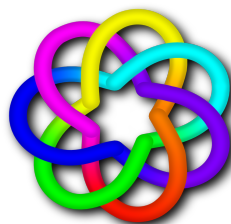


CONFERENCES IN RESEARCH AND PRACTICE IN  
INFORMATION TECHNOLOGY

VOLUME 122

# COMPUTER SCIENCE 2012

AUSTRALIAN COMPUTER SCIENCE COMMUNICATIONS, VOLUME 34, NUMBER 1





# COMPUTER SCIENCE 2012

Proceedings of the  
Thirty-Fifth Australasian Computer Science Conference  
(ACSC 2012), Melbourne, Australia,  
30 January – 3 February 2012

Mark Reynolds and Bruce Thomas, Eds.

Volume 122 in the Conferences in Research and Practice in Information Technology Series.  
Published by the Australian Computer Society Inc.



Published in association with the ACM Digital Library.

**Computer Science 2012.** Proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, 30 January – 3 February 2012

**Conferences in Research and Practice in Information Technology, Volume 122.**

Copyright ©2012, Australian Computer Society. Reproduction for academic, not-for-profit purposes permitted provided the copyright text at the foot of the first page of each paper is included.

Editors:

**Mark Reynolds**

School of Computer Science and Software Engineering  
Faculty of Engineering, Computing and Mathematics  
The University of Western Australia  
Crawley, WA 6009  
Australia  
Email: [mark@csse.uwa.edu.au](mailto:mark@csse.uwa.edu.au)

**Bruce Thomas**

School of Computer and Information Science  
Division of Information Technology, Engineering and the Environment  
University of South Australia  
Adelaide, SA 5001  
Australia  
Email: [bruce.thomas@unisa.edu.au](mailto:bruce.thomas@unisa.edu.au)

Series Editors:

Vladimir Estivill-Castro, Griffith University, Queensland  
Simeon J. Simoff, University of Western Sydney, NSW  
Email: [crpit@scm.uws.edu.au](mailto:crpit@scm.uws.edu.au)

Publisher: Australian Computer Society Inc.  
PO Box Q534, QVB Post Office  
Sydney 1230  
New South Wales  
Australia.

Conferences in Research and Practice in Information Technology, Volume 122.  
ISSN 1445-1336.  
ISBN 978-1-921770-03-6.

Printed, January 2012 by University of Western Sydney, on-line proceedings  
Printed, January 2012 by RMIT, electronic media  
Document engineering by CRPIT

The *Conferences in Research and Practice in Information Technology* series disseminates the results of peer-reviewed research in all areas of Information Technology. Further details can be found at <http://crpit.com/>.



## Table of Contents

### Proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, 30 January – 3 February 2012

Preface .....	vii
Programme Committee .....	viii
Organising Committee .....	ix
Welcome from the Organising Committee .....	x
CORE - Computing Research & Education .....	xi
ACSW Conferences and the Australian Computer Science Communications .....	xii
ACSW and ACSC 2012 Sponsors .....	xiv

### Contributed Papers

FEAS: A full-time event aware scheduler for improving responsiveness of virtual machines .....	3
<i>Denghui Liu and Jinli Cao</i>	
Boosting Instruction Set Simulator Performance with Parallel Block Optimisation and Replacement .	11
<i>Bradley Alexander, Sean Donnellan, Andrew Jeffrey, Travis Olds and Nicholas Sizer</i>	
On the parameterized complexity of dominant strategies .....	21
<i>Mahdi Parsa and Vlad Estivill-Castro</i>	
Single Feature Ranking and Binary Particle Swarm Optimisation Based Feature Subset Ranking for Feature Selection .....	27
<i>Bing Xue, Mengjie Zhang and Will N. Browne</i>	
On the Existence of High-Impact Refactoring Opportunities in Programs .....	37
<i>Jens Dietrich, Catherine McCartin, Ewan Tempero and Syed Ali Shah</i>	
Declarative Diagnosis of Floundering in Prolog .....	49
<i>Lee Naish</i>	
Learning Time Series Patterns by Genetic Programming .....	57
<i>Feng Xie, Andy Song and Vic Ciesielski</i>	
ERA Challenges for Australian University ICT .....	63
<i>Paul Bailes</i>	
Evolutionary Design of Optical Waveguide with Multiple Objectives .....	73
<i>Qiao Shi, Andy Song, Thach Nguyen and Arnan Mitchell</i>	
Real-time Evolutionary Learning of Cooperative Predator-Prey Strategies .....	81
<i>Mark Wittkamp, Luigi Barone, Phil Hingston and Lyndon While</i>	
Trends in Suffix Sorting: A Survey of Low Memory Algorithms .....	91
<i>Jasbir Dhaliwal, Simon J. Puglisi and Andrew Turpin</i>	
Spectral debugging: How much better can we do? .....	99
<i>Lee Naish, Kotagiri Ramamohanarao and Hua Jie Lee</i>	

Importance of Single-Core Performance in the Multicore Era .....	107
<i>Toshinori Sato, Hideki Mori, Rikiya Yano and Takanori Hayashida</i>	
Explaining alldifferent .....	115
<i>Nicholas Downing, Thibaut Feydy and Peter Stuckey</i>	
<b>Author Index</b> .....	125

## Preface

The Australasian Computer Science Conference (ACSC) series is an annual forum, bringing together research sub-disciplines in Computer Science. The meeting allows academics and other researchers to discuss research topics as well as progress in the field, and policies to stimulate its growth. This volume contains papers being presented at the Thirty-Fifth ACSC in Melbourne, Australia. ACSC 2012 is part of the Australasian Computer Science Week which runs from January 30 to February 2, 2012.

The ACSC 2012 call for papers solicited 38 submissions from Australia, New Zealand, Iran, Germany, India, Japan, China and Thailand. The topics addressed by the submitted papers illustrate the broadness of the discipline. These included algorithms, virtualisation, software visualisation, databases, constraint programming and PROLOG, to name just a few.

The programme committee consisted of 29 highly regarded academics from Australia, New Zealand, China, Japan, Canada, France, Italy, and USA. All papers were reviewed by at least three programme committee members, and, in some cases, external reviewers. Of the 38 papers submitted, 14 (or 37%) were selected for presentation at the conference.

The Programme Committee determined that the "Best Paper Award" should go to Mahdi Parsa and Vlad Estivill-Castro for their paper entitled "On the parameterized complexity of dominant strategies". Congratulations.

We thank all authors who submitted papers and all conference participants for helping to make the conference a success. We also thank the members of the programme committee and the external referees for their expertise in carefully reviewing the papers. We are grateful to Professor Simeon Simoff from UWS representing CRPIT for his assistance in the production of the proceedings. We thank Professor Tom Gedeon (President) and Dr Alex Potanin (Conference Coordinator) for their support representing CORE (the Computing Research and Education Association of Australasia).

Thanks to the School of Computer Science and Software Engineering at The University of Western Australia for web support for advertising and refereeing for the conference.

Last, but not least, we express gratitude to our hosts at the RMIT University in Melbourne and, in particular, James Harland.

**Mark Reynolds**

University of Western Australia

**Bruce Thomas**

University of South Australia

ACSC 2012 Programme Chairs

January 2012

# Programme Committee

## Chairs

Mark Reynolds, University of Western Australia  
Bruce Thomas, University of South Australia

## Members

Carole Adam, RMIT University (Australia)  
Stephane Bressan, National University of Singapore (Singapore)  
Fred Brown, University of Adelaide (Australia)  
Rajkumar Buyya, University of Melbourne (Australia)  
Curtis Dyreson, Utah State University (USA)  
Ansgar Fehnker, NICTA (Australia)  
Henry Gardner, Australian National University (Australia)  
Ken Hawick, Massey University - Albany (New Zealand)  
Michael Houle, National Institute for Informatics (Japan)  
Zhiyi Huang, University of Otago (New Zealand)  
Paddy Krishnan, Bond University (Australia)  
Chiou-Peng Lam, Edith Cowan University (Australia)  
Jiuyong Li, University of South Australia (Australia)  
Chris McDonald, University of Western Australia (Australia)  
Tanja Mitrovic, Canterbury University (New Zealand)  
Linda Pagli, University of Pisa (Italy)  
Maurice Pagnucco, University of New South Wales (Australia)  
Alex Potanin, Victoria University of Wellington (New Zealand)  
Yuping Shen, Sun Yat-Sen University, Guangzhou, (China)  
Markus Stumptner, University of South Australia (Australia)  
Xiaoming Sun, Tsinghua University (China)  
David Toman, University of Waterloo (Canada)  
Andrew Turpin, University of Melbourne (Australia)  
Hua Wang, University of Southern Queensland (Australia)  
Burkhard Wuensche, University of Auckland (New Zealand)  
Masafumi Yamashita, Kyushu University (Japan)  
Xiaofang Zhou, University of Queensland (Australia)

## Additional Reviewers

Craig Anslow	Jiwei Jin	David Pearce
Muzammil Mirza Baig	Tuze Kuyucu	A.H.M. Sarowar Sattar
James Birt	Thuc Duy Le	Lili Sun
Quan Chen	Kai-Cheung Leung	Antoine Veillard
Shiguang Feng	Huawen Liu	Hiroshi Wada
Colin Fyfe	Baljeet Malhotra	
Lindsay Groves	Stuart Marshall	
Dirk S. Hovorka	Krzysztof Onak	

# Organising Committee

## Members

Dr. Daryl D'Souza  
Assoc. Prof. James Harland (Chair)  
Dr. Falk Scholer  
Dr. John Thangarajah  
Assoc. Prof. James Thom  
Dr. Jenny Zhang

# Welcome from the Organising Committee

On behalf of the Australasian Computer Science Week 2012 (ACSW2012) Organising Committee, we welcome you to this year's event hosted by RMIT University. RMIT is a global university of technology and design and Australia's largest tertiary institution. The University enjoys an international reputation for excellence in practical education and outcome-oriented research. RMIT is a leader in technology, design, global business, communication, global communities, health solutions and urban sustainable futures. RMIT was ranked in the top 100 universities in the world for engineering and technology in the 2011 QS World University Rankings. RMIT has three campuses in Melbourne, Australia, and two in Vietnam, and offers programs through partners in Singapore, Hong Kong, mainland China, Malaysia, India and Europe. The University's student population of 74,000 includes 30,000 international students, of whom more than 17,000 are taught offshore (almost 6,000 at RMIT Vietnam).

We welcome delegates from a number of different countries, including Australia, New Zealand, Austria, Canada, China, the Czech Republic, Denmark, Germany, Hong Kong, Japan, Luxembourg, Malaysia, South Korea, Sweden, the United Arab Emirates, the United Kingdom, and the United States of America.

We hope you will enjoy ACSW2012, and also to experience the city of Melbourne.,

Melbourne is amongst the world's most liveable cities for its safe and multicultural environment as well as well-developed infrastructure. Melbourne's skyline is a mix of cutting-edge designs and heritage architecture. The city is famous for its restaurants, fashion boutiques, café-filled laneways, bars, art galleries, and parks.

RMIT's city campus, the venue of ACSW2012, is right in the heart of the Melbourne CBD, and can be easily accessed by train or tram.

ACSW2012 consists of the following conferences:

- Australasian Computer Science Conference (ACSC) (Chaired by Mark Reynolds and Bruce Thomas)
- Australasian Database Conference (ADC) (Chaired by Rui Zhang and Yanchun Zhang)
- Australasian Computer Education Conference (ACE) (Chaired by Michael de Raadt and Angela Carbone)
- Australasian Information Security Conference (AISC) (Chaired by Josef Pieprzyk and Clark Thorburn)
- Australasian User Interface Conference (AUIC) (Chaired by Haifeng Shen and Ross Smith)
- Computing: Australasian Theory Symposium (CATS) (Chaired by Julián Mestre)
- Australasian Symposium on Parallel and Distributed Computing (AusPDC) (Chaired by Jinjun Chen and Rajiv Ranjan)
- Australasian Workshop on Health Informatics and Knowledge Management (HIKM) (Chaired by Keryn Butler-Henderson and Kathleen Gray)
- Asia-Pacific Conference on Conceptual Modelling (APCCM) (Chaired by Aditya Ghose and Flavio Ferrarotti)
- Australasian Computing Doctoral Consortium (ACDC) (Chaired by Falk Scholer and Helen Ashman)

ACSW is an event that requires a great deal of co-operation from a number of people, and this year has been no exception. We thank all who have worked for the success of ACSE 2012, including the Organising Committee, the Conference Chairs and Programme Committees, the RMIT School of Computer Science and IT, the RMIT Events Office, our sponsors, our keynote and invited speakers, and the attendees.

Special thanks go to Alex Potanin, the CORE Conference Coordinator, for his extensive expertise, knowledge and encouragement, and to organisers of previous ACSW meetings, who have provided us with a great deal of information and advice. We hope that ACSW2012 will be as successful as its predecessors.

**Assoc. Prof. James Harland**

School of Computer Science and Information Technology, RMIT University

ACSW2012 Chair

January, 2012

# CORE - Computing Research & Education

CORE welcomes all delegates to ACSW2012 in Melbourne. CORE, the peak body representing academic computer science in Australia and New Zealand, is responsible for the annual ACSW series of meetings, which are a unique opportunity for our community to network and to discuss research and topics of mutual interest. The original component conferences - ACSC, ADC, and CATS, which formed the basis of ACSW in the mid 1990s - now share this week with seven other events - ACE, AISC, AUIC, AusPDC, HIKM, ACDC, and APCCM, which build on the diversity of the Australasian computing community.

In 2012, we have again chosen to feature a small number of keynote speakers from across the discipline: Michael Kölling (ACE), Timo Ropinski (ACSC), and Manish Parashar (AusPDC). I thank them for their contributions to ACSW2012. I also thank invited speakers in some of the individual conferences, and the two CORE award winners Warwish Irwin (CORE Teaching Award) and Daniel Frampton (CORE PhD Award). The efforts of the conference chairs and their program committees have led to strong programs in all the conferences, thanks very much for all your efforts. Thanks are particularly due to James Harland and his colleagues for organising what promises to be a strong event.

The past year has been very turbulent for our disciplines. We tried to convince the ARC that refereed conference publications should be included in ERA2012 in evaluations – it was partially successful. We ran a small pilot which demonstrated that conference citations behave similarly to but not exactly the same as journal citations - so the latter can not be scaled to estimate the former. So they moved all of Field of Research Code 08 “Information and Computing Sciences” to peer review for ERA2012. The effect of this will be that most Universities will be evaluated at least at the two digit 08 level, as refereed conference papers count towards the 50 threshold for evaluation. CORE’s position is to return 08 to a citation measured discipline as soon as possible.

ACSW will feature a joint CORE and ACDICT discussion on Research Challenges in ICT, which I hope will identify a national research agenda as well as priority application areas to which our disciplines can contribute, and perhaps opportunity to find international multi-disciplinary successes which could work in our region.

Beyond research issues, in 2012 CORE will also need to focus on education issues, including in Schools. The likelihood that the future will have less computers is small, yet where are the numbers of students we need?

CORE’s existence is due to the support of the member departments in Australia and New Zealand, and I thank them for their ongoing contributions, in commitment and in financial support. Finally, I am grateful to all those who gave their time to CORE in 2011; in particular, I thank Alex Potanin, Alan Fekete, Aditya Ghose, Justin Zobel, and those of you who contribute to the discussions on the CORE mailing lists. There are three main lists: csprofs, cshods and members. You are all eligible for the members list if your department is a member. Please do sign up via <http://lists.core.edu.au/mailman/listinfo> - we try to keep the volume low but relevance high in the mailing lists.

**Tom Gedeon**

President, CORE  
January, 2012

# ACSW Conferences and the Australian Computer Science Communications

The Australasian Computer Science Week of conferences has been running in some form continuously since 1978. This makes it one of the longest running conferences in computer science. The proceedings of the week have been published as the *Australian Computer Science Communications* since 1979 (with the 1978 proceedings often referred to as *Volume 0*). Thus the sequence number of the Australasian Computer Science Conference is always one greater than the volume of the Communications. Below is a list of the conferences, their locations and hosts.

**2013.** Volume 35. Host and Venue - University of South Australia, Adelaide, SA.

**2012. Volume 34. Host and Venue - RMIT University, Melbourne, VIC.**

**2011.** Volume 33. Host and Venue - Curtin University of Technology, Perth, WA.

**2010.** Volume 32. Host and Venue - Queensland University of Technology, Brisbane, QLD.

**2009.** Volume 31. Host and Venue - Victoria University, Wellington, New Zealand.

**2008.** Volume 30. Host and Venue - University of Wollongong, NSW.

**2007.** Volume 29. Host and Venue - University of Ballarat, VIC. First running of HDKM.

**2006.** Volume 28. Host and Venue - University of Tasmania, TAS.

**2005.** Volume 27. Host - University of Newcastle, NSW. APBC held separately from 2005.

**2004.** Volume 26. Host and Venue - University of Otago, Dunedin, New Zealand. First running of APCCM.

**2003.** Volume 25. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Adelaide Convention Centre, Adelaide, SA. First running of APBC. Incorporation of ACE. ACSAC held separately from 2003.

**2002.** Volume 24. Host and Venue - Monash University, Melbourne, VIC.

**2001.** Volume 23. Hosts - Bond University and Griffith University (Gold Coast). Venue - Gold Coast, QLD.

**2000.** Volume 22. Hosts - Australian National University and University of Canberra. Venue - ANU, Canberra, ACT. First running of AUIC.

**1999.** Volume 21. Host and Venue - University of Auckland, New Zealand.

**1998.** Volume 20. Hosts - University of Western Australia, Murdoch University, Edith Cowan University and Curtin University. Venue - Perth, WA.

**1997.** Volume 19. Hosts - Macquarie University and University of Technology, Sydney. Venue - Sydney, NSW. ADC held with DASFAA (rather than ACSW) in 1997.

**1996.** Volume 18. Host - University of Melbourne and RMIT University. Venue - Melbourne, Australia. CATS joins ACSW.

**1995.** Volume 17. Hosts - Flinders University, University of Adelaide and University of South Australia. Venue - Glenelg, SA.

**1994.** Volume 16. Host and Venue - University of Canterbury, Christchurch, New Zealand. CATS run for the first time separately in Sydney.

**1993.** Volume 15. Hosts - Griffith University and Queensland University of Technology. Venue - Nathan, QLD.

**1992.** Volume 14. Host and Venue - University of Tasmania, TAS. (ADC held separately at La Trobe University).

**1991.** Volume 13. Host and Venue - University of New South Wales, NSW.

**1990.** Volume 12. Host and Venue - Monash University, Melbourne, VIC. Joined by Database and Information Systems Conference which in 1992 became ADC (which stayed with ACSW) and ACIS (which now operates independently).

**1989.** Volume 11. Host and Venue - University of Wollongong, NSW.

**1988.** Volume 10. Host and Venue - University of Queensland, QLD.

**1987.** Volume 9. Host and Venue - Deakin University, VIC.

**1986.** Volume 8. Host and Venue - Australian National University, Canberra, ACT.

**1985.** Volume 7. Hosts - University of Melbourne and Monash University. Venue - Melbourne, VIC.

**1984.** Volume 6. Host and Venue - University of Adelaide, SA.

**1983.** Volume 5. Host and Venue - University of Sydney, NSW.

**1982.** Volume 4. Host and Venue - University of Western Australia, WA.

**1981.** Volume 3. Host and Venue - University of Queensland, QLD.

**1980.** Volume 2. Host and Venue - Australian National University, Canberra, ACT.

**1979.** Volume 1. Host and Venue - University of Tasmania, TAS.

**1978.** Volume 0. Host and Venue - University of New South Wales, NSW.



## Conference Acronyms

<b>ACDC</b>	Australasian Computing Doctoral Consortium
<b>ACE</b>	Australasian Computer Education Conference
<b>ACSC</b>	Australasian Computer Science Conference
<b>ACSW</b>	Australasian Computer Science Week
<b>ADC</b>	Australasian Database Conference
<b>AISC</b>	Australasian Information Security Conference
<b>AUIC</b>	Australasian User Interface Conference
<b>APCCM</b>	Asia-Pacific Conference on Conceptual Modelling
<b>AusPDC</b>	Australasian Symposium on Parallel and Distributed Computing (replaces AusGrid)
<b>CATS</b>	Computing: Australasian Theory Symposium
<b>HIKM</b>	Australasian Workshop on Health Informatics and Knowledge Management

Note that various name changes have occurred, which have been indicated in the Conference Acronyms sections in respective CRPIT volumes.

## ACSW and ACSC 2012 Sponsors

We wish to thank the following sponsors for their contribution towards this conference.



CORE - Computing Research and Education,  
[www.core.edu.au](http://www.core.edu.au)



RMIT University,  
[www.rmit.edu.au/](http://www.rmit.edu.au/)



**AUSTRALIAN  
COMPUTER  
SOCIETY**

Australian Computer Society,  
[www.acs.org.au](http://www.acs.org.au)



THE UNIVERSITY OF  
WESTERN AUSTRALIA  
University of Western Australia,  
[www.uwa.edu.au](http://www.uwa.edu.au)

# CONTRIBUTED PAPERS



# FEAS: A full-time event aware scheduler for improving responsiveness of virtual machines

**Denghui Liu, Jinli Cao**

Department of Computer Science & Computer  
Engineering, Engineering & Mathematical Sciences  
La Trobe University, Melbourne Victoria 3086, Australia

d8liu@students.latrobe.edu.au,  
j.cao@latrobe.edu.au

**Jie Cao**

Jiangsu Provincial Key Laboratory of E-Business,  
Nanjing University of Finance and Economics  
Nanjing, China

caojie690929@163.com

## Abstract

Due to the advances in software and hardware support for virtualisation, virtualisation technology has been adapted for server consolidation and desktop virtualisation to save on capital and operating costs. The basic abstraction layer of software that virtualises hardware resources and manages the execution of virtual machines is called virtual machine monitor (VMM). A critical part of VMM is the CPU scheduler which slices and dispatches physical CPU time to virtual machines. Xen's credit scheduler utilised blocked-to-boosted mechanism to achieve low latency on I/O intensive tasks. However, it suppresses event notifications for the guest domain that is not blocked. This may delays the response of a guest domain doing mixed workloads, as its virtual CPU is seldom blocked when processing CPU-intensive tasks. We enhance the credit scheduler by making it full-time aware of inter-domain events and physical interrupt request events. Our proposed scheduler not only improves the responsiveness of domains doing mixed workloads, but also minimises the possibly caused scheduling unfairness. The experimental evaluation demonstrates the benefits of our proposed scheduler.

**Keywords:** Virtual machine, Xen, Paravirtualization.

## 1 Introduction

Virtualisation technology involves the virtualisation of several critical parts of a computer, such as CPU, memory, network and storage. It partitions the underlying physical resources and makes them shared among multiple virtual machines (VMs) (or domains) either by assigning a portion of physical resources to each VM (e.g. hard disk) or by switching from one VM to another in a very short time frame to use the physical resources in turns (e.g. CPU). These VMs run in parallel on a single physical machine under the control of virtual machine monitor (VMM) and they can have different operating systems.

Virtualisation technology opens up the possibility of server consolidation which increases the efficient use of server resources by consolidating multiple servers running different operation systems onto a single physical server. Desktop virtualization is another major application of the

virtualisation technology, in which case users only need a thin client to display the desktop interface locally while have all backend processing done in a dedicated VM that resides remotely in the Cloud. Both situations involve the execution of CPU intensive tasks and I/O intensive tasks, and often need to process a mix of both kinds at one time. The complexity of workloads makes it a great challenge for the VMM scheduler to maximise throughput and minimise latency while ensuring fairness.

This paper is based on the observation of Xen 4.0.1 (Xen 2011) platform. Its default scheduler, named credit scheduler, employs the BOOST mechanism to achieve low I/O response latency which works reasonably well when VMs have relatively monotonous workloads. The schedulable entities of a VM are the virtual CPUs (VCPUs) it has. The priority of an idle VCPU is boosted to get an immediate execution when it receives an event. This allows VMs performing I/O tasks to achieve lower response latency. However, the responsiveness of a VM diverges if it also does CPU intensive tasks at the same time. A VCPU waiting in the run queue does not get properly boosted when it receives an incoming event. The event notification is suppressed and thus has no effect on the scheduling. This might make the event sender wait unnecessarily and delay the following jobs.

An enhanced version of credit scheduler is presented in this paper to improve the responsiveness of busy VMs by taking advantage of Xen's split driver model and even channels. The device driver in Xen is split into two portions. Domain 0 or a dedicated driver domain hosts the front portion that directly interacts with the device, and the other portion resides in unprivileged guest domains. These two parts notify each other of waiting data using the Xen event channel mechanism and exchanged data via the I/O ring mechanism. The proposed scheduler monitors the events sent across VMM and boosts the runnable VCPUs receiving events originating from another domain or physical interrupt requests (PIRQs). To complement credit scheduler, the proposed scheduler prioritises not only blocked VCPUs but also runnable ones, and is called full-time event aware scheduler (FEAS). A VM processing mixed workloads can greatly benefits from prompt scheduling upon receiving an incoming event, particularly if it is an I/O related event.

The rest of this paper is organized as follows. Section 2 discusses previous research on VM scheduling and relates the virtual-machine monitor Xen. Section 3 presents the design of our proposed scheduler for Full-time event aware scheduling. Some experimental tests have been conducted to verify/demonstrate the performance

improvements and the analysis of results has also been included in Section 4. Finally, the conclusions and future work are presented in Section 5.

## 2 Related work

This section firstly discusses previous research on VM scheduling, then describes the architecture of Xen's split driver model and its credit scheduler.

