{cases} N_1/BS & \text{if } T_1 < N_1 \\ (N_1 * N_2)/BS & \text{if } T_1 = N_1 \text{ and } T_2 < N_2 \\ (N_1 * N_2 * N_3)/BS & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 < N_3 \\ (N_1 * N_2 * N_3 * N_4)/BS & \text{if } T_1 = N_1 \text{ and } T_2 = N_2 \text{ and } T_3 = N_3 \end{cases}$$

The per-byte cost of reads is formulated as

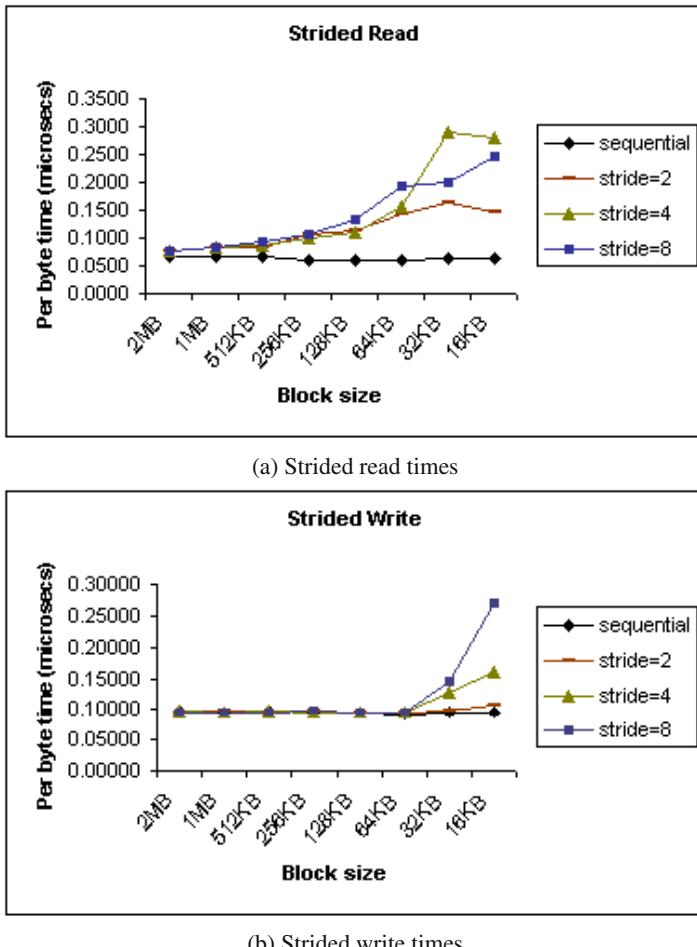


Fig. 2. Strided read/write times on the Pentium II system.

$$Cost = \begin{cases} \left(seq + \frac{\text{avg. access time}}{BS} \right) & \text{if } BS \geq \text{prefetch size} \\ \left(seq * S + \frac{\text{access time}}{64KB} \right) & \text{if } BS < \text{prefetch size} \end{cases}$$

where prefetch size is the extent of read ahead done by the I/O subsystem. The average access time is the sum of average seek time and average latency, as provided in the specification of the disk and seq is the time per-byte for sequential reads. For the platform under consideration, the read ahead size was determined to be 64KB. The average access time was found to be 13 milliseconds and the per-byte sequential access time was determined as 65 nanoseconds.

Writes at different strides have the same cost as sequential writes for large block sizes. This is due to the fact that writes are not synchronous and no additional cost in the form of seek time has to be incurred. Below a certain block size, the lack of sufficient

‘locality’ increases the cost from the minimum possible. In contrast to reads, for small block sizes, the per-byte cost of write steadily increases with stride. This is due to the reordering of writes by the I/O subsystem, which diminishes the influence of actual of data layout on disk.

The graph shows a logarithmic relationship between the per-byte write time and the block size for small block sizes. The per-byte cost for small block sizes is formulated as a set of linear equations, one for each stride. Each of the equations is a function of the logarithm of the block size. The per-byte sequential write time on the system under consideration was determined to be 95 nanoseconds.

4 Out-of-Core Data Locality Optimization Algorithm

4.1 Disk File Layout

Given an imperfectly nested loop structure that specifies a set of tensor contractions, the arrays involved in the contractions fall into one of three categories: *a*) input arrays, which initially reside on disk, *b*) intermediate temporary arrays, which are produced and consumed within the specified computation and are not required at the end, and *c*) output arrays, which must be finally written out to disk.

If an intermediate array is too large to fit into main memory, it must be written out to disk and read back from disk later. Since it is a temporary entity, there is complete freedom in the choice of its disk layout. It is not necessary for it to be represented in any canonical linearized order such as row major order. A blocked representation on disk is often advantageous in conjunction with tiling of the loop computations. For example, consider the multiplication of two large $N \times N$ disk-resident arrays on a system with a memory size of M words, where $M < N^2$. Representing the arrays on disk as a sequence of $k \times k$ blocks, with $k^2 < M/3$, allows efficient read/write of blocks of the arrays to and from disk. We therefore allow the out-of-core synthesis algorithm to choose the disk representation of all intermediate arrays in a manner that minimizes disk I/O time. For the input and output arrays, the algorithm can be used in either of these modes:

- The layouts of the input and output arrays are *unconstrained*, and can be chosen by the algorithm to optimize I/O time, or
- The input and output arrays are *constrained* to be stored on disk in some pre-specified canonical representation, such as row-major order.

Even for the arrays that are not externally constrained to be in some canonical disk layout, some constraints are imposed on their blocked layout on disk and the tile sizes of loop indices that index them:

- All loop indices that index the same array dimension in multiple reference occurrences of an array must be tiled the same. This ensures that multiple reference instances of an array can all be accessed efficiently using the same blocked units on disk. Although the blocks of an array may be accessed in a different order for the different instances due to a different nesting of the loop indices, it is ensured that the basic unit of access from disk is always the same.

```

double MA(1:Ti, 1:Tk)
double MB(1:Ti, 1:Tj)
double MC(1:Tj, 1:Tk)

double A(1:Ni,1:Nk)
double B(1:Ni,1:Nj)
double C(1:Nj,1:Nk)

FOR i
  FOR j
    FOR k
      A(i,k) += B(i,j) * C(j,k)
END FOR k, j, i

(a) Abstract code for matrix multiplication.
Ni=Nk=6000, Nj=2000.
Memory limit=128MB.

double MA(1:Ti,1:Tk)
double MB(1:Ti,1:Tj)
double MC(1:Tj,1:Tk)

FOR iT = 1, Ni, Ti
  FOR jT = 1, Nj, Tj
    FOR kT = 1, Nk, Tk
      MA(1:Ti,1:Tk) = Read disk array A(i,k)
      MB(1:Ti,1:Tj) = Read disk array B(i,j)
      MC(1:Tj,1:Tk) = Read disk array C(j,k)
      FOR iI = 1, Ti
        FOR jI = 1, Tj
          FOR kI = 1, Tk
            MA(1:Ti,1:Tk)
              += MB(1:Ti,1:Tj) * MC(1:Tj,1:Tk)
        END FOR kI, jI, iI
        Write MA(1:Ti,1:Tk) to disk A(i,k)
      END FOR kT, jT, iT

(b) Tiled code with all reads and writes immediately surrounding the intra tile loops.

```

Fig. 3. Abstract and concrete code for matrix multiplication.

- Array dimensions of different arrays that are indexed by the same loop index variable must have the same blocking. This ensures that the unit of transfer from disk to memory for all the arrays match the tiling of the loop computations.

Before searching for the optimal tile sizes for the loops, we first need to identify constraints among loop indices that result from the above two conditions. For computing these constraints, we use a Union-Find data structure for grouping indices into equivalence classes.

First, we rename all loop indices to ensure that no two loop indices have the same name. Initially, each loop index is in its own equivalence class. In the symbol table entry for an array name, we keep track of the index equivalence classes of all array dimensions. Then, in a top-down traversals of the abstract syntax tree, for every loop index i and every array reference $A[\dots, i, \dots]$, we merge the equivalence class of i with the equivalence class of the array dimension indexed by i .

The index equivalence classes found by this procedure will be used to constrain the tile size search, such that all the indices in an equivalence class will be constrained to have the same tile size.

4.2 Tile Size Search Algorithm

We now describe our approach to addressing the out-of-core data locality optimization problem:

- **Input:** An *abstract* form of the computation for a collection of tensor contractions, as a set of imperfectly nested loops, operating directly on arrays that may be too large to fit in physical memory, e.g., as in Fig. 3(a).
- **Input:** Available physical memory on target system.
- **Output:** A *concrete* form for the computation, as a collection of tiled loops, with appropriate placements of disk I/O statements to move data between disk-resident arrays to corresponding memory-resident arrays, e.g., as in Fig. 3(b).

```

Index: {
    String name
    int range
    int tilesize
    int tilecount
}
}

CostModel: {
    double memCost(Index[] tiledIndices)
    double diskCost(Index[] tiledIndices)
}

bool MemoryExceeded (Index[] tiledIndices, CostModel C)
    return (C.memCost(tiledIndices) > memoryLimit)

TileSizeSearch (Index[] tiledIndices, CostModel C):
    foreach Index I ∈ tiledIndices
        I.tilecount = 1
    while (MemoryExceeded (tiledIndices, C)) do
        foreach Index I ∈ tiledIndices I.tilecount += 1
        if (foreach Index I ∈ tiledIndices, I.tilecount = 1) then
            return
    Repeat
        foreach Index I ∈ tiledIndices
            I.tilecount --
            Diff[I] = ΔdiskCost ÷ ΔmemCost
            I.tilecount ++
        Index Best..I = I ∈ tiledIndices with max Diff[I]
        Best..I.tilecount --
        if (MemoryExceeded (tiledIndices, C))
            foreach Index J ∈ tiledIndices(J ≠ I)
                while (MemoryExceeded (tiledIndices, C))
                    J.tilecount ++
                    EffDecrease[J] = ΔdiskCost
                    Tiles[J] = J.tilecount
                    Reset J.tilecount to original value
                Index Best..J = J ∈ tiledIndices
                    with max EffDecrease[J]
                Best..J.tilecount = Tiles[Best..J]
            Until (EffDecrease[Best..J] ≤ 0)
}

```

Fig. 4. Procedure **TileSizeSearch** to determine optimal tile sizes that minimize disk access cost.

Given a concrete imperfectly nested loop, with proper placements of disk I/O statements, the goal of the tile size search algorithm is to minimize total disk access time under the constraint that memory limit is not be exceeded. The input to the algorithm is the cost model and the set of index equivalence classes, determined by the procedure explained in Sec. 4.1.

The disk access cost model depends on the mode of the algorithm, as explained in Sec. 4.1. For the unconstrained case, the disk cost for an array is proportional to the volume of data accessed, as each block access is sequential. On the other hand, for the constrained case, we use the I/O performance model described in Sec. 3.

Fig. 4 presents the pseudo code for the tile size search algorithm. It starts by initializing the number of tiles for each tiled index to 1. This is equivalent to all arrays being completely memory resident, and would be the optimal solution if the memory limit is not exceeded. Otherwise, the algorithm tries to find a feasible solution that satisfies the memory constraint, by iteratively incrementing the tile count for all indices. To illustrate the algorithm, consider the concrete code for matrix multiplication in Fig. 5(a). The memory cost equation for this code is:

$$T_i * N_k + T_i * T_j + T_j * N_k$$

As shown in Fig. 5(b), with a tile count of 4 for all indices i , j and k , the memory cost is 97MB, which is within the memory limit of 128MB. The algorithm then tries to reduce the number of tiles for any of the indices as much as possible, so that the solution just fits in memory. This is chosen as the starting point in the search space. For this example, the tile counts of 3, 4, and 1 are determined as a starting point for indices i , j and k , respectively.

From this starting point, the algorithm attempts to reduce the tilecount for each index by 1. It selects the index that provides the maximum improvement in disk cost and suffers the minimum penalty for memory cost. However, reducing the tile count for this index could cause the memory limit to be exceeded. The algorithm tries to repair this situation

	Number of tiles	Block size	Memory Cost
	$i \ j \ k$	$T_i \ T_j \ T_k$	
FOR iT = 1, Ni, Ti	1 1 1	6000 2000 6000	457MB
MA(1:Ti,1:Nk) = Read disk array A(i,k)	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
FOR jT = 1, Nj, Tj	4 4 4	1500 500 1500	97MB
MB(1:Ti,1:Tj) = Read disk array B(i,j)	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
MC(1:Tj,1:Nk) = Read disk array C(j,k)	3 4 4	2000 500 1500	122MB
FOR kT = 1, Nk, Tk	3 4 3	2000 500 2000	122MB
FOR iI = 1, Ti	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
FOR jI = 1, Tj	3 4 1	2000 500 6000	122MB
FOR kI = 1, Tk	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
MA(1:Ti,1:Nk)	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
+= MB(1:Ti,1:Tj) * MC(1:Tj,1:Nk)	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
END FOR kI, jI, iI, kT	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
END FOR jT	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
Write MA(1:Ti,1:Nk) to disk A(i,k)	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮
END FOR iT	⋮ ⋮ ⋮	⋮ ⋮ ⋮	⋮

(b) Tile size search illustration.

(a) Concrete code with I/O stmts moved outside.

Fig. 5. Concrete code and tile size search illustration for matrix multiplication

by increasing the number of tiles for one of the other indices. It selects the index that provides the maximum effective improvement in disk access cost. If it cannot find such an index, the algorithm terminates.

5 Experimental Results

The algorithm from Sec. 4 was used to generate code for the AO-to-MO index transformation calculation described in Sec. 2. Measurements were made on a Pentium II system with the configuration shown in Table 1. The codes were all compiled with the Intel Fortran Compiler for Linux. Although this machine is now very old and much slower than PCs available today, it was convenient to use for our experiments in an uninterrupted mode, with no interference to the I/O subsystem from any other users. Experiments were also carried out on more recent systems; while the overall trends are similar, we found significant variability in measured performance over multiple identical runs, due to disk activity from other users.

The out-of-core code generated by the proposed algorithm was compared with an unfused but tiled baseline version. With this version, it is necessary to write large intermediate arrays to disk when they are produced, and read them back again when they are consumed. The baseline version is representative of the current state of practice in implementing large out-of-core tensor contractions in state-of-the-art quantum chemistry packages.

Table 2 shows the measured I/O time for the unfused baseline version and the measured as well as predicted I/O time for the fused version for the case of unconstrained disk layout. For the baseline version, all arrays were assumed to be blocked on disk so that I/O was performed in large chunks in all steps. The size of the tensors(double precision) for the experiment were: $N_p = N_q = N_r = N_s = 80$ and $N_a = N_b = N_c = N_d = 70$.

For these tensor sizes and an available memory of 128MB, it is possible to choose fusion configurations so that the sizes of any two out of the three intermediate arrays can

Table 2. Predicted and Measured I/O Time: Unconstrained Layout

	Unfused	T3 on disk	
	Measured time (seconds)	Measured time(seconds)	Predicted time(seconds)
Array A	42.4	43.648	42.6
Array t_1	195.67	-	-
Array t_2	66.74	-	-
Array t_3	48.26	74.09	74.32
Array B	29.078	22.346	38.6
Total time	382.15	140.08	155.52

Table 3. Predicted and Measured I/O Time: Column-Major Layout of Input/Output Arrays

	Unfused	T1 on disk	
	Measured time (seconds)	Measured time(seconds)	Predicted time(seconds)
Array A	148.26	149.31	180.3
Array t_1	140.65	94.37	63.72
Array t_2	52.99	-	-
Array t_3	44.89	-	-
Array B	29.55	22.62	38.61
Total time	416.34	266.3	282.63

be reduced to fit completely in memory, but it is impossible to find a fusion configuration that fits all three intermediates within memory. Thus, it is necessary to keep at least one of them on disk, and incur disk I/O cost for that array. Table 2 reports performance data for the fusion configuration that requires $T3$ to be disk-resident.

The I/O time for each array was separately accumulated. It can be seen that the out-of-core code version produced by the new algorithm has significantly lower disk I/O time than the baseline version. The predicted values match quite well with the measured time. The match is better for the overall I/O time than for some individual arrays. This is because disk writes are asynchronous and may be overlapped with succeeding disk reads — hence the measurements of I/O time attributable to individual arrays is subject to error due to such overlap, but the total time should not be affected by the interleaving of writes with succeeding reads.

Table 3 shows performance data for the layout-constrained case. A column-major representation was used for the input array. For the fused/tiled version, we used the fusion configuration that results in $T1$ being disk-resident. Again, the version produced by the new algorithm is better than the baseline version. As can be expected, the disk I/O overheads for both versions are higher than the corresponding cases where the input array layout was unconstrained. The predicted and measured I/O times match to within 10%.

6 Conclusion

We have described an approach to the synthesis of out-of-core algorithms for a class of imperfectly nested loops. The approach was developed for the implementation in a component of a program synthesis system targeted at the quantum chemistry domain. However, the approach has broader applicability and can be used in the automatic syn-

thesis of out-of-core algorithms from abstract specifications in the form of loop computations with abstract arrays. Experimental results were provided that showed a good match between predicted and measured performance. The performance achieved by the synthesized code was considerably better than that representative of codes incorporated into quantum chemistry packages today.

Acknowledgments. We thank the National Science Foundation for its support of this research through the Information Technology Research program (CHE-0121676 and CHE-0121706), NSF grants CCR-0073800 and EIA-9986052, and the U.S. Department of Energy through award DE-AC05-00OR22725. We would also like to thank the Ohio Supercomputer Center (OSC) for the use of their computing facilities.

References

1. G. Baumgartner, D.E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc Supercomputing 2002*, Nov. 2002.
2. D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. *Proc. of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, June 2002, pp. 177–186.
3. D. Cociorva, X. Gao, S. Krishnan, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam. Global Communication Optimization for Tensor Contraction Expressions under Memory Constraints. *Proc. of 17th International Parallel & Distributed Processing Symposium (IPDPS)*, Apr. 2003.
4. D. Cociorva, J. Wilkins, G. Baumgartner, P. Sadayappan, J. Ramanujam, M. Nooijen, D.E. Bernholdt, and R. Harrison. Towards Automatic Synthesis of High-Performance Codes for Electronic Structure Calculations: Data Locality Optimization. *Proc. of the Intl. Conf. on High Performance Computing*, Dec. 2001, Lecture Notes in Computer Science, Vol. 2228, pp. 237–248, Springer-Verlag, 2001.
5. D. Cociorva, J. Wilkins, C.-C. Lam, G. Baumgartner, P. Sadayappan, and J. Ramanujam. Loop optimization for a class of memory-constrained computations. In *Proc. 15th ACM International Conference on Supercomputing*, pp. 500–509, Sorrento, Italy, June 2001.
6. C. Lam. *Performance Optimization of a Class of Loops Implementing Multi-Dimensional Integrals*, Ph.D. Dissertation, The Ohio State University, Columbus, OH, August 1999.
7. C. Lam, D. Cociorva, G. Baumgartner and P. Sadayappan. Optimization of Memory Usage and Communication Requirements for a Class of Loops Implementing Multi-Dimensional Integrals. *Proc. 12th LCPC Workshop* San Diego, CA, Aug. 1999.
8. C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. In *Proc. Intl. Conf. on High Perf. Comp.*, Dec. 1999.
9. C. Lam, P. Sadayappan and R. Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reductions for Parallel Execution. *Par. Proc. Lett.*, (7) 2, pp. 157–168, 1997.
10. C. Lam, P. Sadayappan and R. Wenger. Optimization of a Class of Multi-Dimensional Integrals on Parallel Machines. *Proc. of Eighth SIAM Conf. on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

Performance Evaluation of Working Set Scheme for Location Management in PCS Networks*

Pravin Amrut Pawar¹ and S.L. Mehndiratta²

¹ Dr. Babasaheb Ambedkar Technological University,
Lonere - 402 103, Mangaon, Raigad,
Maharashtra, India.

Pravin_A_Pawar@relianceinfo.com
² Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay, Mumbai – 400 076,
Maharashtra, India.
slm@cse.iitb.ac.in

Abstract. To overcome drawbacks of existing centralized location management scheme, various approaches including message routing schemes based on a hierarchy of databases have been proposed previously by researchers in wireless network community. We have chosen working set idea, which is originally proposed for location management for mobile IP, to analyze its performance for PCS networks. Variants of working set idea have been developed for existing location management standard (HLR/VLR approach) and hierarchical approaches. Experimental analysis has been performed using a location management simulator with respect to bandwidth requirements, database requirements and time requirements for two real life mobility traces. Results obtained indicate that the working set scheme performs superior for real life traces at the cost of slightly higher penalty. Future work involves analyzing working set scheme in details for synthetic mobility models.

1 Introduction

Personal communication services (PCS) allow mobile users with wireless terminals to receive calls irrespective of their location in a seamless manner. In PCS networks [1], mobile host (MH) communicates using portable handsets. Depending on the network topology, there are location databases in the network, which will keep track of location of mobile hosts in their zones.

Location management refers to accessing and maintaining user information for call routing purposes [2]. User mobility causes two main problems in location management with call set-up and routing that are not encountered in a network containing only stationary users. Firstly, call set-up requires at least one database access to find the current location of the user being called (callee). Secondly, if the

* The first author of this paper is also working as a consultant in Application and Solutions Group, Reliance Infocomm, Dhirubhai Ambani Knowledge City, Navi Mumbai – 400 709, India.

network maintains an up-to-date list of the location of each mobile user, then each location change requires one or more database updates. To address the location management problems such as single point failure, congestion problem, tromboning problem [3], in existing PCS standard (e.g. GSM in Europe), various location management schemes including those based on hierarchical arrangement of location registers or location databases have been proposed earlier by the researchers in wireless networks community. Caching location information in HLR/VLR system has been proposed in [14]. For small networks, full replication of location of MH is suggested in [3]. Analysis for caching based scheme for location management is done in [4]. A concept of partitions in the leaf level of hierarchy is introduced in [5]. HiPER family of location management techniques is analyzed practically in [2]. Dynamic hashing + quorum approach, which uses concept of hot and cold MHs is presented in [6] while uniform hashing approach is discussed in [7].

It has been observed that the performance of any location management scheme depends on the following: *Database requirements* are the measure of number of update, deregistration and lookup operations incurred at location database. *Bandwidth requirements* refer to the utilization of network bandwidth by update, deregistration and lookup messages sent for the location management related operations. *Time requirement* is the measure of the call setup and MH lookup time. Minimum time requirements are desirable, which can be measured by the number of hops required for the message to reach destination.

1.1 Introduction and Motivation behind Work Reported

Working set scheme for location management in mobile IP ($WS_{MobileIP}$) has been proposed in [8], theoretical analysis of which is given in [15]. While the potential set of sources for the MH may be large, the set of sources that a given MH communicates most frequently with is very small, which is referred as *working set of hosts* for the MH (similar to the working set concept in operating systems). The adaptive location management scheme for mobile IP [8] enables the MH to dynamically determine its working set, and trade-off routing and update costs in order to reduce total cost. Comparative analysis of the proposed scheme, using simulation, shows its efficiency over a wide range of call-to-mobility ratios. The working set of mobile host has been referred in [6] for reducing lookup costs in PCS networks. Since Mobile IP and PCS networks are two different domains, this scheme has been chosen to assess its practicability for PCS networks. Two variants of working set scheme viz. working set scheme for HLR/VLR approach ($WS_{HLR/VLR}$) and working set scheme for hierarchical approach ($WS_{Hierarchy}$) have been proposed. Experimental analysis has been performed using two mobility trace files, which contain call and movement data of the MHs based on realistic call and movement patterns for the optimal database hierarchies.

$WS_{HLR/VLR}$ is introduced in Section 2. Section 3 describes the hierarchical database architecture used for experimental assessment and for generation of optimal database hierarchy. Section 4 explains $WS_{Hierarchy}$. Section 5 describes the implementation of working set scheme. Section 6 depicts the trace files used for experimental assessment. Section 7 is on the results of the experimental analysis of working set scheme. Section 8 analyzes the results obtained. Section 9 concludes the paper.

2 Working Set Scheme

The issues facing WS_{MobileIP} [8] are, how to determine the current working set of the MH and which sources in the working set must be updated as the MH moves. For this purpose, following quantities for each source s that actively communicates with the mobile host are computed in WS_{MobileIP}:

f_s : The estimated frequency with which connection set-up requests are received from the source s .

δ_s : The additional cost incurred by s , in setting up a connection request to the mobile host MH. This cost is defined as:

$$cost(s, HA) + cost(HA, MH) - cost(s, MH) \quad (1)$$

where HA = home agent in mobile IP protocol.

In short δ_s is the additional overhead incurred when the source does not know the current location of mobile host.

f_{update} : The estimated frequency with which mobile host changes its location and

U_s : The cost incurred in both, announcing the location of mobile host to the source s and also invalidating the location information at source.

Having determined these parameters, WS_{MobileIP} evaluates in an online manner whether the following inequality holds for the source s :

$$f_s * \delta_s > f_{update} * U_s \quad (2)$$

The left and right hand sides of the inequality denote routing cost and the update cost components, respectively. The above parameters are updated and the inequality is evaluated at the mobile host each time the source sets up a connection or when the mobile host moves. More details about WS_{MobileIP} can be obtained in [8].

2.1 Working Set Scheme for HLR/VLR Approach

It is not feasible to add individual MHs in the working set for PCS networks, as mobile users in PCS networks tend to be highly mobile. If the location information of callee is replicated in VLR of caller and before next call if caller moves to another zone then it won't get the benefit of replicating location information at old VLR. Also, we wish to limit the possible number of elements to be considered for working set.

For these reasons, zones of the network have been considered as the elements of working set instead of individual mobile hosts. If the location information of mobile host is to be replicated at zone (source) s or not is decided based on modifications in parameters described in Section 2.1 as:

f_s : The estimated frequency with which connection set-up requests are received from the MHs residing in zone s .

δ_s : The additional cost incurred in setting up a connection request to the mobile host. This cost is defined as

$$cost(s, HLR) + cost(HLR, Z_{Callee}) - cost(s, Z_{Callee}) \quad (3)$$

Where Z_{callee} is the callee's zone and $\text{cost}(s, \text{HLR}) + \text{cost}(\text{HLR}, Z_{\text{callee}})$ represents the cost incurred in setting up connection request in HLR/VLR scheme. HLR indicated home location register of callee. In short δ_s is the additional overhead incurred when location database of caller zone does not have the current location information of callee.

U_s : The cost incurred in both, announcing the location of mobile host to the zone s and also invalidating the location information at location database serving s .

The inequality equation remains same as in [Eq. 2].

For HLR/VLR scheme, call is said to be served locally if both, caller and callee are in the same zone or caller MH is in the zone of HLR of the callee. Additionally, for $\text{WS}_{\text{HLR/VLR}}$ call is said to be served locally if Z_{caller} (caller zone) is in the working set of the callee.

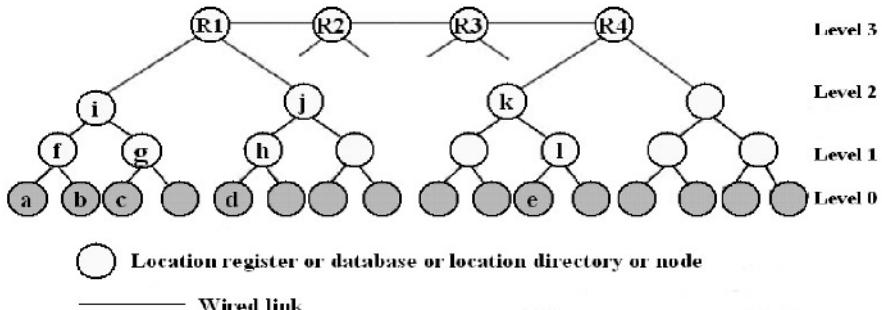


Fig. 1. Conceptual diagram showing hierarchical arrangement of databases

3 Conceptual Structure of Database Hierarchy

Conceptual structure of database hierarchy H is shown in Fig. 1. In H , there is database in each zone of the network, which is shown at level 0 in the hierarchy shown in Fig. 1. Each location database (node of the tree) in the upper level of the hierarchy has at least two children. Bus topology has been considered for connecting root databases. All wired links in the network are bi-directional. In the above hierarchy structure, there is a unique path between every pair of nodes in the tree.

3.1 Update, Deregistration, and Lookup Operations in Hierarchy

Update (Deregistration) operation results in writing (invalidating) location information in the database. In H , each root database contains location information of each MH and points to the root database in the subtree of which, the MH is residing. All databases, except root databases point to the child database in the subtree of which, the MH is residing.

For the intra zone call, a single lookup is sufficient to obtain location information of the callee. If the call is inter zone, and if $LCA(Z_{\text{caller}}, Z_{\text{callee}})$ exists, then lookup message propagates from caller zone to $LCA(Z_{\text{caller}}, Z_{\text{callee}})$ and from $LCA(Z_{\text{caller}}, Z_{\text{callee}})$ to callee

zone. If no $LCA(Z_{\text{caller}}, Z_{\text{callee}})$ exists, then lookup message propagates from caller zone to its root database which points to the root database of callee zone.

3.2 Algorithms for Calculating Optimal Database Hierarchy

A dynamic programming approach for optimization of database hierarchy has been suggested in [9]. A genetic algorithm framework for calculating optimal database hierarchy has been proposed in [10], which has been used here for optimal database hierarchy calculation subject to database and link capacity constraints. The optimal hierarchies obtained for the corresponding trace file have been used for the experimental assessment of the working set scheme.

4 Working Set Scheme for Hierarchical Approach

For database hierarchy structure described in Section 3, there is no concept of HLR and VLR. If a user U is in zone a of hierarchy tree shown in Fig. 1, then only database f contains a pointer to the exact location of user U . Moreover, there is a unique path between every pair of databases in the hierarchy. For these reasons, the set of databases to be considered for working set scheme is different for hierarchical scheme than that for HLR/VLR scheme. For $WS_{\text{Hierarchy}}$, when a call originates from caller residing in zone i for some callee residing in zone j , then the following zones will be considered by callee for working set calculation:

1. If $LCA(i, j)$ exists, then call is said to be originated from zone i . For example, if the call originates from zone a to the callee residing in zone b shown in Fig. 1, then call is said to be originated from zone a , and no other database in the path will be considered for working set calculation.