### 2.1 VM scheduling

Three different VM schedulers have been introduced over the course of Xen's history, which are Borrowed Virtual Time (BVT) scheduler, Simple Earliest Deadline First (SEDF) scheduler, and Credit Scheduler. All these three are Proportional Share schedulers which allocate CPU in proportion to the VMs' weight. Cherkasova et al. (2007) comprehensively analysed and compared the impacts of schedulers and their respective scheduler parameters on the performance of I/O intensive applications running on virtual machines.

Ongaro et al. (2008) study the impact of the credit scheduler with various configurations on the performances of guest domains concurrently running a mixed workload of processor-intensive, bandwidth-intensive, and latency-sensitive applications. They suggest in their work that latency-sensitive applications should be placed in their own VMs to achieve the best performance. The purpose of our paper is to address this problem.

Govindan et al. (2009)'s communication-aware scheduler monitors the I/O ring and preferentially schedules the VMs that receive more data packets or are anticipated to send more data packets. However, the scheduler relies on accumulating the number of packets received or sent over a certain period of time and does not provide the immediate response to an incoming event.

Kim et al. (2009) made scheduler task aware by using the gray-box knowledge. The scheduler infers the guest-level I/O tasks by identifying the tasks using the CR3 register and then monitoring their time slices. A task is considered to be an I/O task based on two grey-box criteria: it immediately pre-empts the running task if the guest VM receives an event and its time slice is short. However, classifying the tasks just based on the CPU usage is not enough (Xia et al., 2009).

Xia et al. (2009) propose a pre-emption aware scheduling (PaS) interface. Same with our scheduler, PaS also improves the responsiveness of busy VMs by allowing the VCPU to pre-empt when an event is pending while it is waiting in the run queue. But in that approach the event channels on which the pre-empting condition is based need to be pre-known and registered to the guest kernel.

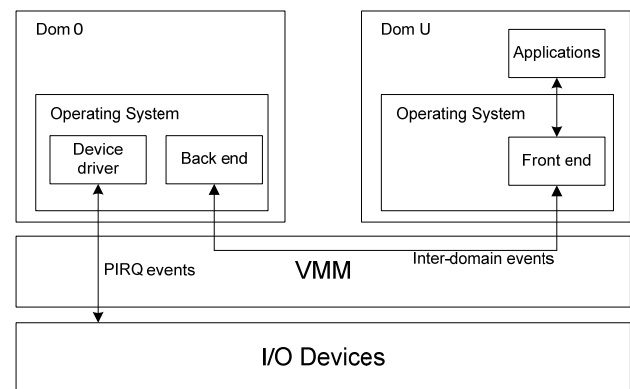
### 2.2 Xen and Split driver model

Xen 4.0.1 is used in our research. Xen adopts the paravirtualization approach and its guest operating systems require modifications to be able to run on the Xen platform. The Xen hypervisor sits between the hardware and the co-existing virtual machines. It has full control over hardware resources and dispatches them to different VMs according to a set of predefined rules.

Device drivers are the essential software for any operating system to communicate with physical hardware.

Xen's split driver model divides the device driver into two portions, the front end and the back end (Figure 1). The back end handles the physical device and the front end acts as the proxy of the back end. The back end is typically in Domain 0 but sometimes in a dedicated driver domain. Unprivileged guest domains have the front end with which they can accomplish a network or disk request.

The two portions of device driver notify each other of critical events using event channels and pass messages using I/O ring buffers. Event channel is the primitive notification mechanism within Xen. When the remote domain is busy or yet to be scheduled, an event is asynchronously delivered from its source to it to indicate the relevant event on the source domain. The ring buffers are implemented in the shared memory pages shared by both driver ends. Front end and back end exchange data by sending over the memory addresses of data pages rather than doing a full copy. This zero-copy feature enables fast message passing and consequently fast I/O.



**Figure 1: Xen split driver model**

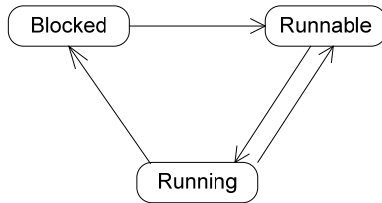
Based on the source of events, there are four types of events sent over the event channel. They are physical interrupt request (PIRQ) events, virtual IRQ (VIRQ) events, inter-domain events and intra-domain events. PIRQ events are mapped to the real IRQs of various physical devices. As an incoming IRQ generated by a device is likely not for the currently running domain, its corresponding PIRQ event is enqueued on the target domain and then processed when the domain is scheduled. Only privileged domains, such as domain 0 and driver domains, can handle PIRQ events. VIRQs are related to virtual devices created by Xen, like the timer virtual device. The main use of inter-domain events is for the front end and the back end of paravirtualised devices to notify each other of waiting data. Intra-domain events are a special case of inter-domain events where the events are delivered between the VCPUs of a single domain.

### 2.3 Credit scheduler

The current default scheduler of Xen is the credit scheduler. It allocates fair shares of processor resources to guest domains. Each slice of physical CPU time is weighted by a certain number of credits. Thus, if domains receive the same number of credits, they should expect an equal amount of CPU time.

Each VCPU's state and credits are managed and scheduled separately. The VCPU's credit balance can be positive or negative, and correspondingly its state can be

*under* or *over*. VCPUs trade credits for CPU time. A tick interrupt is triggered every 10ms. At each tick event, the currently running domain is debited some credits for the period it has run. An accounting event occurs every 30ms. During the accounting process, VCPUs are recharged with credits proportional to their weights. Their states are adjusted accordingly. The *under* state is assigned for VCPUs with positive credit balance while the *over* state for the ones with negative credit balance. To fully utilise available CPU resources, the accounting process caps VCPU's credits at an amount that is worth one rotation's CPU time slice. If a VCPU's accumulating credits exceeds the cap, it is marked inactive and will not receive any more credits until it is active again. Its credits are forfeited and shared by other active VCPUs. A scheduling event occurs when a scheduling decision is needed, which triggers a function that firstly refreshes the current VCPU's credit balance based on how long it has run and then decides the next VCPU to be scheduled. The order of scheduling VCPUs is based on their priorities. VCPUs with the *under* priority are always run before those with the *over* priority. VCPUs with same priority are scheduled in a round robin manner. The scheduled VCPU is allowed to run for 30ms or until pre-empted by other VCPUs with higher priority whichever comes first. If a running VCPU runs out of credits during its scheduled interval, it will not spontaneously yield the CPU. Contrarily, it will continue running and its credit simply goes negative.



**Figure 2: Run state transitions**

A VCPU in Xen could be in one of the following four possible run states, *running*, *runnable*, *blocked* and *offline*. The VCPU that is currently running on a physical CPU is in the *running* state. Since multiple VCPUs share a limited number of CPUs, VCPUs might not be scheduled on any physical CPU as soon as they become runnable. These VCPUs in the *runnable* state are essentially waiting in a queue for their turn. In a conventional system an idle thread occupies the physical CPU when there are no jobs to do. A virtualised system avoids such a waste by letting other busy VCPUs have the unused CPU time when some are idle. Idle VCPUs are given a *blocked* state. An *offline* VCPU is neither runnable nor blocked. Typically it is paused by the administrator. Figure 2 depicts the transition of the states that are tightly related to scheduling.

When an event comes, credit scheduler wakes a blocked VCPU and puts it back to the run queue. Furthermore, if the waken VCPU is in *under* priority, its priority is promoted to *boost*. So it is high likely that the boosted VCPU pre-empts the running one and gets scheduled immediately. This function is carried out by the boost module which lowers the latency of I/O related tasks. However, this only benefits blocked VCPUs with positive credits. A runnable VCPU receiving an I/O related event cannot be promptly scheduled. This often

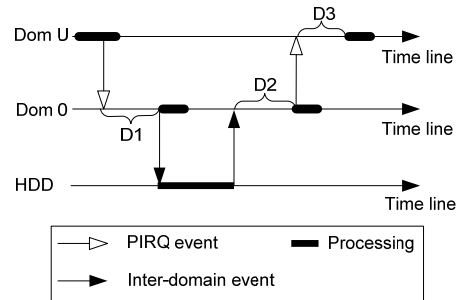
happens with the domains doing mixed workloads of CPU intensive tasks and I/O intensive tasks.

### 3 Full-time event aware scheduling

This section firstly identifies the problems that cause long latency of I/O tasks. Then the full-time event aware scheduler is proposed and detailed.

#### 3.1 Scheduling delays

Figure 3 shows the typical delays happening within a disk reading process. Delay D1 and D2 are associated with the scheduling of Domain 0. D1 is the duration between when the Domain U sends a reading request to Domain 0 and when Domain 0 is scheduled to actually send the reading IRQ to the hard disk. D2 is the duration between when the hard disk notifies Domain 0 the data to be retrieved is ready and when Domain 0 gets scheduled to set up the I/O ring and then notifies the Domain U. D3 happens on the Domain U side. It is the duration between when Domain 0 sends the notification and when Domain U is scheduled to finish the data reading process. A data writing process is similar.



**Figure 3: Scheduling delays within a disk reading process**

All aforementioned scheduling delays can be reduced by scheduling the events' respective target domains soon after the events are received. The events of interest are either inter-domain events or PIRQ events (Chisnall, 2007). The current credit scheduler boosts the VCPU upon an incoming event call only when the VCPU is blocked and has not consumed more than its fair share of CPU time. In other cases, event calls get suppressed and have no effect on the scheduling. Our enhanced version of credit scheduler makes the scheduler always aware of event notifications, in other words, the scheduler is full-time event aware. Once detecting an eligible incoming event the scheduler changes the course of scheduling accordingly.

#### 3.2 FEAS design

In FEAS every physical CPU has two additional queues: *immediate queue* and *postponed queue*, complementing the original run queue (Refer to Figure 4). VCPUs on the immediate queue are always preferentially scheduled over those on the run queue. The postponed queue is the temporary depository of the VCPUs that need to be scheduled as soon as possible but they have been scheduled more often than allowed. Once a designated trigger fires, postponed queue is swapped with immediate queue. In other words, postponed queue becomes immediate queue and the previous immediate queue

becomes the new postponed queue. Therefore, postponed VCPUs can get preferential scheduling after the trigger fires. The VCPU retains its position in run queue when joining or leaving immediate queue or postponed queue. VCPUs on the run queue are executed in round robin fashion. In such way the time slice VCPUs receive each rotation can be tightly controlled, and the side-effects caused on scheduling fairness by frequent pre-emption are kept minimal.

In general, similar with credit scheduler, FEAS also consists of three parts, namely *en-queuing*, *queue processing* and *de-queuing*. Figure 4 depicts the structure of FEAS. VCPUs that need to run are en-queued. Only runnable VCPUs can join queues and wait to be executed by CPU. Thus, *blocked* VCPUs that receive incoming events are set to be *runnable* beforehand. This process is called being *waken*. Credit scheduler boosts waken VCPUs by given them a *boost* priority to lower response latency because incoming events are often I/O related. FEAS prioritises VCPUs differently. It boosts VCPUs by assigning them to immediate queue or postponed queue (Explained in Sec. 3.2.2). Queue processing takes on heavy workloads off the de-queuing process since de-queuing process is in the critical path and meant to be fast. After queue processing, the de-queuing process usually de-queues the head VCPU in a queue straightaway and scheduled it to run next.

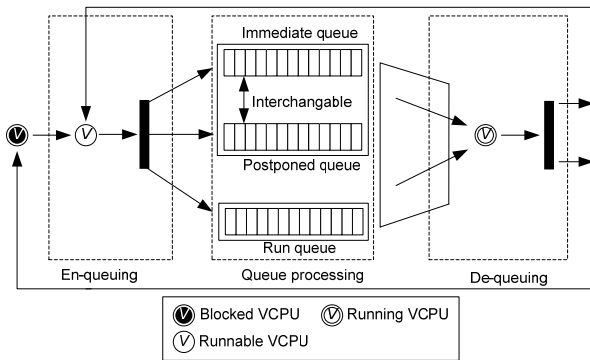


Figure 4: FEAS structure

The proposed scheduler involves three modules complementing the original credit scheduler.

### 3.2.1 New scheduling decision trigger

In credit scheduler a scheduling decision is made when a VCPU blocks, yields, completes its time slice, or is awoken (Xen wiki, 2007). A runnable VCPU with an incoming event call is not prioritised properly. It cannot process the event until it crawls to the head position of run queue. This may delay the processing of some events that are related to latency sensitive tasks. The proposed scheduler captures the event notifications for runnable VCPUs and then tickles the scheduler for a new scheduling decision. When scheduler is tickled, if the currently running VCPU is rather than prioritised, it is pre-empted and a new VCPU is selected to run next. Since the runnable VCPU receiving an event is prioritised, it is very likely for it to get an immediate execution.

FEAS is configured to react merely on inter-domain events and PIRQ events received merely by runnable VCPUs. As such an event may also wake and boost a

blocked VCPU with *under* priority, a *waken* flag is set if the VCPU is woken. The scheduler only promotes runnable VCPUs whose *waken* flag is off.

### 3.2.2 Interchangeable immediate queue and postponed queue

FEAS prioritises VCPUs doing I/O tasks no matter whether the VCPU is blocked and whether its priority is under or over. However, if there is no constraint, an I/O intensive domain can hold the PCPU for an unfair amount of time by taking advantage of this preference. So two new queues: immediate queue and postponed queue, are introduced to limit the frequency of VCPUs getting scheduled. A limit on the number of times a VCPU can get scheduled within a counting cycle is enforced on every VCPU. Let  $n_{limit}$  denote the upper limit of the number of scheduling times,  $c$  denote the  $c$ th counting cycle and  $n_{i,c}$  denote the scheduling times of VCPU  $i$  during the  $c$ th counting cycle. A trigger which is usually a timer indicates the start of a new counting cycle.  $n_{limit}$  is a constant over all counting cycles and  $n_{i,c}$  is initialised to 0 at the start of every counting cycle. VCPUs join immediate queue or postponed queue following the two rules below. Note that FEAS prioritises runnable VCPUs even if they have negative credits, so that, VCPUs can achieve optimal responsiveness.

- ⎧ If  $n_{i,c} < n_{limit}$ , then VCPU  $i$  joins the immediate queue
- ⎨ If  $n_{i,c} \geq n_{limit}$ , then VCPU  $i$  joins the postponed queue

On every scheduling decision, the scheduler always preferentially schedules the VCPUs in the immediate queue. If the immediate queue is empty, the VCPUs in the run queue are executed in the decreasing order of their priorities as usual.

Postponed queue is used as the temporary depository for those VCPUs that needs prompt execution but have already run more often than allowed. Upon the start of each counting cycle, the scheduler checks the status of immediate queue and postponed queue. If it finds the immediate queue empty while the postponed queue is not, it swaps these two queues and then sends a rescheduling request. As a result, postponed VCPUs can receive preferential and properly delayed scheduling.

The *boost* priority of a VCPU in credit scheduler is replaced by the action of joining the immediate queue. A waken and boosted VCPU can be recognised by checking the *waken* flag (refer to sub-section 3.2.1) and whether it is in the immediate queue.

Every VCPU also has an *is\_immediate* flag. It is turned on when a VCPU scheduled from the immediate queue and turned off when it is de-scheduled or a tick event fired. Conditional pre-emption of the running VCPU is decided based on the *is\_immediate* flag instead of by comparing priorities. A VCPU with the *is\_immediate* flag set cannot be pre-empted.

### 3.2.3 Guaranteed VCPU's time slicing

Credit scheduler is a weighted round-robin (WRR) based fair scheduler. It achieves proportional fairness by adjusting VCPU's credits to control the frequency that the VCPU is selected to run and by running each VCPU for the same size of time quantum. Ideally, a VCPU is only



pre-empted when its time slicing expires or it spontaneously yields the CPU. However, credit scheduler pre-empts the running VCPU when a blocked VCPU is waken and boosted. Also, in the proposed scheduler the running VCPU is pre-empted when a runnable VCPU receives an inter-domain or PIRQ event. Both cases of pre-emption can happen anytime. So the pre-empted VCPU becomes the victim of pre-emption, because once being pre-empted it will lose its remaining time slice in this rotation and have to wait in the run queue until its next turn. Time slices allocated to VCPUs are loosely controlled in credit scheduler and usually the long term CPU time received by VCPUs is bound and balanced by the credits they receive. However, this does not work well with CPU affinity. Scheduling unfairness may be caused when VCPUs are pinned to some specific CPUs since they earn more credits than they could spend. Over-earned credits allow VCPUs that block and wake regularly to excessively pre-empt their competitors. Also, as FEAS allows temporary overdraft of future quantum and it prioritises VCPUs receiving events even if they have negative credit balance, this module limits the overdraft to one rotation's range.

In this module, every VCPU is allocated a quantum of 30ms each rotation and this quantum is guaranteed to be exhausted in this rotation. The quantum is deducted at the same time with debiting credits. After being pre-empted, the VCPU is inserted to the head position of the run queue if its quantum is still more than 1ms. Therefore, it can keep consuming its quantum later on. Otherwise, it is inserted into the run queue in the conventional way. For those de-scheduled VCPUs de-queued from the immediate queue, their position in the run queue is reserved. So the usage of their allocated quantum for each rotation can be accurately recorded.

FEAS maximises busy VCPUs' responsiveness by allowing temporary overdraft of future quantum. A runnable VCPU receiving many I/O related events may exhaust its quantum early by frequently joining immediate queue or postponed queue. If that is the case, the VCPU can no more be prioritised and have to wait in the run queue for its turn. When it gets its turn, it is given a minor slice of 500 microseconds to run. Therefore, the overdraft is limited within the quantum that is worth one rotation.

#### 4 Performance study

FEAS is implemented based on the credit scheduler of Xen 4.0.1 and tested on Linux-2.6.18.8. Since it works by monitoring the events sent between domains and between domain 0 (or driver domain) and physical devices, all source code modifications made are within the hypervisor and none is needed within the guest kernel. In our implementation,  $n_{limit}$  is set to 1 and the *tick* which fires every 10ms is reused as the trigger that indicates the start of a new counting cycle. The machine we are testing with has an Intel Core2 Duo 3.16 HZ CPU, 3.2 GB RAM and a Gigabit Ethernet network interface. A separate machine is used as the client for the network related experiments. The client machine is guaranteed of no bottleneck in any experiments. Two machines are connected via a 10/100M switch.

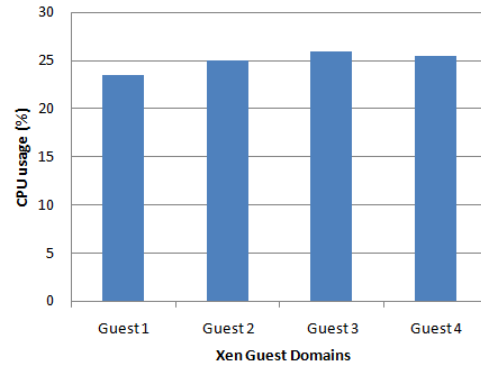
Four guest domains each of which has one VCPU are created. They all have the default weight of 256. Domain 0 is chosen as the driver domain. Its VCPU is pinned to CPU core 1 and VCPUs of four guests are pinned to CPU core 2. Dedicating a CPU core to Domain 0 can achieve better system performance since all I/O requests have to go through Domain 0 and this can reduce the number of CPU context switches required. This is also a common configuration in production servers. Guest 4 is connected to the client machine in all experiments. In the experiments guest domains, excluding Domain 0, are loaded in groups with CPU intensive tasks to simulate various production environments. Table 1 enumerates the different setups. The VCPUs are kept busy using *cpuburn* 1.4 (Softpedia 2011).

Domains	Explanation
All busy	VCPUs of all guest domains are running at 100%.
Others busy	VCPUs of all guest domains except Guest 4 are running at 100%.
All idle	VCPUs of all guest domains are idle most of the time.

**Table 1: Experimental setup**

#### 4.1 Scheduling fairness

Fairness is one of the main goals of a scheduler, especially the scheduler of a VMM. Domains should not be starved and a malicious domain cannot take an unfair amount of CPU time slicing at any time. When CPU resources are in contention, each domain should receive CPU time proportional to their weights. Domains with the same weights are expected to receive the same amount of CPU time.



**Figure 5: CPU time distribution**

This experiment proves the scheduling fairness of FEAS by keeping all domains busy. Each domain runs *cpuburn* and eats as much CPU time as they are given. As can be seen from Figure 5, every domain can receive roughly equal amount of physical CPU time. This figure holds in all experiments where all guest domains are CPU hogging and guest 4 constantly receives ping or downloading requests. The CPU time slicing guarantor module described in section 3.2.3 keeps the influences of pre-emption on round robin scheduling to a minimum. A VCPU is marked inactive and its credits are forfeited when it accumulates too many credits. Ideally, excessive credit accumulation is due to the VCPU's little demand.

However, it may also be caused by undesirable starvation. Undesirable starvation is more frequent in FEAS than in credit scheduler, since the former boosts runnable VCPUs greedily to achieve high responsiveness even if its credit balance is negative. The CPU time slicing guarantor can efficiently ease this kind of starvation by guaranteeing that pre-empted VCPUs fully use their CPU slice in every rotation and that pre-empting VCPUs cannot overdraft their CPU time too much.

#### 4.2 Latency sensitive processes

This experiment tests the performance of latency sensitive tasks in a domain doing mixed workloads. The client machine sends ping requests to the Guest 4 for 100 times and its response time is recorded.

All domains including Guest 4 are running CPU intensive tasks. As a result, they are hardly blocked. So under credit scheduler, even though Guest 4 constantly receives ping requests, it is not boosted. It cannot get scheduled and respond to the request until other VCPUs ahead of it in the run queue finish execution. So the latency under the credit scheduler is high and unpredictable.

On the other hand, FEAS can capture the incoming event notification all the way and scheduled the target domain immediately. It prioritises runnable VCPUs receiving an event call at most once per 10ms no matter whether its priority is under or over. Since the ping request is sent every second which is way longer than 10ms, the domain doing CPU intensive tasks can always respond to ping requests immediately.

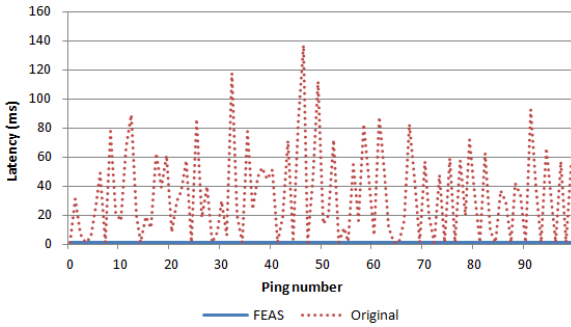


Figure 6: Ping Latency

#### 4.3 Network intensive processes

The performance of a CPU-consuming domain on network intensive tasks is evaluated in this experiment. Guest 4 hosts a FTP server using vsftpd-2.3.4 (vsftpd, 2011) and the client machine downloads files of different sizes from it. The average transfer speed is recorded.

Figure 7 illustrates the results achieved in different situations. Both schedulers achieve similar results when Guest 4 is idle no matter whether other domains are busy. The *boost* mechanism from credit scheduler efficiently ensures the performance of idle domains on I/O related tasks when they co-exist on the same server with other domains that are doing CPU intensive tasks. However, when Guest 4 is also doing CPU intensive tasks, the download speed under FEAS doubles that under credit scheduler. Since Guest 4's VCPU is runnable most of the time when fully loaded, credit scheduler does not discriminate it from other co-currently running busy

domains. Scheduling delays regarding handling I/O related events are thus much longer than when Guest 4 is idle. However, FEAS keeps I/O devices busy by promptly handling of incoming events and thus speeds up the transmission process.

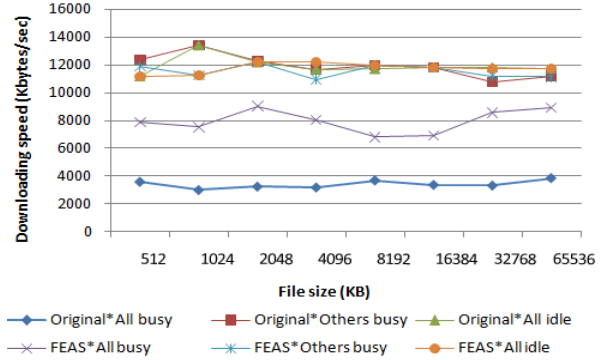


Figure 7: FTP downloading speed

#### 4.4 Impact of co-working VMs

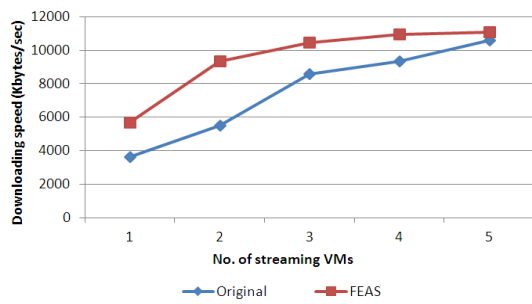
This experiment examines the I/O performances under FEAS when multiple co-located VMs concurrently process mixed workloads. Five duplicated guest domains are set up for this experiment on the same physical server used in previous experiments. Also, Domain 0 is pinned to CPU core 1 and Domain 1-5 are pinned to CPU core 2. All five guest domains are configured with a FTP server facilitated by vsftpd-2.3.4, and each hosts a 16384 KB file ready for download. To keep guest domains' VCPUs under constant pressure, they all run cpuburn. A multi-threaded C# program is designed to simultaneously download the hosted file from guest domains. Hence, during the period of concurrent network streaming dense inter-domain events and PIRQ events are fired across by all domains at the same time. All these events are caught by FEAS and its decision has a direct impact on the I/O performances and CPU fairness.

The multi-threaded program is modified to run with 1 to 5 threads respectively on the client machine and each thread downloads the test file from an exclusive VM. Suppose that  $n$  threads simultaneously stream files from  $n$  VMs (where  $n = \{1, 2, \dots, 5\}$ ), and that the  $i$ th streaming thread starts at  $t_i^s$  and finishes at  $t_i^f$  (where  $i = \{1, \dots, n\}$ ). Downloading speed  $s_n$  is evaluated to reflect the system performance as

$$s_n = \frac{n * size_f}{latest(t_1^f, \dots, t_n^f) - earliest(t_1^s, \dots, t_n^s)}$$

where  $size_f = 16384KB$

The results are recorded and shown in Figure 8. The overall system throughput increases when downloading files from more VMs, and the I/O devices tend to operate at full speed if downloading files from all VMs. The reason is that the overall system performance depends on the CPU time and the scheduling latency of all streaming VMs. Thus, if all five VMs are delivering files, no matter which VM is scheduled, they all contribute to the overall system throughput. All in all, FEAS performs better than the original credit scheduler in terms of I/O throughput even when multiple VMs do mixed workloads at the same time.



**Figure 8: Download from multiple VMs**

#### 4.5 Scheduling overhead

Seconds	Others busy	All idle	Domain 0
Original	32.43754	8.78261	8.123744
FEAS	32.5931	8.80411	8.146923

**Table 2: Duration for prime searching in seconds**

To quantify the scheduling overhead FEAS causes over the original credit scheduler, we observe the running time of the prime searching function which is a lengthy and CPU-intensive process. This experiment finds all the 664,579 prime numbers less than  $10^7$  using the trial division algorithm (Wikipedia, 2011). The time required to complete the process in three cases is illustrated in Table 2. The percentage increase of running time introduced by FEAS in all three cases is less than 1%, which indicates that the overhead is negligible.

#### 5 Conclusion

VMs sharing the same hardware contend for limited resources. It is important to appropriately allocate shared resources among VMs that are running simultaneously. While fairness requires that each VM receives CPU time proportional to their weights, low latency is achieved by scheduling a VM as soon as it needs CPU especially if the VM has I/O tasks pending. Our scheduler is an enhanced version of credit scheduler that prioritises VCPUs doing I/O tasks by monitoring inter-domain events and PIRQ events sent between domains and physical devices. FEAS makes VMM full-time event aware and promptly schedules with best effort runnable VCPUs that receive I/O related events. The experiments show that VMs under FEAS performs better on I/O intensive tasks than those under credit scheduler if they also do CPU intensive tasks at the same time. The cost for the modifications needed to realise FEAS is proved to be negligible.

Currently, FEAS is only implemented and tested with the guest domains virtualised in para-virtualisation mode. Since hardware virtual machine (HVM) does not requiring the guest operating system to be modified, it can run proprietary operating systems like Windows as guest. Split drive model is implemented differently in PV-on-HVM kernels. Our future research will try to apply FEAS on fully virtualised virtual machines.

#### 6 References

Barham, P., Dragovic, B., Fraser, K., & et al (2003): Xen and The Art of Virtualization, *ACM Symposium on Operating Systems Principles*.

Cherkasova, L., Gupta, D. & Vahdat, A. (2007): Comparison of the Three CPU Schedulers in Xen, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 35, Iss. 2, pp. 42–51.

Chisnall, D. (2007): *The Definitive Guide to the Xen Hypervisor*. Sydney, Prentice Hall.

Goldberg, R.P. (1974), Survey of Virtual Machine Research, *IEEE Computer*, Vol. 7, Iss. 6, pp. 34–45.

Govindan, S., Nath, A., Das, A., Urgaonkar, B. and Sivasubramanian, A. (2007): Xen and co.: communication-aware CPU scheduling for consolidated xen-based hosting platforms, *Proceedings of the 3rd international conference on Virtual execution environments*, New York, USA.

Gupta, D., Cherkasova, L., Gardner, R. & Vahdat, A. (2006): Enforcing performance isolation across virtual machines in Xen, *In Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference*, Melbourne, Australia.

Iyer, R., Illikkal, R., Tickoo, O., Zhao, L., Apparao, P. & Newell, D. (2009): VM3: Measuring, modeling and managing VM shared resources, *Computer Networks*, Vol. 53, Iss. 17, pp. 2873–2887.

Kim, H., Lim, H., Jeong, J., Jo, H. & Lee, J. (2009): Task - Aware Virtual Machine Scheduling for I/O Performance, *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environment*. Washington, DC, USA.

Lin, B., Dinda, P. & Lu, D. (2004): User - Driven Scheduling of Interactive Virtual Machines. *5th IEEE/ACM International Workshop on Grid*. Washington, DC, USA.

Ongaro, D., Cox, A. and Rixner, S. (2008): Scheduling I/O in virtual machine monitors, *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, New York, USA.

Softpedia (2011), cpuburn 1.4, <http://www.softpedia.com/get/System/Benchmarks/cpuburn.shtml>, Accessed May, 2011

vsftpd: vsftpd - Secure, fast FTP server for UNIX-like systems. <https://security.appspot.com/vsftpd.html>. Accessed May, 2011.

Weng, C., Wang, Z., Li, M. & Lu, X. (2009): The Hybrid Scheduling Framework for Virtual Machine Systems, *ACM International Conference on Virtual Execution Environments*. Washington, DC, USA.

Wikipedia: Trial division. [http://en.wikipedia.org/wiki/Trial\\_division](http://en.wikipedia.org/wiki/Trial_division). Accessed May, 2011.

Xia, Y.B., Yang, C. & Cheng, X. (2009): PaS: A Preemption-aware Scheduling Interface for Improving Interactive Performance in Consolidated Virtual Machine Environment. *International Conference on Parallel and Distributed Systems*. Shenzhen, China.

Xen: Home of the Xen Hypervisor. <http://xen.org/>. Accessed Jun, 2011.



# Boosting Instruction Set Simulator Performance with Parallel Block Optimisation and Replacement

Brad Alexander<sup>1</sup>

Sean Donnellan<sup>1</sup>

Andrew Jeffries<sup>3</sup>

Travis Olds<sup>3</sup>

Nicholas Sizer<sup>1</sup>

<sup>1</sup> School of Computer Science,  
University of Adelaide,  
Adelaide 5005,  
Email: brad@adelaide.edu.au

<sup>2</sup> Ultra Electronics Avalon Systems,  
12 Douglas Drive,  
Mawson Lakes, South Australia 5095,  
Email: andrewj@avalon.com.au

<sup>3</sup> Australian Semiconductor Technology  
Company,  
Level 5, 76 Waymouth Street,  
Adelaide, South Australia 5000,  
Email: travis.olds@astc-design.com

## Abstract

Time-to-market is a critical factor in the commercial success of new consumer devices. To minimise delays, system developers and third party software vendors must be able to test their applications before the hardware platform becomes available. Instruction Set Simulators (ISS's) underpin this early development by emulating new platforms on ordinary desktop machines. As target platforms become faster the performance demands on ISS's become greater. A key challenge is to leverage available simulator technology to produce, at low cost, incremental performance gains needed to keep up with these demands. In this work we use a very simple strategy: in-place-block-replacement to produce improvements in the performance of the popular QEMU functional simulator. The replacement blocks are generated at runtime using the LLVM JIT running on spare processor cores. This strategy provides a very lightweight way to incrementally build an alternate code generator within an existing ISS framework without incurring a substantial runtime cost. We show the approach is effective in reducing the runtimes of the QEMU user-space emulator on a number of SPECint 2006 benchmarks. *Keywords: Instruction Set Simulation, Dynamic Binary Translation, Background Optimisation, LLVM, QEMU*

## 1 Introduction

Instruction Set Simulators (ISSs) are software platforms that run on a host hardware architecture and emulate a guest hardware architectures. An ISS allows developers to test and use systems and application software whenever using the actual hardware platform is not an easy option. ISSs are used for reasons of security and safety[Vachharajani et al., 2004], cross-platform-support[Adams and Agesen, 2006, Bellard, 2005, Chernoff et al., 1998] or just because they may be more readily available than the actual hardware. In the extreme, an ISS can provide

a platform for software development *before* the corresponding hardware platform exists. This last mode of use for ISSs permits the parallel-development of hardware and software for new mobile and embedded devices. Such parallel-development reduces time-to-market which is vital in the commercial success of these devices[De Michell and Gupta, 1997]. With mobile devices becoming faster, and competitive pressures shortening production cycles, there is strong demand for faster emulation from ISSs.

ISSs emulate at a variety of levels. Cycle-accurate ISSs[Lee et al., 2008] emulate the timing of hardware devices to allow debugging at the hardware/system interface. Functional ISSs[Adams and Agesen, 2006, Bellard, 2005, Magnusson et al., 2002, Cmelik and Keppel, 1994, Witchel and Rosenblum, 1996] emulate architectures at the behavioural level providing a platform for interactive testing of systems and application software. Functional ISSs are generally much faster than cycle-accurate ISSs making them attractive for high-level system and application developers. Functional ISSs can further subdivided into whole-system simulators, able to emulate an entire operating system[Adams and Agesen, 2006, Bellard, 2005, OVP, 2011] and process-VMs or user-mode emulators which can run a single application. This work describes enhancements to the performance of the QEMU ISS[Bellard, 2005]. QEMU is one of the most popular ISS's. It provides fast emulation for a variety of target and source architecture and forms the basis of a number of industrial emulators including the Android mobile device emulator[Google .Inc, 2011]. QEMU provides both system-mode emulation for whole-systems and user-mode emulation for single-applications. In this article we focus on enhancements to the simpler, and faster, user-mode QEMU.

Like many ISSs, QEMU uses Dynamic Binary Translation (DBT) to translate blocks of guest platform code to host platform code at runtime. Once blocks are translated, they can be run natively on the host platform – greatly improving performance over simple interpretation of guest instructions[Arnold et al., 2005]. However, good performance is only possible if translation is done quickly. On single processor systems, any attempt at optimising translated code is time-constrained: time spent optimising translated code is time *not* spent running translated code. When emulating on a multi-core host these constraints are

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

less severe. After an initial fast translation, idle cores can be utilised to perform background optimisation in a separate thread. Such background optimisation is relatively common in high-level-language process-VMs[Krintz et al., 2001, Alpern et al., 2005] but relatively rare in DBT-Based ISSs[Qin et al., 2006]. For simulators where the host and guest Instruction-Set-Architecture (ISA) is the same, this rarity is unsurprising: we would expect gains from further optimisation of already-optimised code in same-ISA simulation to be small[Adams and Agesen, 2006, Arnold et al., 2005]. However, this rarity is harder to explain in different-ISA ISSs, the architectural gap between the guest ISA and the host ISA is usually large enough to leave room for further optimisation.

In this article we describe an implementation that experiments with the use of background optimisation to improve the speed of user-mode QEMU running ARM binaries on a multi-core x86-64 host. In our experiments we use LLVM components, running on a spare core, to perform background optimisation and replacement of cached blocks of dynamically translated code. We show that this approach is successful in producing speedups in the simulation of a number of the SPECint 2006[Standard-Performance-Evaluation-Corporation, 2011] benchmarks.

At this point it should be emphasised that the focus of this work is in utilising spare cores to improve code quality and hence the simulation speed of *each-thread* of an application. This is in contrast to the orthogonal, and well-studied, problem of using multiple cores to help simulate *multi-threaded* applications[Jiang et al., 2009, Almer et al., 2011, Wentzlaff and Agarwal, 2006]. The primary aim of this work is to improve thread code quality with the goal of reducing overall execution time.

This paper makes the following contributions:

- It is the first successful attempt to boost the performance of QEMU through background optimisation.
- Moreover, to the best of our knowledge, it is the second successful attempt to use background optimisation of already translated code to boost the performance of any different-ISA ISS, the other being SIMIT-ARM[Qin et al., 2006] and the first to use spare cores on the same machine.
- It demonstrates that background optimisation can be added cheaply and incrementally and still achieve faster execution (section 4.5) and better quality host-code (section 4.4). Performance gains can be had *before*, covering all instructions and before implementing heuristics for selecting blocks for replacement. Moreover, our optimisation is still useful at the level of the basic block - allowing gains to be enjoyed before block aggregation and inter-block optimisation is implemented. In summary, there is a path to retro-fit changes cheaply, incrementally and productively to an existing ISS, using standard tools.

The rest of this paper is structured as follows. In the next section we describe related work. In section 3 we describe our approach starting with the base components of QEMU and LLVM followed by an explanation how these are combined to make Augmented-QEMU which uses background optimisation for faster execution. In section 4 we describe our experimental results. Finally, in section 5 we summarise our results and propose future directions for research.

## 2 Related Work

The most closely related work to ours is Scheller's LLVM-QEMU project[Scheller, 2008]. Like our work, LLVM-QEMU used the LLVM Just-in-Time compiler (LLVM-JIT). However, in his work he uses the LLVM-JIT to *replace* the role QEMU's native code generator – the Tiny Code Generator (TCG). In our work we use the LLVM-JIT to *complement* the role of TCG. We allow TCG to perform a fast initial translation for immediate execution and use the LLVM-JIT to replace selected blocks as QEMU is running. Our work also differs in the use of a separate thread<sup>1</sup> to run the LLVM-JIT. This combination of differences results in our implementation achieving faster performance overall with speedup, rather than slowdown, on most benchmarks.

Also related is the identically named LLVM-QEMU project by Chipounov and Candea[Chipounov and Candea, 2010]. They also used the LLVM-JIT to bypass TCG in QEMU. They compiled all micro-operations into C functions which were then translated into LLVM Intermediate Representation (LLVM IR) and then optimised before execution. Again, the overheads of running the LLVM-JIT were substantial, resulting in slowdown when compared to the original QEMU.

Again, our work differs most from the above approaches in leaving TCG intact. By leaving most of the QEMU infrastructure in place we are able to focus on incrementally increasing the number of blocks we handle as we add new, carefully-handwritten, translations from individual ARM instructions to LLVM-IR code. These direct hand-written translations appear, a-priori, easier to optimise than LLVM-IR generated from C or TCG intermediate code.

Looking more broadly at related DBTs, same-ISA DBTs[Watson, 2008, Adams and Agesen, 2006] have the option of preserving most optimisations already present in the source binary. This can give very good performance[Adams and Agesen, 2006] though, in some cases, knowledge gained from runtime profiles can be used for even more optimisation[Bruening et al., 2003]. Unfortunately most ISSs are by necessity different-ISA DBTs[OVP, 2011, Magnusson et al., 2002, Qin et al., 2006, Bellard, 2005]. These face greater challenges than same-ISA DBTs due to the unavoidable loss of platform-specific optimisations during translation between ISAs.

The trade-off between running translated code and optimising translated code means that optimisation must be done sparingly on a single processor machine. On multi-core machines, optimisation can be carried out in a separate thread. Such background optimisation is relatively common in process VMs for high level languages such as Java (Kulkarni[Kulkarni et al., 2007] gives an overview of the impact of these optimisations). In contrast, the use of such background optimisation is rare in different-ISA DBTs.

One exception is SIMIT-ARM[Qin et al., 2006] which, concurrently with interpretation of guest code, translates guest code into large, fixed-size blocks of C code and then compiles these into dynamically-linked-libraries (DLLs) using gcc running on separate hosts. As the new DLLs are produced they are loaded in and run. The amount of optimisation performed by gcc is trivially controlled using compiler flags. The DLLs in SIMIT-ARM are persistent between runs which lets applications run much faster the second time they are invoked. In earlier experiments[Lee, 2009] it was

<sup>1</sup>It should be noted that LLVM-QEMU was a short summer-project and multi-threading was on the list of things to do.



found that SIMIT-ARM was able to run at speeds comparable to QEMU in user-mode when it runs on these cached DLLs.

Our work differs from SIMIT-ARM in our use of spare cores rather than separate hosts, and the focus of optimisation on small basic blocks rather than large fixed-sized blocks with multiple entry points<sup>2</sup>. Our approach, through the use of a lightweight-JIT, working in a shared memory space, provides a lower latency on block optimisation – which is likely to be important in the context of the relatively fast QEMU simulator.

Finally, there is substantial work that applies optimisation to the *initial* translation phase of an ISS. Almer [Almer et al., 2011] uses the LLVM-JIT to translate and optimise hot traces in their multi-threaded ISS. Wentzlaff[Wentzlaff and Agarwal, 2006] also performs translation and optimisation of code blocks in their parallel ISS. Both these works differ from ours by applying optimisation only during the initial translation of each trace or block as such they do not have to perform runtime block-replacement of translated code.

This concludes our overview of related literature. In the next section we describe our approach.

### 3 Approach

In this section we describe the implementation used in this work: Augmented-QEMU. The goal of Augmented-QEMU is to provide faster emulation by performing background optimisation of guest (ARMv5) instructions to host (x86-64) instructions. Augmented-QEMU is built using the following software components:

- QEMU version 0.10.6, running in user-mode, which we call vanilla-QEMU,
- The LLVM optimisers and x86 code-generator that are part of the LLVM just-in-time compiler version 2.6 (LLVM-JIT).

We describe each of these in turn before explaining how they are combined to form Augmented-QEMU in section 3.3.

#### 3.1 Vanilla-QEMU

QEMU[Bellard, 2005] is a widely used, versatile and portable DBT system. QEMU is cross-platform. It supports a variety of host and guest ISAs. It is able to exploit the features of its host architecture. For example, when run on a 64bit host such as the x86-64 architecture it is able to exploit the extra registers to provide better performance. QEMU is a functional simulator, it emulates program behaviour rather than simulating accurate timings of events in hardware. QEMU has two modes: a system-mode with detailed hardware models for emulating entire systems; and a, somewhat faster, user-mode that acts as a process-VM on which to run a single application. QEMU is quite fast, with quoted emulation speeds of 400 to 500 MIPS. In terms of actual runtimes, on our experimental machine, we found that QEMU running cross-compiled ARM binaries ran approximately ten times slower than native x86 compiled benchmarks. The

<sup>2</sup>The approach of SIMIT-ARM is highly original and starkly different from the basic-block-oriented approach taken by most DBTs. The use of multiple-entry points leads to blocks being cast as large switch statements. The potential impact of this format on different levels of optimisation is unknown and warrants further study.

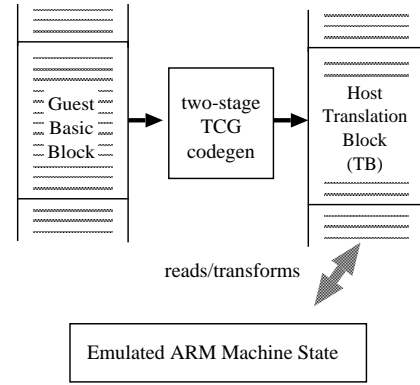


Figure 1: Basic Translation Schema for QEMU

following is a brief outline of how QEMU CPU simulation works. For a more detailed overview see [Bellard, 2005].

#### 3.1.1 The QEMU Simulation Process

QEMU performs computation by maintaining an abstract CPU state for the guest architecture. This state includes program counters, stack pointers, control flags and general-purpose registers. QEMU translates guest binary code to host code that then acts on this processor state. At the start of simulation, this state is initialised as it would be at the start of program execution on the guest architecture. For Augmented-QEMU, the guest architecture is a 32 bit ARM machine and the state is called `CPUARMState`. Figure 1 illustrates this basic execution schema. Note that, unlike some other VMs[Bruening et al., 2003, Krintz et al., 2001], QEMU does not initially directly interpret guest code. Instead it relies on a fast translator, called Tiny Code Generator (TCG) to quickly produce basic blocks of host code. These blocks, called Translation Blocks (TBs), are cached in an area called the translation cache and then immediately executed. TCG runs in two-phases. The first phase translates the guest ISA to TCG code – a generalised intermediate form. The second phase converts TCG code to the host ISA. This two-phase design aids portability by decoupling the source and host ends of the translation process.

The translation schema described above sits in the context of the broader control structure for QEMU shown in figure 2. In the figure, control flows are indicated with thin lines and flows of code with thick dashed grey lines. At the core of QEMU simulation is the `cpu_exec()` function. This function is responsible for the controlling the translation and execution of basic blocks of guest code. During simulation, `cpu_exec()` operates as follows. First, when a block of guest code needs to be executed, the source program counter (SPC) is read from the guest CPU state (step (1) in figure 2). Next, the SPC is looked up in the map table (step (2)). At this point one of two things can happen:

1. The block starting at the SPC is found to have already been translated and stored in the translation cache. In this case the map table will return the address of the relevant TB in the translation cache. Or,
2. The block starting at the SPC is not in the translation cache and so is not found in the map table. In this case the guest block is new and will need to be translated prior to execution.

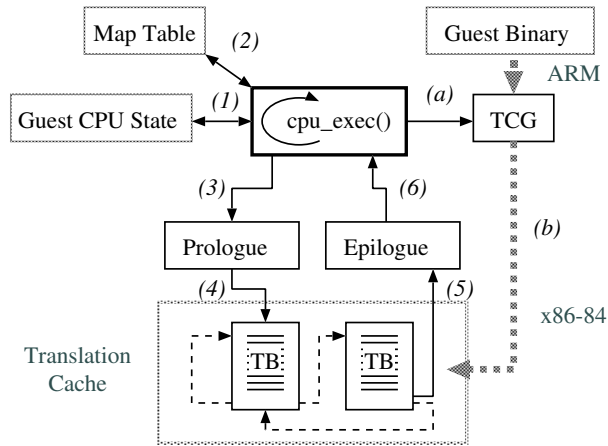


Figure 2: Control structure for QEMU centred around the core execution loop: `cpu_exec()`. Showing the actions creating new TBs ((a) and (b)) and actions to execute extant TB's ((1) to (7)).

In the case of the relevant TB being found, control is passed from `cpu_exec()` to a block of prologue code (step (3)). The prologue saves QEMU's register state before jumping to the relevant TB (step (4)). The TB then executes. In many cases, when the end of the TB is reached, control is not passed back to `cpu_exec()` but instead the TB branches to itself or other TBs. This internal branching within the translation cache (indicated with thin dashed lines in figure 2) is called *chaining*. Chaining is a simple optimisation that often avoids the overhead of going back to `cpu_exec()`. QEMU is able to spend substantial time running in chained TBs of blocks within the translation cache. Eventually, one of the TBs will return control back to some epilogue code (step (5)) which updates the SPC in the guest CPU state and restores the QEMU register state before returning to `cpu_exec()` (step (6)).

In the case of the required TB *not* being found, a new TB will have to be generated from the guest binary. This is done by calling TCG with the current SPC (step (a)) and inserting it into the translation cache (step (b)). Note that the chaining between TBs is performed by TCG as blocks are inserted into the translation cache. Once the new TB is in the cache, an entry for it is inserted into the map table and its execution can proceed.

It should be noted that, overall, this simulation infrastructure is quite efficient, in earlier experiments[Jeffery, 2010] we found that QEMU spent a large majority of its time running in the translated code cache when run in user-mode on SPECint 2006 benchmarks.

This concludes our brief overview of vanilla QEMU. It is worth noting that our modifications, which we describe shortly, leave this basic structure intact. Our changes basically leverage LLVM-JIT described next, to improve the code in the TBs.

## 3.2 LLVM

The LLVM compiler Infrastructure [Lattner and Adve, 2004] is a set of open-source components which can be used as building blocks for custom compilers. The aim of the LLVM project is to provide modular components, such as optimisers, and code-generators that can be reused in different language implementations. The key to reusability is the use of a general purpose intermediate representation, LLVM-IR, to

act as an interface between components. The LLVM project envisages that any compiler using LLVM components can have easy access to the frequent improvements made to these components.

LLVM is well-supported and being used in many significant projects [LLVM-Project, 2011]. In this work, we combine a short series of optimisation passes, described in the section below and the LLVM-JIT code generator to produce blocks of fast x86-64 code from LLVM-IR that our own custom translator generates from ARM instructions. Next, we describe the process by which we exploit these components in Augmented-QEMU.

### 3.3 Augmented-QEMU

Augmented QEMU performs background optimisation and replacement of blocks, using LLVM, to improve the performance of QEMU. The modifications used to produce Augmented-QEMU in the context of vanilla-QEMU are shown in figure 3. Note, to provide context, the components of vanilla-QEMU from figure 2 are shown in grey. The new and modified components are drawn in black. Control flows are represented by solid black arrows and data flows (labelled with their type) by dashed grey arrows. There are two threads, the original QEMU thread and a new LLVM thread. Communication between the threads is managed by two queues, the block translation queue, which contains pointers to guest binary blocks awaiting further optimisation; and the block replacement queue, which contains pointers to optimised replacement TBs. A brief summary of how Augmented QEMU works follows. Each part of the summary is cross-referenced to the steps in figure 3. In addition, more detailed descriptions of these parts follow this summary.

**Steps (i) and (ii), Testing Eligibility:** Whenever TCG produces a new TB from a block of guest binary code, our modified `cpu_exec` performs an eligibility check. The eligibility check scans the block of guest binary and checks that we have implemented LLVM translations for every instruction in the block (step (i)). If so, a pointer to this block is queued for translation (step (ii)). This process is described in section 3.3.1.

**Steps (iii) and (iv) , Block Translation:** The addition of a new pointer to the block translation queue wakes the LLVM thread (step (iii)) which immediately passes the block of guest binary to our own LLVM-IR code generator for translation (step (iv)). This process is described in section 3.3.2.

**Steps (v) and (vi), Block-optimisation/Code Generation:** The new block of LLVM IR is given to the LLVM-JIT (step (v)). The LLVM-JIT performs a series of optimisations and then generates a block of host code. This process is described in section 3.3.3. A pointer to this new code is added to the block replacement queue (step (vi)).