2. If $LCA(i, j)$ doesn't exist, then call is said to have originated from zone i and all its ancestors including the nearest root database to zone i . In $WS_{\text{Hierarchy}}$, we restrict the number of possible elements for working set. If all the databases in the path from caller zone to callee zone are considered for working set calculation, then the databases in the path to the old location of callee will be updated unnecessarily when callee moves to the new location. If the location information of mobile user U is to be replicated at the source (database) considered for working set or not is decided based on modifications in parameters described in Section 2.1 as:

f_s : The estimated frequency of lookups for obtaining location information of mobile user U at source s where s is a node in hierarchy which is considered for working set of user U .

δ_s : The additional cost incurred by s , in obtaining exact location of mobile host. This cost is defined as:

$$\text{cost}(s, Z_{\text{Callee}}) - 2 \quad (4)$$

where $\text{cost}(s, Z_{\text{Callee}})$ represents number of lookup messages required to obtain exact location of mobile host. The second entity in [Eq. 4] is always 2 because if exact location information is replicated at source s , then only two lookup messages are required, first lookup at source s and another lookup at Z_{Callee} . Thus δ_s is the additional

overhead incurred in terms of number of lookups when queried location database in the network does not have the exact location information of callee.

f_{update} : The estimated frequency with which mobile host changes its location and U_s : The cost incurred in both, announcing the location of mobile host to the source s and invalidating the location information at s . This cost is measured in terms of number of hops. The inequality equation remains same as [Eq. 2]. For hierarchical approach, call is said to be served locally only if caller and callee are in the same zone. Additionally, For $WS_{Hierarchy}$ call is said to be served locally only if the Z_{caller} is in the working set of callee.

5 Implementation of Working Set Scheme

The working set scheme is implemented at the MH. The frequency measures, f_s and f_{update} are determined using a simple averaging procedure. For $WS_{HLR/VLR}$ ($WS_{Hierarchy}$) the basic idea is to determine if it is more efficient to supply the source with the location information or to allow the source to route the connection request using HLR/VLR (hierarchical) approach. For computing the average arrival rate, the last n connection requests that arrived at the MH from source s are observed. Similarly, the last m moves made by the MH are used to compute the average update rate. f_s is computed by dividing the number of connection requests received by the time between the last and the first connection request. f_{update} is similarly computed. The algorithm for adaptive location management scheme has two parts, as explained below.

5.1 MH Actions after a Connection Request Is Received from Source S

If the source s is already a member of the update set, i.e. it has location information of the MH, the only action that needs to be taken at the MH is to record the time at which this connection request arrived from the source s . This will be used to compute f_s . On the other hand, if s is currently not a part of the update set, MH needs to make a decision about whether or not to include the source in its update set, by evaluating inequality equation. The parameters of the inequality, f_s and an estimated value of f_{update} are computed using the frequency estimation procedure explained earlier. δ_s is computed using [Eq. 3] for $WS_{HLR/VLR}$ and using [Eq. 4] for $WS_{Hierarchy}$. To determine U_s , the MH constructs a temporary spanning tree, which includes s and all the current update set members. This gives an estimate of the update cost that will be incurred if s is included in the update set [8]. If the inequality holds, s is included in the update set and location information is supplied over the already established connection.

5.2 MH Actions When It Changes Location

When the MH changes its location in the network, it evaluates f_{update} . The MH must determine if the sources currently in the update set should continue to be updated

from the new location. It also determines if there are other sources currently not in the update set, but which may have qualified to be put in the update set since the last move. It thus reevaluates the inequality equation for each source in its working set. If the inequality holds, the source will be included in the update set. Otherwise, the location information at the source must be invalidated (deregistered). Update and deregistration messages to the sources are sent along the minimum spanning tree which covers the elements of working set. Sources that are invalidated will then be removed from the update set. Since the precise estimation of the frequencies may require long-term samples, we have used mobility traces for 24 hours duration.

6 Mobility Traces Used for Simulation

We have used two mobility trace files for simulation. The mobility trace file contains the actual trace, a list of movement and call events in the PCS network. The events are arranged chronologically in the trace file.

The first mobility traces are for San Francisco Bay Area topology (93 zones, 11000 users per zone, 24 hours duration) is a benchmark trace and well validate against call and user data¹. The other traces for Greater Mumbai region (100 zones, total 126000 users, 24 hours duration) have been developed using already available survey data [11] for movements and using a realistic call model proposed in [12]. These traces are close to realistic traces but not validated due to unavailability of information. The call traffic model for these traces uses information that for the peak hour, the mean call arrival rate and the mean call duration during busy hours are 2.8 calls/hour and 2.6 min/call, respectively. Call distribution model uses parameters of Zipf's law given in [12] for one day.

7 Experimental Assessment of Working Set Scheme

Experimental assessment of working set scheme has been performed using location management simulator (LMSimulator) developed for analyzing performance of location management scheme and proposed in [10]. LMSimulator generates various statistics such as number of calls completed locally, database operations at each level of the hierarchy, messages sent over the network per unit time, total utilization of databases and links, maximum utilization of single database and links etc. given as input encoded location management strategy. More details about functionality of LMSimulator can be obtained from [10].

Other simulator offering such functionality is Pleiades² proposed in [2]. Each call setup has two phases. During the first phase, MH knows that there is a call for it and during second, the call is routed from mobile host to caller. For example, in HLR/VLR scheme, call is routed to caller from callee through HLR. Here, we

¹ These traces are generated using trace generator SUMATRA, which is well validated against real call and movement data. SUMATRA is not available in public domain. More information about these traces can be found at [13].

² The simulator Pleiades is not available in public domain.

concentrate on the first stage and not second stage. We also assume that there are no calls between the period during which MH changes zone and new location information is updated. We have used optimal database hierarchies for Bay area and Mumbai traces determined using the genetic algorithm framework proposed in [10]. Though LMSimulator generates various statistics for the location management scheme to be analyzed, we have given here some representative statistics related to the performance of working set scheme.

Table 1. Results of the Experimental Assessment

		San Francisco Bay Area				Greater Mumbai Region			
		H/V	WS H/V	HIE	WS HIE	H/V	WS H/V	HIE	WS HIE
1	Database updates	1.97	3.08	3.26	3.79	1.93	2.26	4.06	4.18
2	Database deregistrations	0.97	1.61	1.35	1.81	0.93	1.03	1.43	1.59
3	Database lookups	2.11	1.80	3.27	2.65	2.11	2.08	5.05	4.05
4	Total database operations	2.22	2.04	3.45	3.04	2.27	2.29	5.14	4.42
5	Update messages	0.97	5.15	2.26	2.86	0.93	4.77	3.06	2.97
6	Deregistration messages	0.97	0.93	2.07	4.29	0.93	0.10	2.31	4.06
7	Lookup messages	1.94	0.80	2.27	1.65	1.12	1.08	4.05	3.05
8	Total messages sent	1.94	1.51	2.55	2.39	1.28	1.70	4.33	3.92
9	Update hops	5.23	6.25	2.26	2.86	4.85	5.71	3.06	2.97
10	Deregistration hops	0.97	0.93	2.43	4.29	0.93	0.10	2.75	4.06
11	Lookup hops	11.2	3.59	2.62	2.62	8.57	8.18	5.56	5.56
12	Total hops	10.5	3.93	2.90	3.23	7.96	7.45	5.62	5.88
13	Updates over a link	1	1.03	1	1.09	1	1.23	1	1.02
14	Dereg over a link	1	0.81	1	0.57	1	0.09	1	0.43
15	Lookups over a link	1	0.25	1	1	1	0.95	1	1
16	Total msgs sent over a link	1	0.32	1	1.21	1	0.94	1	1.09
17	Updates for a database	1	1.79	1	1.05	1	1.03	1	1.01
18	Dereg for a database	1	1.49	1	1.25	1	1.02	1	1.08
19	Lookups for a database	1	0.82	1	0.65	1	0.98	1	0.70
20	Total ops for a database	1	0.89	1	0.74	1	1	1	0.78
21	Local to overall lookup ratio	0.44	0.75	0.42	0.58	0.20	0.32	0.14	0.37

Table 1 shows the results obtained. Rows 1 to 12 give number of operations per move for update and deregistration, per call for lookups and per event (call or move) for total operations. Note that the total is not merely sum of maximum update, deregistration and lookup operations but is the result of combined operations per event. Rows 14 to 16 (17 to 20) give the values for maximum usage of a single link (single database) considering the corresponding values are 1 for HLR/VLR and hierarchical approach. Row 21 shows local to overall lookup ratio (ratio of number of calls served locally to total number of calls) for corresponding schemes. Columns 3 to 6 and 7 to 10 show the results of HLR/VLR scheme, $WS_{HLR/VLR}$, hierarchical approach and $WS_{Hierarchy}$ for San Francisco Bay Area traces and Greater Mumbai Region traces respectively. Average call to mobility ratio is 6.43 and 3.59 for Bay Area traces and Mumbai Region traces respectively.

8 Analysis of the Results Obtained for Working Set Scheme

From the results obtained for Bay area traces it can be observed that $WS_{HLR/VLR}$ has shown overall performance improvement over HLR/VLR approach. Local to overall lookup ratio has shown an improvement of 71% over HLR/VLR scheme, which signifies reduction in call setup time. Total number of database operations, messages sent and hops taken by the messages is significantly less for $WS_{HLR/VLR}$. Also, maximum operations on a single database per unit time and maximum messages sent over a link per unit time are less than that of HLR/VLR approach. For $WS_{Hierarchy}$, 38% improvement has been observed in local to overall lookup ratio. In the case of $WS_{Hierarchy}$, 58% calls have been served locally. The penalty paid for it is 26% more messages sent for updates, 16% more database updates and slight increase in total number of hops required to send messages compared to HLR/VLR approach.

From the results obtained for Greater Mumbai traces it can be observed that $WS_{HLR/VLR}$ requires more number of database operations and messages per event as compared to HLR/VLR approach, but local to overall lookup ratio has 60% improvement. $WS_{Hierarchy}$ shows performance improvement in terms of local to overall lookup ratio at the penalty of slight increase in number of hops required per event.

For both the traces, peak database and link requirements for lookup messages are less or same and for updates they are more for working set approach compared to HLR/VLR and hierarchical approach. Peak number of total operations on databases is less for working set scheme while peak number of messages sent over a link is slightly more in case of $WS_{Hierarchy}$. Thus, for these traces, it can be concluded that implementing $WS_{HLR/VLR}$ will reduce overall load on the network and improve call setup time at the cost of additional functionality at the MH to calculate working set. For the same, $WS_{Hierarchy}$ will require slight increase in bandwidth and database requirements.

For robust and conclusive analysis of working set scheme, future work involves analysis of the working set scheme for synthetic mobility traces generated using some mobility and call models. Such analysis has been done in [10] for mobility traces generated using uniform random call and movement model (though not validated). For these traces, results obtained in [10] show some positive and negative aspects of working set scheme.

9 Conclusions and Future Work

Variants of working set idea proposed for location management in mobile IP have been developed for PCS networks. Experimental performance evaluation of these variants using optimal database hierarchies and two trace files modeling real life call and movement pattern show that the working set scheme performs superior to HLR/VLR approach in terms of call setup time and overall database and bandwidth requirements. For hierarchical approach, working set scheme shows promising results at the cost of a slightly increase in bandwidth and database requirements. The results reported in this paper are not as conclusive as they should be and hence work is in progress to be more conclusive.

References

1. Jan Jannink, Derek Lam, Narayanan Shivkumar, Jennifer Widom, Donald C. Cox: Modeling location management in personal communication services. Proceedings of the 1996 IEEE International Conference on Universal Personal Communications, volume 2 pages 596–601, Cambridge, Massachusetts (September 1996)
2. Jan Jannink, Derek Lam, Narayanan Shivakumar, Jennifer Widom, Donald C. Cox: Efficient and flexible location management techniques for wireless communication systems. ACM/Baltzer Wireless Networks, 3(5) pages 361–374 (October, 1997)
3. Karunaharan Ratnam, Ibrahim Matta, Sampath Rangarajan: A fully distributed location management scheme for Large PCS. Technical Report BU-CS-1999-010, Computer Science Department, Boston University, Boston, MA 02215 (August 1999)
4. Karunaharan Ratnam, Ibrahim Matta, Sampath Rangarajan: Analysis of Caching Based Location Management Scheme for Personal Communication Networks. Proceedings of IEEE ICNP, Toronto (1999)
5. B. R. Badrinath, T. Imielinski, and Aashu Virmani: Locating Strategies for Personal Communication Networks. Proceedings of IEEE Globecom 92 (December, 1992)
6. Ravi Prakash and Mukesh Singhal: Dynamic Hashing + Quorum = Efficient Location Management for Mobile Communication Systems. Proceedings of the Principles of Distributed Computing, Santa Barbara, CA (August, 1997)
7. Walter H. Truitt, Nalini Mysore: Distributed Location Management.
http://www.utdallas.edu/~wtruitt/netlocmgt_toc.html (1998)
8. Subhashini Rajagopalan, B. R. Badrinath: An adaptive location management strategy for mobile IP. Proceedings of the first annual international conference on Mobile computing and networking, November 13–15, Berkeley, CA USA (1995)
9. V. Ananthram, M. L. Hoing et. Al.: Optimization of a Database Hierarchy for Mobility Tracking in a Personal Communication Network. Performance evaluation, vol. 20 pages 287–300 (1994)
10. Pravin Amrut Pawar: Location management in personal communication services network. M.Tech. dissertation, Indian Institute of Technology, Bombay, India (March, 2002)
11. S. L. Dhingra and Transportation System Engineering group, IIT Bombay: Traffic And Transportation Study For Worli Bandra Sea Link. WBSL study draft report (1999)
12. D. Lam, D. C. Cox, and J. Widom: Teletraffic modeling for Personal Communications Services. IEEE Communications Magazine Special Issue on Teletraffic Modeling, Engineering and Management in Wireless and Broadband Networks, 35(2): pages 79–87 (February, 1997)
13. Stanford University database group homepage: www-db.stanford.edu/Sumatra
14. Ravi Jain, Yi_Bing Lin, Charles Lo and Seshadri Mohan: A Caching Strategy to Reduce Network Impacts of PCS. IEEE Journal on selected areas of communication, pages 1434–1444 (October, 1994)
15. R.Yates, C.Rose, S.Rajagopalan, B.R.Badrinath: Analysis of a Mobile Assisted Adaptive location Management Strategy. ACM Wireless Networks (1995)

Parallel Performance of the Interpolation Supplemented Lattice Boltzmann Method

C. Shyam Sunder¹, G. Baskar¹, V. Babu¹, and David Strenski²

¹ Department of Mechanical Engineering,
Indian Institute of Technology,
Madras, India 600 036.

vbabu@iitm.ac.in

² Applications Analyst
Cray Inc., Seattle,
Washington, USA.
stren@cray.com

Abstract. The interpolation supplemented lattice Boltzmann (ISLB) method was proposed by He *et. al.* (*J. Comput. Phys.*, **129**, 357 (1996)) as a modification of the traditional lattice Boltzmann (LB) method for simulating incompressible flows. The traditional LB method has been shown to be a highly parallel method (*Computers in Physics*, **8**, No. 6,705 (1994)). In this paper, the parallel performance of the ISLB method, which has different computational requirements than the traditional LB method, is investigated. The performance of the ISLB method on Cray X1, Cray T3E-900 and SGI Origin 3000 is presented. The noteworthy feature of the present implementation of the ISLB method is that it was able to achieve a sustained speed of 550 Gflops on a 124 processor Cray X1.

Keywords: Lattice Boltzmann Parallelization SHMEM

1 Introduction

The lattice Boltzmann method is a very promising method for simulating incompressible fluid flows [1]. The method is appealing due to its simplicity - simplicity in the mesh used (cartesian mesh) and simplicity in the operations performed (collision, advection and boundary update). However, the simplicity of the uniform cartesian mesh is also a liability for two reasons: one, complex, curved geometries cannot be handled without losing boundary integrity and two, uniform mesh everywhere results in a large number of mesh points, which in turn, increases the computational cost tremendously. The first issue has been addressed by Mei *et al.*, [2,3]. The second issue has been addressed by He *et al.*, [4], wherein they proposed the interpolation supplemented lattice Boltzmann (ISLB) method. This allows non-uniform as well as curvilinear body-fitted meshes to be used in the lattice Boltzmann calculations. An alternative approach for using non-uniform meshes based on the concept of hierarchical grid refinement was

proposed by Filippova and Hanel[5,6]. In the present work we have chosen to use the former approach as it requires only a relatively straightforward modification of the traditional lattice-BGK procedure for simulating incompressible flows and is thus quite easy to implement.

2 A Brief Review of the Lattice Boltzmann Method

In this work, the 2D LB method on a square lattice is considered. Some of the basic features of the LB method alone are presented here. The details are available elsewhere [4]. Each node of the lattice is populated by three kinds of particles - a rest particle that resides in the node, particles that move in the coordinate directions and particles that move in the diagonal directions. The total number of particles in each node in this model (called the d2q9i) model is 9. The speed of the particles are such that they move from one node to another during each time step. These speeds can be written as

$$\mathbf{e}_i = \begin{cases} 0, & i = 0 \\ c(\cos((i-1)\pi), \sin((i-1)\pi)), & i = 1, 2, 3, 4 \\ \sqrt{2}c(\cos((i-5)\pi/2 + \pi/4), \sin((i-5)\pi/2 + \pi/4)), & i = 5, 6, 7, 8 \end{cases} \quad (1)$$

Here $c = \delta_x/\delta_t$, where δ_x and δ_t are the lattice spacing and the time step respectively. In the traditional LB method the particles at each node undergo collision followed by advection. In terms of distribution functions, this can be written as [4]

$$p_i(\mathbf{x} + \mathbf{e}_i \delta_t, t + \delta_t) - p_i(\mathbf{x}, t) = -\frac{1}{\tau} \left[p_i(\mathbf{x}, t) - p_i^{(eq)}(\mathbf{x}, t) \right], \quad (2)$$

where τ is the dimensionless collisional relaxation time and is related to the kinematic viscosity ν of the lattice fluid as [4]

$$\nu = \frac{(2\tau - 1)}{6} \frac{\delta_x^2}{\delta_t} \quad (3)$$

Equation (2) above describes the evolution of the LB automaton. The equilibrium functions that appear in the right hand side of this equation can be evaluated [4] as follows:

$$p_i^{(eq)} = w_i \left[p + \rho_0 \left((\mathbf{e}_i \cdot \mathbf{u}) + \frac{3}{2} \frac{(\mathbf{e}_i \cdot \mathbf{u})^2}{2} - \frac{1}{2} \mathbf{u}^2 \right) \right], \quad (4)$$

where, p is the pressure and \mathbf{u} is the velocity vector of the fluid and are given as

$$p = \sum_{i=0}^8 p_i \quad (5)$$

$$\rho_0 \mathbf{u} = \frac{1}{c_s^2} \sum_{i=1}^8 \mathbf{e}_i p_i \quad (6)$$

where c_s is the speed of sound and is equal to $c/\sqrt{3}$ for the model under consideration. The quantity ρ_0 can be thought of as a reference density and has been taken to be equal to 1.

In case of the ISLB method, the lattice Boltzmann automaton is assumed to reside on a *lattice* with spacing exactly equal to the finest mesh spacing overlaid on the *computational grid*. During each time step, the particles on this *lattice* undergo collision followed by advection as dictated by the lattice-BGK model[4]. The particle distribution function at each node of the *computational grid* can then be determined by using second order accurate Lagrangian interpolation [4] (this is not required in the finest mesh region as the mesh spacing here equals the lattice spacing). The interpolation is done by using values upwind (based on the direction of the particle velocity not the fluid velocity) of the node. The particle distributions on the boundary nodes of the *computational grid* are modified next according to the imposed boundary conditions. This way, although the automaton resides on a fine mesh, the particle distribution functions are calculated and stored only on the nodes of the *computational grid* which can be tailored to the problem being solved *i.e.*, fine mesh in regions of strong gradients and coarse mesh everywhere else.

3 Parallel Implementation of the ISLB Method

3.1 Computational Issues

The computations of the ISLB method can be summarized as follows:

1. Start with an initially guessed fluid velocity field
2. Calculate the equilibrium particle distribution based on the velocity and pressure field based on equation (4)
3. Perform the collision operation at each node (right hand side of equation (2))
4. Communicate the post collision distribution data from appropriate nodes to neighboring processors that need the data.
5. Calculate particle distribution function at each node from the post collision distribution using second order accurate interpolation
6. Update the particle distributions on physical boundaries based on the prescribed boundary conditions
7. Calculate the fluid velocity and pressure at each node using equations (5,6)
8. Repeat steps (2) through (7) until the flow field reaches a steady state or until enough cycles of the unsteady flow field have been obtained.

From a computational perspective, steps (2), (3), (5) and (7) are the most important as they account for nearly all the computations. In the actual implementation, steps (7), (2) and (3) can be combined which results in considerable

reuse of data already in the cache thereby improving the computational speed. Mapping the arrays onto the L1 and L2 cache properly with pads between them and stripmining the loops also results in further improvement of speed. All the computational kernels have been written in Fortran77 and everything else in C. In the present case, the total number of floating points operations per node per time step comes out to be 120 multiplications and 101 additions/subtractions. This allows the computational speed to be calculated very accurately, once the execution time is measured. In calculating the computational speed, both additions/ subtractions and multiplications were treated equally. It should be noted that step (5) above is not applicable in the traditional LB method, but in the ISLB method it accounts for about 45% of the total computations. This increase in computations is more than offset by the decrease in the number of nodes in the computational grid for the ISLB method.

3.2 Domain Decomposition

In the present work, all the parallel computations were done by dividing the computational domain into equal parts and then assigning each part to a processor. The size of each part was determined based on considerations of surface-to-volume ratio [7], as follows. Let the computational domain be of size $N_x \times N_y$ points. Further, let the domain be decomposed onto a two dimensional network of $P_x \times P_y$ processors. This implies that each processor is assigned a portion of the computational domain that is $N_i \times N_j$, where $N_i = N_x/P_x$ and $N_j = N_y/P_y$. Thus the surface-to-volume ratio of the part of the computational domain assigned to each processor is $2(N_i + N_j)/N_i N_j$, or, $2(P_x/N_x + P_y/N_y)$. By defining $P_x \times P_y = P$, where P is the total number of processors, the surface-to-volume ratio becomes equal to $2(P_x/N_x + P/(P_x N_y))$. The optimal processor layout for solving a given problem (*i.e.*, N_x, N_y specified) using P number of processors can be obtained by minimizing the previous expression with respect to P_x . This gives $P_x = \sqrt{PN_x/N_y}$. This, of course, can result in fractional values for N_i and N_j . In the actual runs, the value for P_x can be chosen such that it results in integral values for N_i and N_j , while still being reasonably close to the value given by the above formula. This means that P will not necessarily be a power of 2.

3.3 Communication Issues

The amount of data to be communicated to the neighboring processors in step (4) above is more in the ISLB method than the traditional LB method. This is because the second order accurate interpolation procedure (step 5) at each node requires data from that node plus the nearest as well the next nearest neighboring node in the upwind direction. This means that, for particles with velocity in the directions $i = 1, 2, 3, 4$ in equation (1) above, data from three nodes is required, while for particles with velocity in the directions $i = 5, 6, 7, 8$ data from a total of 9 nodes is required. In the current implementation, communication between processors has been done using SHMEM routines, which offer high data transfer

rates between processors on the Cray and the SGI machines. Porting to MPI is currently in progress. Also, `SHMEM_PUT` routines have been used throughout, as they offer better performance than the `SHMEM_GET` routines.

3.4 Load Balancing Issues

The computations involved in steps (2),(3) and (7) require only local data and so these steps can be parallelized trivially and they also do not pose any difficulties related to load balancing. Step (4), namely communication of data has to be completed before the computations in steps (5) and (6) can be done (it should be kept in mind that step (7) has been merged with step (2)). This poses some load balancing problems as the processors that have been assigned the interior of the computational domain communicate more data than the others. Step (6), namely, implementation of the boundary conditions also presents difficulties in load balancing as the processors assigned the interior of the domain do less computations (if any) in this step. However, good load balance can be achieved without too much additional effort as will be shown in the next section.

4 Results and Discussion

4.1 Cray X1

Parallel performance of the code on a 31 node Cray X1 machine is shown in Figs. 1 and 2. The Cray X1 is a scalable vector MPP architecture with a peak flop rate of 12.8 Gflops (64-bits) per MSP processor or 24 Gflops for 32 bit calculations. The latter number is relevant in the present case as all the calculations have been done using 32 bit floating point numbers. Each node of the machine contains 4 MSP processors. It is also possible to run parallel applications in an SSP mode where each MSP processor behaves as though it is a collection of 4 tightly coupled SSP processors. The Cray X1 is a shared memory architecture within each node whereas the memory is logically shared but distributed across the nodes. It is also possible to specify the number of processors to be used per node for parallel applications. For example, an 8 processor run can be done by using 8 nodes with 1 MSP processor per node or by using 4 nodes with 2 MSP processor per node or by using 2 nodes with 4 MSP processors (the maximum possible) per node. The same concept can be used for execution in SSP mode also, except that, now, upto 16 SSP processors are available per node.

The performance curves in Fig.1 show the results for parallel runs with MSP processors. In these figures, speedup values are presented for the slowest execution times. Also, the solid line in this figure (as well as the subsequent figures) indicates linear speedup. The general trend in Fig. 1 is that the load balance is quite good when the number of processors is less than 50 and it worsens somewhat afterwards (the difference in CPU time between the processor finishing first and last in the worst case was about 3% of the average CPU time). The maximum speed achieved in MSP mode was 10 Gflops per processor and the minimum was

5 Gflops. In general, the performance deteriorated somewhat as the number of processors per node was increased. This can be attributed to the fact that the bandwidth available for each processor in a particular node decreases as more processors in that node are used. The maximum speed achieved was **550 Gflops** for a run with 124 MSP processors.

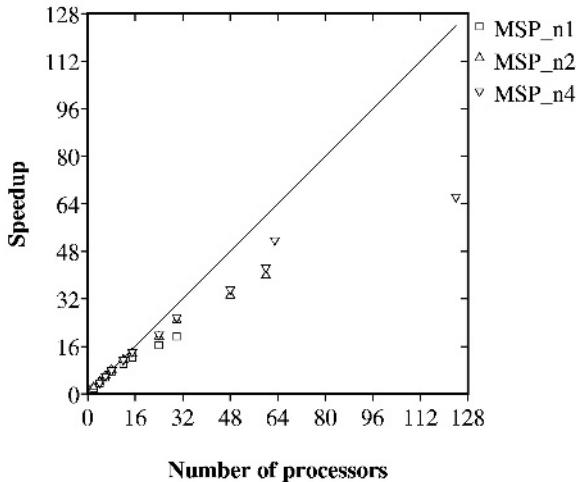


Fig. 1. Performance curve on Cray X1 for 1, 2 and 4 MSPs per node

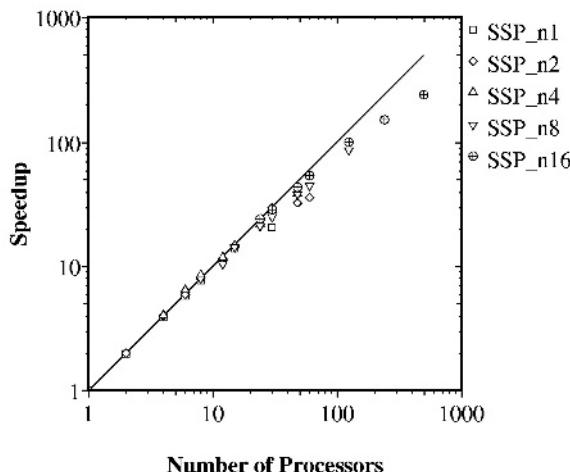


Fig. 2. Performance curve on Cray X1 for 1, 2, 4, 8 and 16 SSPs per node

Performance curves for parallel calculations with SSP processors are shown in Fig. 2. The speedup is very close to being linear for all the cases for num-

ber of processors upto 30 or so. Beyond that, as the number of SSPs per node is increased, the speedup falls off considerably, as the available bandwidth per node has to be shared among more SSPs. This trend is similar to what is seen with increasing number of MSP processors per node. However, the load balance, even for the run with 496 processors is quite good as can be seen from this figure. This clearly shows that as the problem size per processor is reduced, the increased communication cost more than offsets the performance gains due to better positioning in the cache. The maximum speed achieved was **436 Gflops** for a run with 496 SSP processors. The overall performance of the MSP processors is better because the parallelization is at the streaming level. In the case of SSP processors, more parallelization is at the process level and the overheads associated with this are more than those of the MSP processors.

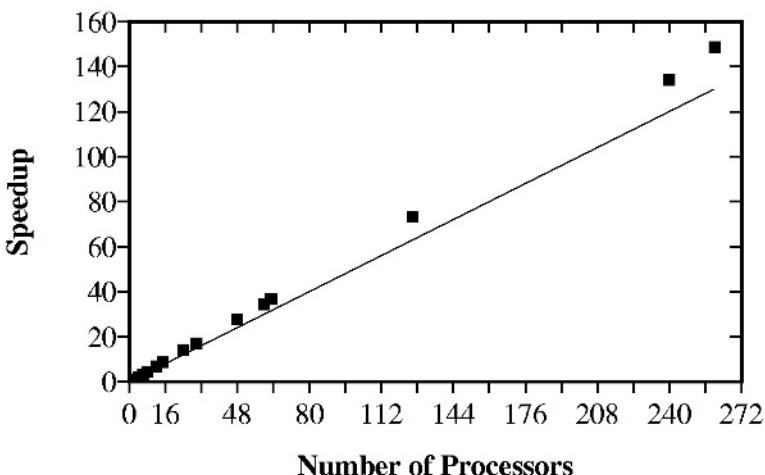


Fig. 3. Performance curve on Cray T3E

4.2 Cray T3E

Parallel performance of the code on the distributed memory Cray T3E-900 architecture is discussed next. This architecture utilizes 450 MHz DEC EV5 processors and on the machines that were used for the timing studies, each processor has 256 MB of memory, 8 KB of primary cache and 96 KB of secondary cache. In this work, performance studies were carried out on two T3E machines, which are identical except for the number of processors. All the calculations on the T3E were done with 64 bit numbers as it is not possible to do 32 bit calculations. The computational speed per processor for a 2 processor run came out to be 144 Mflops (the size of the problem in each processor was too big for single processor runs). The speed increased to 165 ± 3 Mflops per processor for runs using larger number of processors, which indicates that the problem fits better

in the cache as the number of processors is increased. This is reflected in the super-linear speed-up seen in Fig. 3 right from the beginning. For the run with 260 processors, the difference in CPU time between the processor finishing first and last was about 0.1% of the average CPU time, indicating extremely good load balance. The speed-up is 14% super-linear for the run with 260 processors. The noteworthy trend from Fig. 3 for the T3E architecture is that the degree of super-linearity continues to increase for number of processors upto 260. This is in contrast to the usual trend on distributed memory machines where the speed-up tapers off beyond a certain number of processors as applications become communication bound. The maximum sustained speed achieved for this architecture was 43 Gflops for a run with 260 processors. It should be kept in mind that these speeds are for 64 bit calculations, so when comparing with the speeds from the other architectures such as the Cray X1 above or the SGI Origin (see below), these should be multiplied by a factor of two.