**Steps (vii),(viii) and (ix), Block-Replacement:** The replace-block function, de-queues pointers to any newly generated blocks (step (vii)) and, if the block fits in the space allocated by the original TB, it uses a memcpy operation (step (viii)) to overwrite the original TB block. Finally, a jump instruction is added from the end of the new code block to the end of the space occupied by the original



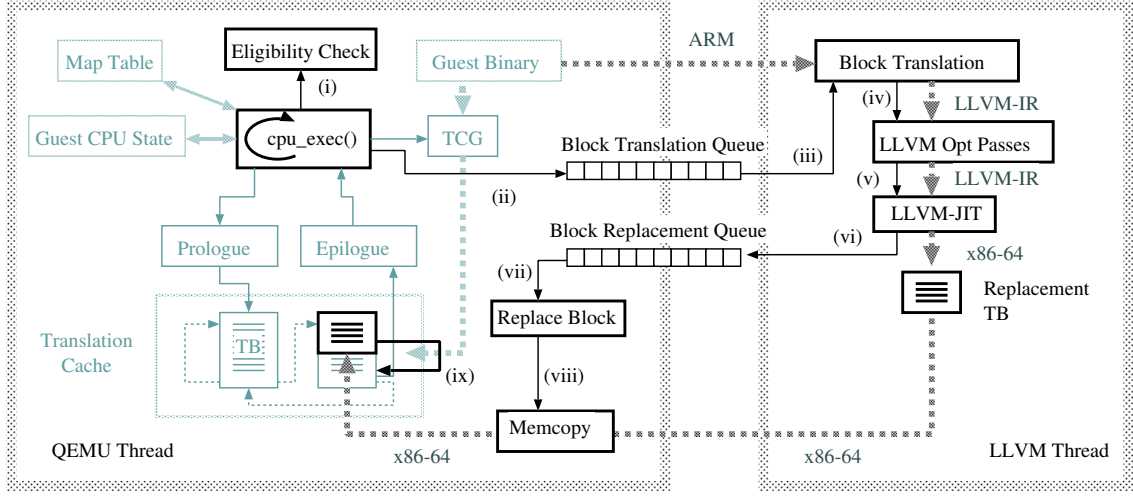


Figure 3: The modifications used to produce Augmented-QEMU (in black) in the context of vanilla-QEMU (in grey).

Execution Count	
seng	
ldr	2594556723
add	618512755
str	595786363
sub	432138506
cmp	428270933
mov	408249153
bne	204144055
lsl	170450711
beq	96700869
bl	94330545
asr	87404495
bx	63627453
and	58944875

Table 1: Top instruction execution counts for seng SPECint 2006 benchmark (instructions we translate are in green)

TB (step (ix)). This process is described in section 3.3.4

More detailed explanations of each of these steps follow.

### 3.3.1 Testing Eligibility

Augmented-QEMU uses a slightly modified version of `cpu_exec()` to guide the block-optimisation and replacement process. Immediately before any new TB is formed by TCG, an eligibility check is performed on the same block of an guest binary code to determine if it can also be translated by the LLVM thread. This check needs to be done because not all of the guest binary instruction set is handled by our LLVM optimisation thread. The choice of which instructions to handle first is guided by benchmarking. We ran a number of the SPECint 2006 benchmarks compiled to ARM by GCC 4.4.3 (arm-softfloat-linux-gnueabi) to get counts of each instruction. The counts for the sjeng chess-playing benchmark on test workload are shown in table 1. The instructions marked in green are among the ones currently translated by our block

translator<sup>3</sup>. Branches, marked in yellow, are excluded because they appear only after the end of each translation block, either as part of chaining or jumps back to the epilogue. Instructions in red are yet to be implemented in the block translator. Other SPECint 2006 benchmarks had roughly similar distributions. In our tests, data movement, arithmetic and comparison operators tended to dominate guest-code so we focused on implementing these. Note that only blocks that we can translate *entirely* are considered eligible. By this measure, currently more than half the total number of blocks in the SPECint 2006 benchmarks are eligible.

After a guest block is found to be eligible for translation, a pointer to that block is queued for translation by the LLVM thread. Access to the translation queue is guarded by a lock to prevent race-conditions. No lock is needed for accessing the guest binary code because it is read-only in this context<sup>4</sup>.

### 3.3.2 Block Translation

The addition of the new block-pointer to the queue wakes up the LLVM thread which de-queues the pointer and gives it to the block translator. This stage generates LLVM-IR for each ARM instruction in the block in the guest binary referenced by the de-queued pointer. The block translator is hand-written by us and its creation was the most labour-intensive part of this project. This block translator is, essentially, a partially complete ARM binary to LLVM-IR frontend for the QEMU virtual machine. In detail, the block-translation:

1. Sets up a function prototype with a call to `LLVMFunctionType()`, so that a pointer to the guest CPU state (`CPUARMState`) structure is passed as the first parameter making it accessible to the code generated by LLVM.
2. Allocates a new empty LLVM IR function using a call to `LLVMAddFunction()` with the prototype created in step 1 above.

<sup>3</sup>We also implemented translations for a number of logic operators such as `eor` and `oor` not on the list above.

<sup>4</sup>Any attempt at self-modification by the guest-binary causes QEMU to flush it's entire translation cache and map-table.

```

define { [16 x i32] }* @block_test({ [16 x i32] }*) {
entry:
  %tmp1 = getelementptr { [16 x i32] }* %, i32 0, i32 0, i32 0
  %tmp2 = getelementptr { [16 x i32] }* %, i32 0, i32 0, i32 2
  %tmp3 = load i32*, %tmp2
  store i32 %tmp3, i32* %tmp1
  ret { [16 x i32] }* %0
}

```

Figure 4: LLVM intermediate representation of `mov r2, r0`

3. Transcribes the ARM instructions one at a time with the help of an LLVM builder object made by a call to `LLVMCreateBuilder()` on the function described in step 2 above. Using this builder LLVM IR instructions can be added to the function by calling functions such as: `LLVMBuildAdd()`, `LLVMBuildStore()`, `LLVMBuildIntToPtr()` and several others.
4. Finally, finishes off the function with a call to `LLVMBuildRet()` to build a return statement.

At the end of this process we have a pointer to the new LLVM IR function.

To help visualise block-translation, consider a hypothetical block consisting of a single ARM instruction<sup>5</sup>: `mov r2, r0`. Our Block Translator would build the LLVM IR function shown in figure 4.

A short semantics of this code is as follows. `%tmp1` holds a pointer into state pointing to `r0`, while `%tmp2` holds a pointer to `r2`. The value at the address in `%tmp2` is then loaded into `%tmp3`, which is subsequently stored at the location pointed to by `%tmp1`. At this stage our `mov` instruction is complete and the function returns.

Note that each basic block is mapped by our block translator to one LLVM-IR function. Each LLVM-IR function created by the block translator is immediately passed to the optimisation and code generation stages. We describe these stages next.

### 3.3.3 Code optimisation and generation

These stages are calls to pre-existing LLVM compiler components. Thus, from the implementer's point of view, this stage is straightforward. We first add optimisation passes to the LLVM JIT and then call the LLVM JIT to optimise and generate the LLVM IR block created by the block translator described in the previous section. In this project, we add optimisation passes by calling the following functions:

```

LLVMAddConstantPropagationPass();
LLVMAddPromoteMemoryToRegisterPass();
LLVMAddDeadInstEliminationPass();
LLVMAddDeadStoreEliminationPass();
LLVMAddInstructionCombiningPass();
LLVMAddGVNPass();

```

The purpose of each of these passes is fairly self-explanatory except for the `LLVMAddGVNPass` which does global value numbering to help eliminate equivalent values.

Once the passes are set up, then the LLVM JIT can be called on each block to perform translation. This translation can be triggered simply by calling `LLVMGetPointerToGlobal()` which returns a pointer to the optimised and translated host (x86-64) block. The host block produced from the code in figure 4 is shown in figure 5. This code bears some explana-

```

mov %r14, %rdi
mov 0x8(%rdi), %eax
mov %eax, (%rdi)
mov %rdi, %rax
ret

```

Figure 5: The x86-64 block produced by the LLVM JIT for the code shown in figure 4.

tion as the first instruction is not in-fact generated by LLVM. `mov %r14, %rdi` is related to the LLVM function prototype outlined previously: The x86-64 C ABI designates that the first struct pointer provided as a parameter to a function should be passed in register `%rdi`, however QEMU pins the CPU state pointer in `%r14`. As LLVM doesn't support pinned registers<sup>6</sup> the `mov` instruction is introduced as a work-around to move the CPU state pointer to `%rdi` where it can be accessed by the LLVM-generated code. The next two lines implement the actual move and the last two lines return the pointer to the register state. These last two lines are not required when the code is copied back to the TB during block replacement so they are dropped during this process.

Once the x86 block has been generated as above, it is added to the block-replacement queue. We describe how block replacement works next.

### 3.3.4 Block Replacement

The task of block replacement is to copy the host code blocks made by the LLVM JIT back into the correct TBs in the translation cache. Block replacement has to be carefully timed so as not to overwrite the contents of a block while it is running. In our implementation we took the simple choice of only allowing block replacement just before TCG is called to translate another new block. This choice guarantees safety – QEMU is guaranteed not to be running in the code cache when it is about to invoke TCG. Moreover, as we shall see in section 4.2 it allows for tolerably fast block replacement.

Block replacement is done by a simple memcpy back to the address of the original TB. The block replacing the original TB is usually shorter than the original TB. When it is shorter, we add code at the end of our new block to jump to the chaining section of the original TB (step (ix) of figure 3). In rarer cases where the optimised block is too big to fit back into the original TB we simply discard the new block on the assumption that, being so large, it is unlikely to be efficient in any case.

There is one more important detail to add. We had to make a special arrangement for dealing with the ARM compare instruction: `cmp`. This instruction is not easily implemented efficiently in TCG intermediate code so, instead, QEMU uses a pre-compiled

<sup>5</sup>We actually do get a few such one-instruction blocks in our benchmarks.

<sup>6</sup>which would have allowed us to use `%r14` directly

bench	total	queued	gen	repl	discard
hammer	5231	2417	2413	2297	116
gcc	69411	40912	40886	37461	3425
sjeng	4717	2529	2525	2460	65

Table 2: Comparison of blocks queued, generated, replaced and discarded compared to the total number of blocks in three SPECint 2006 benchmarks.

helper function to implement `cmps` semantics. However, when using the LLVM-JIT, by far the easiest option is to *inline* the x86 code for `cmp`. This adds, in the worst case, 48 bytes of extra space to our compiled block. To account for this we added a small amount of logic to the QEMU code that allocates space for each new block in its code cache. This logic allocates 48 bytes of additional space in each new TB to allow for a bigger inlined version of the TB to be copied back in.

This concludes our description of the modifications made to implement augmented QEMU. Next we assess the impact of these modifications on QEMU performance.

## 4 Results

This section presents our experimental results. All experiments were carried out on an Intel Core 2 Quad Processor Q8200, running at 2.3GHz, with 4GB of memory. This is an x86-64 architecture providing, more general purpose registers than IA-32 architectures.

We summarise our results in terms of code coverage (next), measured code size (section 4.3), timeliness of block-replacement 4.2, informal measures of code quality (section 4.4), and, importantly, code speed (section 4.5). We discuss each of these in turn.

### 4.1 Code Coverage

Code coverage is the number of blocks we are able to replace in our benchmarks. In Augmented QEMU this is, primarily, a function of choice and number of instructions supported by our block translator. We found our code coverage to be quite consistent across benchmarks. Table 2 presents statistics collected for three SPECint 2006 benchmarks running on test data. As can be seen in all three applications slightly more than half the total number of basic blocks were found to be eligible for block-translation and queued. Of these, most had time to be generated and most of these were small enough to replace the original block. A very small percentage were discarded because they were too large - perhaps indicating reasonably effective optimisation. In summary, Augmented QEMU, is able to achieve moderately good code coverage with a small number of implemented instructions. A very high percentage of eligible blocks go on to be replaced.

### 4.2 Timeliness of Replacement

As we can see from above, a reasonable number of blocks are being replaced on the benchmarks. However, these blocks will not do much good if they are not being replaced in a timely manner.

To assess the speed of block replacement we plotted a count of blocks queued and blocks replaced over time during the running of the `sjeng` benchmark. Figure 6 shows the blocks queued (blue dots) and the

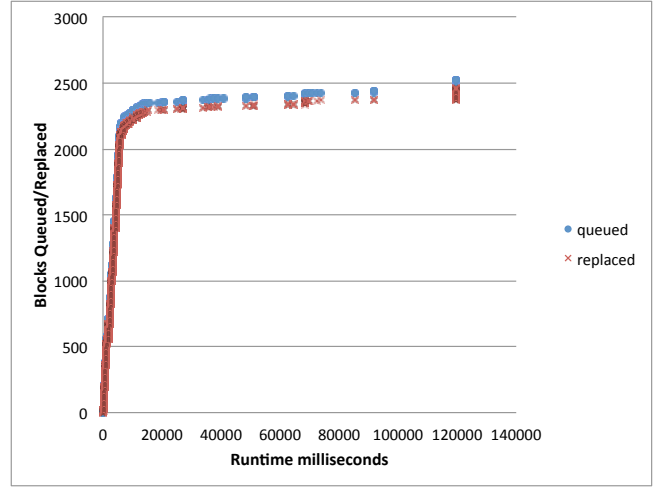


Figure 6: Cumulative block totals for queueing (blue dots) and replacement (red dots) over length of `sjeng` benchmark on test workload

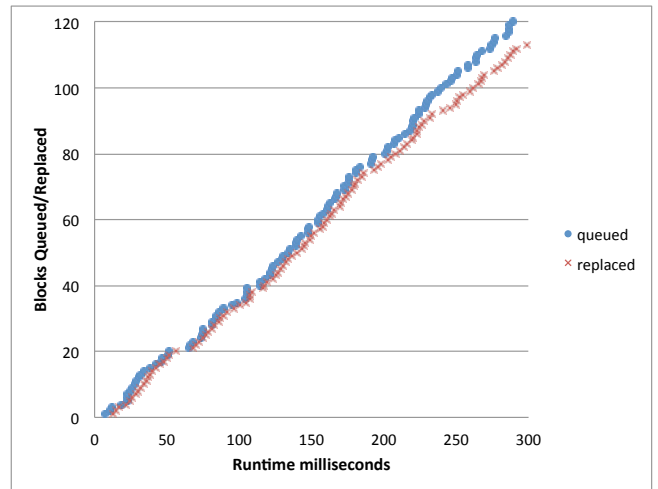


Figure 7: Zoom in on first part of figure 6.

blocks replaced (red dots) over the entire length of the `sjeng` benchmark on the test workload. As can be seen in this benchmark, queueing and replacement of blocks track each other closely. Moreover the great majority of blocks are replaced quickly. This fast replacement helps maximise their chance of being run. Note that a small gap grows between blocks queued and blocks generated as the run proceeds. This gap is due to the small number of blocks being discarded for being too large. Our manual tracking of blocks queued and replaced on other benchmarks revealed a similar pattern to above.

The sheer number of blocks replaced in the graph above makes it difficult to discern the pattern of individual replacements. In order to see some of these figure 7 zooms in on the first part of of figure 6. As can be seen, even at this scale, the rate of queueing and replacement is relatively steady. However the small gaps visible in the graph indicate either:

1. A temporary lack of *need* to replace blocks due to the replacement queue being empty; or
2. A temporary lack of *opportunity* to replace blocks due to `cpu_exec()` finding all the blocks it needs in the map table, thus not triggering TCG/replacement; or

Benchmark	raw size	opt size	reduction
hammer	79.7	44.8	41%
gcc	85.44	46.6	42%
sjeng	104.9	56.1	44%

Table 3: Percentage of optimised blocks that are smaller and larger and the percentage of code saved on replaced blocks for three benchmarks

```
0x400818b4: add ip, pc, #0
0x400818b8: add ip, ip, #147456
0x400818bc: ldr pc, [ip, #1796]!
```

Figure 8: An example ARM code block

3. A temporary lack of *opportunity* to replace blocks because QEMU is running inside a chain in the translation cache and is not passing control to `cpu_exec()`.

The first scenario above is benign. The second and third are not. The second scenario can occur because we only trigger replacement when TCG is triggered. To test the impact of the second scenario we changed the replacement strategy so replacement could also occur at regular intervals when TCG is not triggered. We found that, across a range of benchmarks, this alternative strategy of periodic checking had almost no impact on the timing of replacements and a negative impact on runtimes (due to the maintenance of an interval counter in `cpu_exec()`). We tested for the third scenario and found that, while on most benchmarks QEMU spends only short intervals running chained blocks in the translation cache there were some benchmarks which spent long times. The mcf benchmark in particular spent a substantial proportion of its execution time running within a single chain. Such behaviour, while good for performance, has the potential to impact negatively on the current replacement strategy of augmented QEMU.

### 4.3 Code Size

Augmented QEMU produces significant reductions in the amount of code in the blocks it replaces. Table 3 shows results for three SPECint 2006 benchmarks, gcc, hammer and sjeng on test workloads (adjusted for the inlined `cmp` instructions mentioned previously). The results show that almost all blocks that are optimised result in smaller code with substantial savings in space overall. In general we assume that this reduction in code size correlates with more efficient code. We look more closely at this issue next.

### 4.4 Code Quality

In this context we define code quality by efficiency and lack of redundancy. To ascertain the relative quality of the code being produced by the LLVM-JIT we performed a side-by-side inspection of a number of optimised and unoptimised cache blocks. Some of these revealed very substantial improvements. For example the ARM code block shown in figure 8 is translated by TCG into the reasonably compact x86 code shown in figure 9. However, the code produced by the LLVM-JIT is much better still, just two instructions:

```
mov %r14, %rdi
mov 0x400a58bc, 0x30(%rdi)
```

```
xor %r12d,%r12d
mov $0x400818bc,%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
mov $0x24000,%r12d
mov 0x30(%r14),%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
```

Figure 9: TCG translation of code from figure 8

the first of which is just moving the pointer to the CPU state struct. The savings come from eliminating a redundant move of the `pc` to `ip` and realising that, in this case `pc` can be recognised as a constant and folded in. These are just standard optimisations but TCG, which has to stamp its code out very quickly, doesn't have time to perform optimisations stretching over more than one ARM instruction.

The LLVM-JIT was also successful in removing a lot of redundant loads and stores which saved values out to the CPU state struct only to read them in again.

### 4.5 Code Speed

To test the raw speed of Augmented QEMU we ran 11 SPECint 2006 benchmarks on ARM code produced by GCC 4.4.3 (arm-softfloat-linux-gnueabi). All benchmarks were run on test workloads with the exception of `specrand`, which was run against its reference workload and `gobmk` which was run against its 13x13.txt reference workload in order to get longer runtimes.

Depending on length, each benchmark was run between 5 and 30 times and averaged. An exception was the `gobmk` benchmark which, due to its very long run-time on a reference workload was run twice. Absolute runtimes varied between three seconds for `libquantum` and 40 minutes for `gobmk`. In our experiments we measured real runtimes, with all QEMU logging turned off. The machine was dedicated to these experiments and so was lightly loaded at the time giving very consistent results - usually to within half-a-percent variation.

Figure 10 shows the relative speeds of:

**raw-qemu:** The time for vanilla-QEMU always normalised to one.

**no-replace:** The time for augmented-QEMU but without replacing any blocks - this gives an indication of overhead.

**replace:** The time for augmented-QEMU - blocks are replaced.

**net:** The net cost of augmented-QEMU assuming that there are no overheads.

Note, in order to make-visible the impact of overheads, the y-scale in figure 10 starts at 0.75. Speedup between vanilla-QEMU and augmented-QEMU varied from negative one percent on `gcc`, `omnetpp`, and `specrand` to 12 percent on the `mcf` benchmark. Speedup seemed not to correlate strongly with the size of the benchmark in terms of run time or in terms of the number of blocks it contained. It can be conjectured that the benchmarks that exhibited some regularity in their computation over time gained most but this will require further investigation to confirm.

The overhead of running augmented-QEMU can be estimated as the difference between `raw-qemu` and

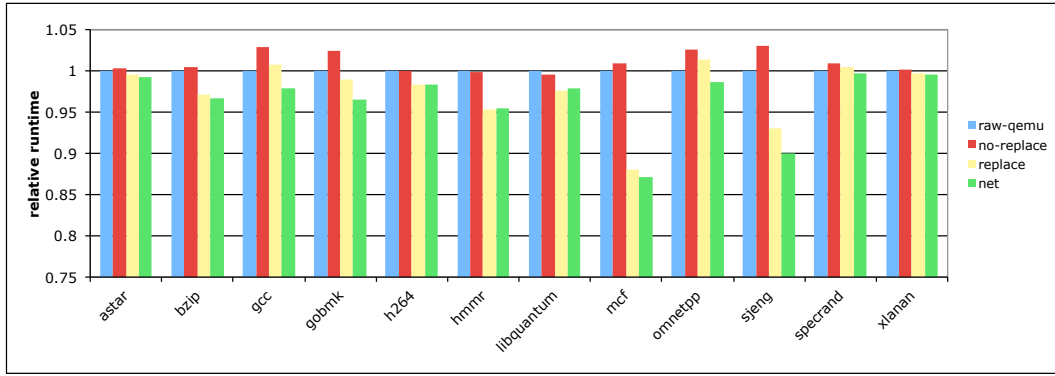


Figure 10: Relative run-times of various forms of QEMU on SPECint 2006 benchmarks. (note: scale starts at 0.75).

no-replace in figure 10. In our experiments with the sjeng benchmarks almost all of this overhead is the cost of detecting which blocks are eligible for translation. Running the translation and the LLVM-JIT in a separate thread had no discernible impact on times and locks on the ring-buffers suffered almost no contention.

If the costs of the overhead are subtracted from the augmented-QEMU run-times then we have a rough estimate of the impact of code improvement sans overheads. The last *net* column for each benchmark in figure 10 represents this measure. In all cases the impact of the code replacement was either positive or neutral<sup>7</sup>.

Note that any of the improvements shown above can only come from TBs that are run after replacement. If a replaced TB is run only for a short time or not run at all after replacement it can have very limited impact. We briefly investigated this issue by modifying the test workload input to the sjeng benchmark so it was forced to repeat the computation involved in the test workload three times. This gives more time for the replaced blocks to have an impact in reducing runtime. We found that there was very little reduction in runtime from running with the replaced blocks for longer. This seems to indicate, that on this benchmark, that the replacement-blocks that reduce runtime appear in the translation-cache early.

## 5 Conclusions, Limitations and Future work

In this article we have described Augmented-QEMU, a set of small modifications to QEMU that leverage the LLVM-JIT, running on a separate processor core, to improve simulation performance. This work demonstrates that it is feasible to improve the speed of an already-fast ISS by conservative, incremental, and minimally intrusive changes using an established compiler framework. While work to date is promising, it is a work in progress, and there are a number of limitations to the current framework that we plan to address in our future work. These limitations are:

**Code Coverage** Currently, not all ARM instructions have translations to LLVM-IR. We will continue to implement these incrementally to improve coverage and performance.

**Single Background Core** Currently we only use one core for optimisation. This limitation is

<sup>7</sup>Though the libquantum benchmark actually displayed a very slightly shorter runtime in the no-replace case. This is likely to be due to sampling noise between runs since libquantum has a very short run-time.

partly due to LLVM versions 2.6 and, to a lesser extent, 2.7 not being perfectly amenable to multi-threaded execution. This limitation appears to be addressed in version 2.9 and we plan to expand background optimisation to more cores using this version.

**ARM-only Guest** The current implementation is specialised to translate only ARM guest code. We made this choice primarily because of the expressiveness and relative ease of translation in the LLVM framework of the ARM instruction set relative to TCG intermediate code. However, full-portability, could be achieved if we do translate from TCG intermediate code to LLVM IR and this is a worthy future goal.

**Timing of Replacement** The limiting of block replacement to when TCG is running is safe and is low-overhead but can potentially lead to blocks not being replaced until after they are needed. A better future strategy is to build our own translation cache, complete with chaining, and perform thread-safe fix-ups to do indirect jumps to the translation cache.

**Exception Emulation** QEMU keeps track of every register in the processor state except the SPC while it runs the TB. When an exception or interrupt is triggered vanilla-QEMU rebuilds the real value of the SPC by abstractly re-executing from the start of the current TB, keeping track of what would happen to the SPC on the way. When the host PC reaches the point at which the exception was triggered the SPC is recorded in the processor state and the exception can then be handled. Our block replacement strategy prevents this abstract-re-execution to get the SPC value. While this is not an issue for SPECint benchmarks in user-mode QEMU it will need to be addressed in future. One good way to address this is to build our own translation cache, this would leave QEMU's Translation Cache intact allowing QEMU's current mechanism to work.

Note that the last two of these limitations can be addressed by building a separate translation-cache for our LLVM-JIT generated code. A new code cache would also open up opportunities to improve chaining form and perform new inter-block optimisations. Finally, using a separate code cache would give us opportunity to build super-blocks with the potential to be efficiently saved either in binary form or as LLVM-IR to speed up future runs as is done in SIMIT-ARM. We believe that an incrementally constructed

combination of background optimisation, building of large chains, and code persistence has the potential to greatly increase the execution speed of ISSs in future.