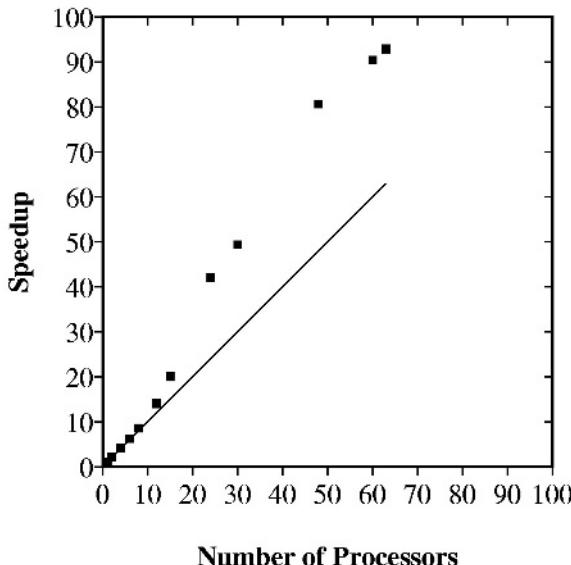


Fig. 4. Performance curve on Origin 3000

4.3 SGI Origin 3000

Parallel performance of the code on a shared memory SGI Origin 3000 machine running 500 MHz, R14000 processors is presented next. This machine has 64 processors, main memory of 32 GB and primary and secondary caches of size 32 KB and 8 MB respectively. Figure 4 shows the performance (using 32 bit floating point numbers) of the current implementation of the ISLB method on this machine. The speed-up is slightly super-linear for number of processors upto 10.

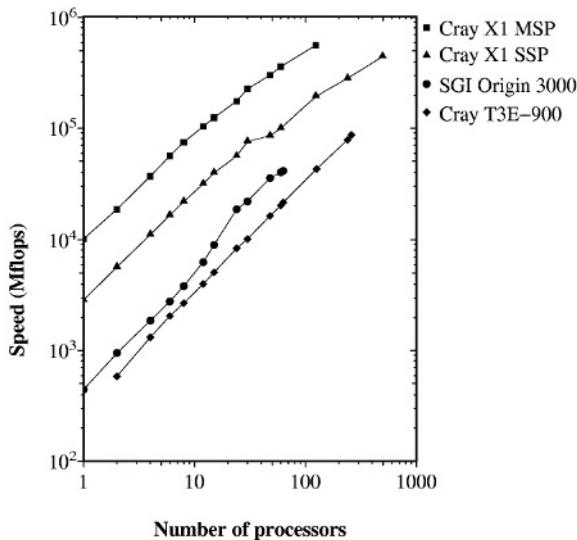


Fig. 5. Computational Speeds on Cray X1, Cray T3E and Origin 3000

Beyond this, as the problem size in each processor becomes small enough to fit in the secondary cache, the speed-up is highly super-linear. The computational speed per processor also increases to about 650 Mflops from 430 Mflops. For number of processors equal to 24, the speed-up is 75% super-linear, but for 63 processors, it is only 50% super-linear, indicating that the application is becoming communication bound. The maximum speed achieved on this architecture was 40 Gflops for a run with 63 processors.

Finally, the overall speeds achieved on the three architectures are shown in Fig. 5. Here, the computational speed (calculated based on the CPU time of the processor finishing last) is plotted against the number of processors using a *log-log* scale. The values plotted in this figure for the case of the Cray T3E machine are actual speeds (for 64 bit calculations) multiplied by two. For a given number of processors, say, for example 60, the Cray X1 in MSP mode gives speeds that are about 3.6 times better than the Cray X1 in SSP mode, about 9 times better than the SGI Origin 3000 and about 18 times better than the Cray T3E-900.

5 Conclusions

The parallel performance of the Interpolation Supplemented Lattice Boltzmann method on different architectures has been investigated. Although the communication and computational requirements per node per time step of the ISLB method are higher than the traditional LB method, the parallel performance is, nevertheless, quite good, particularly on cache based machines. Some further improvement in the single processor performance of the implementation

appear to be possible and these are currently being explored. Performance of the method when using MPI is of considerable interest as the data transfer rates between processors is lesser in the MPI environment than with SHMEM routines. This is very likely to introduce some load imbalance and thus affect the parallel performance.

Acknowledgements. The authors would like to thank the Computer Center, IIT Madras for granting dedicated time on their SGI Origin 3000. The authors would also like to thank Cray Inc, for allowing timing studies to be conducted on the Cray X1 and the Cray T3E at their Chippewa Falls facility in Wisconsin. The authors would further like to thank the Arctic Region Supercomputer Center at Fairbanks, Alaska for granting permission to perform timing studies on their Cray T3E.

References

1. S. Chen and G.D. Doolen, Lattice Boltzmann Method for Fluid Flows, *Annual Review of Fluid Mechanics*, **30**, 329 (1998).
2. R. Mei, L.-S Luo and W. Shyy, An Accurate Curved Boundary Treatment in the Lattice Boltzmann Method, *Journal of Computational Physics*, **155**, 307 (1999).
3. R. Mei, W. Shyy, D. Yu and L.-S Luo, Lattice Boltzmann Method for 3D Flows with Curved Boundary, *Journal of Computational Physics*, **161**, 680 (2000).
4. X. He, L.-S. Luo and M. Dembo, Some progress in lattice Boltzmann method. Part 1. Nonuniform mesh grids, *J. Comput. Phys.*, **129**, 357 (1996).
5. O. Filippova and D. Hanel, Grid refinement for lattice-BGK models, *J. Comput. Phys.*, **147**, 219 (1998).
6. O. Filippova and D. Hanel, Acceleration of lattice-BGK schemes with grid refinement, *J. Comput. Phys.*, **165**, 407 (2000).
7. M. J. Quinn, *Parallel Computing, Theory and Practice*, 2nd Edition, McGraw-Hill Inc, 1994, pp. 241–243.

Crafting Data Structures: A Study of Reference Locality in Refinement-Based Pathfinding*

Robert Niewiadomski, José Nelson Amaral, and Robert C. Holte

Department of Computing Science, University of Alberta
Edmonton, AB, Canada {niewiado, amaral, holte}@cs.ualberta.ca

Abstract. The widening gap between processor speed and memory latency increases the importance of crafting data structures and algorithms to exploit temporal and spatial locality. Refinement-based pathfinding algorithms, such as Classic Refinement, find near-optimal paths in very large sparse graphs where traditional search techniques fail to generate paths in acceptable time. In this paper we present a performance evaluation study of three simple data structure transformation oriented techniques aimed at improving the data reference locality of Classic Refinement. In our experiments these techniques improved data reference locality resulting in consistently positive performance improvements upwards of 51.2%. In addition, these techniques appear to be orthogonal to compiler optimizations and robust with respect to hardware architecture.

1 Introduction

Pathfinding has applications in many industries such as computer games, freight transport, travel planning, circuit routing, network packet routing, etc. For instance, in the Real Time Strategy (RTS) video-game genre refinement-based search and its variants are used to conduct pathfinding for movements on the game map [6]. In these games pathfinding consumes up to half of total computation time [1,15]. Using a modification to Dijkstra's algorithm, the shortest path between two vertices in a graph $G(V, E)$ can be computed in $O(E \log V)$ [4]. However, in some time sensitive applications where $|V|$ is very large we may want to visit only a fraction of the vertices in V to find an approximation of the shortest path [10]. *Refinement-based search* (RBS) is often used to restrict the search space to generate quality paths [9]. *Classic Refinement* (CR), a variation of RBS, partitions a large graph into many subgraphs, and generates an *abstract graph* that describes the interconnections among the subgraphs. A path between two vertices, u and v in the original graph is found by: (1) identifying the vertices in the abstract graph that correspond to the partitions containing u and v ; (2) finding a path, in the abstract graph, between the identified vertices; (3) using this abstract path to find a path in the original graph.

* This research is partially funded by grants from the Natural Sciences and Engineering Research Council of Canada and by the Alberta Ingenuity Fund.

This paper presents a performance evaluation study of three techniques for improving the data reference locality of CR: (1) data duplication; (2) data re-ordering; (3) and merging of independent data structures into a common memory area. We demonstrate that combining these techniques can result in performance improvements upwards of 51.2% and that, through analysis of hardware event profiles, these results stem from improved page level and cache line level locality. By testing on four systems with various compilers and optimization settings we also demonstrate that our techniques are robust to changes in hardware architecture as well as the level of compiler optimization.

Section 2 presents the CR algorithm. Section 3 describes the baseline implementation and our three techniques. Section 4 presents experimental results and analysis. Section 5 discusses related work.

2 Abstraction and Search

Let $G_0(V_0, E_0)$ be the input graph to CR. Let $G_1(V_1, E_1)$ be an *abstraction* of G_0 . G_0 and G_1 are both undirected, unweighted graphs. G_0 is partitioned into connected subgraphs. G_1 must have one vertex for each subgraph of G_0 . If vertex v_i^0 in G_0 maps to vertex v_p^1 in G_1 , v_p^1 is called the *image* of v_i^0 at abstraction level 1 (Note: v_p^1 should be read as “vertex p at abstraction level 1”). The set of vertices in G_0 that map to vertex v_p^1 in G_1 is the *pre-image* of v_p^1 . G_1 has an edge (v_p^1, v_q^1) if and only if there is an edge (v_i^0, v_j^0) in G_0 such that v_i^0 is in the pre-image of v_p^1 and v_j^0 is in the pre-image of v_q^1 . This ensures that paths in G_0 can be mapped to corresponding paths in G_1 . Because we can create an abstract graph for any undirected graph we can create an abstraction of an abstraction to generate a hierarchy of abstractions. A sequence of graphs $\{G_0, G_1, \dots, G_{n-1}\}$ is an abstraction hierarchy for source graph G_0 if for $0 \leq i < n - 1$ G_{i+1} is an abstraction of G_i . To generate an abstraction hierarchy we use the “max-degree” STAR algorithm [9]. Given G_0 and a constant r , the STAR algorithm partitions G_0 into subgraphs whose maximum diameter is at most $2r$.

Definition 1. An ordered list of G_a vertices, $P = \{v_0^a, v_1^a, \dots, v_{k-1}^a\}$, is a **path** in G_a if and only if G_a contains the edges (v_0^a, v_1^a) , (v_1^a, v_2^a) , \dots , (v_{k-2}^a, v_{k-1}^a) . We use $P[j]$ to refer to the j^{th} element in path P .

Definition 2. A path $P = \{v_0^a, v_1^a, \dots, v_{k-1}^a\}$ in G_a is a **constrained path** if and only if it is the shortest path between v_0^a and v_{k-1}^a , such that vertices $v_0^a, v_1^a, \dots, v_{k-1}^a$ belong to the pre-image of the same vertex v_p^{a+1} . Because the pre-image of v_p^{a+1} is a connected subgraph of G_a , when computing a constrained path, a search algorithm restricts its search space to the vertices in the pre-image of v_p^{a+1} .¹

¹ In a constrained path all vertices are in the same pre-image. G_0 may contain a shorter path between v_0^a and v_{k-1}^a than the constrained path P , but any such path contain at least one vertex outside the pre-image of v_p^{a+1} , and therefore is not a constrained path.

Definition 3. Let v_p^{a+1} and v_q^{a+1} be two vertices in G_{a+1} such that (v_p^{a+1}, v_q^{a+1}) is an edge in G_{a+1} . Let v_i^a be a vertex in the pre-image of v_p^{a+1} . Then there exist a path from v_i^a to any vertex in the pre-image of v_q^{a+1} . A **constrained jump path** J from v_i^a to the pre-image of v_q^{a+1} is the shortest path between v_i^a and any vertex in the pre-image of v_q^{a+1} , such that any edge traversed by J connects vertices that belong either to the pre-image of v_p^{a+1} or to the pre-image of v_q^{a+1} .²

```

CLASSICREFINEMENT( $A, s^0, g^0, n$ )
1:    $s^{n-1} \leftarrow \text{LOOKUPVERTEXIMAGE}(s^0, n - 1)$ 
2:    $g^{n-1} \leftarrow \text{LOOKUPVERTEXIMAGE}(g^0, n - 1)$ 
3:    $P_{n-1} \leftarrow \text{FINDPATH}(s^{n-1}, g^{n-1}, n - 1)$ 
4:   if  $|P_{n-1}| = 0$  then
5:     return  $\text{NULL}$ 
6:   for  $i = n - 2$  to  $i = 0$ 
7:      $P_i \leftarrow \{\}$ 
8:      $b \leftarrow \text{LOOKUPVERTEXIMAGE}(s^0, i)$ 
9:     for  $j \leftarrow 0$  to  $j = |P_{i+1}| - 1$ 
10:     $J \leftarrow \text{FINDCONSTRAINEDJUMPPATH}(G_i, b,$ 
            $P_{i+1}[j + 1])$ 
11:     $P_i \leftarrow \text{APPEND}(P_i, J)$ 
12:     $b \leftarrow \text{LASTVERTEX}(J)$ 
13:  endfor
14:   $g_i \leftarrow \text{LOOKUPVERTEXIMAGE}(g^0, i)$ 
15:   $C \leftarrow \text{FINDCONSTRAINEDPATH}(b, g_i, i)$ 
16:   $P_i \leftarrow \text{APPEND}(P_i, C)$ 
17: endfor
18: return  $P_0$ 

```

Fig. 1. Classic Refinement Algorithm.

```

EMBEDDEDQUEUECONSTRAINEDPATH( $G(V, E), s, I$ )
1:    $EQP(s) \leftarrow \text{NULL}$ 
2:    $w \leftarrow s$ 
3:    $w' \leftarrow \text{NULL}$ 
4:   while TRUE
5:     while  $w \neq \text{NULL}$ 
6:       for  $v \in V$  such that  $(w, v) \in E$ 
7:         if  $\text{Image}(v) = I$ 
8:            $BST\_BP(v) \leftarrow w$ 
9:           return  $v$ 
10:        if  $\text{Image}(v) \neq \text{Image}(s)$ 
11:          continue
12:        if  $TVM(v) = \text{TRUE}$ 
13:          continue
14:         $BST\_BP(v) \leftarrow w$ 
15:         $EQP(v) \leftarrow w'$ 
16:         $w' \leftarrow v$ 
17:         $TVM(v) \leftarrow \text{TRUE}$ 
18:      endfor
19:       $w \leftarrow EQP(w)$ 
20:    endwhile
21:     $w \leftarrow w'$ 
22:     $w' \leftarrow \text{NULL}$ 
23:  endwhile

```

Fig. 2. Embedded Queue Constrained Path Algorithm with Abstract Map

Figure 1 presents pseudocode for the CR algorithm. Given a source graph G_0 and an abstraction hierarchy $A = \{G_0, G_1, \dots, G_{n-1}\}$ CR finds a path in G_0 between a source vertex s^0 and a goal vertex g^0 . The CR algorithm starts by finding a path, P_{n-1} , between s^{n-1} and g^{n-1} , the images of the source and goal vertices in the highest level of the hierarchy, G_{n-1} . If no such path exists then the algorithm returns NULL (steps 1-5). $\text{LOOKUPVERTEXIMAGE}(g^0, n - 1)$ returns the image of g^0 at abstraction level $n - 1$. $\text{FINDPATH}(s^{n-1}, g^{n-1}, n - 1)$ returns a path from s^{n-1} to g^{n-1} at abstraction level $n - 1$. If a path is found, CR iterates through each level of abstraction (for loop at step 6). Let $P_{i+1} = \{s^{i+1}, v_1^{i+1}, \dots, v_{k-2}^{i+1}, g^{i+1}\}$ be the path in G_{i+1} . To compute the path P_i , CR initializes b to the image of s^0 at abstraction level i . CR then computes the constrained jump path J from b to a vertex in the pre-image of the next P_{i+1} vertex, $P_{i+1}[j + 1]$ (step 10). By definition the last vertex in J is the first vertex

² Again, a shorter path from v_i^a to the pre-image of v_p^{a+1} may exist in G_0 , but it would have to include at least one vertex outside the pre-image of v_p^{a+1} or v_q^{a+1} and thus not be constrained.

in the pre-image of $P_{i+1}[j + 1]$ visited by J . CR appends the constrained jump path J to P_i , updates b to be the first vertex visited in $P_{i+1}[j + 1]$ and iterates until the pre-image of g^{i+1} is reached. Finally, when b is the initial vertex in the pre-image of g^{i+1} , CR computes a constrained path C between b and g^i , the image of g^0 in G_i (step 15) and appends C to P_i . When the recursion finishes, CR returns the path P_0 .

3 The Three Data Layout Techniques

Our baseline implementation of CR uses an adjacency list representation. We use the graph in Figure 3 as an example. In the baseline the vertex v_0^0 of Figure 3 has the data structure shown in Figure 4(a). ID is a unique identification field, the traversal visit marker (TVM) indicates if the vertex has been visited, BP is a back pointer, IMG is a pointer to the vertex's image, and DEG is the number of neighbors. The use of a 32-bit field for the TVM allows us to not have to reinitialize it upon the start of each search. All TVMs are initialized to zero and henceforth a global search counter is maintained. Whenever a vertex v_a^i is visited during the z^{th} search, its TVM is set to z . Therefore any vertex that has a TVM smaller than z during search z , has not been visited yet.

We use Breadth First Search (BFS) to search for constrained paths and constrained jump paths. BFS stores vertices to be visited in a working queue. This queue is sometimes implemented as a circular buffer due to performance and memory considerations [4]. However we found that the overhead of checking for wrap-around and overflow is high. We eliminate this bookkeeping by simply allocating enough memory to contain a pointer to each vertex in the graph.

Vertex Clustering: Figure 5 shows a layout of the vertex data structures in memory for the graph of Figure 3. Each small box represents a 32-bit memory field. For convenience of drawing we present eight 32-bit fields per line. We identify the 32-bit field where the data structure corresponding to each vertex starts. Consider a search, in Figure 3, for a constrained jump path from v_0^1 to v_2^1 starting at v_0^0 and ending at v_3^0 . The shaded areas in Figure 5 show the memory locations that are accessed in this search. Besides the irregular memory access pattern shown in Figure 5, the baseline implementation also keeps a work queue in a separate region of memory. Accesses to this queue are interleaved with the accesses shown in Figure 5. Such accesses hurt spatial locality and are potential source of data cache conflict misses. Our *vertex clustering* technique re-arranges the vertices, such that vertices that map to the same image are located in close proximity of each other in memory. Figure 6 shows the memory layout after vertex clustering has been applied. Shaded areas are locations accessed for the same constrained jump path search. Notice how the memory accesses are much closer to each other in memory. Though not addressed in this paper we expect vertex clustering benefit abstractions generated with larger radius.

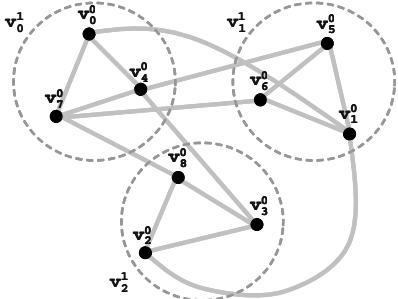


Fig. 3. Running Example.

(a) Baseline.

ID	TVM	BP	IMG	DEG	Adjacency List
0	0	NULL	v_0^1	3	v_1^0 , v_4^0 , v_7^0 , v_8^1

(b) Abstract map.

ID	TVM	BP	EQP	IMG	DEG	Adjacency List
0	0	NULL	NULL	v_0^1	3	v_1^0 , v_1^1 , v_4^0 , v_6^0 , v_7^0 , v_8^1

(c) Abstract map with embedded queue.

Fig. 4. Fields in the data structure of a vertex.

0x000	v_0^0							
0x020	v_1^0							
0x040		v_2^0						
0x060		v_3^0						
0x080		v_4^0						
0x0A0			v_5^0					
0x0C0			v_6^0					
0x0E0			v_7^0					
0x100				v_8^0				
0x120								

Fig. 5. Memory Layout without Vertex Clustering.

0x000	v_0^0							
0x020	v_4^0							
0x040		v_7^0						
0x060			v_1^0					
0x080				v_5^0				
0x0A0				v_6^0				
0x0C0				v_2^0				
0x0E0				v_3^0				
0x100				v_8^0				
0x120								

Fig. 6. Memory Layout with Vertex Clustering.

Image Mapping: Even after vertex clustering, the constrained jump path discussed above still has poor spatial locality. In order to search a path of vertices that map to v_0^1 we need to access the IMG field of each vertex encountered during the search. As a result the search accesses memory locations that are far from the clustered vertices (see the shaded box at the bottom of Figure 6). Our *image mapping* augments the vertex data structure as shown in Figure 4(b) to include the IMG field of each neighbor of the vertex. Thus when finding constrained jump paths or constrained paths the search does not access remote memory locations to determine the pre-image of a neighboring vertex.

Embedded Queue: The next source of poor memory reference pattern is the working queue of BFS that resides in a remote memory region. The interleaving of accesses between the vertex cluster region and the working queue region may

cause *cache thrashing* — entries that will be used later are discarded because of memory conflicts — and reduces the benefits of the free prefetching due to large cache lines. Our *embedded queue* technique stores the information about vertices yet to be visited by BFS within the vertex’s data structures. To implement a BFS embedded queue we augment the vertex data structure with an additional field, the embedded queue pointer (EQP), as shown in Figure 4(c). The EQP field contains a pointer to the last vertex that was added to the working queue. Using this technique entails a modification to the manner in which BFS is performed.

The pattern of vertex visitation in BFS can be viewed as an expanding wave that starts at the initial vertex. If we divide this expansion into phases, in phase 0 we visit the starting vertex s , in phase 1 we visit all the immediate neighbors of s . In phase 2 we visit all the vertices that are two hops away from the starting vertex, and so on. The embedded queue algorithm, shown in Figure 2, uses w to access the linked list formed by the EQP’s of the vertices that are being visited in the current phase. It uses w' to build the linked list of the vertices to be visited in the next phase.

When traversing a list in a given phase of BFS, we use *EQP* to find the next vertex to be visited. In the initialization (steps 1-3) the *EQP* of the starting vertex s is assigned NULL to ensure that the phase 0 will terminate. NULL is also assigned to w' to ensure that the next phase will also terminate. The first vertex of phase 0 is s . The algorithm will terminate when a vertex whose image is I is encountered (step 7).³ Adjacency lists ensure that the accesses in the for loop (step 6) benefit from spatial locality. Vertices that are not in the same image as the starting vertex (step 10) or that have already been visited (step 12) are not included in the working list for the next phase. Spatial locality is promoted because: (1) the comparison between the image of v and the image of the starting vertex s (step 10) accesses data within v (*image mapping*); and (2) accesses to *EQP* (steps 15 and 19) are also within v and w (*embedded queue*). The direction in which the embedded queue is constructed and traversed matters. We build a backward queue in the sense that the newly discovered vertex v is placed at the front of w' , not the rear. The advantage of this traversal direction is that when we finish building the queue, we start to visit vertices in the reverse order in which they were added to the queue. Thus we are likely to visit vertices that we have recently visited and benefit from temporal locality.

4 Experimental Results

We studied embedded queues (Q), vertex clustering (V), and image mapping (I) by writing eight versions of CR: **Baseline**, Q--, -V-, --I, QV-, Q-I, -VI, and QVI (the three characters in the version denote either the presence or the absence of each one of the features). We used three graph types: (1) **2D-Plane**, a $h \times w$ two-dimensional plane, (2) **3D-Cube**, a $h \times w \times d$ three-dimensional cube, and (3) **Airway-Road**, an airline route network where each vertex is an instance of a

³ The algorithm assumes that if the start vertex s is in abstraction level a , then G_{a+1} has an edge between the image of s and the destination image I .

road network graph of the city of Pittsburgh. Multiple instances were generated for each graph type. For **2D-Plane** and **3D-Cube** graphs we varied dimensions while for **Airway-Road** graphs we varied the size of the road network graph. The average vertex degree was 4 in **2D-Plane** graphs, 6 in **3D-Cube** graphs, and between 2.4 and 2.7 in **Airway-Road** graphs. Table 1 contains a summary of the systems utilized in our experiments as well as the compilers used to compile the implementations. For each system we used GCC and the processor vendor's compiler to compile each implementation with **-O0** and **-O3** (all results presented in this paper were obtained with **-O3** and a vendor compiler, except for SGI where we used GCC). All measurements encompassed the computation of 10,000 paths between random pairs of vertices.⁴ Computing each path involved searching for it and reconstructing it into linked list form via *BP* pointer traversal.⁵ To generate abstraction hierarchies we used the STAR method with a radius of 2. We recursed until an abstraction graph with a single vertex was constructed.⁶

We found that combining all three techniques results in the best overall performance improvements over **Baseline**. Although **-VI** occasionally produced better performance gains than **QVI**, we prefer **QVI** because it always improved performance, whereas **-VI** did not. Figure 7 shows the percentage reduction in execution time produced by **QVI** over **Baseline** for all systems and graph instances. The figure shows consistently positive improvements on all systems ranging from 1.0% to 51.2%, highlighting **QVI** being robust to changes in hardware architecture. Is **QVI** also robust to compiler changes? In short the answer is *Yes*. Figure 8 presents the average percentage reduction in execution time produced by **QVI** over **Baseline** using all compilers with **-O0** and **-O3**. In all instances **QVI** produced non-trivial performance gains. In addition, because of the relative similarity of the performance gains obtained with **-O0** and **-O3**, it would seem that the manner in which **QVI** improves performance is orthogonal with respect to compiler driven optimizations.

Hardware event profiles indicate that in the case of **2D-Plane** graphs **QVI** improved data reference locality at the page level while degrading it at the cache line level. The same profiles show **QVI** improving locality at both levels for **3D-Cube** and **Airway-Road** graphs. As an example, we present Figures 9 and 10. The figures show the percentage reductions produced by **QVI** over **Baseline** in the total number of L1 and L2 cache misses and TLB misses on the IBM and INTEL systems. We also observed that queue embedding eliminated code otherwise required to maintain a detached working queue. As a result, **QVI** generally graduated about 10% less instructions than **Baseline**. Our results suggest that the bulk of **QVI** performance gains stem from data reference locality improvements at the page level and to a lesser degree at the cache line level.

When in isolation, the technique of vertex clustering proved to be the most effective, followed by image mapping and queue embedding. Our techniques ap-

⁴ Because every graph is connected our experiments never include dead-end searches.

⁵ Reconstruction of paths took between 1% and 17% of the measured execution times.

⁶ When generating G_{i+1} we sequentially iterated through all vertices in G_i using yet to be classified vertices as starting points for new subgraphs.

Table 1. Systems and compilers used in our experiments (All systems had 1GB of main memory and a UNIX based OS).

System	Processor	Compilers	Caches
SGI	R12K 350Mhz	MIPSPro (7.2.1) GCC (2.7.2)	L1 32KB L2 4MB
IBM	Power3 450Mhz	IBM XLC (6.0) GCC (2.9)	L1 64KB L2 8MB
AMD	XP 1667Mhz	Intel (6.0) GCC (2.96)	L1 64KB L2 256KB
INTEL	P4 2260Mhz	Intel (6.0) GCC (2.96)	L1 8KB L2 256KB

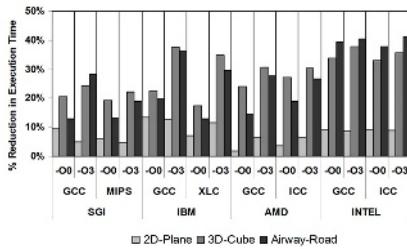


Fig. 8. Average execution time reductions produced by QVI over Baseline using various compilers with -00 and -03.

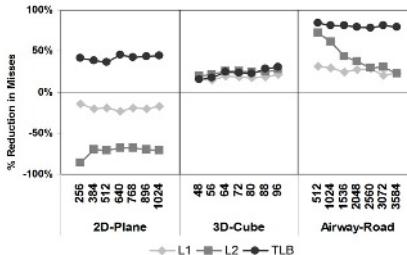


Fig. 10. L1, L2, and TLB miss-total reductions by QVI over Baseline on the INTEL system.

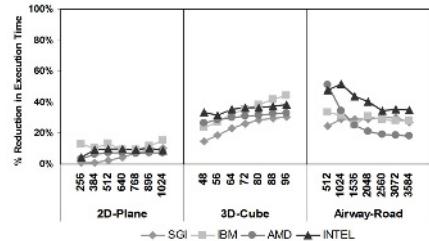


Fig. 7. Execution time reductions produced by QVI over Baseline.

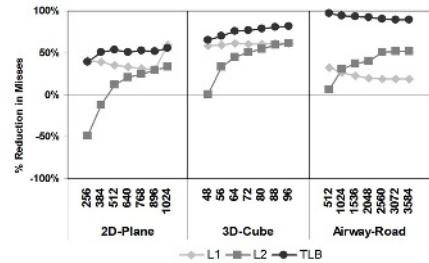


Fig. 9. L1, L2, and TLB miss-total reductions by QVI over Baseline on the IBM system.