## References

- K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, pages 2–13, 2006.
- O. Almer, I. Böhm, T. E. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *International Symposium on Systems, Architectures, Modelling and Simulation (SAMOS'11)*, 2011.
- B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005. ISSN 0018-8670. doi: 10.1147/sj.442.0399.
- M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840305.
- F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.
- D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275, 2003.
- A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. Fx132 a profile-directed binary translator. *Micro, IEEE*, 18(2):56–64, 1998. ISSN 0272-1732. doi: 10.1109/40.671403.
- V. Chipounov and G. Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, 2010.
- R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS*, pages 128–137, 1994.
- G. De Michell and R. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, Mar. 1997. ISSN 0018-9219. doi: 10.1109/5.558708.
- Google .Inc. Android emulator, 2011.
- A. Jeffery. Using the llvm compiler infrastructure for optimised, asynchronous dynamic translation in qemu. Master’s thesis, School of Computer Science, University of Adelaide, 2010.
- W. Jiang, Y. Zhou, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu. Cfs optimizations to kvm threads on multi-core environment. In *ICPADS*, pages 348–354, 2009.
- C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.
- P. A. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE*, pages 94–104, 2007.
- C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.
- J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. Facsim: a fast and cycle-accurate architecture simulator for embedded systems. In *LCTES*, pages 89–100, 2008.
- W. H. Lee. Arm instruction set simulation on multi-core x86 hardware. Master’s thesis, School of Computer Science, University of Adelaide, 2009.
- LLVM-Project. Llvm users, 2011. URL <http://llvm.org/Users.html>.
- P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002. ISSN 0018-9162. doi: 10.1109/2.982916.
- OVP, April 2011. URL <http://www.ovpworld.org/>.
- W. Qin, J. D’Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS*, pages 193–198, 2006.
- T. Scheller. LLVM-QEMU, Google Summer of Code Project, 2008.
- Standard-Performance-Evaluation-Corporation. Spec cpu2006 cint2006 benchmarks, 2011. URL <http://www.spec.org/cpu2006/>.
- N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254, 2004.
- J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J*, 2008, February 2008. ISSN 1075-3583.
- D. Wentzlaff and A. Agarwal. Constructing virtual architectures on a tiled processor. In *CGO*, pages 173–184, 2006.
- E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24:68–79, May 1996. ISSN 0163-5999. doi: <http://doi.acm.org/10.1145/233008.233025>. URL <http://doi.acm.org/10.1145/233008.233025>.

---

We’d like to thank Tillmann Scheller for sharing his insights from the LLVM-QEMU project; Wanghao Lee for invaluable early input on this project; and the research team at ASTC for their expert advice and enthusiastic support.



# On the parameterized complexity of dominant strategies

Vladimir Estivill-Castro

Mahdi Parsa

School of ICT, Griffith University, Queensland, Australia 4111  
Email: {v.estivill-castro, m.parsa}@griffith.edu.au

## Abstract

In game theory, a strategy for a player is dominant if, regardless of what any other player does, the strategy earns a better payoff than any other. If the payoff is strictly better, the strategy is named strictly dominant, but if it is simply not worse, then it is called weakly dominant.

We investigate the parameterized complexity of two problems relevant to the notion of domination among strategies. First, we study the parameterized complexity of the MINIMUM MIXED DOMINATING STRATEGY SET problem, the problem of deciding whether there exists a mixed strategy of size at most  $k$  that dominates a given strategy of a player. We show that the problem can be solved in polynomial time on win-lose games. Also, we show that it is a fixed-parameter tractable problem on  $r$ -sparse games, games where the payoff matrices of players have at most  $r$  nonzero entries in each row and each column. Second, we study the parameterized complexity of the ITERATED WEAK DOMINANCE problem. This problem asks whether there exists a path of at most  $k$ -steps of iterated weak dominance that eliminates a given pure strategy. We show that this problem is  $W[2]$ -hard, therefore, it is unlikely to be a fixed-parameter tractable problem.

**Keywords:** Algorithm, Complexity, Computational game theory, dominant strategies, parameterized complexity

## 1 Introduction

Game theory is a mathematical framework for the study of conflict and cooperation between intelligent agents. This theory offers models to study decision-making situations and proposes several long-standing solution concepts.

A game consists of a set of players, a set of strategies for each player, and a specification of payoffs for each combination of strategies. Each single strategy in the set of strategies of a player is called a pure strategy. However, if a player randomly chooses a pure strategy, we say that the player is using a mixed strategy. In each game, players want to optimize their payoff which depends both on their own choices and also the choices of others.

Here, we use the Prisoners' Dilemma, a classical example in game theory, to introduce the concept of dominant strategy. In this game, two prisoners, the row player and the column player, are collectively charged with a crime and held in separate cells with no way of communicating. Each prisoner has two choices, cooperate ( $C$ ) which means not defect his partner or defect ( $D$ ), which

Table 1: Payoff matrix of the players in Prisoners' Dilemma.

		column player	
		$C$	$D$
row player	$C$	-1,-1	-10,0
	$D$	0,-10	-9,-9

means betray his partner. The punishment for the crime is ten years of prison. Betrayal gives a reduction of one year for the confessor. If a prisoner is not betrayed by its partner, he is convicted to one year for a minor offense. This situation can be summarized in Table 1. The numbers in the table represent the payoff for the players and there are two payoffs at each position: by convention the first number is the payoff for the row player and the second number is the payoff for the column player. In this game, the strategy defecting ( $D$ ) gives a better payoff for both players no matter how that player's opponents may play. Strategies like the strategy ( $D$ ) are called *dominant* strategies.

The notion of dominant strategies is a more elementary than the well-known solution concept Nash equilibrium. A Nash equilibrium is a set of strategies, one for each player, such that all players have no incentive to unilaterally change their decision. The elimination of dominated strategies can be used as a preprocessing technique for computing Nash equilibria. For example, in the Prisoners' Dilemma, after eliminating the dominated strategies  $C$  for each player, the remaining respective strategies  $D$  specify a Nash equilibrium.

Gilboa, Kalai and Zemel (Gilboa et al. 1993) used classical complexity theory and showed that many decision problems regarding to computation of dominant strategies are **NP**-complete for two-player games. Later, other researches (Brandt et al. 2009, Conitzer & Sandholm 2005) extended their hardness results to other classes of games such as win-lose games.

In this paper, we study two problems relevant to the notion of domination from the perspective of parameterized complexity. Hence, we are interested in algorithms that compute exact optimal solutions, while attempting to confine the inevitable exponential-running time of such algorithms to an input-length independent parameter.

First, we study the parameterized complexity of MINIMUM MIXED DOMINATING STRATEGY SET problem.

### MINIMUM MIXED DOMINATING STRATEGY SET

**Instance** : Given the row player's payoffs of a two-player game  $G$  and a distinguished pure strategy  $i$  of the row player.

**Parameter** : An integer  $k$ .

**Question** : Is there a mixed strategy  $x$  for the row player that places positive probability on at most  $k$  pure strategies, and dominates the pure strategy  $i$ ?

A strategy may fail to be dominated by a pure strategy, but may be dominated by a mixed strategy. Here, we focus on specializations of two-player games (by revisiting the original NP-completeness proof (Conitzer & Sandholm 2005) we can discover that this is a parameterized reduction). Thus, the problem is  $W[2]$ -hard, and it is unlikely to be in FPT for general two-player games. First, we focus on win-lose games, games where the payoff values are limited to 0 and 1. We show that this problem can be solved in polynomial time on win-lose games (Lemma 3.4). Second, we investigate this problem on  $r$ -sparse games. Here the payoff matrices have at most  $r$  nonzero entries in each row and each column. We show that MINIMUM MIXED DOMINATING STRATEGY SET is fixed-parameter tractable for  $r$ -sparse games (Theorem 3.5).

Next, we study the parameterized complexity of the ITERATED WEAK DOMINANCE problem.

#### ITERATED WEAK DOMINANCE

**Instance** : A two-player game and a distinguished pure strategy  $i$ .

**Parameter** : An integer  $k$ .

**Question** : Is there a path of at most  $k$  steps of iterated weak dominance that eliminates the pure strategy  $i$ ?

It is well-known that iterated strict dominance is path-independent, that is, the elimination process will always terminate at the same point, and the elimination procedure can be executed in polynomial time (Gilboa et al. 1993). In contrast, iterated weak dominance is path-dependent and it is known that whether a given strategy is eliminated in some path is NP-complete (Conitzer & Sandholm 2005). We show that this problem is  $W[2]$ -hard, therefore it is unlikely to be fixed-parameter tractable (Theorem 4.1).

The rest of the paper is organized as follows. In Section 2 we give formal definitions for games, and parameterized complexity theory. In Section 3, we show the fixed-parameter tractability results. In Section 4 we show our parameterized hardness results. In Section 5 we discuss further the implications of our results and some open problems.

## 2 Preliminaries

In this section, we review relevant concepts game theory, and computational complexity theory including parameterized complexity.

### 2.1 Parameterized complexity theory

Parameterized complexity aims at providing an alternative to exponential algorithms for NP-complete problems by identifying a formulation where the parameter would take small values in practice and shifting the exponential explosion to this parameter while the rest of the computation is polynomial in the size of the input.

A *parameterized problem* is a language  $L \subseteq \Sigma^* \times \mathbb{N}$  where the second part of the problem is called the parameter. A parameterized problem  $L$  is *fixed-parameter tractable* if there is an algorithm that decides in  $f(k)|x|^{O(1)}$  time whether  $(x, k) \in L$ , where  $f$  is an arbitrary computable function depending only on the parameter  $k$ . Such an algorithm is called FPT-time algorithm, and **FPT** denotes the complexity class that contains all fixed-parameter tractable problems.

In order to characterize those problems that do not seem to admit an FPT algorithm, Downey and Fellows (Downey & Fellows 1998) defined a *parameterized reduction* and a hierarchy of classes  $W[1] \subseteq W[2] \subseteq \dots$

including likely fixed parameter intractable problems. A (many-to-one) *parameterized reduction* from a parameterized problem  $L$  to a parameterized problem  $L'$  is an FPT-time mapping  $\Phi$  that transforms an instance  $(x, k)$  of  $L$  into an instance  $(x', k')$  such that,  $(x, k) \in L$  if and only if  $(x', k') \in L'$ , where  $k'$  bounded by some function depends only on the parameter  $k$ .

The class  $W[t]$  is defined to be the class of all problems that are reducible to a parameterized version of the satisfiability problem for Boolean circuits of weft  $t$  (see Downey and Fellows (Downey & Fellows 1998) for the exact definition).

The above classes may be equal (if  $\text{NP}=\text{P}$ , for example); however, there is evidence to suspect (Downey & Fellows 1998) that  $W[2]$ -completeness is a strong indication of intractability in the FPT sense. The best known algorithm for any  $W[2]$ -complete problem is still just the brute force algorithm of trying all  $k$  subsets which has a running time  $O(n^{k+1})$ .

### 2.2 Games and dominant strategies

A two-player *normal form* game  $\mathcal{G}$  consists of two matrices  $A = (a_{ij})_{m \times n}$  and  $B = (b_{ij})_{m \times n}$ , where  $a_{ij}$  denotes the payoff for the first player and  $b_{ij}$  denotes the payoff for the second player when the first player plays his  $i$ -th strategy and the second player plays his  $j$ -th strategy. We identify the first player as the row player and the second player as the column player. Each single strategy in the set of strategies of a player is called a *pure strategy*. However, if a player randomly chooses a pure strategy, we say that the player is using a mixed strategy.

**Definition 2.1** An ordered  $n$ -tuple  $\mathbf{x} = (x_1, \dots, x_n)$  with  $\sum_{i=1}^n x_i = 1$  and  $\mathbf{x} \geq 0$  is a *mixed strategy*.

Thus, a mixed strategy is a probability distribution over the pure strategy space. The *support* (denoted  $\text{supp}(\mathbf{x})$ ) of a mixed strategy  $\mathbf{x}$  is the set of pure strategies which are played with positive probability, that is  $\{i : 1 \leq i \leq n, x_i > 0\}$ .

In a two-player game  $\mathcal{G}=(A, B)$ , a strategy  $i$  of the row player is said to *weakly dominate* a strategy  $i'$  of the row player if for every strategy  $j$  of the column player we have  $a_{ij} \geq a_{i'j}$  and there exists a strategy  $j_0$  of the column player that  $a_{ij_0} > a_{i'j_0}$ . The strategy  $i$  is said to *strictly dominate* strategy  $i'$  if for every strategy  $j$  of the column player  $a_{ij} > a_{i'j}$ . A similar definition is used to define the domination relation of the column player, but now using the payoff matrix  $B$ .

If a strategy is dominated, the game (and thus the problem) can be simplified by removing it. Eliminating a dominated strategy may enable elimination of another pure strategy that was not dominated at the outset, but is now dominated. The elimination of dominated strategies can be repeated until no pure strategies can be eliminated in this manner. In a finite game this will occur after a finite number of eliminations and will always leave at least one pure strategy remaining for each player. This process is called *iterated dominant strategies* (Gilboa et al. 1993).

Note that a strategy may fail to be strongly eliminated by a pure strategy, but may be dominated by a mixed strategy.

**Definition 2.2** Consider strategy  $i$  of the row player in two-player game  $(A, B)$ . We say that the strategy  $i$  is *dominated by a mixed strategy*  $\mathbf{x} = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$  of the row player, if the following holds for every strategy  $j$  of the column player

$$\sum_{i' \neq i} x_{i'} a_{i'j} \geq a_{ij}.$$



### 3 FPT results on mixed strategy domination

Recall that sometimes a strategy is not dominated by any pure strategy, but it is dominated by some mixed strategies. Example 3.1 illustrates the differences between these two types of strategies.

**Example 3.1** Consider the payoff matrix of the row player that is given as follows:

$$\begin{pmatrix} 4 & 0 & 2 \\ 0 & 4 & 0 \\ 1 & 1 & 0 \end{pmatrix}.$$

In this situation, no pure strategy can eliminate any other. However playing the first and the second strategy with probability  $1/2$ , dominates the third strategy. Because, the expected payoff of those two strategies is equal to  $1/2 \cdot (4, 0, 2) + 1/2 \cdot (0, 4, 0) = (2, 0, 1) + (0, 2, 0) = (2, 2, 1)$ .

Moreover, we can test in polynomial time, whether a given strategy of a player is dominated by a mixed strategy of the same player. The following proposition shows the tractability of this issue.

**Proposition 3.2** Consider a two-player game  $G = (A_{m \times n}, B_{m \times n})$ , a subset  $S'$  of the row player's pure strategies, and a distinguished strategy  $i$  for the row player. We can determine in polynomial time (in the size of the game) whether there exists a mixed strategy  $\mathbf{x}$ , that places positive probability only on strategies in  $S'$  and dominates the pure strategy  $i$ . Similarly, for the column player, a subset  $S'$  of the column player's pure strategies, and a distinguished strategy  $j$  for the column player. We can determine in polynomial time (in the size of the game) whether there exists a mixed strategy  $\mathbf{y}$ , that places positive probability only on strategies in  $S'$  and dominates the pure strategy  $j$ . This applies both for strict and weak dominance (Conitzer & Sandholm 2005).

Nevertheless, finding such a mixed strategy that dominates a pure strategy with the smallest support size (MINIMUM MIXED DOMINATING STRATEGY SET) is computationally hard (NP-complete) (Conitzer & Sandholm 2005). Moreover, it is not hard to obtain a proof of  $W[2]$ -hardness for this problem. The original proof (Conitzer & Sandholm 2005) introduced a reduction from SET COVER, a  $W[2]$ -complete problem, to MINIMUM MIXED DOMINATING STRATEGY SET. We just need to verify that it is a parameterized reduction. Furthermore, this  $W$ -hardness result shows that it is unlikely to find an FPT algorithm for this problem by considering only the size of the support as the parameter.

Moreover, the review of the proof reveals that the constructed instances of the MINIMUM MIXED DOMINATING STRATEGY SET problem in the reduction have limited payoffs, which are  $\{0, 1, k + 1\}$ . Therefore, a natural question to ask next is whether it is possible to find an FPT algorithm by considering extra conditions on the problem instances. Our first step would be specializing the games to win-lose games. Recall that in win-lose games, the given payoffs are in  $\{0, 1\}$ . The following lemma shows that this restriction makes the problem easy.

**Lemma 3.3** In a win-lose game  $G = (A, B)$  every pure strategy that is weakly dominated by a mixed strategy is also weakly dominated by a pure strategy.

*Proof:* Consider a mixed strategy  $\mathbf{x} = (x_1, x_2, \dots, x_m)$  that dominates a pure strategy  $i$  (without loss of generality, both of course, of the row player). Clearly, for any strategy  $j$  of the column player where  $a_{ij} = 0$ , the expected payoff of playing the mixed strategy in the

column  $j$  is at least 0. Therefore, we only need to consider columns  $j$  where  $a_{ij} = 1$ . Let  $j_0$  be a first column where  $a_{ij_0} = 1$ . Because  $\mathbf{x}$  dominates the strategy  $i$  there is a row (strategy)  $r$  in the mixed strategy  $\mathbf{x}$  where  $x_r > 0$  and  $a_{rj_0} = 1$ . We claim that row  $r$  weakly dominates row  $i$ . We just need to show that  $a_{rj} = 1$  for any column  $j$  where  $a_{ij} = 1$ . However, if  $a_{rj} = 0$ , then for the  $j$ -th column we have  $\sum_{i=1}^m a_{ij}x_i = \sum_{i \neq r} a_{ij}x_i + r_j x_j < 1$ . This contradicts the hypothesis that  $\mathbf{x}$  dominates  $i$ .  $\square$

**Lemma 3.4** MINIMUM MIXED DOMINATING STRATEGY SET is in **P** (that is, it can be decided in polynomial time) if it is limited to win-lose games.

*Proof:* By Lemma 3.3, if a pure strategy  $i$  is dominated by a mixed strategy  $\mathbf{x}$ , then there exists a pure strategy  $i'$  that dominates  $i$ . Therefore, the problem reduces to the problem of finding a pure strategy that dominates  $i$ . This can be done in polynomial time in the size of the game.  $\square$  Our first effort for specializing the problem makes it an easy problem (class **P**). Therefore, instead of limiting the payoffs, we will work on limiting the number of non-zero entries in each row and each column of the payoff matrix of the row player. The MINIMUM MIXED DOMINATING STRATEGY SET problem remains NP-complete even on  $r$ -sparse games with  $r \geq 3$  (Conitzer & Sandholm 2005, Garey & Johnson 1979).

**Theorem 3.5** MINIMUM MIXED DOMINATING STRATEGY SET problem for  $r$ -sparse games (when considering  $r$  as the parameter) is in the class FPT.

*Proof:* Consider an  $r$ -sparse instance of MINIMUM MIXED DOMINATING STRATEGY SET. Without loss of generality we can assume the last row of the first player is the strategy to be dominated by a mixture of another  $k$  strategies. Because of Proposition 3.2, finding a mixed strategy that weakly dominates the distinguished strategy reduces to the problem of determining the support of the mixed strategy. Consider the following procedure.

**Step 1:** We remove (in polynomial time) all columns where the last row has a zero payoff. Because, all payoffs are at least zero in each column, any mixed strategy that dominates those columns with positive entries of the distinguished strategy also does so where the distinguished strategy has zeros. As the game is  $r$ -sparse, this step reduces the size of the payoff matrix of the row player to a matrix with  $r$  columns.

**Step 2:** If there is a column where all entries in that column are less than the last entry in the column, then the instance is a no-instance.

**Step 3:** Now remove all rows that are made completely of zeros. Because there are at most  $r$  entries different than zero in each column, the matrix now has at most  $r^2$  rows. We can test exhaustively all subsets of rows of size  $k$  of the first  $r^2 - 1$  rows for domination of the now  $r^2$ -th row. If none of the tests results in domination, we have a no-instance, otherwise we have a yes-instance and a certificate of the domination.

The only step that is not polynomial is the exhaustive verification at the end; however, this is polynomial in  $r$  as there are  $\binom{r^2-1}{k} = O(r^{2k})$  such subsets. This problem can be solved in  $f(r)poly(n)$  because  $k < r^2$ .  $\square$

#### 4 ITERATED WEAK DOMINANCE (IWD)

As discussed earlier, iterated elimination of strictly dominated strategies is conceptually straightforward in a sense that regardless of the elimination order the same set of strategies will be identified, and all Nash equilibria of the original game will be contained in this set. However, this process becomes a bit trickier with the iterated elimination of weakly dominated strategies. In this case, the elimination order does make a difference, that is, the set of strategies that survive iterated elimination can differ depending on the order in which dominated strategies are eliminated. Therefore, the problem such as deciding whether a strategy can be eliminated in a path of iterated weakly dominated absorbed more attention. ITERATED WEAK DOMINANCE is a **NP**-complete problem (Conitzer & Sandholm 2005) even in games with payoffs in  $\{(0, 0), (0, 1), (1, 0)\}$  (Brandt et al. 2009). Here, we show its hardness in terms of parameterized complexity theory.

**Theorem 4.1** *The IWD STRATEGY ELIMINATION problem is  $W[2]$ -hard.*

We prove this by providing a parameterized reduction from SET COVER. Therefore, consider an instance of SET COVER. That is, we are given a set  $S = \{1, 2, \dots, n\}$  and a family  $\mathcal{F}$  of proper subsets  $S_1, \dots, S_r$  that cover  $S$  (that is,  $S_i \subset S$ , for  $i = 1, \dots, r$  and  $S = \bigcup_{i=1}^r S_i$ ). The question is whether there is a sub-family of  $k$  or fewer sets in  $\mathcal{F}$  that also covers  $S$ .

Our proof constructs, a game  $\mathcal{G} = (A, B)$  and the question, whether the last row of the matrix  $A$  can be eliminated by iterated weak domination in  $k+1$  or fewer steps. Because  $k$  is the parameter of the SET COVER instance and  $k' = k+1$  is the parameter of the ITERATED WEAK DOMINANCE (IWD) this would be a parameterized reduction.

We start by describing the payoff matrices of the game  $\mathcal{G} = (A, B)$ . The number of rows of the matrices is  $|\mathcal{F}| + 1 = r+1$ . The number of columns is  $r+n+1$ .

We first describe the payoff matrix  $A$  of the row player. The last row of  $A$  will be

$$a_{r+1,j} = \begin{cases} 1, & j < n+r+1, \\ 0, & \text{otherwise.} \end{cases}$$

That is, this row has a 1 everywhere except for the last column.

The last column of  $A$  has a similar form.

$$a_{i,n+r+1} = \begin{cases} 1, & i < r+1, \\ 0, & \text{otherwise.} \end{cases}$$

That is, this column has a 1 everywhere except for the last row.

Now, the first block of  $r$  columns and  $r$  rows of  $A$  have a diagonal full with the value 0 and the value 1 everywhere else. We let the following entries of  $A$  defined by

$$a_{i,j} = \begin{cases} 1, & i \leq r \text{ and } j \leq r \text{ and } i \neq j, \\ 0, & i \leq r \text{ and } j \leq r \text{ and } i = j. \end{cases}$$

Finally, after the  $r$ -th column, the  $i$ -row has the characteristic vector of the set  $S_i$  scaled by  $k$ .

$$a_{i,j} = \begin{cases} k, & j-r \in S_i \text{ } i \leq r \text{ and } r+1 \leq j \leq r+n, \\ 0, & j-r \in S \setminus S_i, i \leq r \text{ and } r+1 \leq j \leq r+n. \end{cases}$$

We illustrate this construction with an example. Consider the set  $S = \{1, 2, 3, 4, 5, 6, 7, 8\}$ , and the parame-

ter  $k = 2$ . The family  $\mathcal{F}$  is defined as follows

$$\begin{aligned} S_1 &= \{1, 2, 3\}, & S_2 &= \{3, 5, 7\}, & S_3 &= \{4, 5, 6\}, \\ S_4 &= \{6, 7, 8\}, & S_5 &= \{1, 2, 4\}, & S_6 &= \{1, 3, 5, 7\}, \\ S_7 &= \{2, 4, 6, 8\}, & S_8 &= \{3, 4, 5\}, & \text{and } S_9 &= \{2\}. \end{aligned}$$

Therefore, the matrix  $A$  is given by

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 2 & 0 & 2 & 0 & 2 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 2 & 2 & 0 & 2 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 2 & 0 & 2 & 0 & 2 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 2 & 0 & 2 & 0 & 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 2 & 2 & 2 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

**Observation 4.2** *In the resulting matrix  $A$  it is impossible to perform a row elimination to eliminate the  $r+1$ -th row.*

Any convex combination of strategies in  $\{1, 2, \dots, r\}$  would add to less than one in one column in  $\{1, 2, \dots, r\}$ . Thus, there would be one column blocking such elimination.

**Observation 4.3** *Consider a yes-instance of the SET COVER problem, where  $I$  is the set of indexes in  $\{1, 2, \dots, r\}$  such that  $|I| \leq k$  and  $S \subseteq \bigcup_{i \in I} S_i$ . Removing the columns in  $I$  from  $A$  results in a configuration where the linear combination of rows in  $I$  with probability  $1/|I|$  eliminate row  $r+1$  in one step.*

To confirm this observation first note that any convex combination of rows in  $\{1, 2, \dots, r\}$  produces domination in the  $r+1$ -th column, and thus in particular the rows in  $I$ .

Now we show that the removal of the columns in  $I$  causes no longer a blockage. First, consider a column  $j \leq r$ . Since  $r$  is not in  $I$ , when we consider the convex combination of rows in  $I$ , that combination will add to a payoff of 1, which is equal to the value in row  $r+1$  and column  $j$ .

Finally, consider a column  $j$  with  $r < j < r+n+1$ . Because  $I$  is the set of indexes of a cover, all entries in the rows indexed by  $I$  have value  $k$  in column  $j$ . Therefore the linear combination with uniform probability  $1/|I|$  on the rows with index  $I$  will have at least one entry with weight  $k/|I| \geq 1$  since  $|I| \leq k$ .