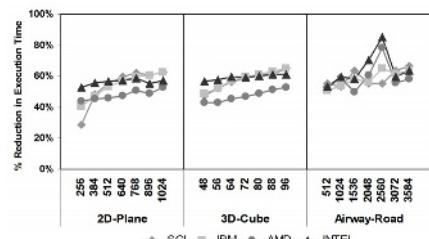


Fig. 11. Execution time reductions produced by QVI over Baseline with scrambled input graph vertex order.

pear to compliment each other since when combined they produced better performance gains than when on their own. Applying image mapping and/or queue embedding increases the memory footprint of the abstraction hierarchy (between 8.7% and 65% compared to Baseline). However, the increase often translated into a decrease in the overall amount of heap memory touched during search, especially at page sized granularities. We note that our experiments generated

virtually no page faults, which is indicative of memory traffic being exclusive to internal memory.

In the absence of vertex clustering our implementations place vertices in memory in G_0 in the order in which they appear in the input graph. Vertices in our input graphs are ordered in a manner that reflects the topology of the graph. For instance, vertices in our 2D-Plane input graphs are ordered as left-to-right rows stacked in a top-down fashion. When we scrambled our orderings we observed dramatic increases in performance gains. Figure 11 shows QVI producing improvements ranging from 28.4% to 85.2% under these circumstances. Scrambling reduced the likelihood of vertices in the same pre-image appearing in close proximity of each other in memory, thus making vertex clustering and image mapping more viable. Although such a scenario may seem artificial, consider a graph representing a national road system, where vertices correspond to cities and junctions. Here, it is possible for vertices to end-up in memory in an order based on an alphabetical sort of their labels (ie. city names), thereby having a similar effect on pre-image data proximity that vertex order scrambling had on our graphs.

5 Related Work

Edelkamp and Schrödl address the problem of thrashing of pages at the virtual memory level [7]. They apply their localized A* to improve the page level locality of a route planning system. In the field of *external memory* algorithms we find various techniques aimed at improving the I/O efficiency of graph search [12, 13]. Typically these methods use techniques akin to vertex clustering (grouping) and image mapping (data redundancy). For instance, blocking is used to minimize the number of page faults incurred during the traversal of paths in planar graphs. Variants of vertex grouping are also used to increase the performance of sparse matrix multiplication [14]. Graph partitioning, needed for abstraction generation, is a well studied problem [5,8,11]. Better partitioning could yield improvements in page level locality.

The Artificial Intelligence community focuses on reducing the search space (for example, [16]), which can produce improvements of orders of magnitude. The gains obtained with the data structure transformation oriented techniques presented in this paper are orthogonal to the search space reduction, and the two techniques can be easily combined. They are also in line with performance improvements obtained through compiler transformations that improve data placement [2,3]. Notice however that the automated techniques found in contemporary compilers are quite inept at improving data locality with respect to graph search in general. Even with the ongoing development of profile oriented compilation we foresee this to continue to be the case because techniques such as our embedded queue and image mapping methods not only require a change in the manner data is layed out in memory but also require changes to the search algorithms themselves.

6 Conclusion

Research in the Artificial Intelligence and computer game communities has produced algorithms to quickly find short paths in very large sparse graphs. However, the effects of temporal and spatial locality in the implementation of these algorithms has been mostly overlooked. This paper demonstrates that three simple data structure transformation oriented techniques can consistently improve the performance of CR pathfinding for sparse graphs. In our experiments these techniques improved data reference locality resulting in performance gains ranging from 1.0% to 51.2%. In addition, these techniques appear to be orthogonal to compiler optimizations and robust with respect to hardware architecture.

References

1. Personal correspondence with David C. Pottinger of Ensemble Studios.
2. B. Calder, K. Chandra, S. John, and T. Austin. Cache-conscious data placement. In *In Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, 1998.
3. Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1999.
4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter Chapter 24, Dijkstra’s algorithm, pages 598–599. MIT Press, September 2001.
5. Josep Díaz, Jordi Petit, and Maria Serna. A survey of graph layout problems. *ACM Computing Surveys (CSUR)*, 34(3):313–356, 2002.
6. Mark DeLoura. *Game Programming Gems Vol 1*. Charles River Media, 2000.
7. S. Edelkamp and S. Schrödl. Localizing A*. In *17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 885–890, 2000.
8. Norman E. Gibbs, Jr. William G. Poole, and Paul K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software (TOMS)*, 2(4):322–330, 1976.
9. R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1-2):321–361, 1996.
10. R.C. Holte, C. Drummond, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Searching with abstractions: A unifying framework and new high-performance algorithm. In *10th Canadian Conf. on Artificial Intelligence (AI’94)*, pages 263–270, 1994.
11. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in vlsi domain. In *Proc. 34th annual conference on Design automation conference*, pages 526–529. ACM Press, 1997.
12. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.
13. M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

14. Ali Pinar and Michael T. Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 30. ACM Press, 1999.
15. David C. Pottinger. Terrain analysis in realtime strategy games. In *Game Developers Conference Proceedings*, 2000.
16. Stuart J. Russell. Efficient memory-bounded search methods. In *10th European Conference on Artificial Intelligence Proceedings (ECAI 92)*, pages 1–5, 3–7 August 1992.

Improving Performance Analysis Using Resource Management Information*

Tiago C. Ferreto¹ and César A.F. De Rose²

¹ CPAD-PUCRS/HP, Porto Alegre, Brazil

ferreto@cpad.pucrs.br

² Catholic University of Rio Grande do Sul,
Computer Science Department, Porto Alegre, Brazil

derose@inf.pucrs.br

Abstract. In this paper we present *Clane*, a performance analysis environment for clusters. It uses a novel approach combining resource management and monitoring data to provide reliable information for cluster users and administrators in application and system performance analysis. *Clane* uses the XML standard to represent its internal information base, providing more flexibility in data manipulation and simplicity to extend the environment with other analysis tools. The environment is composed by an *Information Server*, which stores performance information provided by the monitoring system and user events dispatched through the resource management system, and an *Analysis Tool* to present the combined information and events using statistics, graphs and diagrams. It also enables performance comparisons among distinct executions of the same application in the cluster.

1 Introduction

Resource management and monitoring systems are among the main basic services available in a cluster environment. They are usually implemented as distinct tools and used for different purposes. Resource management is usually directly related to the cluster user allowing him to get access to a group of cluster nodes, whereas the monitoring system is designed for the cluster administrator to provide a view on how well the machine resources are being exploited by the end users. Some resource management systems provide also a basic monitoring service, but the information is usually presented in real time and without any relation to the applications that are running on each node. This view is well suited to the cluster administrator to evaluate the global performance of the machine, but not to the end user to evaluate the performance of his application.

Believing that a stronger integration of resource management and monitoring tools could lead to a better system and application analysis we developed *Clane*, an analysis environment for clusters. *Clane* combines resource management events and monitoring information to provide performance information related to the applications that are executed in a cluster. The acquired information is formatted to better suit the needs of system administrators as well as end users. This approach differs from system monitoring that

* This research was done in cooperation with HP-Brazil

is provided by some resource management systems because the information is stored for future analysis (post-mortem monitoring), and is related to the applications running in the cluster, enabling comparisons among several runs of an user application, evaluation of the system performance impact of a change in the machine configuration, and other useful performance analysis.

2 Resource Management and Monitoring

Resource management systems are responsible for distributing applications among computers to maximize their throughput. It also enables the effective and efficient utilization of the available resources [1].

Some of the main functionalities that should be provided by resource management systems for clusters are: resources allocation and freeing, application execution using the allocated resources through interactive or batch jobs, and allocation queue management using an allocation policy. Other functionalities that can be provided by the resource management system are: load balancing, process migration, support to many allocation policies, allocation queue management based on priorities, suspension and restart of applications, etc.

There are several cluster resource management systems freely available. Some of the most used resource management systems are OpenPBS [2] and CCS [3].

OpenPBS is a flexible batch queuing and workload management system originally developed by Veridian Systems for NASA. It operates on networked, multi-platform Unix environments, including heterogeneous clusters of workstations, supercomputers, and massively parallel systems. The main goal of the PBS system is to provide additional controls over initiating or scheduling execution of batch jobs; and to allow routing of those jobs between different hosts. The batch system allows a site to define what types of resources and how many resources can be used by different jobs. Some of the main features of OpenPBS are configurable job priority, transparent job scheduling, easy integration with other applications through a comprehensive API, and automated load-leveling based on HW configuration, resource availability and keyboard activity. OpenPBS supports real time system monitoring at application level through an internal tool called *xpbsmon*.

CCS is a resource management system developed at the PC² (Paderborn Center for Parallel Computing) at the University of Paderborn, Germany. It has been designed for the user-friendly access and system administration of parallel high-performance computers and clusters. It supports a large number of software and hardware platforms and provides a homogeneous, vendor independent user interface. For system administrators, CCS provides mechanisms for specifying, organizing and managing various high-performance systems that are operated in a computing service center. Robustness, portability, extensibility, and the efficient support of space sharing systems, have been among the most important design criteria. CCS provides system and application monitoring integrated to its environment through the *ccsMon* and *SPROF* tools.

Monitoring systems are designed to collect system performance parameters such as node's CPU utilization, memory usage, I/O and interrupts rate, and present them in a form that can be easily understood by the system administrator [4]. This service is important

for the stable operation of large clusters because it allows the system administrator to spot potential problems earlier. Moreover, other parts of the systems software can also benefit from the information provided. For example, the information can be used to modify the task scheduling, in order to improve load balancing. Some of the main characteristics that should be provided by the monitoring tools are:

Low intrusion. One of the main issues considered in the project and implementation of monitoring systems is the intrusion generated by the monitoring system. To obtain the highest possible performance in a cluster, the parallel application should be able to get all of the available processing power. However, the monitoring system has some processing and communication costs and it will compete with the running application. Therefore, to enable high performance execution in the presence of monitoring, the monitoring tool should have low intrusion, producing minimal interference.

High configurable. Due to the diversity of cluster topologies, and the increase of heterogeneous clusters, monitoring systems need to be high configurable. Some of the features that should be provided by monitoring systems are: adjustable monitoring frequency, selection of monitoring metrics, selection of resources to be monitored, online and offline monitoring, etc.

Extensibility. Extensibility is a critical issue for any monitoring system survival. The extension of the system to support new metrics is really necessary to enable information gathering of new hardware technologies (e.g. Myrinet [5] and SCI [6] network boards). This feature is also useful to provide interconnection between the monitoring system and other systems, in which the last uses the information obtained by the monitoring system to execute some task (e.g. a load balancing system), or just to present it (e.g. a graphical tool).

There are several cluster monitoring systems freely available. Some of the most used are PCP [7] and Ganglia [8].

Performance Co-Pilot (PCP) was developed by SGI as a commercial monitoring system for IRIX. In February 2000, SGI released the PCP infrastructure as Open Source software. Their goal was to provide a readily available, feature-rich and extensible framework for managing performance in large Open Source systems. PCP is composed of a framework and services to support system-level performance monitoring and performance management. It provides a range of services that may be used to monitor and manage system performance. Some of the main features provided by PCP include public interfaces and extensible frameworks at all levels, integrated logging (real time) and retrospective (historical) data analysis support, single API for all performance data, and coverage of a large range of activity areas (e.g. CPU, memory, Cisco routers, Web servers).

Ganglia is a monitoring environment initially developed at the University of California, Berkeley Computer Science Division as a way to link clusters across the Berkeley campus together in a logical way. Since it was developed at a university, it is completely open-source and has no proprietary components. All data is exchanged in well-defined XML and XDR to ensure maximum extensibility and portability. Ganglia provides a complete real time monitoring and execution environment that is in use by hundreds of

universities, private and government laboratories and commercial cluster implementations around the world.

3 Clane: An Analysis Environment for Clusters

Clane provides a complete analysis environment with generic interfaces and a graphical analysis tool, which presents information based on cluster resources allocation, applications execution, and performance information acquired through monitoring. The environment presents a novel approach combining system monitoring information with resource allocation and execution events, providing a different analysis view of the cluster.

3.1 Motivation

Resource management and monitoring systems are most commonly used in clusters as two separated systems. This fact is clear by the increasing number of specialized cluster monitoring systems, and the few number of resource management systems that provide also system monitoring. Some resource management systems try to integrate a specialized monitoring system in its architecture to enable other functionalities (e.g. load balancing), but this integration is usually not easy and just the basic functionalities of the monitoring system are used. In most of the cases only online monitoring is used, the metrics monitored or the monitoring frequency can't be changed, and the information is gathered from all cluster nodes.

In the other hand specialized cluster monitoring systems provide online and offline information with a collection of metrics and configurable capture frequencies. These systems are really useful to supply a global view of the system during the time. However, the acquired information is not related to the applications being executed in the cluster.

We believe that the correlation of monitoring information and allocation/execution events, which are controlled by the resource management, are useful for cluster administrators and users. The information resultant of this correlation can be divided for each user, and internally for each application executed in the cluster, as presented in Figure 1. Figure 1(a) presents monitoring information for a group of nodes. Each rectangle drawn represents an individual application execution. Figure 1(b) presents the allocation and execution table for the monitored period. The rectangles drawn around the user names are correlated to the rectangles drawn in (a). This information can be used in application and cluster utilization analysis. Some of the benefits achieved by this approach are: it optimizes the utilization of cluster resources by the applications, detects possible bottlenecks in the cluster that can be minimized, and enables the comparison between several executions of an application in the cluster.

3.2 Architecture

The architecture of *Clane* is presented in Figure 2. The environment is highly connected to the resource management and monitoring systems. This connection is established using functions provided by the *Information Storage Interface* which gathers the information

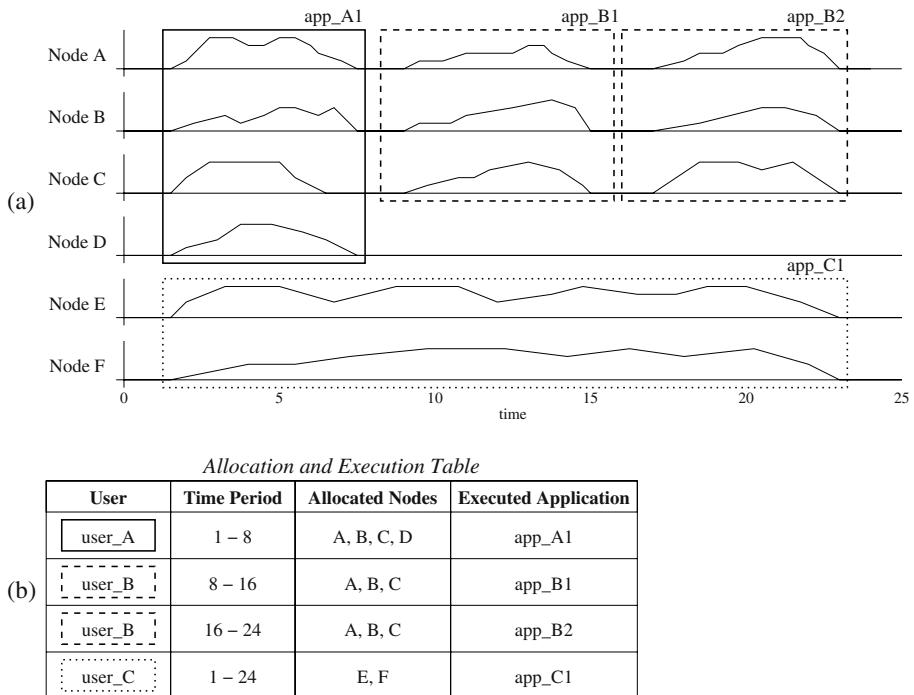


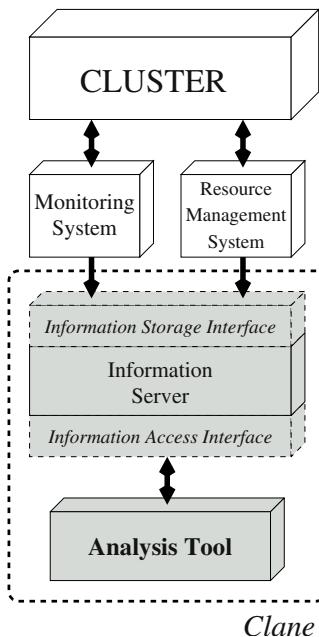
Fig. 1. Correlation between resource management events and monitoring information. (a) Monitoring information. (b) Allocation and execution table.

and forwards it to be stored by the *Information Server*. The information is stored in XML format providing simplicity and flexibility in data manipulation. The *Analysis Tool* access this information using the functions available in the *Information Access Interface*. These functions allow the selection of fragments from the total amount of stored information. The result of this selection is stored in an XML file. After the selection, the *Analysis Tool* is able to process the resulting XML file and present the information. The *Analysis Tool* provides different formats to present the information, facilitating the analysis process.

3.3 Implementation

Information Server. The *Information Server* is responsible for storing performance information, and resources allocation and applications execution events, and to provide access to it. All information is stored in XML format due to its high flexibility for storing and accessing the information and to provide extensibility and portability.

The information is obtained using the resource management and monitoring systems with the functions available in the *Information Storage Interface*. These functions are divided in three groups, one to store information related to the cluster allocation, another to store information about the applications executed during the allocation, and the last one to store system status information.

**Fig. 2.** Clane architecture

Functions *is_initalloc()* and *is_endalloc()* records respectively information about the start and end of an user allocation, such as: allocation identifier, username, nodes allocated, timestamp, etc. Functions *is_initexec()* and *is_endexec()* stores respectively information about the start and end of an application execution, such as: execution identifier, the allocation identifier related to this execution, timestamp, application name and parameters, application returned status, etc. Function *is_monentry()* records information about the each node of the cluster, such as percentage of CPU, memory and swap utilized, percentage of L1 cache misses, etc. These functions are all available in a shared library (written in C) and as individual programs, called *clane_addallocation*, *clane_addexec*, and *clane_addmonentry*.

To store all received information, the *Information Server* generates at least two files, one for information related to resource allocation and applications execution, and other one for monitoring information. The location of these files can be configured in the environment. It is also possible to create an allocation and execution file for each user, minimizing the overhead of multiple accesses to the same file.

The *Information Access Interface* provides functions to extract a subset of the information monitored, and store it in an XML file. It provides a shared library, with specific functions to filter the XML files, and also a program, called *clane_query* which receives as parameters a group of filters used to provide to the user the subset of information desired. The filters can select information based on a specific user, application and period of time.

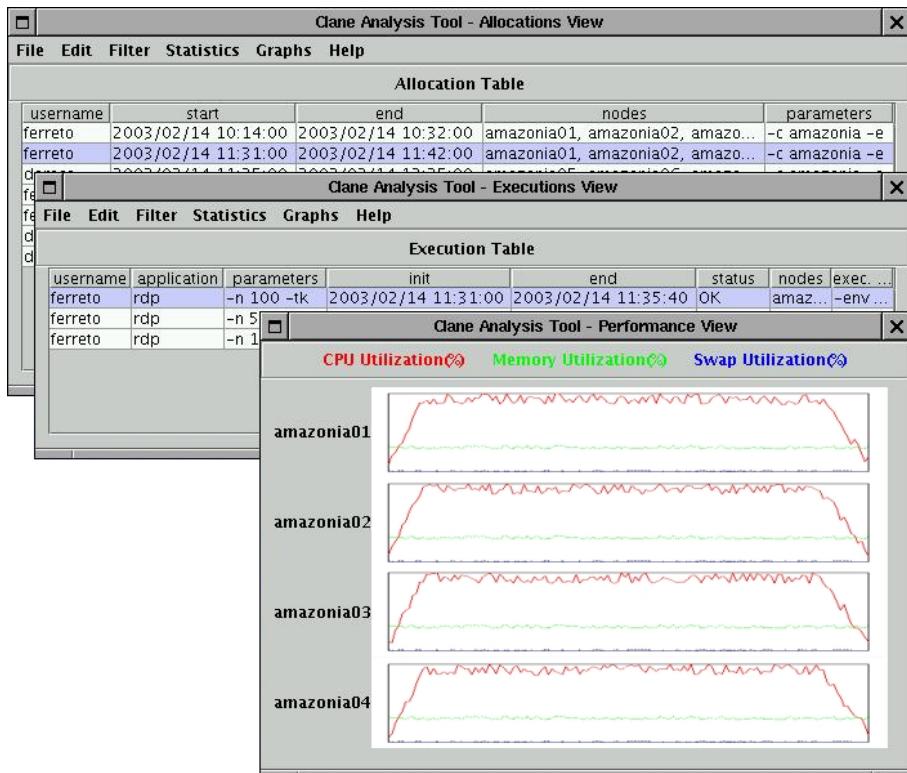


Fig. 3. Clane Analysis Tool

Analysis Tool. The *Analysis Tool*, presented in Figure 3 is written in Java due to its design and programming simplicity, code portability, and XML support. Each time the *Analysis Tool* is executed, the user needs to specify the filters that should be applied to the information stored by the *Information Server*. The tool uses these filters to generate the resulting XML file with the information selected using the functions of the shared library provided by the *Information Access Interface*. After this, the tool parses the XML file, presents the collection of allocations selected, and keeps waiting for user interaction. The main features of the *Analysis Tool* include statistics for one or more allocations and application executions, graphs representing resource utilization for allocations and executions, and textual or graphical comparison between a collection application executions.

Figure 3 presents a typical view of the analysis tool in use. The window in the back, the *Allocations Window*, shows the allocations selected initially. The user selects one of these allocations and requests for the executions realized during this allocation, which appear in the *Executions Window*. After this, one of these executions is selected, and the monitoring information is displayed based on the monitored metrics, in this case,

percentage of CPU, memory and swap utilization. *Clane* also provides some useful statistics to the user, based on the information monitored.

The statistics are divided in allocation statistics and execution statistics. The main allocation statistics provided include average and standard deviation for a group of selected allocations based on the allocation time requested, number of applications executed, and number of nodes used. The main execution statistics provided include average and standard deviation for a group of selected executions based on the execution time, and for a group of performance metric.

The comparison between allocations or executions is based on the information and statistics generated for these events. The comparison can be presented textually or graphically. The *Analysis Tool* can be used by the cluster administrator or user. The only difference between the functionality of the tool for each one is based on the amount of information that is available for analysis. The administrator can view information of all users, whereas the user can just see information of its own allocations and executions.

4 Configuration

Clane was configured at our lab, the Research Center of High Performance Computing (CPAD-PUCRS/HP) [9]. The environment was connected to Crono [10] and RVision [11], the resource management system and monitoring system used in the center respectively. These systems where developed in our lab and are being used in all clusters. The inclusion of the functions and programs provided by the *Information Storage Interface* to Crono and RVision was relatively simple, due to the high configurability and flexibility presented in these systems.

Crono executes a script for each started allocation (*mpreps*) and another one for each finished allocation (*mpostps*). These scripts are used to set necessary configurations before user interaction. The program to log allocation events, *clane.addallocation*, was inserted in these scripts to log information related to resources allocation.

To simplify the utilization of the system, Crono provides an unique application dispatcher, called *crrun*, which encapsulates all execution environments available to the users. This script receives the execution parameters and forwards it to the real execution environment which is responsible for the application execution. The program *clane.addexecution* was included in this script before and after calling the execution environment to log the information related to the application execution.

The connection between RVision and Clane was implemented with a special RVision monitoring client. This client calls the function *is_monentry()*, provided by the *Information Storage Interface*. This monitoring client was implemented as a daemon, due to its unique functionality of gathering information and sending it to the *Information Server*, instead of showing it.

The monitoring client was implemented with an internal area of memory, which is used to buffer the information received from the core of the monitoring system (*RVCores* module). This method was used to decrease the number of file openings and writings, therefore, the information is written periodically (each 30 seconds) in larger blocks.

RVision divides the information gathering in monitoring sessions. Each monitoring session is responsible for capturing a defined set of information from a group of nodes

in a given periodicity. Therefore, each allocation starts a new monitoring session for the group of nodes allocated. The same approach used to log the allocation starting and ending events was used to start a new monitoring session and finishing it. The calls to start the monitoring client and to finish it were included respectively in the *mpreps* and *mpospts* scripts provided by Crono.

The metrics defined to be monitored are percentage of CPU utilization, percentage of memory utilization, and percentage of swap utilization. These metrics are monitored each 2 seconds on each node and sent to the *RVCORE* module based on a threshold of 2%, i.e. if the information gathered is 2% higher or smaller than the last one captured, then it is sent to the *RVCORE* module, otherwise, the information is rejected. This method decreases the amount of data transmitted through the network (reducing the intrusion), and decreasing also the total amount of data stored.

The *Clane* environment was configured to generate an allocation and execution record file for each user, instead of a unique file. This approach was chosen based on its better division of the data, simplifying the storing and parsing process. Since all information is date and time based, the network time protocol (NTP) [12] was configured to realize the clock synchronization in all machines, to allow a correct interpretation of the captured information.

4.1 Utilization

The *Analysis Tool* is used by the users mostly to compare the performance obtained for a specific application with different parameters. Since the tool generates graphs showing the performance and also statistics for each metric measured, it is possible to optimize the application based on this information. The tool is used by the administrator to visualize the percentage of resources utilization for each user. It is also used to determine the periods of the highest and lowest utilization.

In addition to the features provided by the *Analysis Tool*, a *Report Generator* was implemented. This tool uses the functions of the *Information Access Interface* to obtain information about user's allocation, executions and obtained performance. It was configured at CPAD to execute the following tasks: generate a weekly report based on users utilization (hours of utilization, number of machines); send weekly to each user an email containing the amount of hours allocated, the number of nodes, and the 10 best and worst executions based on the performance obtained through monitoring; and send monthly to the administrator (via mail) the periods of highest and lowest utilization of the clusters, and the name of the users that have most used the cluster.

5 Conclusions and Future Work

In this paper we presented a new approach to enhance system and application monitoring in cluster architectures based on the utilization of resource management information. Traditional monitoring data like processor and memory usage is combined with information about executed jobs like start and finishing times and number of allocated resources allowing the analysis among several runs of the same application. This is especially useful to generate statistics about resource use and also for application tuning.

The proposed architecture, called *Clane*, has defined interfaces in both ends to allow the adaptation to other monitoring and resource management system that rely on XML for data storage allowing a broader utilization of the system.

To investigate this new concept we implemented a prototype of the proposed architecture and linked together two tools that are already in production in our lab, the resource manager Crono and the monitoring system RVision. The initial results are promising and we are now studying the implementation of data compression and the support for more resource management systems and monitoring systems through the utilization of patches.

We believe that the correlation of monitoring and resource management information will allow administrators and users to better analyze the behavior of their systems and the executed parallel applications opening a new range of opportunities for the enhancement of monitoring tools.

References

1. Buyya, R.: High Performance Cluster Computing: Architectures and Systems. Volume 1. Prentice-Hall (1999)
2. Henderson, R.L., et al.: Portable batch system: Requirement specification. Technical report, NASA Ames Research Center (1995)
3. Keller, A., Reinfeld, A.: Anatomy of a Resource Management System for HPC Clusters. Annual Review of Scalable Computing **3** (2001)
4. Baker, M.: Cluster computing white paper (2000)
5. et al, C.L.S.: Myrinet – a gigabit-per-second local-area network. IEEE Micro **vol. 15** (1995)
6. IEEE standart 1596-1992 New York: IEEE: IEEE Standart for Scalable Coherent Interface (SCI). (1993)
7. Goodwin, M., et al.: Performance Co-Pilot User's and Administrator's Guide. Silicon Graphics, Inc. (1999)
8. Team, G.D.: Ganglia Toolkit. University of California, Berkeley. (2002) <http://ganglia.sourceforge.net/docs/>.
9. Center, C.R.: CPAD-PUCRS/HP. <http://www.cpad.pucrs.br> (2003)
10. Netto, M.A., Rose, C.D.: Crono: A configurable management system for linux clusters. In: Proceedings of the 3rd LCI International Conference on Linux Clusters: The HPC Revolution 2002 (LCI'2002), St. Petersburg, Florida, USA (2002)
11. Ferreto, T., Rose, C.D., DeRose, L.: Rvision: An open and high configurable tool for cluster monitoring. In: Proceedings of the 2nd IEEE/ACM International Symposium on Cluter Computing and the Grid (CCGrid'2002), Berlin, Germany (2002) 75–82
12. Mills, D.L.: Internet time synchronization: the network time protocol. IEEE Trans. Communications (1991) 1482–1493

Optimizing Dynamic Dispatches through Type Invariant Region Analysis*

Mark Leair¹ and Santosh Pande²

¹ STMicroelectronics, Inc., mark.leair@st.com

² Georgia Institute of Technology, santosh@cc.gatech.edu

Abstract. Object oriented languages lead to dynamic dispatches of most method calls due to limitations on static analysis which have significant run-time overheads. In this work, we propose a new optimization to cut this overhead which is based on the following key observation: run-time type testing of a method at *every* call-site is extraneous. The technique makes use of an important observation: most of the times a polymorphic object, although statically unknown in type, is fixed at run-time and never mutates. In such cases, run-time type testing for its methods is only needed when an object is instantiated. The type check is hoisted at the entry point of a region and its results are shared over all dispatches that lie within the region. We implemented this optimization in Vortex, an optimizing object-oriented compiler back-end, and with our framework, we observed an average speed-up of 9.22%. We avoided 91% of the dynamic dispatches with a speed-up of 17.9% in the best case.

1 Introduction

Dynamic message sends pose a major performance bottleneck in Object-Oriented Programming Languages (OOPLs). Current approaches to static type inferencing and type feedback attempt to optimize OOPLs by reducing dynamic message sends the program. The two main limitations of static type inferencing are long analysis times and sometimes limited precision on statically binding call-sites. The two disadvantages of profile-derived type feedback are that it is only as good as its instrumented run and it requires a recompilation. This motivates the need for a new technique based on the following observation: many of the run-time type checks are redundant since the type of the underlying object does not mutate over large program regions; thus, significant potential savings could be gleaned by eliminating redundant run-time type checks.

One of the assumptions made by current optimizations is that a receiver's type may *mutate* or change. This assumption forces run-time checking of the receiver type on all *statically unknown* method calls. The fact is, not all method calls are dynamically bound during run-time. Grove et al. [2] showed that 50% of the dynamic dispatches in their Cecil programs had a single receiver class. This would indicate that dynamically typed objects, although statically unknown in

* This research is partially supported by NSF under grant # CCR 9696129

their type bindings, have a fixed type during a particular region in the program. This fact is critically supported by programming styles that tend to result in *type locality* for program regions. In other words, large program regions contain objects whose types do not change within those regions. In these cases, type tests are only needed at the first use of the object within the region. Run-time type tests or cache probes for subsequent uses of these objects are extraneous. Our work, which is presented in the next section, is motivated by this fact.

In our work, we use simple value analysis to determine program regions where the underlying type although unknown would be guaranteed to be invariant. Therefore, if one performs a type check at the entry point of the region, the type would be known throughout the region allowing elimination of all the type checks at respective call sites and leading to optimized calls with low overheads. This is the essence of our work.

2 Type Invariant Region Analysis

In order to eliminate unneeded type tests, we perform a static analysis to determine the program region where the type (although statically unknown) is invariant once it is dynamically fixed. Methods dispatched on objects whose types are fixed in a particular program region are directly called; saving on the overhead of type tests and/or dynamic dispatches. To maximize the call-sites that are covered by the type invariant region, we take the transitive closure of the object's reaching definitions and use a flow-sensitive analysis for better precision. After calculating this information, we perform one type test and share its results for all call-sites in the region. The end result is fewer type checks and more direct calls in a given program. The general idea of our framework is to detect type invariance due to non-modification of a value. The non-modification of a value is detected through reaching definition analysis (in other words, if a definition reaches a point, it must not be getting killed and thus the original type of the value must remain invariant). Intraprocedurally, we take a transitive closure of the invariant values to detect more invariances. We precede this with an interprocedural analysis to find reaching definitions based on MOD and DEF sets. We iteratively refine invariance interprocedurally. Due to lack of space, detailed algorithm is not presented here.

3 Implementation

We implemented our framework in Vortex [1], an optimizing OOPL back-end with front-ends for Cecil, Modula-3, Java etc. We implemented our phase after class hierarchy and type analysis phases to focus on those methods which could not be tackled by these phases. We first perform in-lining and then resort to TIRA to detect invariant regions.

After inserting our regions into the CFG, we generate the necessary C++ run-time code described below during the *Codegen* phase. Although Vortex can gen-

erate portable C++ or Sparc Assembly, our analysis presently generates portable C++ only.

3.1 Run-Time Support for Type Invariant Regions

We now describe the run-time system that we have developed to support the type invariant regions calculated in the compile-time analysis above. The run-time system is implemented on top of the dynamic dispatch mechanisms provided by Vortex. The run-time system is crucially responsible for maintaining efficacy of our solution through low run-time overheads. We will now discuss these mechanisms and how we augmented them.

Our framework uses a hybrid approach of the polymorphic inline cache (PIC) [3] and inline cache (IC) [4] run-time systems. It uses two ICs and one PIC. The first IC, called the *region IC*, is used to cache the region's receiver type from the last time the program entered the region. The second IC works in conjunction with the PIC. Basically it can be viewed as a 2-level cache. The IC sits on top of the PIC. Whenever the region IC has a cache miss, PICs are used to dispatch the region's call-sites. When the PIC-dispatched method returns, the second IC caches the called method's address. When the program revisits the region and there is a region IC hit, the program directly calls the cached method rather than consult the PIC. Figure 1 illustrates this in a C++ like code.

We must check for predicate objects at run-time which is done in Vortex run-time system. Initially, the compiler generates a PIC for a dynamic message send. At run-time, the PIC checks to see whether the receiver is a predicate or not. If it is, the PIC falls back to a slower lookup mechanism. This slower lookup mechanism performs the predicate dispatching by evaluating its expression. Because our framework builds upon the existing PICs, our framework too falls back to a slower predicate dispatch.

4 Results and Discussion

Table 1 summarizes the benchmarks used in our study. Benchmarking was performed on an unloaded Sparc Ultra 1/200 MHZ workstation with 128MB of RAM. Table 2 reports the number of dispatches that remain in each benchmark. The *% in Regions* column was calculated from table 3. The number of dispatches we were able to include in a region may seem low in comparison, however, a number of the remaining dispatches are a part of a false branch in a run-time type test inserted by an earlier optimization phase (e.g., static type analysis). Our framework concentrates on optimizing dispatches that could not be optimized by earlier phases, therefore we do not touch any dispatches that are a part of a run-time type test.

The *region size* measures the number of call-sites in the region. Table 3 gives the region sizes that were detected by our analysis. As shown, the majority of the regions detected only had one method. This has a lot to do with the object-oriented programming (OOP) paradigm, rather than our framework. Compared

```

OOP X;
/*assume some object X*/
OOP_Map * Xtype = X->type();
if( Xtype != CACHE_REGION_1 ){
    CACHE_REGION_1 = Xtype;
    /*set cache region to new type*/
    Meth_1_0 = 0; /*flush region call-site IC*/
}

/* Pre-amble code for a call-site*/

if(Meth_1_0) {
    (*Meth_1_0) (arguments);
    /*directly call cached method*/
} else{
    (*PIC_Meth_1_0) (arguments);
    /*cache miss, use PIC*/
    Meth_1_0 = PIC_Meth_1_0->cachedMethod;
    /*update 1st. level IC*/
}

```

Fig. 1. Implementation of two level cache method invocation

to imperative languages, the size of methods (procedures) are generally smaller in OOP. However, the fact that we could detect a region with 32 call-sites indicates that large methods do occur in OOP.

Table 4 shows the number of regions used according to region size. In general, the number of regions used is less than the ones detected. This is probably due to the 11,000 line standard library included by each benchmark. The benchmarks, of course, do not use every method in the standard library. Therefore, many regions are not used.

Table 5 presents the different types of method lookups performed in regions with one or more methods. *IC Lookups* measures the number of times a region call-site inline cache (IC) was used. Ideally, we want this number to be high, as its the fastest lookup for our regions. Whenever we get an IC hit, we can jump directly to the cached method. *PIC Lookups* measures the number of times a region call-site polymorphic inline cache was used. If we have an IC miss, we dispatch the call through the PIC and rebind the IC to the cached method. The *Table Lookup* field measures the number of times a region call-site was dispatched through a table. When this happens, the call-site's PIC fell back to a slower table lookup. The following are three cases when a PIC will fall back to a slower lookup, as implemented in the Vortex run-time system:

1. One or more of the receiver objects is a predicate.

Table 1. The benchmarks used in our study.

Cecil Benchmark Applications		
Program	Lines ^a	Description
richards	400	Operating systems simulator
deltablue	650	Incremental constraint solver
instr sched	2,400	Global MIPS instruction scheduler
typechecker	20,000	Typechecker for the old Cecil type system
new-tc	23,500	Typechecker for the new Cecil type system
compiler	50,000	Old version of the Vortex Compiler

^aDoes not include an 11,000 line standard library.

Table 2. Remaining dynamic message sends in benchmarks.

Remaining Dynamic Message Sends		
Program	No. Remain ^b	% In Regions ^c
richards	841	15.00
deltablue	975	13.43
instr sched	3,521	8.70
typechecker	8,452	7.15
new-tc	8,751	7.54
compiler	18,542	6.51

^bIncludes dynamic dispatches that are in the false branch of a run-time type test.

^cPercentage calculated for regions with one or more methods.

2. The PIC has grown too large due to a *megamorphic* call-site¹.
3. There is an insufficient number of available scratch registers for holding the dispatching arguments.

In the first case, we cannot rely on type alone to determine the target method. Therefore, the PIC falls back to a slower *predicate dispatch*. The *Pred Lookup* field measures the number of times a region call-site required a predicate dispatch. The second case occurs when the type is mutating too much. The PIC is typically implemented with a linked-list. Whenever a PIC is consulted, it traverses the linked list to find the target method. This is relatively efficient for small lists. However, for larger lists, a table lookup is as fast as traversing the linked list. The last case has to do with how PICs are implemented. PICs exploit self-modifying machine code. Each node in the PIC's linked-list is a small stub for invoking the method. Therefore, it requires the use of scratch registers when it compares the types of the dispatching arguments. If no scratch registers are available, the PIC must fall back to a table lookup. Once a call-site switches to a table lookup, it never returns to PIC dispatching.

¹ We use the term megamorphic call-site to describe a method selector that has a large number of receiver types (e.g. greater than 10 in the Self implementation [3]).

Table 3. Total regions detected and their sizes

		Region Sizes and No. Regions Detected																		
Prg	Tot	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	32
rich	126	101	21	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
delta	131	104	23	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
inst	306	234	57	3	7	4	0	0	0	0	0	0	0	0	0	0	0	0	1	
type	604	505	83	4	10	0	1	0	0	0	0	0	0	0	0	0	0	0	1	
new	659	554	86	8	10	0	0	0	0	0	0	0	0	0	0	0	0	0	1	
comp	1209	973	159	20	24	10	9	4	2	1	0	1	2	0	3	0	0	0	1	

Table 4. Total regions used and their sizes

		No. Regions Used and Sizes																		
Prg	Tot	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	32
rich	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
delta	8	6	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
inst	24	20	2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	
type	151	127	16	6	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
new	173	141	25	4	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
comp	560	459	61	20	11	3	2	0	1	0	0	1	0	0	2	0	0	0	0	

Because our framework’s call-site caches use the PIC for caching the method, our framework must fall back to table lookups too when the above criteria is met. Fortunately, the fall back is on a *per call-site* basis. Therefore, one call-site fall back does not necessarily mean the entire region falls back to table lookups.

Table 5. Lookups for Region_Size ≥ 1

Lookups for Region_Size ≥ 1				
Program	IC Lookups	PIC Lookups	Table Lookups	Pred Lookups
richards	5	3	723,683	0
deltablue	15,726	13	76,090	0
inst sched	21,713	52	100,653	0
typechecker	361,116	254,218	10,671	4,494
new-tc	265,151	2,371	378,247	4,476
compiler	536,270	5,032	654,098	0

The performance results are given in tables 7 and 8. The *Hit* and *Miss* columns report the number of hits and misses for the region caches. The *Call* column reports the number of region call-sites executed. *Hit %* measures the hit rate of the region IC. The *Time_{base}* field gives the benchmark’s performance time when compiled with all but our optimization phase. The *Time_{tira}* column gives the benchmark’s performance time when compiled with our framework

augmented with all other optimization phases. The *Waste* field is equal to the *Calls* minus the sum of *Hits* and *Misses*. This indicates whether we are wasting CPU cycles on region tests. A zero in this field indicates that we are breaking even, a negative number means we are wasting CPU cycles, and a positive number means we are avoiding dispatches. All but one case has a negative number in this field. One other fact is revealed by table 8. One can see that with the exception of the richards benchmark, the average region type mutation is only 8% for $\text{REGION_SIZE} \geq 1$. This confirms our claim that objects do not always mutate at run-time.

Table 6 reports the lookups for regions with at least two methods. Interestingly, IC lookups dominate. Table 5 saw more Table lookups than IC lookups. As can be seen we get good speed-ups for all the benchmarks in this case. In every benchmark many dispatches were avoided. Deltablue avoided 91% (20/22) of the dispatches due to our optimization. Another interesting aspect to the deltablue case is the zero region cache hits and 17.9% speedup. This is because the region call-sites were in loops. Once the receiver object type was set, it did not mutate inside the loop. The method target in the IC was used for subsequent iterations in the loop. Moreover, the deltablue case demonstrates that avoiding just a small number of dispatches can improve overall performance by a substantial amount. The results in general show that it is profitable to perform the optimization based on type invariance using IC/PIC mechanisms as long as the region size > 2 .

Table 6. Lookups for $\text{Region_Size} \geq 2$

Lookups for $\text{Region_Size} \geq 2$				
Program	IC Lookups	PIC Lookups	Table Lookups	Pred Lookups
richards	N/A	N/A	N/A	N/A
deltablue	18	4	0	0
inst sched	1,796	20	20,852	0
typechecker	259,673	117,681	0	0
new-tc	77,522	97	56,816	1,492
compiler	113,321	430	495,890	0

Table 7. Performance for $\text{Region_Size} \geq 2$

Performance for $\text{Region_Size} \geq 2$								
Prg	Hit	Miss	Call	Hit%	Waste	Time _{base}	Time _{tira}	Speedup%
rich	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
delta	0	2	22	0	20	46 ms	39 ms	17.9
inst	5,556	2,315	22,668	71	14,797	1,125 ms	1,102 ms	2.1
type	171,858	932	377,354	99	204,564	11 s	10 s	10.0
new	70,282	414	135,927	99	65,231	9 s	8 s	12.5
comp	233,307	39,599	609,641	85	336,735	29 s	28 s	3.6

Table 8. Performance for Region_Size ≥ 1

Performance for Region_Size ≥ 1								
Prg	Hit	Miss	Call	Hit%	Waste	Time _{base}	Time _{tira}	Speedup%
rich	332,706	390,985	723,691	46	0	125 ms	145 ms	-13.79
delta	93,122	1,678	91,829	98	-2,971	49 ms	50 ms	-2.00
inst	94,681	12,819	122,418	88	14,918	1,125 ms	1,136 ms	-0.97
type	611,944	67,993	630,499	90	-49,438	11 s	12 s	-8.30
new	618,537	68,727	650,245	90	-37,019	9 s	10 s	-10.00
comp	1,292,227	82,483	1,195,400	94	-179,310	29 s	30 s	-3.33

Table 9 summarizes the dispatches that were avoided based on the number of methods in the region. The net result for each region size was calculated by subtracting the sum of the region IC hits and misses from the number of executed calls. As mentioned earlier, all but one case avoided any dispatches when optimizing regions with one or more methods. A negative value indicates a wasted region IC type test. This occurs when program control does not reach a region call-site. If this happens, then the region IC type test is wasted. However, for regions with more than one method, we can potentially share the results of the region type test with other call-sites. Therefore, if program execution does not reach a call-site in a region of size greater than one, it may still reach other region call-sites. Our overhead breaks even as long as program execution reaches one region call-site. If it reaches more than one, we gain in terms of performance.

Table 9. Dispatches Avoided based on Region Size

Dispatches avoided based on Region_Size						
Size	richards	deltablue	inst sched	typechecker	new-tc	compiler
1	0	-2,991	121	-254,002	-102,250	-516,045
2	N/A	20	3,997	195,974	60,472	149,956
3	N/A	N/A	1,644	4,738	710	70,880
4	N/A	N/A	N/A	3,852	4,049	54,056
5	N/A	N/A	9,156	N/A	N/A	6,567
6	N/A	N/A	N/A	N/A	N/A	35,729
7	N/A	N/A	N/A	N/A	N/A	0
8	N/A	N/A	N/A	N/A	N/A	8,145
9	N/A	N/A	N/A	N/A	N/A	0
10	N/A	N/A	N/A	N/A	N/A	0
11	N/A	N/A	N/A	N/A	N/A	10,772
12	N/A	N/A	N/A	N/A	N/A	0
13	N/A	N/A	N/A	N/A	N/A	0
14	N/A	N/A	N/A	N/A	N/A	630

Table 10 presents the analysis time² required for calculating the type invariant regions. The *du Chain* field presents the time it took to calculate the du

² Measured on an unloaded 168 Mhz UltraSparc with 128MB of RAM.

chains for each benchmark. The *TIRA phase* is the time it took to detect the type invariant regions. The third column is the time it took to analyze the program with all but our analysis. The good news is that *all* the benchmarks could be analyzed; unlike some interprocedural analyses. The du-chain analysis tends to dominate the analysis time. The main reason behind it is that du-chain analysis is performed in an interprocedural sense and is interleaved with invariance detection which is invoked iteratively.

Table 10. Analysis time

Analysis Time			
Program	du Chain Calc _{seconds}	TIRA phase _{seconds}	Standard Time _{seconds}
richards	823	69	770
deltablue	1,427	89	842
instr sched	2,948	195	1,977
typechecker	6,035	517	5,175
new-tc	6,599	550	5,306
compiler	14,380	1,081	10,820

The executable code size is presented in table 11. Because our type invariant regions require an additional run-time support, the code size does increase. Fortunately, the increase in code size is reasonably low when considering regions with at least two methods.

Table 11. Executable Code Size

Executable code size					
Prg	Base	Size $\geq 1_{Bytes}$	% increase _{Size ≥ 1}	Size $\geq 2_{Bytes}$	% increase _{Size ≥ 2}
rich	1,094,032	1,173,712	7.3	N/A	N/A
delta	1,121,356	1,204,520	7.4	1,150,364	2.5
instr	2,296,840	2,509,284	9.3	2,384,352	3.8
type	5,952,956	6,318,920	6.1	6,047,304	1.6
new	6,240,852	6,636,584	6.3	6,340,412	1.6
comp	10,555,804	11,381,432	7.8	10,875,364	3.0

5 Conclusion

In this work, we have developed a framework that augments existing intraprocedural and interprocedural type analyses for OOPLs. Because our analysis works off of reaching definition values rather than types, our analysis can optimize message sends in which no available type information is present. This is especially useful in the separate compilation of library routines where the types of the incoming formal parameters are statically unknown but fixed at run-time. Moreover, any message send that could not be optimized by an earlier analysis

could potentially be optimized by our framework. In the best case, our framework could optimize 15% of the remaining dynamic messages.

As demonstrated in section 4, our framework performs best for regions with two or more messages to give good speed-ups. Moreover, messages found on the false branch of a run-time type test are generally not executed due to correct type inferencing by earlier optimization phases. Therefore, we do not lose potential benefits when we consider our simpler region heuristic.

The type invariant region analysis is the second contribution of this work. The *Hit %* field in tables 5 and 6 measures the rate at which the region's type definition(s) mutate at the region entry point. The hits were quite high for most of the benchmarks. This further validates our claim that types generally remain invariant in a region of the program and thus, there is a good potential for carrying out other optimizations based on type invariance. The typechecker benchmarks in table 6 saw a 99% hit rate. This probably explains why they performed well (10% and 12.5% speedup).

Acknowledgments. We would like to express our sincere thanks to the Cecil group at the University of Washington for making the Vortex compiler available for our work.

References

1. Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers, *Vortex: An Optimizing Compiler for Object-Oriented Languages*. In OOPSLA '96, 11th Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Jose, CA, October 1996.
2. David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers, *Profile-Guided Receiver Class Prediction*. OOPSLA '95, Austin, TX, October 1995.
3. Urs Hözle, Craig Chambers, and David Ungar, *Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches*. In ECOOP '91 Conference Proceedings, Geneva, 1991. Published as Springer Verlag Lecture Notes in Computer Science 512, Springer Verlag, Berlin, 1991.
4. L. Peter Deutsch and Alan Schiffman, *Efficient Implementation of the Smalltalk-80 System*. In Proceedings of the 11th. Symposium on the Principles of Programming Languages, Salt Lake City, UT, 1984

Thread Migration/Checkpointing for Type-Unsafe C Programs*

Hai Jiang and Vipin Chaudhary

Institute for Scientific Computing
Wayne State University
Detroit, MI 48202 USA
`{hai, vipin}@wayne.edu`

Abstract. Thread migration/checkpointing is becoming indispensable for load balancing and fault tolerance in high performance computing applications, and its success depends on the migration/checkpointing-safety, which concerns the construction of an accurate computation state. This safety is hard to guard in some languages, such as the C, whose type-unsafe features may cause the dynamic types associated with memory segments to be totally different from the static types declared in programs. To overcome this, we propose a novel intra-procedural, flow-insensitive, and context-insensitive pointer inference algorithm to systematically detect and recover unsafe factors. The proposed scheme conducts a thorough static analysis at compile time and inserts a minimized set of primitives to reduce the overhead of possible dynamic checks at run-time. Consequently, most unsafe features will be removed. Programmers can then code in any style, and nearly every program will be qualified for migration/checkpointing.

1 Introduction

Thread migration and checkpointing has become an imperative means in redistributing computation [1] and fault-tolerance for parallel computing. All such schemes need to fetch (or construct), transfer, and re-install the computation state. The correct execution of this three-step procedure will guarantee correct results upon resuming work on a new machine. Since portable application-level schemes build the state from the source code, certain type unsafe features in C programs, such as pointer casting, structure casting with pointers, and pointer arithmetic, may cause the dynamic types associated with memory segments to be different from the static types declared in programs. In C, it is impossible to acquire the actual dynamic data types from the memory blocks themselves. This might prevent migration/checkpointing systems from getting the actual state and may result in migration/checkpointing failure. To avoid this, most schemes [3,4] rely on type-safe programming styles.

* This research was supported in part by NSF IGERT grant 9987598, NSF MRI grant 9977815, and NSF ITR grant 0081696.

We have designed and implemented a thread migration/checkpointing package, *MigThread* [5], to migrate threads safely for adaptive parallel computing. As an application-level approach, *MigThread* demonstrates good portability and scalability. Furthermore, we have identified pointer operations and library calls as the factors for unsafe migration/checkpointing [6]. To provide a reliable solution and cover all possible cases, we extend the previous work by proposing a novel intra-procedural, flow-insensitive, and context-insensitive pointer inference algorithm to detect unsafe pointer operations, and then systematically trace their propagation. To reduce the complexity, this algorithm focuses on C functions themselves, ignoring all flow control instruments and contexts. We argue that with the dynamic characteristic of programs, it is not always possible to predict the actual execution and function call path. Permutating all possibilities is intractable. Therefore, a heuristic algorithm is essential.

At compile time, a static analysis module scans source code, identifies unsafe operations, and calculates unsafe pointer casting closure. A complete, but minimized closure would insert less primitives into the transformed source code. On the other hand, at run-time, a dynamic check and update module is activated through those inserted primitives to capture the affected events. At each migration/checkpointing point the precise thread state can be constructed, or a warning message is issued to users to abort any un-recoverable unsafe operations, such as illegal library calls. With this pointer inference algorithm, *MigThread* becomes practical and applicable for more real applications.

The remainder of this paper is organized as follows: Section 2 introduces the thread migration/checkpointing package *MigThread*. Section 3 discusses pointer operations in C. In Section 4, we describe the pointer inference algorithm in detail. Some experimental results and microbenchmarks are shown in Section 5. Section 6 gives an overview of related work. Our conclusions and continuing work are presented in Section 7.

2 Thread Migration/Checkpointing

Migration/checkpointing schemes can be classified by the levels at which they are applied. They fall into three categories: kernel-level, user-level and application-level [1]. Although only application-level ones support portability across heterogeneous platforms, they suffer from transparency and complexity drawbacks.

2.1 *MigThread*

MigThread is an application-level thread migration/checkpointing package [5], designed to take advantage of the portability and scalability in heterogeneous distributed computing environments. *MigThread* consists of two parts: a preprocessor and a run-time support module.

The preprocessor is designed to transform user's source code into a format from which the run-time support module can construct the thread state efficiently. Its power can improve the "weak transparency" drawback of application-level schemes. The thread state is moved from its original location (libraries or

kernels) and abstracted up to the language level. Therefore, the physical thread state is transformed into a logical form to achieve platform-independence. All related information with regards to stack variables, function parameters, program counters, and dynamically allocated memory regions, is collected into certain pre-defined data structures. *MigThread* also supports user-level memory management. Therefore, both thread stacks and heaps are moved out to the user space and handled by *MigThread* directly. Since the address space of a thread could be different on source and destination machines, pointer values may be invalid after migration/checkpointing. It is the preprocessor's responsibility to identify and mark pointers at the language level so that they can easily be traced and updated later.

The run-time support module constructs, transfers, and restores thread state dynamically. Since the preprocessor has collected enough information, thread state is constructed promptly [5,6]. Internally, thread stacks and heaps are rebuilt, while pointers are updated. In addition, *MigThread* takes care of all heterogeneity issues, such as byte ordering, data alignment, and data/pointer resizing.

2.2 Migration/Checkpointing-Unsafe Factors

Migration/checkpointing-safety concerns precise state construction for successful migration/checkpointing. Since the thread state consists of thread stacks and heaps, heterogeneous migration/checkpointing schemes need to interpret all memory segments precisely. They usually have difficulties in dealing with programs written in type-unsafe languages, such as C [3], where memory blocks' dynamic types could be totally different from the types declared in the program.

Type-safe languages can avoid such type uncertainty. That is, static types declared within a program will be the same as the dynamic types at run-time. No type casting is allowed, and no type change occurs once programs start running. Thus, migration/checkpointing schemes can interpret contents of memory segments accurately and build thread states correctly. But “type-safe languages” restriction is too conservative since many programs written in such languages might be safe for migration/checkpointing. Most schemes rely on type-safe programming styles [3,4]. It is impractical to rely on programmers to identify unsafe language features and avoid using them. Major unsafe uses come from pointers and their operations. If a pointer is cast into an integral-type variable before migration/checkpointing and cast back to a pointer variable later, a scheme might fail to identify the real data type in the memory block statically declared as an integral type, and miss the pointer updating procedure during the migration/checkpointing. Then, subsequent use of the pointer with invalid values can lead to errors or incorrect results. Therefore, events generating hidden pointers such as pointer casting are the actions we must forbid.

Another unsafe factor is third-party library calls which may leave undetectable memory blocks and pointers out of the control scope. Without precise understanding, migration/checkpointing schemes will fail to determine their existence. *MigThread* issues compile time warnings of the potential risks.

Table 1. Assignments in Normal Form

Assignment	Explanation
$x = y$	Direct Copy Assignment
$x = \&y$	Direct Address-of Assignment
$x = *y$	Direct Load Assignment
$*x = y$	Indirect Copy Assignment
$*x = \&y$	Indirect Address-of Assignment
$*x = *y$	Indirect Load Assignment

3 Pointer Operations in C

From the programming language’s point of view, only the unsafe pointer casting is of concern for migration/checkpointing-safety. *MigThread* attempts to trace all pointer casting cases and detect potential pointers in any memory location.

3.1 Data Updating Operations

When variables are initially declared, they refer to memory locations directly or indirectly. Later on, their memory contents can be modified to hold data with other types. According to the C syntax, memory contents can be changed by increment and decrement statements, assignment statements, and library calls. Since the increment and decrement statements only change the data values instead of the types, type casting can only come from the other two manners.

Assignment statements propagate the type obligation for the left-hand side to the right-hand side [7]. Combined with type casting, they are the major mechanism to hide pointers in non-pointer type memory locations. To simplify the presentation of the algorithm, normalized assignment statements are listed in Table 1. *MigThread* scans a complicated assignment and breaks it into several basic items without introducing actual temporary variables as in other analyses [2,7]. Table 1 only illustrates the top-level address-of operator “ $\&$ ” and pointer dereference operator “ $*$.” Both x and y may contain more of such operators inside, and type casting qualifier “ (τ_{opt}) ” might be inserted ahead. Without type casting, each memory location will use its default type declared within the program. Otherwise, pointers can be placed in non-pointer type locations by a special pointer casting.

Another way to change memory contents is using library calls, such as **memcpy()**, which directly copy memory blocks from one place to another in memory space. When they contain pointer fields, unsafe casting should be screened out.

3.2 Pointer Casting

Data in memory can be referenced by the following:

- *Named Locations*: Variables in stacks referring to unique memory locations
- *Unnamed Locations*: Fields within dynamically allocated memory blocks in thread heaps do not have static names

- *Pointer Dereference*: For pointer p , $*p$ indicates the content of the memory location that p is pointing to

Pointer casting can be done directly or indirectly, and explicitly or implicitly. For the normalized assignment statements in Table 1, if pointers are sent into *named locations*, we say it happens *directly*. Otherwise, through *unnamed locations* or *pointer dereference*, the casting is defined as *indirectly*. The direct casting of non-scalar types like structures is not permitted in ANSI C [8]. However, we can always achieve this effect indirectly by *pointer dereferencing*. Structures are flattened down into groups of scalar type members which are cast one-by-one. Pointer casting might take place here if some members are pointers. For such cases, we say it happens *implicitly*. Otherwise it happens *explicitly*.

Sometimes type casting might happen in locations without valid static types. They can be created by `malloc()` without a real type association. Another case might be the misalignment in data copy. Some addresses might not point to the beginning of scalar type data after certain pointer arithmetic operations. Such “*notype*” cases should be recorded for future interpretation.

4 Pointer Inference System

Compile-time analysis techniques, such as point-to and alias analyses [9], can be classified by flow- and context-sensitivity. Flow-sensitive analysis takes into account the order in which statements are executed whereas context-sensitive analysis takes into account the fact that a function must return to the site of the most recent call, and produce an analysis result for each calling context of each function [2]. Flow and context sensitive approaches generally provide more precise results, but can also be more costly in terms of time and/or space.

In this paper, the proposed pointer inference system for hidden/unhidden pointer detection has the following features:

Intra-procedure : Analysis works within the function’s scope. It does not consider the ripple effect of propagating unsafe pointer casting among functions.

Flow-insensitivity : Control flow constructs are ignored. We argue that it is impossible to predict the exact execution order because of the potential dynamic inputs for the program. Conservative solutions might be safer for our case (pointer casting).

Context-insensitivity : We do not distinguish contexts to simplify our algorithm. For larger programs the exponential cost of context-sensitive algorithms quickly becomes prohibitive.

4.1 Pointer Inference Rules

After *MigThread* identifies the types of the left-hand and right-hand sides of assignments, the pointer inference system has to apply pointer inference rules, shown in Fig. 1 and Fig. 2, to detect unsafe pointer casting. The function `typeof()` is used to check the data types in variables or memory locations. The

<pre> x = y ∈ Prog typeof(x) != POINTER (typeof(y) == POINTER) ((typeof(y) != POINTER) && (y ∈ Closure))) </pre> <hr/> <pre> Pointer_Casting(x, y) </pre>
<pre> x = &y ∈ Prog typeof(x) != POINTER </pre> <hr/> <pre> Pointer_Casting(x, &y) </pre>
<pre> x = *y ∈ Prog typeof(x) == SCALAR ((typeof(*y) == POINTER) ((typeof(*y) != POINTER) && (*y ∈ Closure))) </pre> <hr/> <pre> Pointer_Casting(x, *y) </pre>
<pre> x = *y ∈ Prog typeof(x) == STRUCTURE typeof(x.α) == SCALAR ((typeof((*y).β) == POINTER) ((typeof((*y).β) != POINTER) && ((*y).β ∈ Closure))) offset(x.α) == offset((*y).β) </pre> <hr/> <pre> Pointer_Casting(x.α, (*y).β) </pre>

Fig. 1. Pointer inference rules for direct assignments

useful types are POINTER, SCALAR and STRUCTURE. The term *Closure* concerns the Pointer Casting (PC) Closure which is an approximate superset of possible variables and locations holding hidden pointers, maintained by *Pointer_Casting()* dynamically. Structure types are flattened down into scalar type groups whereas Union types' activation fields are traced since it is possible to create a pointer value under one field and then manipulate its value through another overlaid non-pointer field of the union. Letters α and β indicate the scalar fields which should be enclosed in PC Closure instead of entire structures or unions.