To continue, we now need to describe the payoff matrix  $B$  for the column player. This matrix is made of two blocks. The first block is the first  $r$  columns, while the second block is the last  $n+1$  columns. All values are 0 for the first block and all values are 1 for the second block.

$$B = (0_{r+1 \times r} | 1_{r+1 \times n+1}).$$

**Observation 4.4** *The only columns that can be eliminated by a column elimination are one of the first  $r$  columns.*

This observation follows trivially from the structure of  $B$ , since the only dominations are strict dominations from a column in the later block of columns full of the value 1 to a column in the first  $r$  columns full of the value 0.

**Observation 4.5** *A row elimination cannot happen in matrix  $A$  until a set  $I \subseteq \{1, 2, \dots, r\}$  of columns is eliminated by column eliminations, and the set  $I$  defines a cover of  $S$ .*

We know the process of elimination must start with a column elimination. Because of the structure of the first  $r$  columns of  $A$ , the only row elimination possible after some columns eliminations must be a convex combination of a subset of indexes of the already eliminated indexes.

However, this would be a possible row elimination only if the linear combination also implies a set cover because of the structure of the next  $n$  columns of matrix  $A$ .

Now clearly if there is a path of length  $k + 1$  (or less) that eliminates row  $r + 1$  in matrix  $A$  it must consist of  $k$  (or less) column eliminations defining  $k$  (or less) indexes of the covering sets, and the last elimination is the corresponding row elimination with uniform weight on the same indexes. This completes the proof.

## 5 Conclusion

There are many interesting decision problems regarding the notion of dominant strategies (Gilboa et al. 1993, Conitzer & Sandholm 2005). We showed that one of those problems is parameterized hard problem (e.g. MINIMUM MIXED DOMINATING STRATEGY SET). Furthermore, we showed that special cases of those problems are in **P** or **FPT**. However, many other problems are still open. For example, the paper “On the Complexity of Iterated Weak Dominance in Constant-Sum Games” (Brandt et al. 2009) indicates that the ITERATED WEAK DOMINANCE problem is in **P** for Constant-Sum games, but still **NP**-complete for win-lose games. In fact, the paper is more precise. If of each pair of actions  $(i, j)$ , where  $i$  is for the row player and  $j$  is for the column player, the corresponding entries in  $A$  and  $B$  are only  $(1, 0)$ ,  $(0, 1)$ ,  $(0, 0)$  the problem remains **NP**-complete. Disallowing  $(0, 0)$  makes the problem restricted to constant-sum games and thus becomes a problem in **P**. However, our parameterized hardness proof uses other entries different from  $\{0, 1\}$ . We do not know the fixed-parameter complexity of ITERATED WEAK DOMINANCE in win-lose games or in the more restricted class of games where only  $(1, 0)$ ,  $(0, 1)$ ,  $(0, 0)$  are allowed.

## References

- Brandt, F., Brill, M., Fischer, F. & Harrenstein, P. (2009), On the complexity of iterated weak dominance in constant-sum games, in ‘SAGT-09, Proceedings of the 2nd International Symposium on Algorithmic Game Theory’, Springer-Verlag, Berlin, Heidelberg, pp. 287–298.
- Conitzer, V. & Sandholm, T. (2005), Complexity of (iterated) dominance, in J. Riedl, M. J. Kearns & M. K. Reiter, eds, ‘EC-05, Proceedings of the 6th ACM Conference on Electronic Commerce’, ACM, Vancouver, BC, Canada, pp. 88–97.
- Downey, R. & Fellows, M. R. (1998), *Parameterized Complexity*, Monographs in Computer Science, Springer, New York.
- Garey, M. R. & Johnson, D. S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, USA.
- Gilboa, I., Kalai, E. & Zemel, E. (1993), ‘The complexity of eliminating dominated strategies’, *Mathematics of Operations Research* **18**(3), 553–565.



# Single Feature Ranking and Binary Particle Swarm Optimisation Based Feature Subset Ranking for Feature Selection

Bing Xue

Mengjie Zhang

Will N. Browne

School of Engineering and Computer Science

Victoria University of Wellington, Wellington, New Zealand

Email: (Bing.Xue, Mengjie.Zhang, Will.Browne)@ecs.vuw.ac.nz

## Abstract

This paper proposes two wrapper based feature selection approaches, which are single feature ranking and binary particle swarm optimisation (BPSO) based feature subset ranking. In the first approach, individual features are ranked according to the classification accuracy so that feature selection can be accomplished by using only a few top-ranked features for classification. In the second approach, BPSO is applied to feature subset ranking to search different feature subsets. K-nearest neighbour (KNN) with n-fold cross-validation is employed to evaluate the classification accuracy on eight datasets in the experiments. Experimental results show that using a relatively small number of the top-ranked features obtained from the first approach or one of the top-ranked feature subsets obtained from the second approach can achieve better classification performance than using all features. BPSO could efficiently search for subsets of complementary features to avoid redundancy and noise. Compared with linear forward selection (LFS) and greedy stepwise backward selection (GSBS), in almost all cases, the two proposed approaches could achieve better performance in terms of classification accuracy and the number of features. The BPSO based approach outperforms single feature ranking approach for all the datasets.

**Keywords:** Feature selection, Particle swarm optimisation, Single feature ranking, Feature subset ranking

## 1 Introduction

In many fields such as classification, a large number of features may be contained in the datasets, but not all of them are useful for classification. Redundant or irrelevant features may even reduce the classification performance. Feature selection aims to pick a subset of relevant features that are sufficient to describe the target classes. By eliminating noisy and unnecessary

features, feature selection could improve classification performance, make learning and executing processes faster, and simplify the structure of the learned models (Dash & Liu 1997).

The existing feature selection approaches can be broadly classified into two categories: filter approaches and wrapper approaches. The search process in filter approaches is independent of a learning algorithm and they are argued to be computationally less expensive and more general than wrapper approaches (Dash & Liu 1997). On the other hand, wrapper approaches conduct a search for the best feature subset using the learning algorithm itself as part of the evaluation function. In a wrapper model, a feature selection algorithm exists as a wrapper around a learning algorithm and the learning algorithm is used as a “black box” by the feature selection algorithm. By considering the performance of the selected feature subset on a particular learning algorithm, wrappers can usually achieve better results than filter approaches (Kohavi & John 1997).

A feature selection algorithm explores the search space of different feature combinations to optimise the classification performance. The size of search space for  $n$  features is  $2^n$ , so it is impractical to search the whole space exhaustively in most situations (Kohavi & John 1997). Single feature ranking is a relaxed version of feature selection, which only requires the computation of the relative importance of the features and subsequently sorting them (Guyon et al. 2003). Feature selection can be accomplished by using only the few top-ranked features for classification. However, not much work has been done on wrapper based single feature ranking (Neshatian & Zhang 2009). Single feature ranking is computationally cheap, but the combination of the top-ranked features may be a redundant subset. The performance obtained by this subset could possibly be achieved by a smaller subset of complementary features.

In order to avoid exhaustive search, greedy algorithms are introduced to solve feature selection problems such as sequential forward selection (SFS) (Whitney 1971) and sequential backward selection (SBS) (Marill & Green 1963). They are the two most commonly used greedy search algorithms that are computationally less expensive than other approaches. Thus they are used as the basis for bench-

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

mark techniques to test novel approaches. Existing feature selection approaches, such as greedy search algorithms, suffer from a variety of problems, such as stagnation in local optima and high computational cost. Therefore, an efficient global search technique is needed to address feature selection problems. Particle swarm optimisation (PSO) is such a global search technique, which is computationally less expensive, easier to implement, has fewer parameters and can converge more quickly than other techniques, such as genetic algorithms (GAs) and genetic programming (GP). PSO has been successfully applied in many areas and it has been shown to be a promising method for feature selection problems (Yang et al. 2008, Unler & Murat 2010, Yang et al. 2008). However, PSO has never been applied to feature subset ranking (See Section 4), which is expected to obtain many feature subsets to meet different requirements in real-world applications.

### 1.1 Goals

This paper aims to develop a new approach to feature subset ranking for feature selection in classification problems with the goal of using a small number of features to achieve better classification performance. To achieve this goal, we will develop two new algorithms for finding a subset of features for classification. The two algorithms will be examined and compared with conventional feature selection approaches on eight benchmark datasets with different numbers of features and instances. Specifically, we will

- develop a simple wrapper based single feature ranking algorithm and investigate whether the combination of some top-ranked features generated by this algorithm can achieve better performance than using all features and can outperform conventional approaches; and
- develop a feature subset ranking algorithm using BPSO with heuristic search and investigate whether this algorithm can outperform the method of using all features, conventional approaches and the single feature ranking algorithm.

### 1.2 Organisation

The remainder of the paper is organised as follow. Background information is provided in Section 2. Section 3 describes the proposed wrapper based single feature ranking algorithm. The BPSO based feature subset ranking algorithm is proposed in Section 4. Section 5 describes experimental design and Section 6 presents experimental results with discussions. Section 7 provides conclusions and future work.

## 2 Background

### 2.1 Particle Swarm Optimisation (PSO)

PSO is an evolutionary computation technique proposed by Kennedy and Eberhart in 1995 (Kennedy & Eberhart 1995). In PSO, each solution can be represented as a particle in the search space. A vector  $x_i = (x_{i1}, x_{i2}, \dots, x_{iD})$  presents the position of particle  $i$ , where  $D$  is the dimensionality of the search space. The velocity of particle  $i$  is represented as  $v_i = (v_{i1}, v_{i2}, \dots, v_{iD})$ . The best previous position of each particle is recorded as the personal best called Pbest and the best position obtained thus far is called Gbest. The swarm is initialised with a population of random solutions and searches for the best solution by updating the velocity and the position of each particle according to the following equations:

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \quad (1)$$

$$\begin{aligned} v_{id}^{t+1} = & w * v_{id}^t + c_1 * r_1 * (p_{id} - x_{id}^t) \\ & + c_2 * r_2 * (p_{gd} - x_{id}^t) \end{aligned} \quad (2)$$

where  $t$  denotes iteration  $t$  in the search process.  $c_1$  and  $c_2$  are acceleration constants.  $r_1$  and  $r_2$  are random values uniformly distributed in  $[0, 1]$ .  $p_{id}$  presents the Pbest and  $p_{gd}$  stands for the Gbest.  $w$  is inertia weight. The velocity  $v_{id}^t$  is limited by a predefined maximum velocity,  $v_{max}$  and  $v_{id}^t \in [-v_{max}, v_{max}]$ .

PSO was originally introduced as an optimization technique for real-number search spaces. However, many optimization problems occur in a space featuring discrete, qualitative distinctions between variables and between levels of variables. To extend the implementation of the original PSO, Kennedy and Eberhart (Kennedy & Eberhart 1997) developed a binary particle swarm optimisation (BPSO) for discrete problems. The velocity in BPSO represents the probability of element in the particle taking value 1 or 0. Equation (2) is still applied to update the velocity while  $x_{id}$ ,  $p_{id}$  and  $p_{gd}$  are integers of 1 or 0. A sigmoid function  $s(v_{id})$  is introduced to transform  $v_{id}$  to the range of  $(0, 1)$ . BPSO updates the position of each particle according to the following formulae:

$$x_{id} = \begin{cases} 1, & \text{if } rand() < s(v_{id}) \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

where

$$s(v_{id}) = \frac{1}{1 + e^{-v_{id}}} \quad (4)$$

where  $s(v_{id})$  is a sigmoid limiting transformation.  $rand()$  is a random number selected from a uniform distribution in  $[0, 1]$ .

### 2.2 BPSO for Feature Selection

Generally, when using BPSO to solve feature selection problems (Unler & Murat 2010, Yang et al. 2008), the representation of a particle is a  $n$ -bit binary string,

where  $n$  is the number of features and the dimensionality of the search space. The feature mask is in Boolean such that “1” represents the feature will be selected and “0” otherwise. Many BPSO based filter and wrapper feature selection approaches have been proposed in recent years.

Chakraborty (2008) compares the performance of BPSO with that of GA in a filter feature selection approach with fuzzy sets based fitness function. The results show that BPSO performs better than GA in terms of classification accuracy.

Inertia weight can improve the performance of BPSO by properly balancing its local search and global search. Yang et al. (2008) propose two strategies to determine the inertia weight of BPSO. Experiments on a wrapper feature selection model suggest that the two proposed BPSOs outperform other methods, including sequential forward search, plus and take away, sequential forward floating search, sequential GA and different hybrid GAs. In order to avoid the particles converging at local optima, Yang et al. (2008) propose a strategy to renew the Gbest during the search process to keep the diversity of the population in BPSO. In the proposed algorithm, when Gbest is identical after three generations, a Boolean operator ‘and(.)’ will ‘and’ each bit of the Pbest of all particles in an attempt to create a new Gbest. Experimental results illustrate that the proposed method usually achieves higher classification accuracy with fewer features than GA and standard BPSO.

Chuang et al. (2008) also develop a strategy for Gbest in BPSO for feature selection in which Gbest will be reset to zero if it maintains the same value after several iterations. Experiments with cancer-related human gene expression datasets show that the proposed BPSO outperforms the algorithm proposed by Yang et al. (2008) in most cases.

Wang et al. (2007) propose an improved BPSO by defining the velocity as the number of elements that should be changed. The performance of the improved BPSO is compared with that of GA in a filter feature selection model based on rough sets theories. Experimental results show that the improved BPSO is computationally less expensive than GA in terms of both memory and running time. They also conclude that most of the running time is consumed by the computation of the rough sets, which is a drawback of using rough sets to solve the feature selection problems.

Unler & Murat (2010) modify the standard BPSO by extending social learning to update the velocity of the particles. Meanwhile, an adaptive feature subset selection strategy is developed, where the features are selected not only according to the likelihood calculated by BPSO, but also according to their contribution to the subset of features already selected. The improved BPSO is applied to a wrapper feature selection model for binary classification problems. Experimental results indicate that the proposed BPSO method outperforms the tabu search and scatter search algorithms.

Alba et al. (2007) combine a geometric BPSO with a support vector machine (SVM) algorithm for feature selection, where the current position, Pbest and Gbest of a particle are used as three parents in a three-parent mask-based crossover operator to create a new position for the particle instead of using the position update equation. Experiments on high dimensional microarray problems show that the proposed algorithm could achieve slightly higher accuracy than GA with SVM in most cases. Meanwhile, experiments also show that the initialisation of the BPSO had a great influence in the performance since it introduces an early subset of acceptable solutions in the evolutionary process.

Talbi et al. (2008) propose a geometric BPSO and compare it with GA using SVM for the feature selection in high dimensional microarray data. They conclude that the performance of the proposed BPSO is superior to GA in terms of accuracy. Liu et al. (2011) propose a multiple swarm BPSO (MSPSO) to search for the best feature subset and optimise the parameters of SVM. Experimental results show that the proposed feature selection methods could achieve higher classification accuracy with a smaller subset of features than grid search, standard BPSO and GA. However, the proposed MSPSO is computationally more expensive than other three methods because of the large population size and complicated communication rules between different subswarms.

Huang & Dun (2008) develop a wrapper feature selection method based on BPSO and SVM, which uses BPSO to search for the best feature subset and continuous PSO to simultaneously optimise the parameters in the kernel function of SVM, respectively. Experiments show that the proposed algorithm could determine the parameters, search for the optimal feature subset simultaneously and also achieve high classification accuracy.

Many studies have shown that BPSO is an efficient search technique for feature selection. Therefore, it is selected as the basic tool for developing new feature subset ranking algorithms in this paper.

### 3 Wrapper Based Single Feature Ranking

We propose a wrapper based single feature ranking approach, where the relative importance of each feature is measured by its classification accuracy.

Algorithm 1 shows the pseudo-code of the proposed wrapper based single feature ranking approach. In this approach, each dataset is divided into two sets: a training set and a test set. In both the training set and the test set, K-nearest neighbour (KNN) with  $n$ -fold cross-validation is employed to evaluate the classification accuracy. A detailed discussion of why and how  $n$ -fold cross-validation is applied in this way is given by Kohavi & John (1997). In this algorithm, firstly, in order to make sure  $n$ -fold cross-validation is always performed on the  $n$  fixed folds, both the training set and the test set are divided into  $n$  folds when

**Algorithm 1:** The wrapper based single feature ranking algorithm

---

```

1 begin
2   divide the training set to  $n$  folds;                                // n-fold cross-validation
3   divide the test set to  $n$  folds;
4   for  $d=1$  to number of features do
5     keep feature  $d$  and remove all the other features from training set ;    // training set only
6     contains feature  $d$ 
7     use KNN with n-fold cross-validation to evaluate the classification accuracy of feature  $d$  for the
8     training set;
9   end
10  rank the features according to the classification accuracy;
11  for  $d=1$  to number of features do
12    keep  $d$  top-ranked features and remove the others from the test set;
13    use KNN with n-fold cross-validation to evaluate the classification accuracy of  $d$  top-ranked
14    features for the test set;
15  end
16  return classification accuracy achieved by each feature;
17  return the order of features;
18  return the classification accuracies achieved by the successive numbers of the top-ranked features;
19 end

```

---

**Algorithm 2:** The BPSO based feature subset ranking algorithm

---

```

1 begin
2   divide the training set to  $n$  folds                                // n-fold cross-validation
3   divide the test set to  $n$  folds;
4   initialise a feature subset  $S$  by randomly selecting 1 feature;
5   for  $d=1$  to number of features do
6     initialise half of the swarm in BPSO with  $S$ ;
7     initialise the other half of the swarm with a subset randomly selecting  $d$  features;
8     while maximum iteration or fitness=1 is not met do
9       for  $p=1$  to number of particles do
10        calculate  $sum$  (number of the selected features by particle  $p$ );
11        if  $sum > d$  then
12          randomly exclude ( $sum - d$ ) features;
13        end
14        else if  $sum < d$  then
15          randomly include ( $d - sum$ ) features;
16        end
17        use KNN with n-fold cross-validation to evaluate the fitness of particle  $p$ 
18        // classification accuracy of  $d$  features selected by particle  $p$  for the
19        training set
20      end
21      for  $p=1$  to number of particles do
22        update  $Pbest_p$  and  $Gbest$ ;
23      end
24      for  $p=1$  to number of particles do
25        update the velocity of particle  $p$  (Equation 2);
26        update the position of particle  $p$  (Equations 3 and 4);
27      end
28    end
29    record the evolved feature subset and the corresponding classification accuracy;
30     $S \leftarrow$  the recorded feature subset in Line 27;
31  end
32  rank the learnt feature subsets;
33  use KNN with n-fold cross-validation to calculate the classification accuracy of the ranked feature
34  subsets for the test set;
35  return the order of feature subsets and classification accuracies;
36 end

```

---



the algorithms starts. Secondly, every feature is used for classification in the training set individually and its classification accuracy is calculated by a loop of  $n$ -fold cross-validation on the fixed  $n$  folds of training data (from Line 4 to Line 7 in Algorithm 1). Thirdly, the features are ranked according to the classification accuracies they achieve. Finally, based on the order of the ranked features, successive numbers of the top-ranked features are selected for classification to show the utility of single feature ranking in feature selection and the classification accuracy is calculated by KNN with  $n$ -fold cross-validation on the fixed  $n$  folds of the test data (from Line 9 to Line 12 in Algorithm 1).

The proposed algorithm is simple and easy to implement (around 20 lines of code). In each dataset, the aim is to determine the number of successive top-ranked features that can achieve classification accuracy close to or even better than the classifier with all features.

#### 4 BPSO Based Feature Subset Ranking

The top-ranked feature set resulting from the single feature ranking algorithm might contain potential redundancy. For example, the combination of the two top-ranked features might not perform as well as the combination of one top-ranked feature and a low-ranked feature if the two top-ranked features are highly dependent (redundant). To overcome this problem, we propose a feature subset ranking algorithm based on BPSO. Different feature subsets are evolved and ranked according to the classification accuracy on the training set.

Algorithm 2 shows the pseudo-code of BPSO for feature subset ranking. In this approach, each dataset is firstly divided into two sets: a training set and a test set. KNN with  $n$ -fold cross-validation is employed to evaluate the classification accuracy (Kohavi & John 1997) in both of the training set and the test set, which are divided into  $n$  folds, respectively. If a dataset includes  $D$  features,  $D$  feature subsets will be evolved and ranked. The feature subsets search process starts from finding the best subset including 1 feature and ends with the feature subset with  $D$  features. The  $d$ th feature subset includes  $d$  features, where  $d$  is a positive integer from 1 to  $D$ . There are many combinations for a feature subset with a particular number of features, and we use the  $d$ th feature subset to represent the best combination with  $d$  features in this method.

The process of selecting a certain feature subset is one step in this approach. For a dataset including  $D$  features,  $D$  feature subsets will be evolved and  $D$  steps are needed. Each step can be regarded as a process of using BPSO to select a certain number of the most relevant features (from Line 8 to Line 26 in Algorithm 2). The  $d$ th step is actually the process of using BPSO to search for the  $d$  most relevant features and the fitness function of BPSO is to maximise

Table 1: Datasets

Dataset	Number of features	Number of classes	Number of instances
Vowel	10	11	990
Wine	13	3	178
Australian	14	2	690
Zoo	17	7	101
Vehicle	18	4	846
German	24	2	1000
WBCD	30	2	569
Sonar	60	2	208

the classification accuracy. During the search process of BPSO, if a particle selects more than  $d$  features, a deletion strategy is employed to randomly exclude some features to reduce the number of features to  $d$ . On the other hand, if the number of selected features is smaller than  $d$ , an addition strategy is applied to randomly include some features to increase the number of the selected features to  $d$ .

During the search process, when searching for the  $d$ th feature subset, half of the population in BPSO is initialised with the  $(d-1)$ th feature subset achieved in the  $(d-1)$ th step. This is due to the expectation that some of the features in the  $(d-1)$ th subset are useful and should be retained in the  $d$ th subset. Meanwhile, each particle in the other half of the population is initialised with a feature subset that randomly selects  $d$  features to ensure the diversity of the swarm.

All the evolved feature subsets are ranked according to the classification accuracy on the training set and then their classification performance are evaluated by KNN with  $n$ -fold cross-validation on the test set. In each dataset, the aim is to determine the number of top-ranked feature subsets that can achieve classification accuracy close to or even better than the classifier with all features.

### 5 Experimental Design

#### 5.1 Datasets and Parameter Settings

Eight datasets chosen from the UCI machine learning repository (Frank & Asuncion 2010) are used in the experiments, which are shown in Table 1. The eight datasets were selected to have different numbers of features, classes and instances as the representative samples of the problems that the two proposed approaches could address. For two proposed approaches, in each dataset, the instances are divided into two sets: 70% as the training set and 30% as the test set. Classification accuracy is evaluated by 5NN with 10-fold cross-validation implemented in Java machine learning library (Java-ML) (Abeel et al. 2009). The classification accuracy is determined according to Equation 5:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (5)$$

where TP, TN, FP and FN stand for true positives, true negatives, false positives and false negatives, respectively.

The parameters of BPSO are set as follows: inertia weight  $w = 0.768$ , acceleration constants  $c_1 = c_2 = 1.49618$ , maximum velocity  $v_{max} = 6.0$ , population size  $P = 30$ , maximum iteration  $T = 100$ . The fully connected topology is applied in BPSO. These values are chosen based on the common settings in the literature (Van Den Bergh 2002).

For BPSO based feature subset ranking, the experiment has been conducted for 30 independent runs. The results achieved in different runs are similar to each other in terms of the classification accuracy of the evolved feature subsets. Therefore, the results from a typical run and the best results from 30 independent runs are shown in Section 6.

## 5.2 Benchmark Techniques

Two conventional wrapper feature selection methods, linear forward selection (LFS) and greedy stepwise backward selection (GSBS), are used as benchmark techniques to examine the performance of the two proposed approaches. They were derived from SFS and SBS, respectively.

LFS (Gutlein & Frank 2009) is an extension of best first algorithm. The search direction can be forward, or floating forward selection (with optional backward search steps). In LFS, the number of features considered in each step is restricted so that it does not exceed a certain user-specified constant. More details can be seen in the literature (Gutlein & Frank 2009).

Greedy stepwise (Caruana & Freitag 1994), implemented in Waikato Environment for Knowledge Analysis (Weka) (Witten & Frank 2005), is a steepest ascent search. It can move either forward or backward through the search space. Given that LFS performs a forward selection, a backward search is chosen in greedy stepwise to conduct a greedy stepwise backward selection. GSBS begins with all features and stops when the deletion of any remaining attribute results in a decrease in evaluation, i.e. the accuracy of classification.

Weka (Witten & Frank 2005) is used to run the experiments when using LFS and GSBS for feature selection. During the feature selection process, 5NN with 10-fold cross-validation in Weka is employed to evaluate the classification accuracy. In order to make fair comparisons, all the feature subsets selected by LFS, GSBS and two proposed methods are tested by 5NN with 10-fold cross-validation in Java-ML on the test sets.

When using Weka to run the experiments, all the settings are kept to the defaults except that backward search is chosen in the greedy stepwise approach to perform GSBS for feature selection and 5NN with 10-fold cross-validation is selected to evaluate the classification accuracy in both LFS and GSBS.

## 6 Results

Figure 1 shows the classification accuracy of each feature achieved by the wrapper based single feature ranking on the training set. The eight charts correspond to the eight datasets used in the experiments. In each chart, the horizontal axis shows the feature index in the corresponding dataset. The vertical axis shows the classification accuracy.

Figure 2 compares the classification performance of the two proposed methods, LFS and GSBS on the *test set*. Each plot corresponds to one of the eight datasets. In each plot, the horizontal axis shows the number of features used for classification and the vertical axis shows the classification accuracy. “SFR” in the figure stands for the results achieved by the successive numbers of top-ranked features in the wrapper based single feature ranking. For the BPSO based feature subset ranking, “FSR-Best” shows the best results in 30 independent runs and “FSR” shows the results achieved in a typical run. Both LFS and GSBS produce a unique feature subset, so have a single result for each test set. The red star denotes the classification accuracy achieved by LFS and the blue dot presents the result of GSBS. In addition, the red star and the blue dot in the plot of Vowel dataset are in the same position, which means both methods selected the same number of features and achieved the same classification accuracy.