To simplify the presentation, we do not list type casting qualifiers in pointer inference rules, although in fact they are handled properly. At run-time, type casting phases modify dynamic types which are not recorded in the type system, and the dynamic type of the left-hand side determines the size of data to be copied from the location pointed to by the right-hand side of the assignment.

4.2 Static Analysis and Dynamic Check

With the intra-procedural algorithm, our pointer inference system focuses on functions themselves and ignores their inter-relationship for efficiency. Since it is not guaranteed to be able to predict the actual execution path, flow- and context-insensitive approach is adopted to avoid permutation of all possible cases.

The static analysis at compile time is the actual process that applies the pointer inference rules to the source code. Pointer Casting (PC) Closure is a conservative superset of hidden pointer locations, which is implemented in a red-black tree and used as a clue to insert run-time check primitives. After local variables and types are identified, the static analysis uncovers questionable

```


$$\frac{*\mathbf{x} = \mathbf{y} \in \text{Prog} \\
\text{typeof}(*\mathbf{x}) == \text{SCALAR} \\
(\text{typeof}(\mathbf{y}) == \text{POINTER}) \\
|| ((\text{typeof}(\mathbf{y}) != \text{POINTER}) \& (\mathbf{y} \in \text{Closure}))}{\text{Pointer\_Casting}(*\mathbf{x}, \mathbf{y})}$$



$$\frac{*\mathbf{x} = \mathbf{y} \in \text{Prog} \\
\text{typeof}(*\mathbf{x}) == \text{STRUCTURE} \\
\text{typeof}((*\mathbf{x}).\alpha) == \text{SCALAR} \\
(\text{typeof}(\mathbf{y}.\beta) == \text{POINTER}) \\
|| ((\text{typeof}(\mathbf{y}.\beta) != \text{POINTER}) \\
&& (\mathbf{y}.\beta \in \text{Closure})) \\
\text{offset}((*\mathbf{x}).\alpha) == \text{offset}(\mathbf{y}.\beta)}{\text{Pointer\_Casting}((*\mathbf{x}).\alpha, \mathbf{y}.\beta)}$$



$$\frac{*\mathbf{x} = \&\mathbf{y} \in \text{Prog} \\
\text{typeof}(*\mathbf{x}) != \text{POINTER}}{\text{Pointer\_Casting}(*\mathbf{x}, \&\mathbf{y})}$$



$$\frac{*\mathbf{x} = *\mathbf{y} \in \text{Prog} \\
\text{typeof}(*\mathbf{x}) == \text{SCALAR} \\
(\text{typeof}(*\mathbf{y}) == \text{POINTER}) \\
|| ((\text{typeof}(*\mathbf{y}) != \text{POINTER}) \& (*\mathbf{y} \in \text{Closure}))}{\text{Pointer\_Casting}(*\mathbf{x}, *\mathbf{y})}$$



$$\frac{*\mathbf{x} = *\mathbf{y} \in \text{Prog} \\
\text{typeof}(*\mathbf{x}) == \text{STRUCTURE} \\
\text{typeof}((*\mathbf{x}).\alpha) == \text{SCALAR} \\
(\text{typeof}((*\mathbf{y}).\beta) == \text{POINTER}) \\
|| ((\text{typeof}((*\mathbf{y}).\beta) != \text{POINTER}) \\
&& ((*\mathbf{y}).\beta \in \text{Closure})) \\
\text{offset}((*\mathbf{x}).\alpha) == \text{offset}((*\mathbf{y}).\beta)}{\text{Pointer\_Casting}((*\mathbf{x}).\alpha, (\mathbf{y}.\beta))}$$


```

Fig. 2. Pointer inference rules for indirect assignments

assignments by their components on the left-hand side (LHS) and right-hand side (RHS), and suspicious library calls by their parameters. These items could be global variables, local variables in stacks, global/local memory locations in heaps, and function/library calls. They have to be extracted out and applied against the pointer inference rules. If LHS is not in PC Closure and unsafe pointer casting happens in RHS, LHS will be inserted into PC Closure. If the generated PC Closure is not empty, a primitive is appended after each **memcpy()** and structure casting case to detect possible implicit pointer casting.

At run-time, the dynamic check section is activated by the primitives inserted at compile time. Another red-black tree is maintained to record actual unsafe pointer casting variables and locations, from which *MigThread* can decide whether it is safe to conduct migration/checkpointing. This pointer inference system contains both detection and recovery functionalities. Another option is to only maintain a simple detection strategy for minimal overhead by only determining whether unsafe pointer casting has occurred. PC Closure is not created by enumerating all possibilities as above. If pointer casting occurs, migration/checkpointing will abort even though its effect is removed later. This option is useful for large programs since a recovery system could be quite complicated. Users can choose this option in *MigThread*.

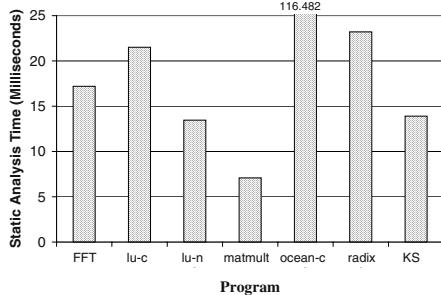


Fig. 3. Execution time of Pointer Inference Algorithm

4.3 Complexity

The space complexity of our pointer inference is $O(N_{prog})$, where N_{prog} is the size of the input programs. The major time complexity is in calculating the PC Closure, which is maintained on a red-black tree with maximum number of related assignment nodes N_{asn} , including large scalar and structure type assignments since these types are large enough to contain pointers. If the execution order of these assignments is the same as the order in the program, the best case will be achieved: $O(N_{asn} * \log N_{asn})$. But if these two orders are completely opposite (one is the reverse of the other), we will get the worst case: $O(N_{asn}^2 * \log N_{asn})$. To avoid it, we establish a threshold: after parsing repeats a certain number of times, we just append corresponding primitives after each related assignment. That means, instead of calculating the smallest PC Closure, we just take a bigger one. This will leave the extra overhead to the dynamic check whose complexity is simply $O(N_{asn} * \log N_{asn})$. Therefore, major overhead can be avoided.

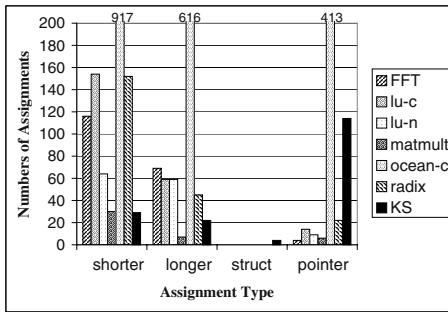
5 Experimental Results and Microbenchmarks

Our testing platform is an Intel Pentium 4 with 1.7GHz CPU, 384MB of RAM, and 700MB of swap space. To evaluate the overall pointer inference algorithm, we apply it to some programs, such as FFT, lu-c, lu-n, ocean-c and radix from the SPLASH-2 suite, matrix multiplication, and KS from the Pointer-Intensive Benchmark Suite [10]. Among them, most static analysis overheads are around 15 milliseconds, except the one for large ocean-c application which consumes 116 milliseconds, as shown in Fig. 3. Since this one-time cost is at compile time and is still relatively small compared to the application's execution time, the overhead of static analysis overhead is minimal.

The detailed information that migration/checkpointing systems can acquire is shown in Table 2 and Fig. 4. The term *shorter* indicates smaller sized data types, such as **char** and **integer**, which normally are not big enough to hold pointer values, and the term *longer* represents larger data types, such as **long** and **double**, which can encompass pointers. The terms *struct* and *pointer* indicate the structure and pointer types in C. The *dir* and *ind* concern values that can be copied into left-hand side directly and indirectly, respectively. The *notype*

Table 2. Assignment statement analysis test (unsafe cases in parentheses)

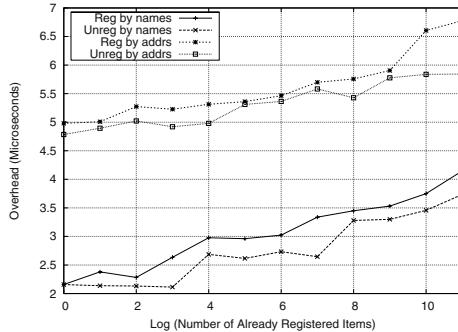
Program	Line No.	Total Asn.	shorter		longer		struct		pointer		no-type	memcpy
			dir	ind	dir	ind	dir	ind	dir	ind		
FFT	1058	183 (0)	113 (0)	3 (0)	68 (0)	1 (0)	0 (0)	0 (0)	4 (0)	0 (0)	0 (0)	0 (0)
lu-c	1051	227 (0)	154 (0)	0 (0)	58 (0)	1 (0)	0 (0)	0 (0)	14 (0)	0 (0)	0 (0)	0 (0)
lu-n	829	132 (0)	64 (0)	0 (0)	58 (0)	1 (0)	0 (0)	0 (0)	9 (0)	0 (0)	0 (0)	0 (0)
matmult	418	43 (0)	30 (0)	0 (0)	6 (0)	1 (0)	0 (0)	0 (0)	6 (0)	0 (0)	0 (0)	0 (0)
ocean-c	5255	1946 (0)	917 (0)	0 (0)	613 (0)	3 (0)	0 (0)	0 (0)	410 (0)	3 (0)	0 (0)	0 (0)
radix	1017	218 (0)	152 (0)	0 (0)	44 (0)	1 (0)	0 (0)	0 (0)	21 (0)	1 (0)	0 (0)	0 (0)
KS	755	169 (0)	29 (0)	0 (0)	15 (0)	7 (0)	4 (0)	0 (0)	84 (0)	30 (0)	0 (0)	0 (0)

**Fig. 4.** Major features of benchmark programs

counts other random type assignments, and the *memcpy* shows the number of *memcpy()* system calls. The numbers in parentheses indicate the number of unsafe pointer casting instances. In this experiment, the *shorter* types are used slightly more than *longer* types. Both of them are used much more frequent than *pointer* types, which is the dominant factor in KS. Finally, structure casting rarely takes place.

Fortunately, among these benchmark programs, no unsafe pointer casting occurs. One major reason is that these programs come from well-known benchmark suites which are primarily created by professionals and type safety rules in C are followed restrictively. But in practice, we cannot count on programmers to achieve this. The pointer inference algorithm has to be applied to deal with the worst case so that programmers can work in any coding style without affecting migration/checkpointing functionality.

To evaluate the dynamic check at run-time, a microbenchmark program is used to investigate the maintenance of PC Closure which is the major operation to register/unregister variables and memory addresses. Fig. 5 illustrates different overheads when PC Closure is saved in a red-black tree with variant numbers of members. The operations with memory addresses normally take more time because they require extra time to convert addresses. Even so, their total cost is about 5-7 microseconds whereas the overhead for variables is just 2-4 microseconds. Since these run-time overhead values are small and increase very slowly as PC Closure expands, the cost of the dynamic check is negligible.

**Fig. 5.** Microbenchmark Results

6 Related Work

Among application-level migration/checkpointing schemes, the Tui system [3] discusses the safety issue, mainly from the point of view of type safety, without providing an effective mechanism about how to prevent it from happening or fixing it once it happens. Another package, SNOW [4], simply declares that it works only for “migration-safe” programs without further explanation or definition of “migration safety”. Both of the above packages rely on programmers to avoid any unsafe cases during migration. Our previous work [6] identified some migration-unsafe features in C without providing a systematic solution.

Modern optimizing compilers and program comprehension tools adopt alias analyses to compute pairs of expressions (or access paths) that may be aliased, and points-to analyses to compute a store model using abstract locations [9,2, 7]. Steensgaard presented a linear-time points-to analysis which contains storage shape graphs to include objects with internal structure and deals with the arbitrary use of pointers [9]. Andersen defined a points-to analysis in terms of constraints and constraint solving [11]. Yong, *et al.*, developed an “offset” approach to distinguishing fields of structures in pointer analysis and makes conservative approximations efficiently [8]. Wilson and Lam proposed a context-sensitive pointer analysis algorithm which is based on a low-level representation of memory locations to handle type casts and pointer arithmetic in C programs [12]. None of these can deal with pointer casting efficiently.

To deal with the unsafe type system in C, CCured [13] seeks to rule out all safety violations by combining type inference and run-time checking. Cyclone compiler performs a static analysis on source code, and inserts run-time checks into the compiled output at places where the analysis cannot determine that an operation is safe [14]. *MigThread* applies a lightweight pointer analysis to solve a different problem: detecting unsafe pointer casting effectively.

7 Conclusions

To overcome the restriction in existing migration/checkpointing schemes where programmers have to write type-safe programs, we have presented an intra-

procedural, flow-insensitive, and context-insensitive pointer inference algorithm, a lightweight pointer analysis, which conducts static analysis to predict possible harmful dynamic types and inserts run-time checks to detect hidden pointers so that the correct thread state can be constructed for migration/checkpointing. Experiments on real applications indicate that the overhead of this scheme is minimal. To reduce overhead, it can be configured to only detect the occurrence of unsafe pointer castings without tracing dynamic types although this might abort migration/checkpointing too conservatively.

This algorithm ensures correct migration/checkpointing whereas other schemes leave the uncertainty to application users. The following research is to provide a full-fledged support for third-party libraries whose procedural calls can introduce undetectable hidden pointers and memory blocks. Therefore, pointer inference algorithms can reach beyond ANSI C.

Acknowledgements. We thank John P. Walters and anonymous reviewers for their helpful comments.

References

1. Milojicic, D., Dougulis, F., Paindaveine, Y., Wheeler, R. and Zhou, S., Process Migration, ACM Computing Surveys, 32(8) (2000) 241–299
2. Rugina, R. and Rinard, M., Pointer Analysis for Multithreaded Programs, Proc. of the Conf. on Program Language Design and Implementation (1999) 77–90
3. Smith, P. and Hutchinson, N., Heterogeneous process migration: the TUI system, Tech rep 96–04, University of British Columbia (1996)
4. Chanchio, K. and Sun, X., Data Collection and Restoration for Heterogeneous Process Migration, Proc. of Int'l Parallel and Dist. Processing Symp. (2001) 51–51
5. Jiang H. and Chaudhary V., Compile/Run-time Support for Thread Migration, Proc. of Int'l Parallel and Distributed Processing Symp. (2002) 58–66
6. Jiang H. and Chaudhary V., On Improving Thread Migration: Safety & Performance, Proc. of the Int'l Conf. on High Performance Computing (2002) 474–484
7. Chandra, S. and Reps, T. W., Physical Type Checking for C, Proc. of Workshop on Program Analysis for Software Tools and Engineering (1999) 66–75
8. Yong, S. H., Horwitz, S. and Reps, T. W., Pointer Analysis for Programs with Structures and Casting, Proc. of the Conference on Programming Language Design and Implementation (1999) 91–103
9. Steensgaard, B., Points-to Analysis by Type Inference of Programs with Structures and Unions, Proc. of the Int'l Conf. on Compiler Construction (1996) 136–150
10. Pointer-Intensive Benchmark Suite, Computer Science Department, University of Wisconsin, Madison, <http://www.cs.wisc.edu/austin/ptr-dist.html>
11. Andersen, L. O., Program Analysis and Specialization for the C Programming Language, PhD thesis, Dept. of Computer Science, University of Copenhagen (1994)
12. Wilson, R. P. and Lam, M. S., Efficient Context-Sensitive Pointer Analysis for C Programs, Proc. of the Conf. on Programming Language Design and Implementation (1995) 1–12
13. Necula, G. C., McPeak, S. and Weimer, W., CCured: type-safe retrofitting of legacy code, ACM Symp. on Principles of Programming Languages (2002) 128–139
14. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y., Cyclone: A safe dialect of C, USENIX Annual Technical Conference (2002) 275–288

Web Page Characteristics-Based Scheduling

Yianxiao Chen and Shikharesh Majumdar

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, CANADA
majumdar@sce.carleton.ca

Abstract. This paper focuses on improvement of system performance through scheduling on parallel web servers. A number of scheduling policies is investigated. A two level policy that uses both the number of embedded objects in a web page as well as the request type demonstrates superior performance in comparison to policies that do not use knowledge of web page characteristics.

1 Introduction

The Internet is becoming an important facet in people's lives for email, web surfing, online shopping and banking. It is also being deployed by companies for performing enterprise computing. Availability of technologies such as secure sockets layer (SSL) and application middleware, combined with cheaper bandwidth and modems, are increasing the popularity of the Internet. A typical web page consists of a main HTML document and multiple embedded objects. An increasing number of audio, video and cgi applications are also being embedded into web pages. Both the performance of the network and server subsystems are being improved. However, the rate of growth in network speed is higher in comparison to the rate of improvement in server processing times [8]. As a result, requests for pages with a large number of embedded objects may push the performance bottleneck to the server system that can lead to an increase of client perceived latency for a web page. Thus, there is a need for effective approaches for improving the performance of web servers. This research focuses on web page characteristic-based scheduling on web servers for achieving high system performance.

Our previous work has demonstrated that parallelizing web page access by storing the embedded objects in a web page on separate web servers that can be concurrently accessed improves system performance significantly [11]. Replication of web pages or a balanced distribution of embedded objects is crucial for achieving this performance improvement. There is a further need for using an effective scheduling policy in such a parallel web server environment that is capable of handling large traffic volumes and improving the overall system performance. This paper concerns this scheduling problem. Scheduling requests on the basis of their communication demands has been proposed in [7]. One of the reasons for the initial focus on the communication subsystem is the low speed of modems used by clients that tends to dominate system performance. However, with continuous increases in modem speeds and the bandwidths of the backbone networks that interconnect the servers with the

clients the performance impact of the server subsystem is increasing in importance. Web pages containing very large audio and video files can stress the server significantly. Dynamic data (as produced by a cgi script for example) is also an important factor and is observed to give rise to a heavy server demand on the 1998 Olympic Web site that recorded a hit of 110,414 hits in a minute during the women's figure skating championship event [5].

Prioritized scheduling of system resources is a means for improving system performance. Scheduling strategies for providing differentiated services to different client classes are discussed in [4]. "Size-based" scheduling is also receiving attention from researchers. For static requests, the size of a request is well approximated by the size of the file, which is well known to the server [9]. When a request size is known, the Shortest-Remaining-Processing-Time (SRPT) scheduling is known to minimize mean response time. The kernel level and application level implementations of SRPT scheduling have been discussed in [9] and [7], respectively. None of these strategies utilize the number of embedded objects in a web page for scheduling that this paper focuses on. Moreover the number of embedded object-based policy does not require a priori knowledge of processing times that may not always be accurately known on real systems especially when web pages have dynamic contents. A detailed survey of existing work on web server performance and scheduling is presented in [6].

The experimental results presented in the paper demonstrate that scheduling policies based on the characteristics of web pages can have a strong impact on performance. Policies that associate a higher priority to web pages with a smaller number of embedded objects are observed to outperform FCFS under a variety of workload conditions. Insights into system behaviour and performance that include the impact of different workload and system parameters on the performance of the scheduling policies are described.

The rest of the paper is organized as follows. The experimental environment is described in Section 2. The scheduling policies and the workload parameters are also described in this section. Performances of the policies are described in Section 3 whereas our conclusions are presented in Section 4.

2 Experimental Environment

An Apache web server Version 1.3.19-based performance prototype is constructed for investigating the scheduling strategies [2] and was run on an isolated "measurement network" of 266MHz Pentium II PCs with 64MB of RAMs running under Red Hat Linux 6.2, kernel 2.2.16-3. The PCs are inter-connected by a 100 MB Ethernet LAN.

Requests for web page components (main HTML file and embedded objects) arrive from clients at a dispatcher that is implemented as an Apache reverse proxy. Requests are then forwarded by the dispatcher to the Apache-based back-end servers. The dispatcher and each back-end server run on separate PC's. A parent process in the Apache-based dispatcher and back-end server spawns a number of children processes that handle requests placed in a central listen queue. The order of the requests in the listen queue is not alterable at the user level. Requests are given priority only after they are extracted from the listen queue. A process changes its priority in accordance with the priority associated with the web page it is serving. The dispatcher and the

back-end servers use the same priority value for each request. The priority value is dependent on the page accessed by a request and the scheduling policy that is discussed in Section 2.1. We have used an open model and client requests are generated on a separate PC by using the open source httpref tool [10] from Hewlett Packard.

As in [11] the back-end servers in the prototype are fully replicated, and any of the servers is eligible for serving any request. In addition to the scheduling policies used for setting the priority of processing a web page component, a dispatching policy that determines which back-end server the request is sent to is required at the dispatcher. The dispatching policy that is used to select a back-end server is Round Robin. Such a policy is easy to implement and is observed to perform well [11].

2.1 Scheduling Policies

The scheduling policies investigated are briefly summarized. In a given experiment, both the dispatcher and the back-end servers use the same scheduling policies (SP).

- ◆ *First Come First Served (FCFS)*

This is a non-priority-based policy. Requests join a First-In-First-Out (FIFO) queue from where they are served by the Apache processes.

- ◆ *Lowest Number of Embedded Objects First (LNEF)*

This policy favors web pages with a fewer number of embedded objects. A request for an embedded object or the main HTML file that belongs to a web page with a smaller number of embedded objects is given a higher priority in comparison to requests for the components of a web page with a higher number of embedded objects. Requests for web pages with the same number of embedded objects are served in a FIFO order. Requests for the main HTML file and for the embedded objects in the same page are given the same priority.

- ◆ *Highest Number of Embedded objects First (HNEF)*

Under this policy, a higher priority is given to the requests for the web pages with more embedded objects. A request for an embedded object or main HTML file that belongs to a web page with a higher number of embedded objects is given a higher priority in comparison to the requests for components of a web page with a smaller number of embedded objects. Requests for web pages with the same number of embedded objects are served in a FIFO order. Requests for the main HTML file and for the embedded objects in the same page are given the same priority.

- ◆ *LNEF_LPH* – is similar to the LNEF policy in which a higher priority is associated with a page with a smaller number of embedded objects. A *lower* priority, however, is given to the request for the main HTML file in comparison to the request for an embedded object in a given web page.
- ◆ *LNEF_HPH* – is similar to the LNEF policy in which a higher priority is associated with a page with a smaller number of embedded objects. A *higher* priority, however, is given to the request for the main HTML file in comparison to the request for an embedded object in a given web page.

The scheduling policies are implemented by using the appropriate Linux real-time policies. Due to space limitations we did not include a discussion of this in the paper. A detailed description is available in [6]. Although the relative

performances of some of these policies are intuitive, it is important to understand the degree of performance improvement from using appropriate request characteristics in scheduling. Moreover, certain workload can lead to non-intuitive results (see Section 3.2 for example).

2.2 Workload and System Parameters

The synthetic workload used in the research allows us to effectively control the workload parameters and to observe their effects on performances. The main workload and system parameters are briefly described.

- ◆ λ – is the arrival rate for web page requests (req/sec). The inter-arrival time is exponentially distributed.
- ◆ S – is the mean service demand in milliseconds (ms). It is associated with each main HTML file or an embedded object within a web page. Simulation of service that includes both file access and CPU execution is achieved by executing a for loop with the desired duration. We have experimented with a range of values of S (50ms, 100ms, 150 ms) for modelling both static as well as dynamic web pages.
- ◆ D – is the distribution of the service demand (fixed or exponential). For the sake of simplicity, we have used an exponential distribution for introducing variability in service demands. Investigation of scheduling for systems with a higher variability in service demand can be interesting and needs further investigation.
- ◆ E – is the number of embedded objects in a web page class. An existing study of a popular proxy trace shows that the number of embedded objects accessed in a page is less than 10 for 95% of the pages [1]. A study of the 1998 World Cup Web site shows that 90% of the pages had 19 or less embedded objects [3]. The levels of E used in this research (1,2,3,10) are in line with these studies.
- ◆ WP_i – describes the web page class i . In most experiments a two-class system ($i=1,2$) is used. Each web page class is characterized by the number of embedded objects, and the distribution and the mean of the service demand. Thus WP_i (E , D , S) represents web page class i that contains E embedded objects and has a mean service demand of S milliseconds with a distribution given by D .
- ◆ $Pr[WP_i]$ – is the probability with which a client makes a request for web page class WP_i . This parameter simulates the popularity of a web page class. [Values used in the paper: $Pr[WP1]=95\%$, $Pr[WP2]=5\%$]
- ◆ B – is number of back-end servers. The default value for this parameter is 4. B is a system parameter.

Performance Metric: The performance metric used for comparing the scheduling policies is the mean response time that is defined next. The response time of a web page request is the elapsed time measured in seconds from the time at which the first request for the main HTML document of a web page is issued to the time of reception of the last embedded object at the client.

An experiment is run multiple times. The number of runs for a given set of parameter values is sufficient to guarantee a confidence interval of less than +/- 5% at a confidence level of 95%. As in [11], caching is turned off on both the dispatcher and the back-end servers so that we can focus only on the impact of scheduling strategies on performance. All requests are HTTP/1.0 requests.

Both the dispatcher and back-end servers use the same configuration parameters. A discussion of the parameter values and the rationale behind their choice are provided in [6]. Due to space limitations we could not include the discussion in this paper.

3 Performance of the Scheduling Policies

A number of experiments are described in [6]. Only a representative set of data is presented in this paper. A factor-at-a-time approach is used: one factor is varied at a time during an experiment while the other factors are held at fixed values. Results of experiments are presented in terms of graphs. The fixed factors (workload parameters, system parameters and scheduling policies) are included in the legend for the corresponding graph.

The scheduling policies FCFS, LNEF, and HNEF are discussed first. Although HNEF demonstrates the worst performance in all of the experiments, we have included it in our discussion to demonstrate the effect of resource hogging by requests for larger pages. The motivation for using LNEF_LPH and the performances of LNEF_LPH and LNEF_PH are described in Section 3.3.

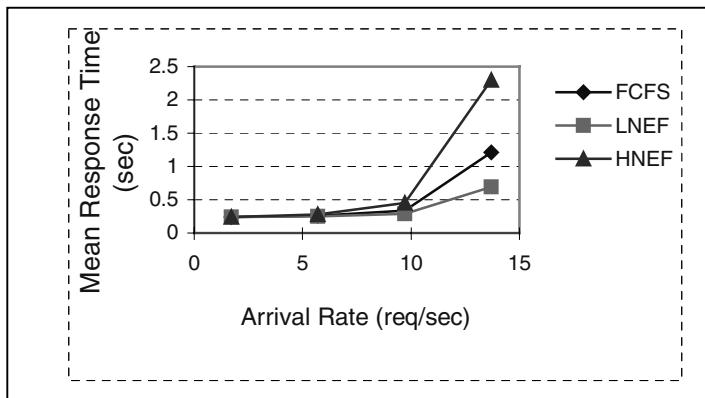


Fig. 1. Benefits of Web Page Characteristics-Based Scheduling. WP1($E=1$, $D=\text{fixed}$, $S = 100\text{ms}$), WP2($E=10$, $D=\text{fixed}$, $S=100\text{ms}$). $\Pr[\text{WP1}] = 95\%$, $\Pr[\text{WP2}] = 5\%$.

The performances of FCFS, LNEF and HNEF are presented in Fig. 1. As expected the mean response time achieved with any policy increases with the arrival rate as more contention for resources occurs at higher arrival rates. LNEF gives rise to the best performance followed by FCFS and HNEF. By giving preference to requests for pages with a smaller number of embedded objects and hence a lower service demand, LNEF achieves the best performance. In case of HNEF, pages with a higher number of embedded objects are processed prior to requests for pages with a smaller number of embedded objects. This introduces a large queueing delay for the requests for pages with a smaller number of embedded objects and results in the largest mean

response time (see Fig. 1). Although with FCFS pages with a larger number of embedded objects are not favoured explicitly, requests for such pages that arrive earlier in comparison to the requests for pages with a smaller number of embedded objects tend to hog the servers resulting in a mean response time that ranks in between LNEF and HNEF. The impact of the using the number of embedded objects in scheduling is observed to be substantial. A large improvement in performance over FCFS is observed with LNEF at moderate to high system load. The performance of HNEF demonstrates the penalty associated with associating a higher priority with requests for components of a web page with a high number of embedded objects that leads to a monopolization of resources by such requests.

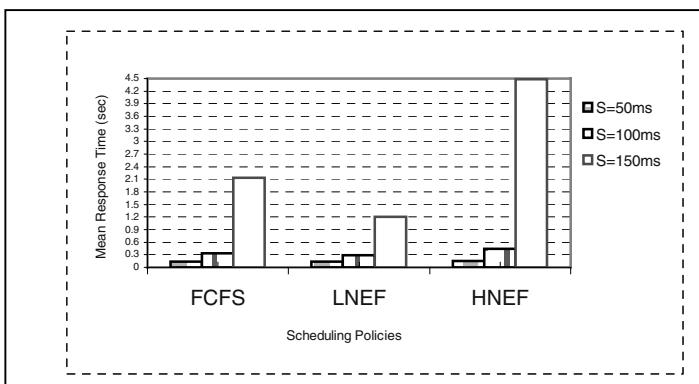


Fig. 2. Impact of Mean Service Demand on Performance. $\lambda=9.7$ req/sec, WP1(E=1, D = fixed), WP2(E=10, D = fixed), $\Pr[WP1] = 95\%$, $\Pr[WP2] = 5\%$

3.1 The Impact of Service Demand

Both the mean and the distribution of service demands can affect the performance of the scheduling policies. The impact of the mean service demand S on the relative performances of the three policies at a given arrival rate is displayed in Fig. 2. A fixed distribution of service times is used. LNEF performs the best followed by FCFS and HNEF (see Fig. 2). The order of the bars in the figure is the same as the order of the legends.

The performance advantage of LNEF over FCFS becomes more and more prominent at higher service demands. A higher service demand at a given arrival rate produces a higher load at the server and tends to increase the queueing delay produced by any given scheduling policy. Using a web page characteristic-based scheduling is observed to become more important when higher service demands are associated with web page components.

The ranking of the policies remains unaltered although the performance differences among the policies are changed with an exponential distribution for service times. However at moderate to high system load, for any given policy and arrival rate a higher mean response time in comparison to the fixed distribution is achieved when the service demand associated with a web page component is exponentially distributed. With a fixed distribution, the processing times for all the web page

components are equal. Existence of variabilities in service times for the different web page components introduced by the exponential distribution, seems to be detrimental to system performance leading to a higher mean response time.

Number of Back-end Servers:

Increasing the number of back-end servers for a given S and arrival rate tends to reduce the mean response time for any scheduling policy. The ranking of the policies is not affected by the increase in B [6].

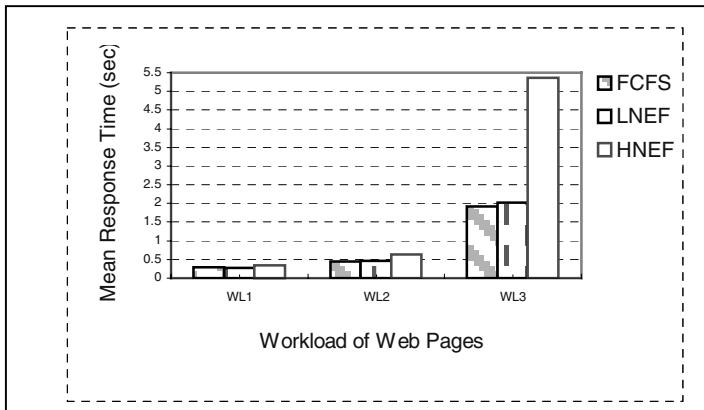


Fig. 3. Impact of E on Performance. $\lambda = 7.8$ req/sec, WP1(D =fixed, $S=100$ ms), WP2($E=10$, D =fixed, $S=100$ ms), $\Pr[WP1] = 95\%$, $\Pr[WP2] = 5\%$

3.2 The Effect of the Number of Embedded Objects

A comparison of the performances of the scheduling policies for three different numbers of embedded objects in WP1 is presented in Fig. 3. The numbers of embedded objects in WP1 are 1, 2, and 3 for workloads WL1, WL2, and WL3 respectively. The number of embedded objects in WP2 for each of these workloads is fixed at 10. Increasing the number of embedded objects in WP1 is observed to reduce the performance differences between LNEF and FCFS at a given arrival rate. The relative performance of HNEF deteriorates heavily at higher values of E . It is interesting to note that for WL3, the performance of LNEF is slightly inferior to that of FCFS (see Fig. 3). The rationale for such a behavior is presented next. Competition for service exists not only among requests with different priorities, but also among requests with the same priority. We will discuss the effect of each of these on performance.

LNEF discriminates against pages with a larger number of embedded objects. The mean response times achieved with WL3 and FCFS at $\lambda=7.9$ req/sec for WP1 and WP2 are 2.38 seconds and 4.59 seconds respectively. The response times produced by LNEF at the same arrival rate are 1.24 seconds and 31.4 seconds for WP1 and WP2 respectively. In comparison to FCFS, LNEF reduces the response time for WP1 by 48%. The corresponding increase in the response time for WP2 is disproportionately large: 584%. As a result, the overall performance advantage of LNEF over FCFS is reduced.

Competition among requests with the same priority is discussed next. If the requests with equal priority abound in the system, there is a lower chance that the embedded objects within a web page belonging to WP1 are served in parallel. In case of WL1, serving a single embedded object request for WP1 completes a web page request. However for WL3, requests for three embedded objects need to be served for each web page. Due to a reduction in concurrency, the processing of all the objects in a page may take a longer time to complete. As a result, the number of incomplete page requests is increased, as the workload is changed from WL1 to WL3 for example. LNEF experiences the combined effects of the competition among web page classes with different priorities and the competition among the requests with the same priority. The overall effect is a deterioration in the performance of LNEF (see Fig. 3).

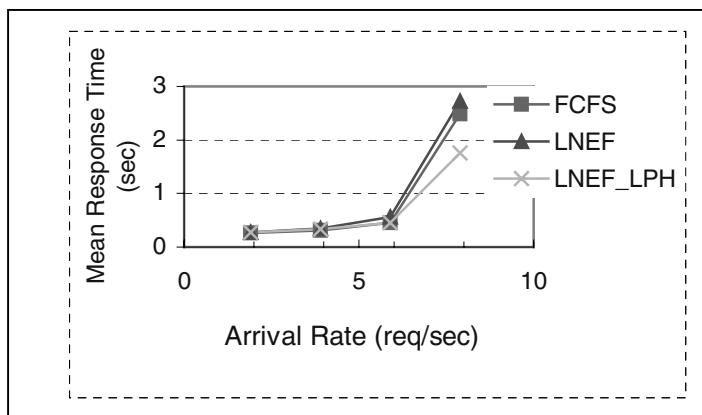


Fig. 4. Effect of Using File Type in Scheduling. WP1(E=3, D=fixed, S=100ms), WP2(E=10, D=fixed, S=100ms), $\Pr[\text{WP1}] = 95\%$, $\Pr[\text{WP2}] = 5\%$.

3.3 Prioritizing Requests Based on Types of Web Page Components

A web page request is associated with two types of files: the main HTML file and embedded objects. Whether or not the performance of LNEF can be further improved by differentiating among requests for different object types is investigated. Two additional scheduling policies LNEF_LPH and LNEF_HPH are experimented with. For both these policies request for files in a page with a smaller number of embedded objects is given a higher priority over requests for files with a higher number of embedded objects. However, request for different types of files for a given page are treated with different priority. In LNEF_HPH, the request for the main HTML file in a page is given a higher priority over requests for the other embedded files in the page whereas in LNEF_LPH, the request for the main HTML file in a page is given a lower priority in comparison to request for the other embedded files in the page. As shown in Fig. 4, LNEF_LPH is superior to the regular LNEF. LNEF_HPH performs worse than the regular LNEF [6]. The rationale for such a behaviour is briefly presented. Giving a higher priority to the main HTML page increases the total number of pages that are simultaneously processed. This is because the HTML files of all the waiting page requests will be processed first before the requests for the embedded objects are

dispatched. As a result the average number of pages that are simultaneously served tends to increase, increasing the mean response time. With LNEF_LPH on the other hand, once an HTML file for a page is processed and the requests for the constituent embedded files are generated, processing these embedded file requests will be given higher priority in comparison to the processing of the main HTML file for a new request. This tends to increase the number of servers divested in the service of the components of the same page, which in turn improves the parallelism in the service of a web page. The average number of distinct web pages served at the same time is reduced and the mean response time is observed to decrease.

We have investigated the performance of the scheduling policies under a number of different workload conditions. Analysis of performance for a system 30 web page classes that models a real system confirmed the superiority of LNEF_LPH. A discussion of these results could not be included due to space limitations. The interested reader is referred to [6].

4 Conclusions

The research reported in this paper focuses on improvement of system performance on a parallel web server by deploying appropriate request scheduling strategies. FCFS and a number of scheduling strategies that are based on the characteristics of the web pages are investigated through a performance prototype subjected to a synthetic workload running on a network of PC's. Insights gained into system behaviour and scheduling from the results of the experiments are summarized.

Impact of Web Page Characteristics: Scheduling strategies based on web page characteristics seem to have a strong impact on performance. LNEF and its variant LNEF_LPH are observed to outperform FCFS under a number of different workload conditions. Both LNEF and LNEF_LPH are based on a known parameter, the number of embedded objects in a web page, and can thus be implemented on a real system.

Influence of Systems Load: Higher the system load, higher is the difference between the performance of FCFS and the request characteristics-based scheduling policies.

Variability in Service Demand: Performance of the system is observed to deteriorate when the distribution of service time is changed from fixed to exponential. The ranking of the scheduling policies does not change with a change in the distribution of service times.

Scheduling Based on Web Page Component Type: An important observation made in this research is the non-intuitive result that the relative performance of LNEF with respect to FCFS deteriorates as the number of embedded objects in the web page class WP1 is increased. Using information on the type of file being served in conjunction to the number of embedded objects is observed to be the solution to this problem. LNEF_LPH demonstrates a superior performance for a variety of different workload parameter values.

Acknowledgments. Financial support for this research is provided by Natural Sciences and Engineering Research Council of Canada.

References

- [1] A. Abhari, S.P. Dandamudi, S. Majumdar. "Structural Characterization of Popular Web Documents", International Journal of Computers and Their Applications Vol. 9, No:1, March 2002, pp. 15–24.
- [2] Apache Software Foundation, "Apache Web Server", Available at: <http://www.apache.org>
- [3] M.F. Arlitt, T. Jin, "Workload Characterization of the 1998 World Cup Web Site", HP Laboratories Technical Report, HPL-1999-35(R.1), HP Laboratories, Palo Alto, September 1999.
- [4] G. Banga, P. Druschel, J. C. Mogul, "Resource Containers: A new facility for Resource Management in Server Systems," in Proceedings of the Symposium on Operating System Design and Implementation (OSDI), New Orleans, USA, February 1999, pp. 45–58.
- [5] J. Challenger, P. Dantzig, and A. Iyengar, "A Scalable and Highly Available System for Serving Dynamic Data at Frequently Accessed Web Sites," IBM Research T. J. Watson Research Center, NY, 1998.
- [6] Y. Chen, "Scheduling on Web Servers", M.C.S. Thesis, Department of Computer Science, Carleton University, Ottawa, Canada, January 2003.
- [7] M. E. Crovella, R. Frangioso, M. Harchol-Balter. "Connection Scheduling in Web Servers," in Proceedings of USENIX Symposium on Internet Technologies and Systems, Boulder, USA, October 1999, pp. 243–254.
- [8] I. Foster, "The Grid: A New Infrastructure for 21 st Century Science", Physics Today, 2002. Available from www.aip.org/pt/vol-55/iss-2/p42.html.
- [9] M. Harchol-Balter, N. Bansal, B. Schroeder, M. Agrawal, "Implementation of SRPT Scheduling in Web Servers," in Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS), Goteborg, Sweden, July 2001, pp. 1–24.
- [10] D. Mosberger and T. Jin, "*httpperf* – A Tool for Measuring Web Server Performance," Proceedings of the *Workshop on Internet Server Performance*, Madison, Wisconsin, USA, June 1998.
- [11] S. Nadimpalli, S. Majumdar, "Performance Enhancement Techniques for Parallel Web Servers", in Proceedings of the IEEE International Conference on Parallel Processing (ICPP), Toronto, Canada, August 2000, pp. 233–241.

Controlling Kernel Scheduling from User Space: An Approach to Enhancing Applications' Reactivity to I/O Events

Vincent Danjean and Raymond Namyst

LaBRI, Université de Bordeaux I, F-33405 Talence, France.

Abstract. In this paper, we present a sophisticated mechanism that allows an application to tightly control the way I/O events are handled within the underlying operating system's kernel. The goal is to provide an efficient user-level interface to allow applications to quickly detect and handle asynchronous I/O events. Reactivity is obviously a crucial property for many time-critical applications. For instance, it is a major issue in distributed applications where poor reactivity to communication events may have a direct influence on the observed latency of communications, and may even impact on the whole application behavior. Our approach is based on an extension of the *Scheduler Activation* kernel scheduling model, originally proposed by Anderson et al. The idea is to minimize the time elapsed between the detection of an I/O event within the kernel and the scheduling of the appropriate user-level thread within the application. The progress of I/O operations still takes place in kernel space, but the scheduling decisions are taken in user space. We have implemented the core of our proposal within the Linux kernel, and the PM² multithreading environment was ported on top of the corresponding user-level API. Our experiments demonstrate that the implementation of our kernel extensions is very efficient and that the overhead introduced by the user-level interface is very small. Moreover, the observed results show that our approach competes favorably against approaches based on the use of signals or kernel threads.

1 Introduction

In the past few years, the widespread use of clusters of SMP workstations for parallel computing has lead many research teams to work on the design of portable multithreaded programming environments [1,2]. One major challenge is to design environments that reconcile the portability and efficiency requirements of parallel applications: applications have to be portable across a wide variety of underlying hardware while still being able to exploit much of its performance. Most noticeably, many efforts have been focused on the problem of performing efficient communications in a portable way [3,4] on top of high-speed networks [5,6].

However, one major property has often been neglected in the design of these distributed runtime systems: the *reactivity* to communication events. We call *reactivity* of an application its ability to handle external, asynchronous events as soon as possible within the course of its regular execution. Trying to minimize (or at least to bound) the response time to a network event is indeed a critical parameter for many applications

since the observed *latency* of messages may become proportionally long. Actually, many distributed environments suffer from this problem since the performance of mechanisms involving synchronous interactions between nodes directly depends on their reactivity to incoming communications. A Distributed Shared Memory (DSM) subsystem, for instance, may behave very poorly if the internal request messages exchanged to fetch remote data are not handled as soon as they reach the destination node[7]. MPI can also be subject to such bottlenecks when using asynchronous primitive such as asynchronous sends (`iSend()`)[8] or remote get and put operations (MPI2). Most communication schemes including a back and forth interaction with a remote agent, such as a remote procedure call, are extremely sensitive to the reactivity of the peer.

In this paper, we investigate the problem of maximizing the reactivity to I/O events in a user-level multithreaded context, focusing on I/O events detected in kernel space, such as those notified by hardware interrupts (see [9] for a discussion about detecting I/O events in a reactive manner by using a user-level polling-oriented policy). More precisely, our goal is to propose a mechanism able to quickly propagate kernel I/O events to the application. That is, we try to minimize the time elapsed between the detection of an I/O event (typically notified by a hardware interrupt) and the execution of the appropriate user-level handler within the application. This mechanism is an extension of the *Scheduler Activations* model introduced by Anderson et al [10]. A flexible user-level API allows the application to tightly control the scheduling of threads when I/O events are detected, thanks to a sophisticated cooperation between the kernel and user space applications.

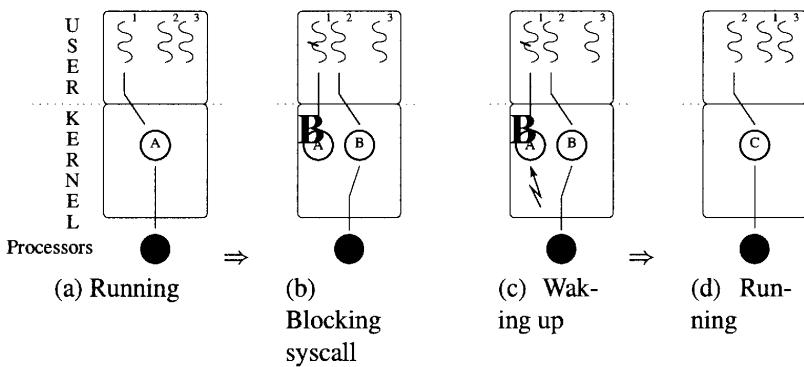
2 Dealing with I/O Events in a Multithreaded Context

To understand how an I/O event is typically handled within a multithreaded application, let us start from the point where I/O events are actually detected: inside the operating system's kernel. For instance, let us consider an I/O event notified by an hardware interrupt. For obvious security reasons, hardware interrupts cannot be handled directly in user-space (at least in traditional operating systems). Indeed, the occurrence of such an interrupt usually triggers the execution of an interrupt handler within the operating system's kernel. Then, the corresponding event may in turn be notified to a process (or kernel thread) either asynchronously or synchronously.

In the second case, a process (or kernel thread) can be notified synchronously by performing a blocking system call, sleeping inside the kernel until the appropriate event occurs. The triggering of the corresponding interrupt will simply wake up the process and, later, let it return back in user space. This mechanism has the advantage to be extremely simple from the application's point of view. However, its efficiency depends on the capability of the thread scheduler to execute the kernel thread as soon as it is woken up, especially if the number of threads in "ready" state exceeds the number of processors.

The *Scheduler Activations* scheduling model, proposed by Anderson et al [10], features a set of mechanisms that enable the kernel to notify a user-level process whenever it makes a scheduling decision affecting one of the process's threads. It is implemented as a set of *up-calls* and *down-calls*.

A traditional system call is a *down-call*, from the user-level down into a kernel-level function. The new idea is to introduce a corresponding *up-call*, from the kernel up into a user-level function. An up-call can pass parameters, just as system calls do. An *activation* is an execution context (*i.e.*, a task control block in the kernel, similar to a kernel-level thread belonging to the process) that the kernel utilizes to make the up-call. The key-point is that each time the kernel takes a scheduling action affecting any of an application's threads, the application receives a report of this fact and can take actions to (re)schedule the user-level threads under its control. Of course, if several events occur at the same time, only one upcall is done to notify all the events. Figure 1 illustrates how a blocking system call is handled by such a system.



- 1(a) The user-level thread 1 is running onto an activation *A* bounded to a processor.
- 1(b) The thread 1 makes a system call blocking the activation *A*. Another activation *B* is launched which runs another user-level thread 2.
- 1(c) An interrupt occurs waking up the blocked activation *A*.
- 1(d) A new activation *C* is created to send the unblocking notice event. To run this activation, the activation *B* is preempted. The activation *C* can chose which user thread to run. Activations *A* and *B* can be discarded once the state of their thread has been sent to the application in the upcall.

Fig. 1. Blocking System Call

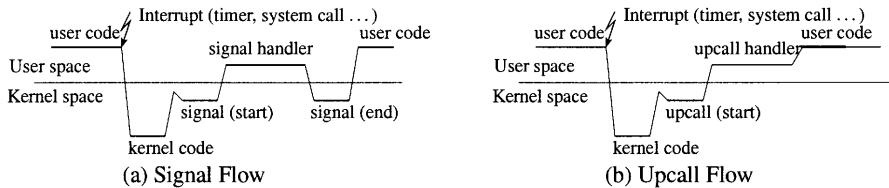
There are several issues with the original Scheduler Activation model that make it difficult to obtain an efficient implementation[11]. First, a lot of information needs to be sent by the kernel to the application. Each time a preemption or an unblocking notice occurs, the kernel needs to send to the application the state of the preempted or unblocked thread. When a thread executes a blocking system call and resumes, the application needs to manage (copy) those information several times. Another difficulty comes from the fact that the application has no control at all over the number of virtual processors assigned to it (it only knows the number of currently assigned virtual processors). This means that the application (in particular upcalls) has to be made completely reentrant. Finally, the fact that the upcalls tell the application about the state of the interrupted (or unblocked) thread makes this model unable to handle internal kernel scheduling state. A blocked

thread has to run in kernel mode until it becomes ready to come back in user mode before being signaled as unblocked for the application.

3 Our Proposition: An Efficient and Reactive Scheduling Model

3.1 A New SA-Like Scheduling Model

To solve the aforementioned issues, we propose an extension of the Scheduler Activation model. The key idea is that the operating system kernel can not change the number of processors assigned to an application during its execution anymore. If a thread blocks, the kernel still has to create a new activation to give back the virtual processor to the application. However, the kernel is not allowed to preempt a virtual processor on its own anymore. More precisely, if the kernel preempts an activation of an application (as this often occurs in a time-shared operating system such as Linux), the application is simply left unnotified.

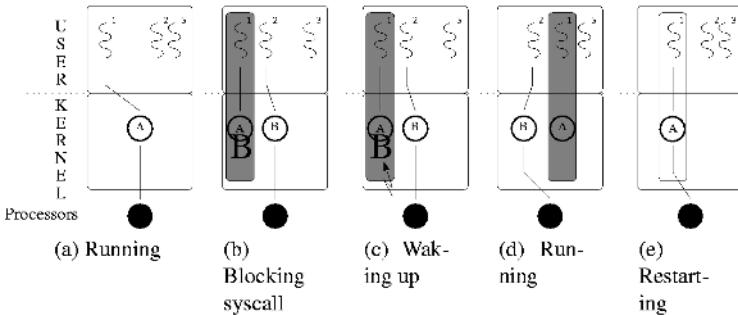


- 2(a) At the end of the signal handler, a system call switch to kernel mode to restore the user environment.
- 2(b) At the end of the upcall, the user library is able to directly continue the interrupted user code. The upcall handler must take care to restore the proper environment, but no system call is needed anymore.

Fig. 2. Restarting After an Event

In our model, upcalls are always performed within the currently running activation, like Unix signals. However, at the end of the upcall, the application does not go back into the kernel using a system call as it is done for signals. Thus, our upcalls are very cheap: there is only the addition of the execution of a user-space function with respect to a standard kernel/user space transition. Figure 2 shows our mechanism used to launch an upcall and Figure 3 illustrates how a blocking system call is handled by our new system. The kernel/user interface features several upcalls used by the kernel to notify the application. Obviously, a few system calls are also needed by the application to interact with the kernel.

Upcalls. The main originality of our model is that the user-level thread scheduler has a complete control over the scheduling of threads, even at times they block or wake up within the kernel. This explains why the upcalls performed by the kernel in our model are slightly different from the ones in the original model. Two kinds of events may be reported by the kernel. *Synchronous events* are reported by upcalls raised synchronously



- 3(a) The user-level thread 1 is running onto an activation *A* bounded to a processor.
- 3(b) The thread 1 makes a system call blocking the activation *A* in the kernel. Another activation *B* is launched which runs another user-level thread 2.
- 3(c) An interrupt occurs. The blocked activation *A* is now ready to wake up (but it will only wake up when requested by the application).
- 3(d) The activation *B* is used to send the unblocking notice event with an upcall. The user thread scheduler puts the thread 1 in the ready-to-run thread queue. Its register state is unknown to the application: the thread is always in the kernel.
- 3(e) When the user thread scheduler wants to restart the unblocked thread, it saves the currently running thread state and discards the current activation in favor of the unblocked thread activation.

Fig. 3. Blocking System Call

with respect to the currently running thread. That is, the upcall is raised when the thread can not follow its standard flow of execution. Such upcalls typically tell the application that the current thread has just blocked (upcall **block**) and indicate if a new activation has been created or if a previously unblocked one has been restarted. On the opposite, *asynchronous events* are reported by upcalls that behave like Unix signals. For instance, the **unblock** upcall is used to notify the application that some previously blocked activations can now be resumed. Thus, the application can choose which one is to be restarted immediately.

System Calls. An application can use several system calls to use and control the Scheduler Activations mechanism. The **init** system call is performed at initialization time to notify the kernel that the application will use the Scheduler Activations mechanism. The kernel is also told about the addresses of the user-space upcalls entries and the addresses of some variables accessed by the kernel. The **revoke** system call tells the kernel that the application does not need the underlying virtual processor anymore until either the application requests this virtual processor again or a blocked activation wakes up. The **request** system call is used by the application to tell the kernel that it wants to use (again) a virtual processor that has been revoked. Finally, the **wakeup(act_id)** system call destroys the current activation so that the unblocked activation of identifier *act_id* can be resumed within this virtual processor.

3.2 A User-Level Interface to Our SA Model Extension

Obviously, such a complex scheduling mechanism is not intended to be used directly by regular applications. Therefore, we developed a stack of software layers to meet the needs of a wide range of applications. The overall design of our implementation is shown on figure 4.

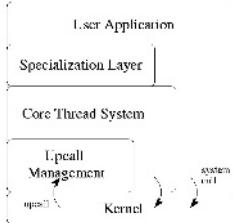


Fig. 4. User Interface Design

The **Core Thread System** is the most important part of our thread package. It is responsible for providing the core locking mechanisms, for managing thread related data structures (thread control blocks) and for managing thread states. It also provides a basic scheduling policy (round robin) that is used by default for scheduling decisions. The **Upcall Management layer** handles all events sent by the kernel to our thread package by the means of upcalls. These functions modify the thread states when needed. For instance, this layer is responsible for removing blocked threads from the ready queue, and for inserting unblocked threads back into this queue. The **Specialization Layer** is intended to be used by the application to customize the behavior of the core thread system. In particular, several scheduling functions can be redefined very easily, such as the function responsible for choosing the next thread to run each time a context switch takes place (`find_next_to_run`). So, virtually any kind of scheduling policy can be implemented on top of our software stack, because the control over the scheduling of kernel activations is entirely done in user space.

The Core Thread System Interface. The core thread system is mainly used by the specialization layer, though it may also be used directly by the application. It features all basic thread operations such as thread creation, destruction, synchronization, *etc.* These functions are often extended in the specialization layer in order to manage more attributes or to offer more advanced features (synchronization barriers, *etc.*). It also provides some low-level entry points to manage the core thread system. One of the most important is the `thread_find_next_to_run` function invoked by the core thread system each time a context switch is performed between user-level threads.

This has been inspired by the OpenThread interface[12] where the application has the ability to manage thread queues and to register call-back functions for each thread state transition. However, the fact that our system can manage more than one processor (ie real parallelism) makes this interface not usable as-is. For example, the locking scheme can

not be the same if the application wants one run queue per virtual processor or one for the whole application. So, for now, such customizations, though possible (and actually implemented), require deep modifications within the library.

4 Implementation Issues

We now discuss the main issues we had to address during the implementation of our model, focusing on the issues related to our extension of the Scheduler Activation (SA) model. Issues related to the original model are discussed in [10].

4.1 On the Number of Kernel Activations

In the original SA model, an unblocked activation is destroyed as soon as the corresponding event has been notified to the application. In our model, as we do not transmit the state of blocked threads to user mode anymore, the activation cannot be destroyed at this point. We must wait for the application to request a restart of the unblocked activation. However, this new model does not require more activations than the original. The activations can live longer, but the kernel creates the same number of activations as in the original model. In particular, when an activation blocks, the kernel tries to restart an unblocked one (if available) instead of creating a new one. So, new activations are created only when a virtual processor becomes free while all activations are either in “running” or “blocked” states.

For now, there is no limit in our implementation about the number of activations that can be created by the kernel. If each user thread executes a blocking system call, then there will be as many [blocked] activations created by the kernel as user threads. To limit kernel resources, we could make blocking system calls either fail or not generate an upcall. However, this leads to other problems: what should be done if all blocked threads need the action of another user thread (not launched due to resource limit on the number of activation) to unblock. Lack of activation could also lead to a special signal so that the application can try to unblock some of its threads or exit with an error code. However, all these behaviors are different from the one of a standard thread library. So, no limit is implemented and, in the bad case, our model can consume as many kernel resources as in the current standard pure-kernel thread library.

4.2 Critical Sections and Upcall Reentrancy

The problem of concurrency between upcalls and user-level code is a hard one, and there are a number of situations where upcalls should be delayed because their execution may conflict with some currently running user-level execution flow, or even with another upcall within the same virtual processor. We solved these problems by implementing a cheap per-virtual processor mutual exclusion mechanism accessed by both user-space and kernel-space. Each virtual processor is equipped with its own lock variable, which is allocated in user-space. Of course, the kernel is told about the location of each of these locks at initialization time.

Just before executing an upcall on a virtual processor, the kernel sets the variable. The user-level core has the responsibility to clean this variable at the end of each upcall. So, when the kernel wants to make an informative upcall while this variable is set, the upcall is delayed. In the case the kernel wants to make an upcall because the previous activation has just blocked, then the event is simply not reported to the application and thus the previous blocked activation is still considered as running. When the activation unblocks, it resumes as if it never stopped. This case (blocking during an upcall) should be rather rare but needs to be correctly handled. We cannot launch yet another upcall if the previous one has been interrupted because the upcall code has been swapped out.

5 Performance

All the features described in this paper were implemented within the Linux kernel (v2.4.19). We now present an evaluation of the cost of some raw kernel mechanisms we did implement. This can be viewed as the overhead of our mechanism compared to a pure kernel-level scheduler.

5.1 The Cost of Kernel Mechanisms

Performing an Upcall. To perform an upcall, the kernel only needs to write a few parameters on the user stack (previous stack pointer and program counter, virtual processor number, ...) before returning as it usually does. On the application side, the upcall handler is executed and then uses a mini assembly trampoline to return to the previously interrupted code. In table 1, one can see the time lost by the application when the kernel preempts the application and executes an upcall, a signal handler or nothing (just returns after regular kernel mode processing). These measures have been taken on an Intel P4 1.8GHz processor. All numbers were gathered from multiple runs. The initial run is not timed to reduce cold cache miss effects. One can observe that upcalls and signal handlers add a few overhead with respect to a regular return. However this overhead is smaller for an upcall than for a signal handler. This is due to the fact that the kernel does not need to save and restore all registers for an upcall.

Table 1. Upcalls Overhead

	Time lost	Overhead
Regular path	13.1 μ s	0 (+0 %)
Upcall	14.1 μ s	+1.0 μ s (+7.63 %)
Signal handler	16.5 μ s	+3.4 μ s (+25.95 %)

Table 2. Restart Overhead and System Calls

upcall overhead	1.0 μ s
getpid()	0.92 μ s
act_cntl(RESTART_UNBLOCKED, id)	2.17 μ s

Restarting an Unblocked Activation. As explained earlier, the restart of an unblocked thread requires a system call in our model, because the underlying activation may be suspended deep in the kernel. Such a system call (`act_cntl(ACT_CNTL_RESTART_UNBLOCKED, id)`) discards the current activation and restarts the unblocked one (with the id `id`) on the current virtual processor. In table 2,

one can see the time needed by a basic system call (`getpid`) and the time needed to restart an unblocked activation. For the latter, we measure the time from the beginning of the system call to the effective restart of the unblocked thread: this time includes the system call and the upcall generated when the unblocked activation restarts. So, the overhead due to discarding the previous activation and switching to the awakened one costs only 0.25 μ s.

5.2 Efficiency and Flexibility

We now evaluate the overhead of our Scheduler Activation model within a user-level thread scheduler, using both a POSIX-compliant API and the API of the PM² distributed programming environment [2], which was ported on top of our software stack. Table 3 shows the time needed to perform a context switch between user-threads, and the execution time of a synthetic program (`sumtime`) that creates, synchronizes and destroys a large number of threads (it makes a sum of the n first integers recursively, each thread managing half of the interval of its father).

Table 3. Basic Thread Operations

	<code>yield()</code>	<code>sumtime(2000)</code>
<code>libpthread</code>	1.16 μ s	7215 ms
POSIX	1.01 μ s	12 ms
PM ²	0.40 μ s	12 ms

We can see that in any case, our user-level libraries perform a lot better than the Linux kernel thread library, while keeping the ability to use SMP machines and correctly manage blocking system calls). The POSIX thread version does not improve a lot the `yield` operation since it is not a pthread call but a system call (`libpthread` does not have `pthread_yield` operation, `sched_yield` is used instead). The little improvement comes from the fact that there is only one running kernel thread with our POSIX library (so no real kernel context switch is performed) while there are two with the `libpthread` library (one for each application thread). If we redefine the `sched_yield()` call as an alias for our `thread_yield()` function in the POSIX library, then POSIX and PM² libraries show similar performance for `yield()` operations. With the `sumtime` program, the lost of performance with the standard `libpthread` library worsens when the number of threads grows. In this example, the program manages about 2000 threads. With our thread libraries, we can easily use this program with more than 100 000 threads. It is not possible with the standard `libpthread` library.