### 6.1 Results of Wrapper Based Single Feature Ranking

According to Figure 1, classification accuracy achieved by each feature varies considerably, which means that they are not equally important for classification. In most cases, the difference between the highest classification accuracy and the lowest one is more than 20%, but it varies with the datasets. For example, the difference in the WBCD dataset is around 50% while the difference is only about 3% in the Vowel dataset. This is caused by the different characteristics in different datasets.

According to the results denoted by “SFR” in Figure 2, a selection of a small number of top-ranked features achieves better results than using all features in all the datasets. In almost all cases, using more top-ranked features, not only does not increase the performance, but actually causes a deterioration, especially for the Wine and Zoo datasets. The results suggest that there are interactions between some features, so the relevance level of a feature changes in the presence or absence of some other features.

### 6.2 Results of BPSO Based Feature Subset Ranking

According to the results (“FSR” and “FSR-Best”) in Figure 2, in all the eight datasets, with many of the feature subsets evolved by BPSO the classifier can achieve higher classification accuracy than with all

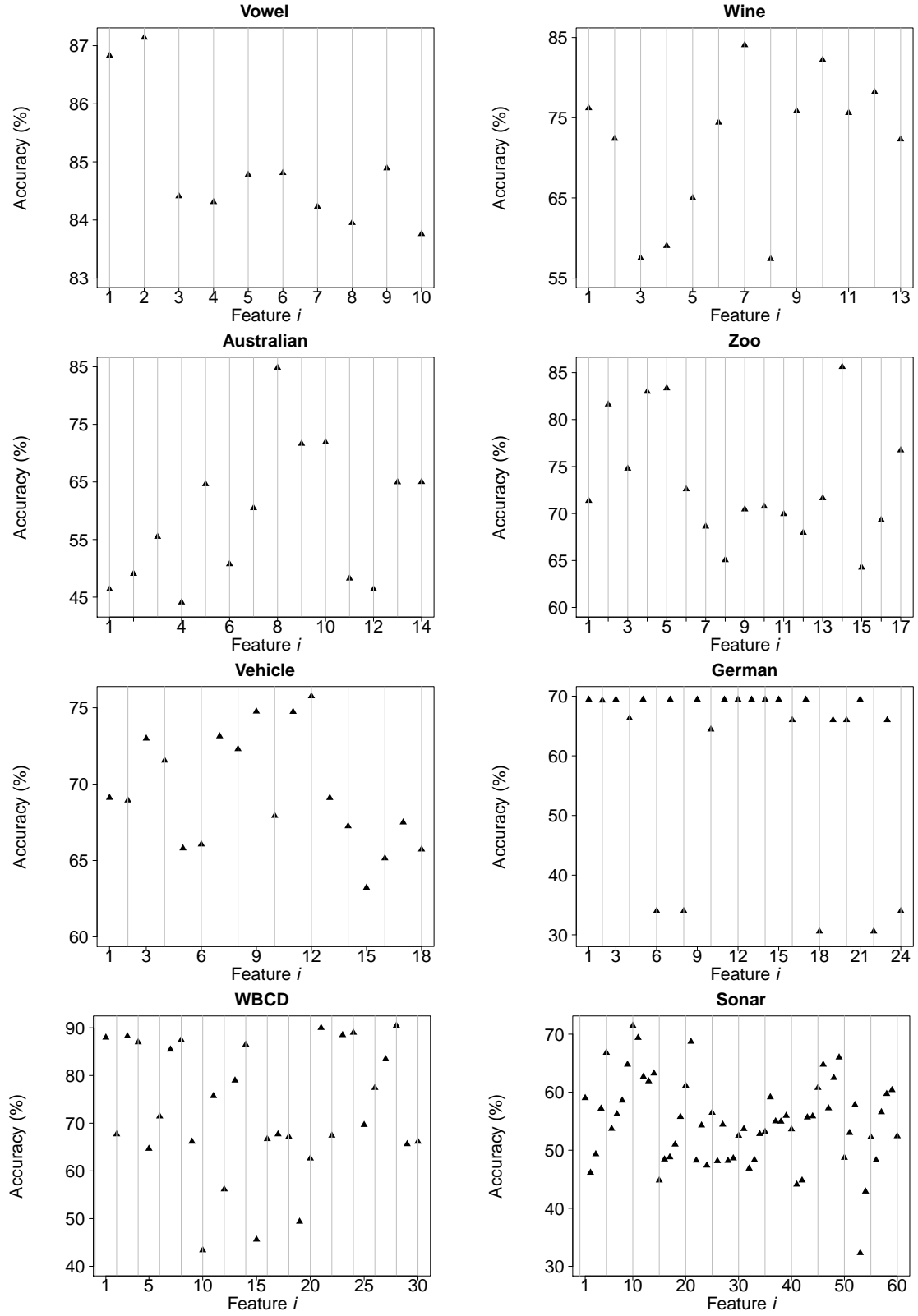


Figure 1: Results of single feature ranking

features. In most cases, the feature subset with which the classifier achieves the best performance contains a small number of features. For example, in the Australian dataset, the second feature subset evolved by BPSO only includes two features, but achieves the highest classification accuracy. This suggests that BPSO can select the relevant features and eliminate some noisy and irrelevant ones.

### 6.3 Comparisons Between Two Proposed Methods

Comparing the two proposed methods for feature selection, leads to the following observations. Firstly, using all features could not achieve the best performance in all the eight datasets. The two proposed methods could select a relatively small number of features with which the classifier could achieve higher

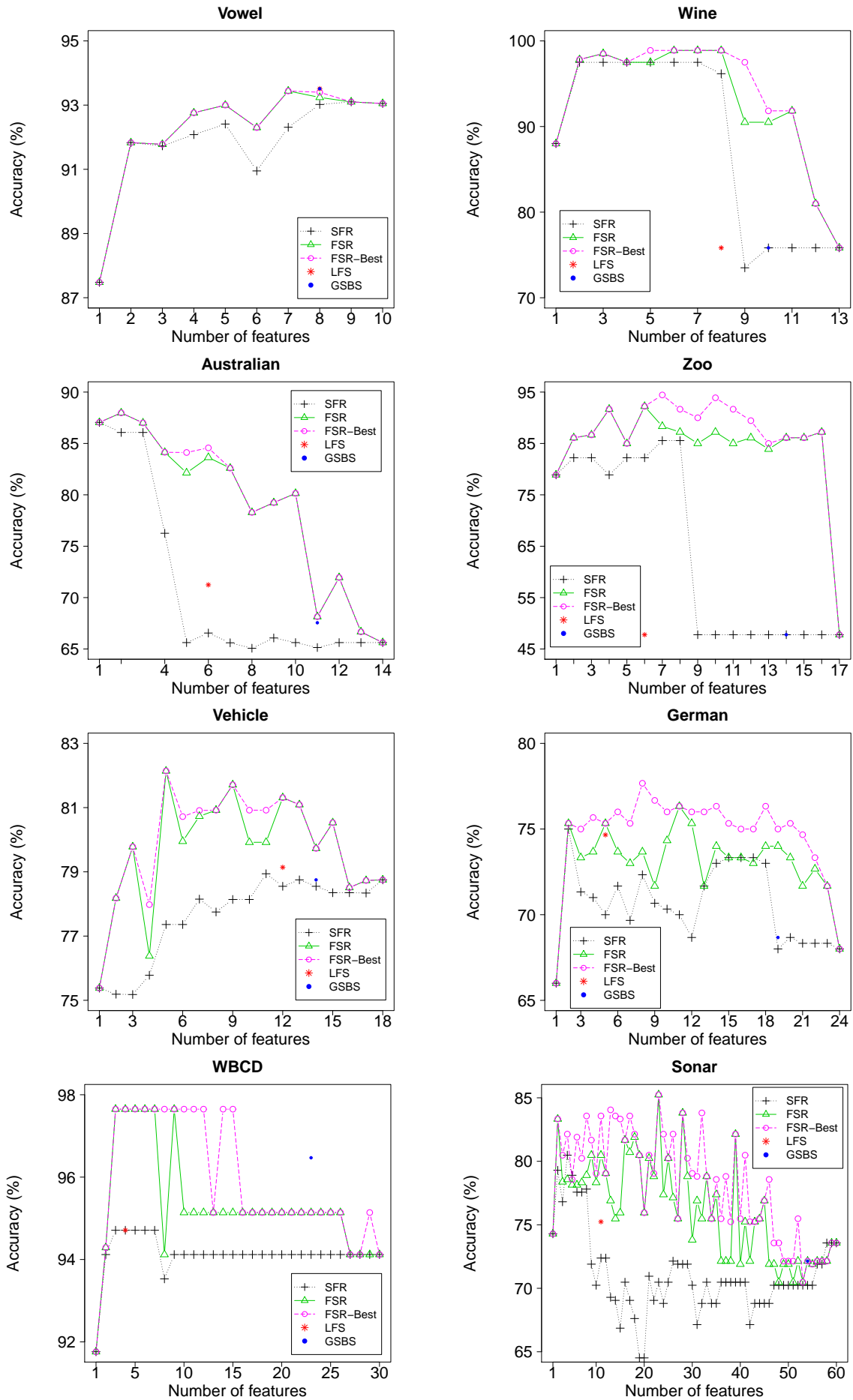


Figure 2: Comparisons between single feature ranking (SFR), feature subset ranking (FSR), the best results of FSR in 30 runs, linear forward selection (LFS) and greedy stepwise backward selection (GSBS)

classification accuracy than with all features. Secondly, in most cases, combining top-ranked features could not achieve the best performance because this combination still has redundancy. Thirdly, feature subset ranking provides an effective way for feature selection. Using the same number of features, BPSO based feature subset ranking can achieve higher classification accuracy than wrapper based single feature ranking. This suggests that BPSO could find a subset of complementary features to improve the classification performance.

#### 6.4 Further Analysis

Results in Figure 2 show that in almost all cases, the feature subset evolved by BPSO is not the combination of the top-ranked features, but a subset of complementary ones.

Considering the Australian dataset as an example, as can be seen in Figure 1, the order of the ranked features is F8, F10, F9, F14, F13, F5, F7, F3, F6, F2, F11, F12, F1, F4, where  $F_i$  denotes the  $i$ th feature in the dataset. The second feature subset evolved by BPSO includes F8 and F12, which are not the two top-ranked features (F8 and F10). According to Figure 2, although with F8 and F10 the classifier can achieve higher classification accuracy than with all features, with F8 and F12 it can obtain better results than with F8 and F10. This suggests that the combination of the two top-ranked features is redundant while the combination of a top-ranked feature (F8) and a low-ranked feature (F12) is a subset of complementary features. Meanwhile, the other 11 (from the 3th to the 13th) feature subsets evolved by BPSO are also not the combinations of the top-ranked features. These results suggest that the BPSO based subset ranking algorithm has great potential to avoid redundant and/or noisy features and reduce the dimensionality of the classifier.

#### 6.5 Comparisons Between Proposed Methods and Benchmark Techniques

The red star and blue dot in Figure 2 show that the number of features selected by LFS is smaller than that of GSBS, but the classification accuracy achieved by LFS is close to or better than that of GSBS in most cases. This suggests that LFS starting with an empty feature subset is more likely to obtain some optimality of the small feature subsets than backward selection methods, but does not guarantee finding the larger feature subsets. GSBS starts with all features and a feature is removed only when its removal can improve the classification performance. The redundant features that do not influence the classification accuracy will not be removed. Therefore, the feature subset selected by GSBS is usually larger than the feature subset selected by LFS because of the redundant features.

Comparing the proposed wrapper based single feature ranking with the two conventional techniques, it

can be observed that using the same number of features, LFS and GSBS could achieve higher classification accuracy than single feature ranking in most cases. This suggests that the combination of top-ranked features could not achieve the best performance because it contains redundancy or noise. However, in most cases, combining a relatively small number of top-ranked features could obtain higher accuracy than LFS and GSBS. The reason might be that the feature subsets selected by LFS and GSBS still have redundancy.

Figure 2 shows that BPSO based feature subset ranking outperforms LFS and GSBS. In seven of the eight datasets, feature subsets obtained by feature subset ranking can achieve higher classification accuracy than the subsets obtained by LFS and GSBS (in the eighth one, the Vowel dataset, the results are almost the same). This suggests that BPSO could find subsets of complementary features that could achieve better classification performance than other combinations of features.

### 7 Conclusions

The goal of this paper was to investigate a feature subset ranking approach to feature selection for classification. This goal was successfully achieved by developing two new wrapper based algorithms, namely a single feature ranking algorithm and a BPSO based feature subset ranking algorithm. The two algorithms were examined and compared with the corresponding method using all features, LFS and GSBS on eight problems of varying difficulty.

The results suggest that both methods can substantially improve the classification performance over the same classifier using all features. In almost all cases, the two proposed approaches could achieve higher classification accuracy whilst using fewer features than LFS and GSBS. The BPSO based feature subset ranking algorithm outperforms the simple single feature ranking algorithm on all the datasets regarding the classification performance. The results also show that on all the eight problems investigated here, it was always possible to find a subset with a small number of features that can achieve substantially better performance than using all features.

The proposed BPSO based algorithm has one limitation, that is, the evolutionary training time is relatively long. While this is usually not a problem as many situations allow offline training (as the test time is shorter using a subset of features than using all features), it might not be suitable for online (real-time) applications. We will investigate efficient feature subset ranking methods for effectively selecting good features in the future.

### References

- Abeel, T., de Peer, Y. V. & Saeys, Y. (2009), 'Java-ML: A Machine Learning Library', *Journal of Ma-*

- chine Learning Research **10**, 931–934.
- Alba, E., Garcia-Nieto, J., Jourdan, L. & Talbi, E.G. (2007), Gene selection in cancer classification using PSO/SVM and GA/SVM hybrid algorithms, in 'IEEE Congress on Evolutionary Computation', pp. 284–290.
- Azevedo, G.L.F., Cavalcanti, G.D.C. & Filho, E.C.B. (2007), An approach to feature selection for keystroke dynamics systems based on PSO and feature weighting, in 'IEEE Congress on Evolutionary Computation', pp. 3577–3584.
- Caruana, R. & Freitag, D. (1994), Greedy Attribute Selection, in 'In Proceedings of International Conference on Machine Learning', pp. 28–36.
- Chakraborty, B. (2008), Feature subset selection by particle swarm optimization with fuzzy fitness function, in '3rd International Conference on Intelligent System and Knowledge Engineering', Vol. 1, pp. 1038–1042.
- Chuang, L.Y., Chang, H.W., Tu, C.J. & Yang, C.H. (2008), 'Improved binary PSO for feature selection using gene expression data', *Computational Biology and Chemistry* **32**(29), 29–38.
- Clerc, M. & Kennedy, J. (2002), The particle swarm - explosion, stability, and convergence in a multi-dimensional complex space, in 'IEEE Congress on Evolutionary Computation', Vol. 6, pp. 58–73.
- Dash, M. & Liu, H. (1997), 'Feature selection for classification', *Intelligent Data Analysis* **1**, 131–156.
- Frank, A. & Asuncion, A. (2010), *UCI Machine Learning Repository*, [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
- Gutlein, M., Frank, E., Hall, M. & Karwath, A. (2009), Large-scale attribute selection using wrappers, in 'IEEE Symposium on Computational Intelligence and Data Mining', pp. 332–339.
- Guyon, I., Elisseeff, A. & Liu, H. (2003), 'An introduction to variable and feature selection', *The Journal of Machine Learning Research* **3**, 1157–1182.
- Huang, C.J. & Dun, J.F. (2008), 'A distributed PSO-SVM hybrid system with feature selection and parameter optimization', *Applied Soft Computing* **8**(4), 1381–1391.
- Kennedy, J. & Eberhart, R. (1995), Particle swarm optimization, in 'IEEE International Conference on Neural Networks', Vol. 4, pp. 1942–1948.
- Kennedy, J. & Eberhart, R. (1997), A discrete binary version of the particle swarm algorithm, in 'IEEE International Conference on Systems, Man, and Cybernetics', Vol. 5, pp. 4104–4108.
- Kennedy, J. & Spears, W.M. (1998), Matching algorithms to problems: an experimental test of the particle swarm and some genetic algorithms on the multimodal problem generator, in 'IEEE Congress on Evolutionary Computation', pp. 78–83.
- Kohavi, R. & John, G.H. (1997), 'Wrappers for feature subset selection', *Artificial Intelligence* **97**, 315–333.
- Langley, P. (1994), Selection of relevant features in machine learning, in 'Proceedings of the AAAI Fall symposium on relevance', pp. 127–131.
- Liu, Y.N., Wang, G., Chen, H.L., & Dong, H. (2011), 'An Improved Particle Swarm Optimization for Feature Selection', *Journal of Bionic Engineering* **8**(2), 191–200.
- Marill, T., & Green, D.M. (1963), 'On the effectiveness of receptors in recognition systems', *IEEE Transactions on Information Theory* **9**(1), 11–17.
- Neshatian, K. & Zhang, M.J. (2009), Genetic Programming for Feature Subset Ranking in Binary Classification Problems, in 'European Conference on Genetic Programming', pp. 121–132.
- Talbi, E.G., Jourdan, L., Garcia-Nieto, J. & Alba, E. (2008), Comparison of population based metaheuristics for feature selection: Application to microarray data classification, in 'ACS/IEEE International Conference on Computer Systems and Applications', pp. 45–52.
- Unler, A. & Murat, A. (2010), 'A discrete particle swarm optimization method for feature selection in binary classification problems', *European Journal of Operational Research* **206**, 528–539.
- Van Den Bergh, F. (2002), An analysis of particle swarm optimizers, Ph.D., University of Pretoria, South Africa.
- Wang, X.Y., Yang, J., Teng, X.L. & Xia, W.J. (2007), 'Feature selection based on rough sets and particle swarm optimization', *Pattern Recognition Letters* **28**(4), 459–471.
- Whitney, A.W. (2007), 'A direct method of nonparametric measurement selection', *IEEE Transactions on Computers* **20**(4), 1100–1103.
- Witten I.H. & Frank E. (2005), *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd Edition, Morgan Kaufmann, San Francisco.
- Yang, C.S., Chuang, L.Y. & Ke, C.H. (2008), Boolean binary particle swarm optimization for feature selection, in 'IEEE Congress on Evolutionary Computation', pp. 2093–2098.
- Yang, C.S., Chuang, L.Y. & Li, J.C. (2008), Chaotic maps in binary particle swarm optimization for feature selection, in 'IEEE Conference on Soft Computing in Industrial Applications', pp. 107–112.

# On the Existence of High-Impact Refactoring Opportunities in Programs

Jens Dietrich<sup>1</sup>

Catherine McCartin<sup>1</sup>

Ewan Tempero<sup>2</sup>

Syed M. Ali Shah<sup>1</sup>

<sup>1</sup> School of Engineering and Advanced Technology  
Massey University, Palmerston North, New Zealand  
Email: {j.b.dietrich,c.m.mccartin,m.a.shah}@massey.ac.nz

<sup>2</sup> Department of Computer Science  
University of Auckland, Auckland, New Zealand  
Email: e.tempero@cs.auckland.ac.nz

## Abstract

The refactoring of large systems is difficult, with the possibility of many refactorings having to be done before any useful benefit is attained. We present a novel approach to detect starting points for the architectural refactoring of large and complex systems based on the analysis and manipulation of the type dependency graph extracted from programs. The proposed algorithm is based on the simultaneous analysis of multiple architectural antipatterns, and outputs dependencies between artefacts that participate in large numbers of instances of these antipatterns. If these dependencies can be removed, they represent high-impact refactoring opportunities: a small number of changes that have a major impact on the overall quality of the system, measured by counting architectural antipattern instances. The proposed algorithm is validated using an experiment where we analyse a set of 95 open-source Java programs for instances of four architectural patterns representing modularisation problems. We discuss some examples demonstrating how the computed dependencies can be removed from programs. This research is motivated by the emergence of technologies such as dependency injection frameworks and dynamic component models. These technologies try to improve the maintainability of systems by removing dependencies between system parts from program source code and managing them explicitly in configuration files.

## 1 Introduction

Software systems are subject to change. However, changing software is risky and expensive. The development of methodologies and tools to deal with change, and to minimise risks and expenses associated with change is one of the great challenges in software engineering. Refactoring is a successful technique that has been developed in order to facilitate changes in the code base of programs. First developed in the late 90s, code refactoring tools have become commodities for many programmers, and refactoring is one of the main supportive technologies for agile process models such as Scrum and extreme programming. The first generation of refactoring tools has focused on the manipulation of source code, using the structure of the source code (in particular the abstract syntax tree (AST)) as the data structure that is being manipulated. In recent years, refactoring has been studied in different contexts, in particular the refactoring of models representing other as-

pects of software systems such as design, architecture and deployment.

The need to refactor systems on a larger scale arises from changing business requirements. Examples include moves from monolithic products to product lines, system integration, or the need to improve some of the “ilities” of systems such as maintainability, security or scalability. While the refactoring of systems at the large scale is difficult, it is a common belief amongst software engineers that the pareto principle, also known as the 80-20 rule, applies: a few targeted actions can have an over-proportional impact.

The main question we would like to answer is, can the pareto principle apply at all? If the answer to this question is no, even with very generous assumptions, then this would be a very important result with significant consequences for when refactoring can be profitably used. If the answer is yes, then, due to our assumptions, that would not necessarily mean efficient refactoring of large scale systems would always be possible, but it would at least provide support for pursuing that goal. Our approach is to create a mathematical model of the systems we would like to refactor, and examine whether small changes to the model will have large impacts on the overall quality of the design.

As a motivating example, consider the program depicted in figure 1. The design of this program can be considered as a graph, the so-called dependency graph (DG). The vertices in this graph are types, and the edges are relationships between these types. This particular program consists of four classes A,B,C and D and three name spaces package1, package2 and package3. It contains several antipatterns [6] that represent design problems:

1. A circular dependency between the packages 1,2 and 3, caused by the path  $A \rightarrow_{\text{extends}} B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} A$
2. A circular dependency between the packages 2 and 3, caused by the path  $B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} D$
3. A subtype knowledge pattern where a type references its own subtype, caused by the edges  $A \rightarrow_{\text{extends}} B$  and  $B \rightarrow_{\text{uses}} C \rightarrow_{\text{uses}} A$

All three antipattern instances can be removed from the graph with the removal of the single edge  $B \rightarrow_{\text{uses}} C$ . An algorithm for finding this edge is simple: for each edge, record the number of occurrences in *all* instances of *each* antipattern, then remove one of the edges with the *highest* score. This method would assign the highest score of three only to the edge  $B \rightarrow_{\text{uses}} C$ . All other edges participate in only one or two pattern instances.

The edge  $B \rightarrow_{\text{uses}} C$  indicates a good starting point for *architectural refactoring*: changing the structure of the system without changing its external behaviour. A refactoring that gives rise to a new system that is modelled by the dependency graph  $DG \setminus (B \rightarrow_{\text{uses}} C)$  would be a *high-impact refactoring* in the following sense: this particular

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

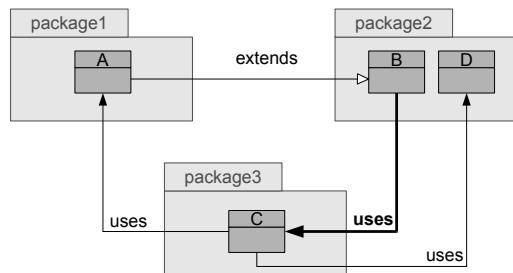


Figure 1: Simple program DG

refactoring removes significantly more (in this case: all) antipattern instances from the model, and can therefore be considered more valuable than alternative refactorings. Such a refactoring would have the highest impact on the overall quality of the system, measured by the number of architectural antipattern instances.

In general, there is no guarantee that such a refactoring can be achieved. However, the fact that a small change in the model has a large impact on the overall quality of the design, in this case, suggests that the pareto principle could apply here. Our aim is to discover whether or not the pareto principle applies in general.

One particular class of architectural refactorings we are interested in is modularisation, and in particular the refactoring of monolithic Java programs into dynamic component models such as OSGi [1] and its clones and extensions. This is a very timely issue, as some of the most complex systems including the Java Development Kit itself [2], and the leading commercial application servers WebLogic and WebSphere [17], are currently being modularised. In our previous work [8], we assessed the scope of the problem by investigating a set of antipatterns that hinder modularisation using an OSGi-style framework. The work presented here has grown out of this approach. We aim to generalise from our previous work to develop a generic approach for using sets of antipatterns to compute refactoring opportunities. The opportunities identified correspond to operations which are applied to a model representing the system. After application, the antipattern analysis is repeated in order to assess whether or not certain characteristics of the system have improved. It turns out that, using this approach, we can compute candidates for high impact refactorings.

The rest of the paper is organised as follows. In Section 2 we review related work. We continue in Section 3 with a short introduction to the framework we have developed for describing antipatterns, and the algorithmic tools used to detect these antipatterns. In Section 4 we motivate our choice of a particular set of antipatterns that hinder modularisation and discuss the algorithm used to detect potential refactoring opportunities. We then describe the organisation of an experiment used to validate our approach, where we analyse a large corpus of open-source Java programs [25] for instances of four antipatterns representing modularisation problems. An analysis of the results of our experiment is presented in Section 5. A discussion of open questions related to our work concludes this contribution.