6 Conclusion and Future Work

In this paper, we present the design and implementation of a fast and flexible extension to the Scheduler Activation model, together with a user-level software stack to allow an easy customization of the thread scheduling policy. The application has a complete

control over this scheduling because all decisions are made in user-space. The kernel mechanisms were fully implemented within Linux (v2.4.19). A POSIX-thread compliant user-level thread library has been developed on top of our software stack, as well as the PM² distributed multithreaded programming environment. We conducted a number of experiments which show that the core mechanisms are very efficient, and that the overhead introduced within the user-level schedulers is very small.

We plan to measure the impact and the improvement due to the use of these mechanisms within large multithreaded networking applications. We also intend to investigate the tuning of specific scheduling policies in the PM² environment, as far as network I/O operations are concerned. The goal is to maximize the reactivity of the threads in charge of handling network events so as to speed up the whole distributed application.

References

1. Foster, I., Kesselman, C., Tuecke, S.: The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing* **37** (1996) 70–82
2. Namyst, R., Méhaut, J.F.: PM2: Parallel multithreaded machine. A computing environment for distributed architectures. In: *Parallel Computing (ParCo '95)*, Elsevier Science Publishers (1995) 279–285
3. Prylli, L., Tourancheau, B.: BIP: A new protocol designed for High-Performance networking on Myrinet. In: *1st Workshop on Personal Computer based Networks Of Workstations (PC-NOW '98)*. Volume 1388 of *Lecture Notes in Computer Science.*, Orlando, USA, Springer-Verlag (1998) 472–485
4. Aumage, O., Bougé, L., Méhaut, J.F., Namyst, R.: Madeleine II: A portable and efficient communication library for high-performance cluster computing. *Parallel Computing* **28** (2002) 607–626
5. Dolphin Interconnect: SISCI Documentation and Library. (1998) Available from URL <http://www.dolphinics.no/>.
6. Myricom: Myrinet Open Specifications and Documentation. (1998) Available from URL <http://www.myri.com/>.
7. Antoniu, G., Bougé, L.: DSM-PM2: A portable implementation platform for multithreaded DSM consistency protocols. In: *Proc. 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS '01)*. Volume 2026 of *Lect. Notes in Comp. Science.*, San Francisco, Held in conjunction with IPDPS 2001. IEEE TCPP, Springer-Verlag (2001) 55–70
8. Prylli, L., Tourancheau, B., Westrelin, R.: The design for a high performance MPI implementation on the Myrinet network. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Proc. 6th European PVM/MPI Users' Group (EuroPVM/MPI '99)*. Volume 1697 of *Lect. Notes in Comp. Science.*, Barcelona, Spain, Springer Verlag (1999) 223–230
9. Bougé, L., Danjean, V., Namyst, R.: Improving Reactivity to I/O Events in Multithreaded Environments Using a Uniform, Scheduler-Centric API. In: *Euro-Par, Paderborn (Deutschland) (2002)*
10. Anderson, T., Bershad, B., Lazowska, E., Levy, H.: Scheduler activations: Efficient kernel support for the user-level management of parallelism. In: *Proc. 13th ACM Symposium on Operating Systems Principles (SOSP 91)*. (1991) 95–105
11. Williams, N.J.: An Implementation of Scheduler Activations on the NetBSD Operating System. In: *USENIX Annual Technical Conference*. (2002) 99–108
12. Haines, M.: On Designing Lightweight Threads for Substrate Software. In: *USENIX Technical Conference, Anaheim, CA, USENIX (1997)* 243–255

High-Speed Migration by Anticipative Mobility

Luk Stoops, Karsten Verelst, Tom Mens, and Theo D'Hondt

Department of Computer Science
Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium

{luk.stoops, karsten.verelst, tom.mens, tjdhondt}@vub.ac.be
<http://prog.vub.ac.be>

Abstract. In the advent of ambient intelligence, introducing ubiquitous mobile systems and services in general and mobile code in particular, network latency becomes a critical factor, especially in wireless, low-bandwidth environments. This paper investigates *anticipative mobility*, a high performance computing technique that exploits parallelism between loading and execution of applications to reduce network latency. The technique sends the mobile application code anticipative to the remote host, long before the actual migration is requested. Then, at the actual migration, we won't transfer the complete application anymore but only the delta of the current computational state with the already migrated computational state. Our experiments show that some applications can migrate in 2% of their original migration time. This allows applications to migrate very fast from host to host without a significant loss of execution time during the migration phase

1 Introduction

An emerging technique for distributed applications involves *mobile code*: code that can be transmitted across the network and executed on the receiver's platform. Mobile code comes in many forms and shapes. The code can be represented by machine code, allowing maximum execution speed on the target machine but thereby sacrificing platform independence. Alternatively, the code can be represented as bytecodes, which are interpreted by a virtual machine (as is the case for Java, Smalltalk and .Net). This approach provides platform independence, a vital property in worldwide heterogeneous networks.

An important problem related to mobile code is *network latency*: the time delay introduced by the network before the code can be executed. This delay has several possible causes (Table 1).

The code must be (1) halted, (2) packed (3) possibly transformed in a compressed and/or secure format, (4) transported over a network to the target platform, (5) possibly retransformed from its compression or security standard, (6) checked for errors and/or security constraints, (7) unpacked, (8) possibly adapted to the receiving host by compiling the byte codes or some other intermediate representation and finally (9) resumed.

Table 1. Migration steps

Step	Action
1	Halt the application
2	Pack it
3	Transform it
4	Transport to the receiver
5	Retransform it
6	Check it
7	Unpack it
8	Adapt it
9	Resume the application

A second important problem is *application availability*. In a classical migration scheme the application that migrates from host to host is temporarily halted and is restarted at the receiving host after the code is completely loaded and restored in its original form. During migration time the application is not available for other processes that need to interact with it. After it is halted it will become available again only when the migration process has completed.

In the advent of ubiquitous mobile applications, network latency and application availability are critical factors. This paper explores the technique of *anticipative mobility* to tackle both problems. Anticipative migration is a form of *progressive mobility*. Progressive mobility allows applications to migrate piece by piece thereby providing early startup at the receiving host and smooth evaluation without any disruption.

Anticipative mobility is developed as an optimization for a progressive mobility technique also known as *application streaming* [10, 11]. Application streaming is inspired by similar techniques of audio and video streaming. The main characteristic of these transmission schemes is that the processing of the digital stream is started long before the load phase is completed. A similar technique called *interlaced code loading* [9] exists where code arrives and starts executing on the receiving host computer in the same manner as interlaced image loading in a web browser. A disadvantage of the application streaming technique is that applications become temporarily distributed during the streaming phase which may slow down some types of applications. As we will show in this paper, anticipative migration avoids this temporary distribution phase thereby allowing the migrating application to run almost continuously at full speed. In this paper we demonstrate the feasibility of anticipative mobility by migrating an application in the Borg mobile agent environment [12].

How can we anticipate migration? In ambient intelligent environments by example, where hosts are moving to each other, one may foresee that an application will migrate to a host that comes physically in the neighborhood. Shopping agents [2] often know in advance their migration paths and time. And even if they plan their itinerary dynamically they are often able to decide for the next host to visit before they start the

actual work on the current host. Another possible application of mobile code is load balancing. Here too, the overload of a host can often be predicted as well as the host to flee to.

The paper is structured as follows. Section 2 presents some basic observations of current network and computer architectures and introduces the technique of progressive and anticipative mobility. Section 3 describes the basic technique in more detail. Section 4 reports on the experiment we have conducted to validate our approach. Section 5 formulates the conclusion and finally we present some related work.

2 Proposed Technique

2.1 Basic Observations, Assumptions, and Restrictions

A first important observation is that code transmission over a network is inherently slower than compilation and evaluation and this will remain the case for many years to come. The speed of wireless data communications has increased enormously over the last years and with technologies as HSCSD (High Speed Circuit Switched Data) and GPRS (General Packet Radio Services) we obtain transmission speeds of 2Mbps [1]. Compared with the raw “number crunching” power of microprocessors where processor speeds of Gbps are common, transmission speed is still several orders of magnitude slower. We expect that this will remain the case for several years to come since, according to Moore’s Law [8], CPU speeds are known to double every year. For this reason, transporting mobile code over a network is in general the most time-consuming activity, and can lead to significant delays in the migration of the application. This is especially the case in low-bandwidth environments such as wireless communication systems or in overloaded networks.

A second observation is that actual computer architectures provide separate processors for input/output (code loading) and main program execution. This enables us to send code to another host in parallel with the execution of the main application.

We assume two preconditions to apply anticipative migration in an efficient way:

1. The internal representation of the computational state is stored in one chunk of memory.
2. The internal representation of the program code remains constant.

Many implementations of programming languages, especially those that deploy garbage collection, store their code and computational state as one chunk of memory. In this chunk we find all the elements necessary for the execution of the program. Again for many language implementations, this is: a representation of the abstract grammar, a stack of some sort and a dictionary with the variable bindings.

The Borg environment, used for our experiments, represents the application code in an *abstract syntax tree*. The computational state of an application is represented by a *stack* (for continuations and intermediate evaluation values) and a *dictionary* that contains the names of the variables and references to a block of memory where the

values of these variables reside, called the *heap*. In the Borg environment these entities are contained in one chunk of memory.

The Borg environment also allows reflection. Reflection is the ability to reason and act upon itself [7], i.e. the ability for a program to manipulate its code and computational state as data during its execution. The expressiveness of Borg allows an application to reason over itself and its computational state during its evaluation. In our first experiments we will allow all kind of reflective behaviour except the adaptation of the abstract syntax tree. This will satisfy precondition 2. Adapting your own code during evaluation is not current practice in the majority of programming languages so this will not compromise our results.

If an executable component is migrated before the application has started it suffices to send over its code and start it up in the same manner as applets are loaded to a web browser and started. If, however, the application was already running before migration, one should send not only the bare code but also the intermediate values of the local variables of that evaluation unit and the information of the exact point in evaluation where the entity was stopped to be able to resume it at the same point. This extra information is usually referred to as *the computational state and runtime stack*. This kind of migration is known as *strong mobility* while the former is called *weak mobility* [4]. In the remainder of this paper we will refer to the computational state to indicate the values of all the variables of the application *including the runtime stack*.

2.2 Anticipative Mobility

The technique of anticipative migration applies a progressive migration scheme. The running application is split in two components: a snapshot of the complete application and the delta of the computational states after a certain time. Basically the technique is a five step process:

1. Take a snapshot of the running application, i.e. take a copy of the code and its computational state, on the sending host.
2. Copy the snapshot to the receiving host while the original application continues to run.
3. Once the copy has arrived at the receiving host halt the original application.
4. Define the changes, called the delta, build up in the original application during the copy phase of the snapshot. This delta contains the changes in the computational state.
5. Migrate and apply this delta to the, already migrated, snapshot and resume its evaluation.

Since each application contains parts that remain constant, the size of the delta will always be smaller than the size of the complete snapshot. This is where we gain in migration time. Suppose we know a few seconds in advance that we are going to migrate. At that moment we capture the complete application including its computational state and forward it already to the receiving host. Then, a few seconds later when we really want to migrate we identify the delta of the current computational state with the

already sent computational state and only transmit this delta across the network. As an example, we use the simple Borg factorial program.

```
fac(n): if (n=1, 1, n*fac(n-1))
fac(100)
```

As shown in Figure 1 we start by calculating $\text{fac}(100)$. Suppose that the application receives an external trigger to migrate at the start of the recursive call with parameter n equal to 2. As a result of this trigger the first 3 steps of the classic migration scheme are launched at Host 1. If we call CS the computational state; then we transfer over the network during step 4: ($\text{code} + \text{CS}_{\text{fac}(2)}$). After the transfer, steps 5..9 of the migration scheme allow the application to resume on Host 2.

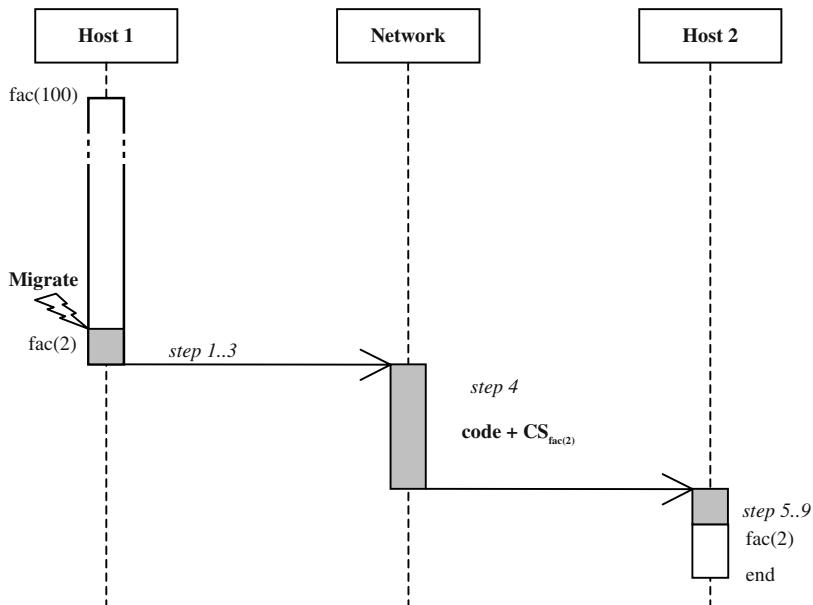


Fig. 1. Sequence diagram – classic strong migration of a factorial calculation

Figure 2 shows an example of anticipative migration of the same Borg factorial program:

We start again by calculating $\text{fac}(100)$. At $\text{fac}(10)$ we anticipatively migrate the bulk of the code and state but the application is not started at Host 2. As shown in the figure we assume that this migration process is able to run in parallel with the application itself. Then, at $\text{fac}(2)$, we transfer the delta and resume the application at Host 2. We note that the migration of the bulk of data $\text{code} + \text{CS}_{\text{fac}(10)}$ takes much longer than the small delta $\text{CS}_{\text{fac}(10)} - \text{CS}_{\text{fac}(2)}$. This allows the application to start up much sooner now at

Host 2, after the external trigger, resulting in the high-speed migration of the application.

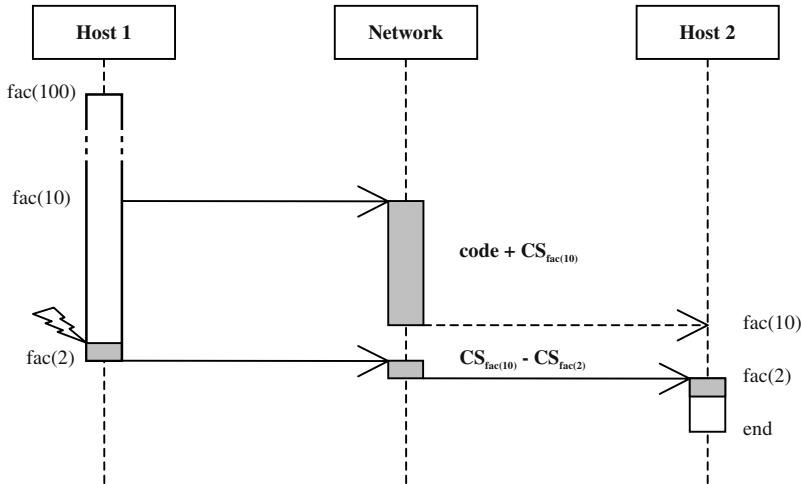


Fig. 2. Sequence diagram - anticipative migration of a factorial calculation

The most challenging part of the technique is the identification, extraction, presentation, migration and reapplication of the delta. The reapplication of the delta can be seen as the reverse process of the extraction so we will focus on the identification and extraction part.

3 Grabbing the Delta

To identify the delta we need to compute the difference between two computational states. Reflection, available in Borg, allows us to capture the current computational state at any time in the Borg language itself. To calculate the delta between two states in an efficient way we add a native function: *delta()* to Borg. This function is written in C, the implementation language of Borg.

Taking a snapshot of the computational state is basically achieved by computing the transitive closure of all elements starting from the root of the stack and the dictionary. This closure will contain complete arrays, and objects pointed to by this root. In Borg this closure will also contain the abstract syntax tree.

To explain the techniques to grab the delta in an efficient way we explore gradually more complex memory operations. To simplify the operations somewhat we assume that the garbage collector is disabled between the capturing of two computational states. We show later in this paper how to deal with garbage collection.

3.1 Non-destructive Memory Operations

Functional languages have the property that they never change the memory in a destructive way. Since there is no assignment operator available in a pure functional language the internal representation of the computational state will have certain properties that can be exploited to calculate the delta.

As in many garbage collected languages, the memory in Borg is allocated sequentially, as on a stack data structure, and since we know that no memory content will be overwritten, the difference in computational states will be the new allocated memory. If we compare the two computational states CS_1 and CS_2 , then the delta will exist of all memory allocated after the first state capture.

For easy identification of this memory block we add a watermark at the end of the CS_1 memory block. This allows the serializer (step 2 of the migration process) to identify the newly allocated memory as the memory after the watermark.

The old memory has been transferred to the receiving host, and when the new computational state has references to the old one, the pointers in memory must be adapted to point to correct addresses on the receiving host. This adaptation takes place during step 8 (Table 1) of the migration process.

3.2 Destructive Memory Operations

Imperative languages allow destructive changes in memory so the internal representation memory block can contain pointers in both directions (Figure 3). Even functional languages will sometimes use invisible destructive operations in memory for performance issues.

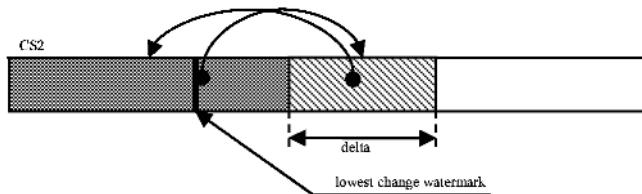


Fig. 3. Possible pointers and watermark in destructive memory

In this case, we will need to compare both memory blocks byte by byte to determine all the differences. Fortunately a stack is a friendly structure, it has no random access and once we find the 'lowest' change, we can be sure that everything above this point has changed too. Putting a watermark on the stack at this point makes it easy for the serializer to determine the part of memory to serialize.

3.3 Random Access Operations

We have discussed how to calculate the delta in non-destructive and destructive memory environments in a stack data structure, heavily used by modern compilers [5]. We

will now discuss how to handle migration and subtraction of computational states in random-access data structures.

When we wish to migrate a random-access structure we must explicitly keep track of the memory involved and changed. How to do this will be largely implementation dependent, but habitually memory can only be accessed through existing variables, so keeping a list of changed variables is often an option. Another alternative involves maintaining an array in which we explicitly keep track of all changed structures.

We could space-optimize this further by using dirty-bits, bits that flag a change of a memory word. In a garbage-collected programming environment we may find that some bits are already reserved for this kind of operation. Under the presumption that we won't garbage collect we can reuse some of the spare garbage collector bits as dirty bits. This restriction isn't so harsh as it seems, if a garbage collect were to trigger, we could just calculate the delta and migrate early. Also, because a garbage collection and a serialization or very similar processes, we could trigger a garbage collect together with the first migration.

3.4 Discussion

3.4.1 Expected Gain

If we run the factorial program and send our first snapshot at `fac(10)` and then really migrate at `fac(2)` we can compare the stack and dictionary at those moments.

Migration of the full stack and dictionary at `fac(2)` would consist of 297 entries: 99 stack, and 99 name/value pairs. When we migrate at `fac(10)` we would have to transfer $89 * 3$ entries. But if we then migrate at `fac(2)` and only need to serialize the entries created between `fac(10)` and `fac(2)`, this would be only $9 * 3 = 27$ entries. This leads to a time compression ratio of $27 / 296 \approx 9\%$.

This gain in time can be expected if we only take the delta between the stacks and dictionaries in account. In reality we also need to include the abstract syntax tree in our first snapshot which will increase our gain in time even more.

3.4.2 Non-recursive Applications

So far we have only looked at the factorial example. Such recursive functions explain clearly what happens to the stack but aren't very common in practice. In practice we expect more sequential function calls. So how does this affect the delta calculation? The answer is: very favorably. Consider the program.

```
main() : { fun100(); ...; fun10(); ...; fun2(); }
```

We start by calling `fun100()`, migrate anticipative at `fun10()` and migrate the delta at `fun2()`.

With each function call the stack expands. However after each function call, it shrinks again. Therefore, at `fun2()`, we will have a much smaller stack as compared to the growing stack in the previous factorial example. In fact it will only contain the data for the main function and for the `fun2` function. This implies that our delta will only contain the `fun2` data. Compared with the bulk of data anticipative sent, this delta is so small that very large time compressions may be expected.

A typical program won't behave like either of the two presented examples but will most likely have a performance situated somewhere between these two extremes. This claim can be supported by the fact that a typical program stack doesn't become very big.

4 Experiment

This is the Borg code (and resulting output) to calculate the size of the streams over the network.

```
a[100]:0;
fac(n):if(n<2,n,{t[n]:=call(cont); n*fac(n-1)});
fac(100);
d:delta(a[10], a[2]);
display(size(serialized(a[10])));
display(size(serialized(a[2])));
display(size(serialized(d)));
>: Size of serialized stream: 25664 bytes.
>: Size of serialized stream: 25408 bytes.
>: Size of serialized stream: 515 bytes.
```

We declare an array *a* of size 100 and then declare and run an instrumented factorial function: *fac()* that fills up the array with the consecutive computational states. The factorial function is instrumented with the native function call: *call(cont)* that returns the complete current computational state.

Then we apply the new native function *delta()* to calculate the delta *d* between $CS_{fac(10)}$ and $CS_{fac(2)}$. This function is written in C, the implementation language of Borg to be able to calculate the difference of two Borg computational states in an efficient way.

Finally we display the sizes of the serialized instances of the computational states $CS_{fac(10)}$, $CS_{fac(2)}$ and the size of the delta between them.

The delta between the computational states contains 515 bytes while the original computational state, including the abstract syntax tree was 25664 bytes. The reduction in size and migration time is $416 / 25664 = 2.01\%$ of the original stream size or a compression ratio of: 97.99%.

5 Conclusion

The technique presented here is very useful when we know in advance when we are going to migrate. If we manage to send the bulk of the application in advance to the receiving host we can, when the real time to migrate has come, obtain a high-speed migration of our running application in a fraction of the time needed for normal migration.

6 Related Work

Code compression is the most common way to reduce overhead introduced by network delay in mobile code environments. Several approaches to compression have been proposed. Ernst et al. [3] describe an executable representation that is roughly the same size as gzipped x86 programs and can be interpreted without decompression.

Exploiting parallelism is another way to reduce network latency. *Interlaced code loading* [9] is a technique that is inspired by interlaced image loading. The Interlaced Graphics Interchange Format (GIF) is an image format that exploits the combination of low bandwidth channels and fast processors by transmitting the image in successive waves of bit streams until the image appears at its full resolution. Interlaced code loading is a technique that applies the idea of progressive transmission to software code instead of images.

Profiling and prefetching: Krintz et al. [6] suggest splitting Java code (at class level) into hot and cold parts. The cold parts correspond to code that is never or rarely used, and hence loading of this code can be avoided or at least postponed. In our approach the code itself can be treated as the cold part which we will send in advance. The hot part will be the computational state or the delta of two computational states.

Acknowledgments. We like to thank Johan Fabry, Julian Down and Gert van Grootel for reviewing the paper, and the members of our lab team for their valuable comments.

References

1. S. Barberis: A CDMA-based radio interface for third generation mobile system. Mobile Networks and Applications Volume 2, Issue 1 ACM Press June 1997
2. A. Chavez, D. Dreilinger, R. Guttman, and P. Maes: A real-life experiment in creating an agent marketplace. Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, April 1997
3. J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, S. Lucco: Code Compression. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation. Volume 32 Issue 5, May 1997
4. Alfonso Fuggetta, Gian Pietro Picco and Giovanni Vigna: Understanding Code Mobility. IEEE Transactions of Software Engineering, volume 24, 1998
5. D. Grune, H. Bal, C. Jacobs, K. Langendoen: Modern Compiler Design. Worldwide Series in Computer Science John Wiley & Sons Ltd London 2000
6. C. Krintz, B. Calder, U. Hölzle: Reducing Transfer Delay Using Class File Splitting and Prefetching. Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications, November, 1999
7. P. Maes: Computational Reflection. PhD thesis, Vrije Universiteit Brussel, 1987.
8. G. Moore: Cramming more components onto integrated circuits. Electronics, Vol. 38(8), pp. 114–117, April 19, 1965.
9. L. Stoops, T. Mens, T. D'Hondt: Fine-Grained Interlaced Code Loading for Mobile Systems. 6th International Conference MA2002, LNCS 2535, pp. 78-92 Barcelona, Spain October 2002

10. L. Stoops, T. Mens, T. D'Hondt: Reducing Network Latency by Application Streaming. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, June 2003
11. L. Stoops, T. Mens, C. Devalez, T. D'Hondt: Migration Strategies for Application streaming. technical report 2003 ftp://prog.vub.ac.be/tech_report/2003/vub-prog-tr-03-06.pdf
12. W. Van Belle, J. Fabry, K. Verelst, T. D'Hondt: Experiences in Mobile Computing: The CBorg Mobile Multi Agent System. Tools Europe 2001, March 2001

Author Index

- Abella, Jaume 34
Agrawal, Banit 44
Agrawal, Gagan 74
Amaral, José Nelson 438
Attali, Isabelle 363

Babu, V. 428
Bak, Sangman 142
Barik, Rajkishore 290
Barli, Niko Demus 393
Baskar, G. 428
Baskiyar, Sanjeev 259
Baumgartner, Gerald 406
Bernholdt, David E. 406
Bhat, Viraj 373
Bhaya, Gaurav 152
Biswas, Rupak 383
Boppana, Rajendra V. 217
Boppana, Rajesh 217
Brodowicz, Maciej 2
Brunett, Sharon 2

Caromel, Denis 363
Casey, Will 204
Chalasani, Suresh 217
Chaudhary, Vipin 469
Chen, Yianxiao 480
Choppella, Venkatesh 406
Cociorva, Daniel 406
Contes, Arnaud 363

D'Hondt, Theo 500
Danjean, Vincent 490
Dhar, Subhankar 130
Dickinson, Christopher 259
Djomehri, M. Jared 383
Duwairi, B. 279

Eleftheriou, Maria 194
Esfahanian, A.-H. 228

Ferreto, Tiago C. 449
Fitch, Blake G. 194
Fujimoto, Noriyuki 162

Germain, Robert S. 194
Goddard, Wayne 66
González, Antonio 34
Gottschalk, T. D. 2
Govindarajan, R. 12
Gu, Qian-Ping 85

Hagihara, Kenichi 162
Hedetniemi, Stephen T. 66
Holte, Robert C. 438
Huang, Yun 238
Hung, Luong Dinh 393

Ino, Fumihiko 162
Iwama, Chitaka 393

Jacobs, David P. 66
Jayaram, B. 174
Jiang, Hai 469
Jiang, Hong 300
Jin, Ruoming 74

Kamakoti, V. 174
Kawasaki, Yasuhiro 162
Khan, Javed I. 311
Kim, Huesung 23
Krishnamoorthy, Sriram 406
Krishnan, Manojkumar 248
Krishnan, Sandhya 406
Kumar, A. Manoj 174
Kumar, Rajeev 343
Kumar, Umesh 290
Kurhekar, Manish P. 290

Lam, Chi-Chung 406
Leair, Mark 459
Lin, Suzhen 268
Liu, Yunhao 228
Lu, Enyue 55
Lysne, Olav 106, 118

Majumdar, Shikharesh 480
Mandal, Swarup 323
Manimaran, G. 268, 279
Manoj, B.S. 152, 333
Mehndiratta, S.L. 418
Mei, Alessandro 95

- Meira, Wagner Jr. 184
Mens, Tom 500
Mishra, Bud 1, 204
Mizutani, Yasuharu 162
Moreira, José E. 194
Murthy, C. Siva Ram 152, 333

Najjar, Walid 44
Namyst, Raymond 490
Ni, Lionel M. 228
Nieplocha, Jarek 248
Niewiadomski, Robert 438

Otey, Matthew Eric 184

Padala, Pradeep 353
Pai, Sukesh 130
Palem, Krishna V. 216
Panda, Dhabaleswar 248
Pande, Santosh 459
Parashar, Manish 373
Parthasarathy, Srinivasan 184
Patil, Abhishek 228
Patt, Yale 105
Pawar, Pravin Amrut 418

Qin, Xiao 300

Ramanujam, J. 406
Reinemmo, Sven-Arne 118
Rieck, Michael Q. 130
Rizzi, Romeo 95
Rose, César A.F. De 449

Sadayappan, P. 406
Saha, Debashis 323
Sakai, Shuichi 393
Sangireddy, Rama 23
Santhanaraman, Gopalakrishnan 248
Sarma, A. Radhika 12

Sasama, Toshihiko 162
Sato, Yoshinobu 162
Scott, David 310
Sem-Jacobsen, Frank Olaf 118
Skeie, Tor 118
Somani, Arun K. 23
Springer, Paul L. 2
Srimani, Pradip K. 66
Stenstrom, Per 405
Stoops, Luk 500
Strenski, David 428
Sunder, C. Shyam 428
Suresh, Dinesh C. 44
Swanson, David R. 300

Tamura, Shinichi 162
Tanaka, Hidehiko 393
Theiss, Ingebjørg 106
Tipparaju, Vinod 248

Upchurch, Ed 2

Veloso, Adriano 184
Venkatasubramanian, Nalini 238
Verelst, Karsten 500
Vidhyashankar, V. 333

Wang, Yong 85
Wilson, Joseph N. 353

Xiao, Li 228

Yanagawa, Yoshimitsu 393
Yang, Ge 74
Yang, Jun 44
Yang, Seung S. 311

Zhang, Chuanjun 44
Zheng, S.Q. 55
Zhu, Yifeng 300