## 2 Related Work

### 2.1 Antipattern and Motif Detection

In our work, we propose to use sets of antipatterns as starting points for architectural refactoring. These patterns can be viewed as the equivalent of smells [10] that are used as starting points for code-level refactorings. While early work on smells and antipatterns has focused on the anal-

ysis of source code, many of these concepts can also be applied to software architecture [18]. Research into code-level antipattern and smell detection has resulted in a set of robust tools that are widely used in the software engineering community, including PMD [7] based on source code analysis, and FindBugs [16] based on byte code analysis. A closely related area is the detection of design patterns [13]. Several solutions have been proposed to formalise design patterns in a platform-independent manner. A good overview is given in [33].

Garcia et al. describe a set of architectural smells [14] using a format similar to the original Gang of Four pattern language [13]. These smells are somewhat different from our patterns. The definitions given by the authors in [14] do not seem to be precise enough for tool-supported detection.

Our approach is based on the detection of antipatterns in the dependency graph extracted from a program. The use of dependency graphs as a basis for program analysis has been investigated by several authors (e.g. [19, 4]).

Patterns in graphs can be formalised as *motifs*. Detection of graph motifs has been widely studied in bioinformatics, and there is a large body of recent work in this area. The concept has also been proposed in the context of complex networks (e.g., Milo et al. [21]). The motifs used in both of these areas are simpler than those that we propose, in that we do not only consider local sets of vertices directly connected by edges, but also sets of vertices indirectly connected by paths.

We have investigated in previous work [9] the potential of the Girvan-Newman clustering algorithm [15] to detect refactoring opportunities in dependency graphs.

### 2.2 Refactoring

Architectural refactorings were first discussed by Beck and Fowler (“big refactorings”) [10], and then discussed by Roock and Lippert (“large refactorings”) [18]. Their work defines the framework for our contribution: starting with the detection of architectural smells by means of antipatterns (for example, cycles) or metrics in architectural models, systems are modified to improve their characteristics while maintaining their behaviour. Large scale refactorings can be broken down (decomposed into smaller refactorings). Our approach fits well into this framework; we compute a sequence of base refactorings that can be performed step by step, using the dependency graph as the architectural model.

Our work is related to the use of graph transformations and graph grammars [27], an area that has been applied in many areas of software engineering such as model transformations. The manipulations of the graphs we are interested in are simple: we only remove single edges. This does not justify the use of the full formalism of a graph grammar calculus. In work by Mens et al. graph transformations are directly used to detect refactorings [20]. There, the focus is on code-level refactoring and the detection and management of dependencies between those refactorings.

Simon et al. try to formalise the notion of smells [30]. The authors use metrics for this purpose, while we use patterns. Tsanatalis and Chatzigeorgiou have identified opportunities to apply the “move method” refactoring [36]. Their proposed algorithm is based on the Jaccard metric between feature sets and preconditions for the respective refactoring. Their aim is to remove only one particular smell (feature envy) from programs. Seng et al. use a genetic algorithm to detect code-level refactorings [29]. Their work is also restricted to the “move method” refactoring. O’Keeffe and O’Cinneide represent object-oriented design as a search problem in the space of alternative designs [22]. They use several search algorithms to



search this space for designs which improve the existing design with respect to a set of given metrics. The refactorings used to traverse the search space are all inheritance-related (extract and collapse hierarchies, move feature up and down the hierarchy) and, therefore, not very expressive.

Bourqun and Keller present a high-impact refactoring case study [5]. They first define the layered architecture of the program to be refactored, and then use Sotograph to detect violations of this architecture. They focus refactoring activities on packages associated with those violations, and validate their approach using violation counts and code metrics. They present their approach as a case study using an enterprise Java application developed by a Swiss telecommunication company. The approach discussed here can be seen as a generalisation of Bourqun and Keller's work [5]: the architecture violations can be expressed using patterns, and the algorithm to compute the artefacts to be refactored from the violations can be recast in our edge-scoring idiom. While our general approach supports and encourages the use of project-specific patterns derived from system architecture, we use a set of general, project-independent patterns for the empirical study.

### 3 Methodology

#### 3.1 Motifs

As stated earlier, our approach to detect high-impact refactoring opportunities is based on the detection of antipatterns in the *program dependency graph* (DG). In the following paragraphs, we formally define this graph and related concepts.

A dependency graph  $DG = (V, E)$  consists of a set of vertices,  $V$ , representing types (classes, interfaces and other types used in the programming language), and a set of directed edges,  $E$ , representing relationships between those types. Both vertices and edges are labelled to provide further information. Vertices have labels providing the name, the name space, the container (library), the abstractness (true or false) and the kind (interface, class, enumeration or annotation) of the respective type. Edges have a type label indicating whether the relationship is an extends, implements or uses relationship.

We formalise architectural antipatterns as network *motifs* in the dependency graph. Given a dependency graph, a motif can be defined as follows:

A motif  $m = (VR, PR, C_V, C_P)$  consists of four finite sets: vertex roles ( $VR$ ) and path roles ( $PR$ ), vertex constraints  $C_V$  and path constraints  $C_P$ . If  $n$  is the cardinality of  $VR$ , a vertex constraint  $c_V \in C_V$  is defined as an  $n$ -ary relation between vertices,  $c_V \subseteq \times_{i=1..n} V$ . If  $n$  is the cardinality of  $PR$ , a path constraint is  $c_P \in C_P$  is defined as an  $n$ -ary relation between sequences of edges ( $SEQ(E)$ ),  $c_P \subseteq \times_{i=1..n} SEQ(E)$ . Intuitively, constraints restrict the sets of possible vertex and path assignments. While vertex constraints are always defined with respect to vertex labels, there are three different types of path constraints:

1. Source and target constraints restricting, respectively, the start and end vertices of a path.
2. Cardinality constraints restricting the length of a path, usually defined using restrictions on the minimum and the maximum length of a path.
3. Constraints defined with respect to edge labels. These constraints have to be satisfied for all edges within a path.

A *binding* is a pair of functions  $\langle inst_V, inst_P \rangle$ , where  $inst_V : VR \rightarrow V$  and  $inst_P : PR \rightarrow SEQ(E)$ . A binding as-

sociates vertex roles with vertices and path roles with sequences of edges. A *motif instance* is a binding such that the constraints are fulfilled, i.e. the following two conditions must be true:

- $(inst_V(vr_1), \dots, inst_V(vr_n)) \in c_V$  for all vertex constraints  $c_V \in C_V$
- $(inst_P(pr_1), \dots, inst_P(pr_n)) \in c_P$  for all path constraints  $c_P \in C_P$

#### 3.2 Motif Definition and Detection

The detection of motif instances in non-trivial dependency graphs is complex. The worst-case time complexity for the type of motif search that we do is  $O(n^k)$ , where  $n$  and  $k$  are the number of vertices in the dependency graph and the number of roles in the motif, respectively. This worst-case time complexity is a consequence of the NP-hardness of the subgraph isomorphism problem, which is essentially the problem that we must solve each time we successfully find an instance of a query motif in a dependency graph. Note that the algorithm that we use to detect motif instances returns all possible bindings of vertex roles, but for each such binding only one selected binding for path roles. Formally, we consider only classes of instances  $(inst_V, inst_P)$  modulo  $(instance_P)$ . This means that two instances are considered as being equal if and only if they have the same vertex bindings.

To detect motifs in the dependency graph we use the GUEY<sup>1</sup> tool. The tool represents dependency graphs in memory, and employs an effective solver to instantiate motifs. The solver used takes full advantage of multi-core processors, and uses various optimisation techniques. It is scalable enough to find motifs in large programs with vertex counts of up to 50000 and edge counts of up to 200000. This kind of scalability is required to analyse real world programs, such as the runtime library of the Java Development Kit, consisting of 17253 vertices and 173911 edges.

Listing 1 shows a motif definition. This motif has two vertex roles  $VR = \{type, supertype\}$  and two path roles  $PR = \{inherits, uses\}$ . The paths roles have source and target constraints defined by the `from` and `to` attributes in the `connectedby` elements, and edge constraints defined in the expressions in line 4. The edge constraints state that all edges in paths instantiating the `inherits` role must be `extends` or `inherits` relationships, and that all edges in paths instantiating the `uses` role must be `uses` relationships. The length of the paths is not constrained in this example, but the language would support this through the `minLength` and `maxLength` attributes defined for the `connectedby` element. The default values are 1 for `minLength` and -1, representing unbound, for `maxLength`.

#### 3.3 Edge Scoring

Motif detection in dependency graphs can be used to assess the quality of architecture and design of systems. The classical example is the detection of circular dependencies between packages, modules and types that has been widely discussed [32, 24]. In general terms, we aim to use motifs to formalise antipatterns and smells and thus facilitate detection of design problems. For a single motif, the number of separate motif instances in a dependency graph can be very large. However, edges can simultaneously participate in many instances.

This raises the question of whether, for a given set of motifs representing antipatterns, there are some edges

<sup>1</sup><http://code.google.com/p/queryframework/>

```

1 motif stk
2 select type, supertype
3 connected by inherits (type>supertype) and uses (supertype>type)
4 where "uses.type=='uses'" and "inherits.type=='extends' || inherits.type=='implements'"

```

Listing 1: The Subtype knowledge motif (STK)

which participate in large numbers of overlapping motif instances. It also raises the question of whether such “high-scoring” edges participate in instances arising from more than one of the motifs in the set. If so, refactorings of the system resulting in the removal of those edges from the dependency graph would be an effective way to improve the overall quality of the architecture and design of the underlying system.

In general, given a dependency graph  $DG = (V, E)$ , a motif  $m = (VR, PR, C_V, C_P)$  and a set of motif instances  $I(m) = \{(inst_V^i, inst_P^i)\}$  of  $m$  in  $DG$ , we define a *scoring function* as a function associating edge-instance pairs with natural numbers,  $score : E \times I(m) \rightarrow \mathbb{N}$ . We also require that a positive score is only assigned if the edge actually occurs in one of the paths instantiating a path role in the motif:

$$\forall i : score(e, (inst_V^i, inst_P^i)) > 0 \Rightarrow \exists pr \in PR : e \in inst_P^i(pr).$$

For a given set of motifs  $M = \{m_j\}$  with sets of instances  $\{I(m_j)\}$  of  $M$  in  $DG$ , the overall score of an edge with respect to  $M$  is defined as the sum of all scores for each instance of each motif:

$$score_M(e) := \sum_{m \in M} \sum_{inst \in I(m)} score(e, inst).$$

The simplest scoring function is the function that just scores each occurrence of an edge in a motif instance as 1. We call this the *default scoring function*. Given a dependency graph, a set of motifs representing antipatterns and a scoring function, we can define the following generic algorithm to detect edges in the dependency graph that may be associated with high impact refactorings of the underlying system:

1. Compute all instances for all motifs.
2. Compute the scores for all edges.
3. Sort the edges according to their scores.
4. Remove some edges with the highest scores from the graph.
5. Recompute all instances for all motifs and compare this with the initial number to validate the effect of edge removals.

Note that this algorithm has several variation points that affect its outcome:

1. The set of motifs used.
2. The scoring function used.
3. If only one edge is to be removed, the selection function that selects this edge from the set of edges with the highest scores.

Depending on the decisions made for these variation points, the effects of edge removal will be different. However, the existence of these variation points supports the customisation of this algorithm in order to adapt it to project-specific settings. For instance, domain-specific antipatterns and scoring functions can be used to represent weighted constraints penalising dependencies between certain classes or packages.

The selection of the edge from the set of edges with high scores can also take into account the difficulty of performing the actual refactoring on the underlying system that would result in the removal of this edge from the dependency graph.

#### 4 Case Study: Detecting High-Impact Refactorings to Improve System Modularity

To demonstrate the use of our generic algorithm, we present a case study that is based on a particular set of antipatterns representing barriers to modularisation. The presence of instances of these antipatterns in dependency graphs implies that packages (name spaces) are difficult to separate (poor *name space separability*), in particular due to the existence of circular dependencies, and that implementation types are difficult to separate from specification types (poor *interface separability*). Both forms of separability are needed in modern dynamic component models such as OSGi, and in this sense the presence of these motifs represents barriers to modularity. For more details, the reader is referred to Dietrich et al. [8].

##### 4.1 Motif Set

###### 4.1.1 Overview

We use the following four antipatterns that represent design problems in general, and barriers to the modularisation in particular:

1. Abstraction Without Decoupling (AWD)
2. Subtype knowledge (STK)
3. Degenerated inheritance (DEGINH)
4. Cycles between name spaces (CD)

These antipatterns can easily be formalised into graph motifs. We discuss each of these motifs in the following subsection. For a more detailed discussion, the reader is referred to Dietrich et al. [8]. We use a simple visual syntax to represent antipatterns. Vertex roles are represented as boxes. Path roles are represented by arrows connecting boxes. These connections are labelled with either *uses* (uses relationships) or *inherits* (extends or implements relationships). They are also labelled with a number range describing the minimum and maximum length of paths, with “M” representing unbound (“many”). If vertex roles have property constraints, these constraints are written within the box in guillemets.

###### 4.1.2 Abstraction Without Decoupling (AWD)

The Abstraction Without Decoupling (AWD) pattern describes a situation where a client uses a service represented as an abstract type, and also a concrete implementation of this service, represented as a non-abstract type extending or implementing the abstract type. This makes it hard to replace the service implementation and to dynamically reconfigure or upgrade systems. To do this, the client code must be updated. The client couples service description and service implementation together.

Techniques such as dependency injection [12] could be used to break instances of this pattern. Fowler discusses

how patterns can be used to avoid AWD [11]. This includes the use of the Separated Interface and the Plugin design patterns. The visual representation of this pattern is shown in figure 2.

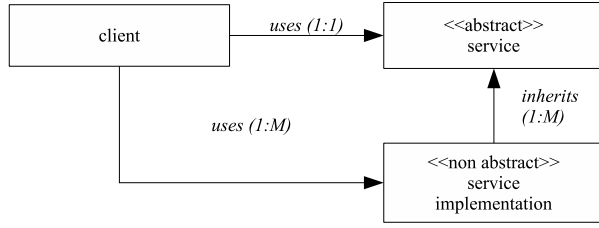


Figure 2: AWD

#### 4.1.3 Subtype Knowledge (STK)

In this antipattern [26], types have *uses* relationships to their subtypes. The presence of STK instances compromises separability of sub- and supertypes. In particular, it implies that there are circular dependencies between the name spaces containing sub- and supertype. Instability in the (generally less abstract) subtype will cause instability in the supertype, and the supertype cannot be used and understood without its subtype. The visual representation of this pattern is shown in figure 3, the definition is given in listing 1.

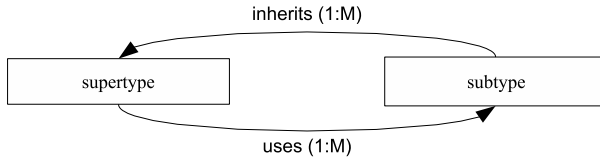


Figure 3: STK

#### 4.1.4 Degenerated Inheritance (DEGINH)

Degenerated inheritance [28, 31], also known as diamond, repeated or fork-join inheritance, means that there are multiple inheritance paths connecting subtypes with supertypes. For languages with single inheritance between classes such as Java, this is caused by multiple interface inheritance. The presence of instances of DEGINH makes it particularly difficult to separate sub- and superclasses.

The visual representation of this pattern is shown in figure 4.

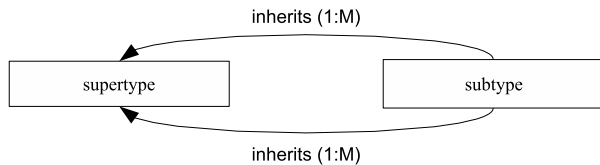


Figure 4: DEGINH

#### 4.1.5 Cycles between Name Spaces (CD)

Dependency cycles between name spaces (CD) is a special instance of cycles between modules [32]. This antipattern implies that the participating name spaces cannot be deployed and maintained separately. In particular, if these name spaces were deployed in several runtime modules (jars), this would create a circular dependency between

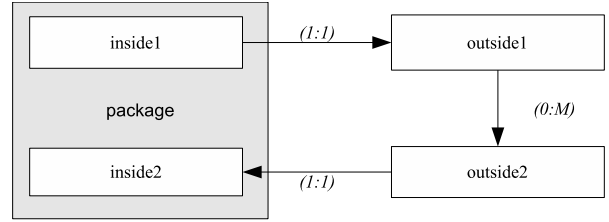


Figure 5: CD

those jars. This antipattern is stronger than the usual circular dependency between name spaces A and B which requires that there be two paths, one connecting A to B and the other connecting B to A. CD requires the existence of one path from A, through B, back into A. The weaker form of circular dependency can sometimes be removed by simply splitting the names spaces involved. CD is more difficult to remove as the path must be broken through refactoring.

The visual representation of this pattern is shown in figure 5. Note that the cardinality constraint for the path connecting outside1 and outside2 is  $[0, M]$ . This means that the path can have a length of 0. In this case, the antipattern instance has a triangular shape and the two outside roles are instantiated by the same vertex.

#### 4.2 Scoring Functions

In this experiment, we have used the default scoring function that increases the score by one for each edge encountered in any path instantiating any path role in each instance for each of the four motifs.

#### 4.3 Data Set

For the validation of our approach we have used the Qualitas Corpus, version 20090202 [34]. For many programs, the corpus contains multiple versions of the same program, sometimes with only minor differences between those versions. We have therefore decided to keep only one version of each program in the data set. We decided to use the latest version available. There are two programs in this set that do not have instances for any antipattern in the set used: `exoportal-v1.0.2.jar` and `jmeter-2.3.jar`. We have removed those two programs from the data set. We have also removed `eclipse.SDK-3.3.2-win32` and `jext-5.0` — these programs already use a plugin-based modularisation model (e.g., through the Eclipse extension registry and the Equinox OSGi container) and therefore many of the antipatterns we are interested in will not be present. Finally, we have removed the Java Runtime Environment (JRE, `jre-1.5.0_14-linux-i586`) — it turns out that our tools are not yet scalable enough to do a full analysis due to the size of the JRE. However, we have done a partial analysis of the JRE, the results are discussed below. This has given us the final set of the 95 programs.

The dependency graphs extracted from the programs in the corpus differ widely in size. The largest graph, extracted from `azureus-3.1.1.0`, has 6444 vertices and 35392 edges. The smallest graph, extracted from `ivatagroupware-0.11.3`, has 17 vertices and 22 edges. The average number of vertices in graphs extracted from corpus programs is 660, the average number of edges 3409.

#### 4.4 Graph Preparation

The dependency graphs can be extracted from different sources, such as byte code and source code of programs

written in different programming languages. We have used the dependency finder library [35] to extract dependency graphs from Java byte code. Dependency graphs built from byte code are slightly different from graphs built from source code. For instance, relationships defined by the use of generic types are missing due to erasure by the Java compiler<sup>2</sup>. We do not see this as a problem as the focus of our investigation is to find refactoring opportunities to improve the runtime characteristics of deployed systems.

Graphs are represented as instances of the JUNG [23] type `edu.uci.ics.jung.graph.DirectedGraph`. This has caused some issues related to the repeatability of results. The GUERY solver we have used to detect motif instances returns all possible bindings of vertex roles, but for each such binding, only one selected binding for path roles. It is possible to override this behaviour and compute all possible path role assignments as well. However, we have found that this is only feasible for very small motifs or graphs and that the combinatorial explosion in the number of possible paths makes a scalable implementation impossible for graphs of a realistic size. Formally, we consider only classes of instances ( $inst_v, inst_p$ ) modulo ( $instance_p$ ), i.e., two instances are considered equivalent if they have the same vertex bindings.

The problem arising from this is that, when the computation is repeated, in some cases different path role bindings are computed for the same binding of vertex roles, since the query engine traverses outgoing/incoming paths in a different order. This is caused by the internal indexing of incoming/outgoing edges in the JUNG API that uses hashing. For this reason, we have modified the JUNG API and to represent outgoing and incoming edges as lists with predictable order. We have also added a method to set a comparator to sort incoming/outgoing edges for all vertices in the graph. In the experiment presented here we have used a comparator that sorts edges according to their betweenness score [15]. If the betweenness value is the same for two edges, they are sorted by the fully qualified names of source and target vertices. The objective of using this particular comparator function is to make it more likely that edges that are more active in the overall topology of the graph will be bound to path roles and thereby gain an increase in score. Thus, this idea should promote the identification of edges with high global impact.

## 5 Results

### 5.1 Impact of Edge Removal

Figure 6 shows the decline of numbers of antipattern instances after removing the edges with the highest score. Data were obtained using the simple scoring function  $score_1$ .

The number of instances is scaled to 100%. Initially, all programs have 100% of their antipattern instances. The values on the x-axis represent the number of edge removal iterations performed. In each iteration, one edge with the highest score is removed, and then the antipattern counts and the edge scores are recomputed. If there is more than one edge with the same highest score, these edges are sorted according to the fully qualified names of source and target vertices, and the first edge is removed. The main reason for using this selection function is to make the experiment repeatable, and to remove only one edge at a time in order to observe the effects of single edge removals corresponding to *atomic* architectural refactorings.

The chart is a boxplot. The dots in the middle represent the medians in the distribution, and the bold bars

<sup>2</sup>Generic type information is only stored using the signature attribute in Java byte code, this information can be used for reflection, but is not used by the Java runtime when loading, linking and initialising classes and objects.

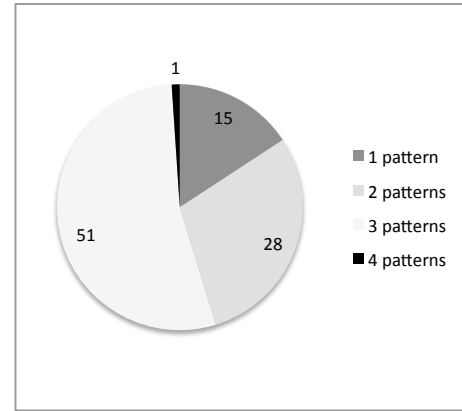


Figure 7: Number of antipatterns instantiated by edge with highest score

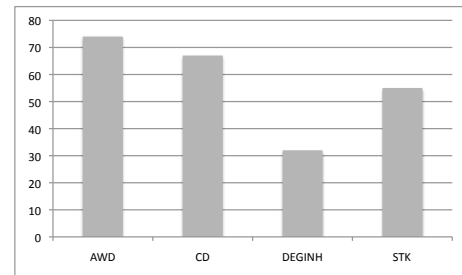


Figure 8: Number of programs with highest scored edge instantiating a given antipattern

around the median represent areas containing 50% of the population.

It is remarkable that the median falls below 50% after only 8 iterations. This means that for half of the programs from the data set, only 8 or fewer edge removals are necessary to remove half of the pattern instances from the model. This suggests that applying refactorings corresponding to these edge removals to the actual programs would have a similarly dramatic effect. The argument is purely statistical: this method works well for most, but not all, programs — the chart shows several outliers.

### 5.2 Pattern Distribution

The question arises whether high scores are caused by single antipatterns, or whether there is an “overlay effect” — edges have high scores because they participate in instances of more than one antipattern. Analysis shows that the latter is the case. For the 95 programs analysed, there are only 15 programs for which the edge with the highest score only participates in instances of a single antipattern. For the majority of programs (51/95), this edge participates in instances of three different antipatterns (figure 7).

Figure 8 shows participation by pattern. For all four patterns we find a significant number of programs where the highest scored edge participates in instances of the respective pattern. This is an indication that we picked a favourable set of patterns in the sense that the combination of these patterns yields synergy effects when detecting edges corresponding to possible high impact refactorings.

The next question we have investigated is whether the simultaneous analysis of multiple patterns yields better results than using one pattern at a time. To answer this question we have created a scoring function for each single pattern. This scoring function increases the score of an edge by one whenever the edge participates in an instance of the respective pattern, and by zero otherwise. That means that

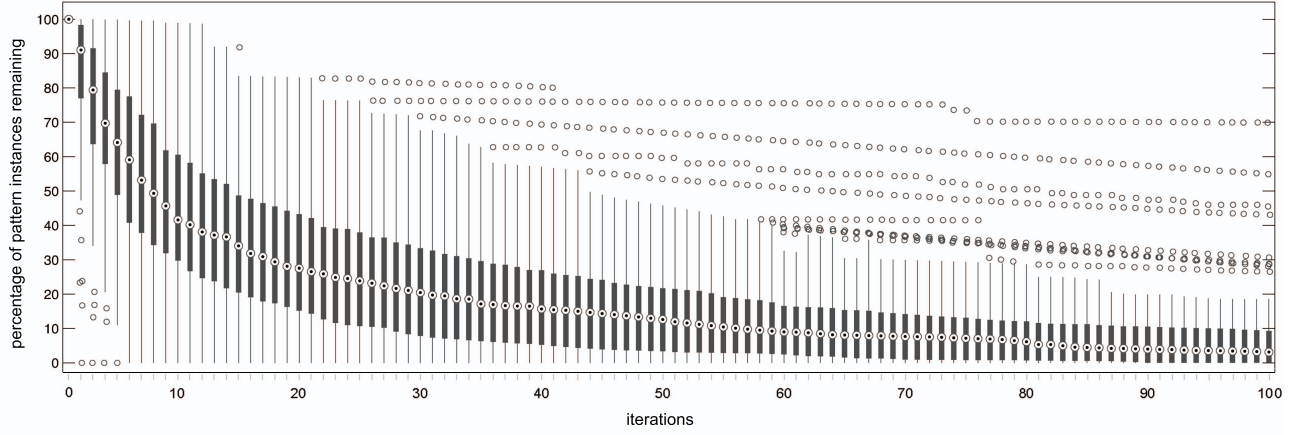


Figure 6: Number of antipattern instances by number of refactorings performed

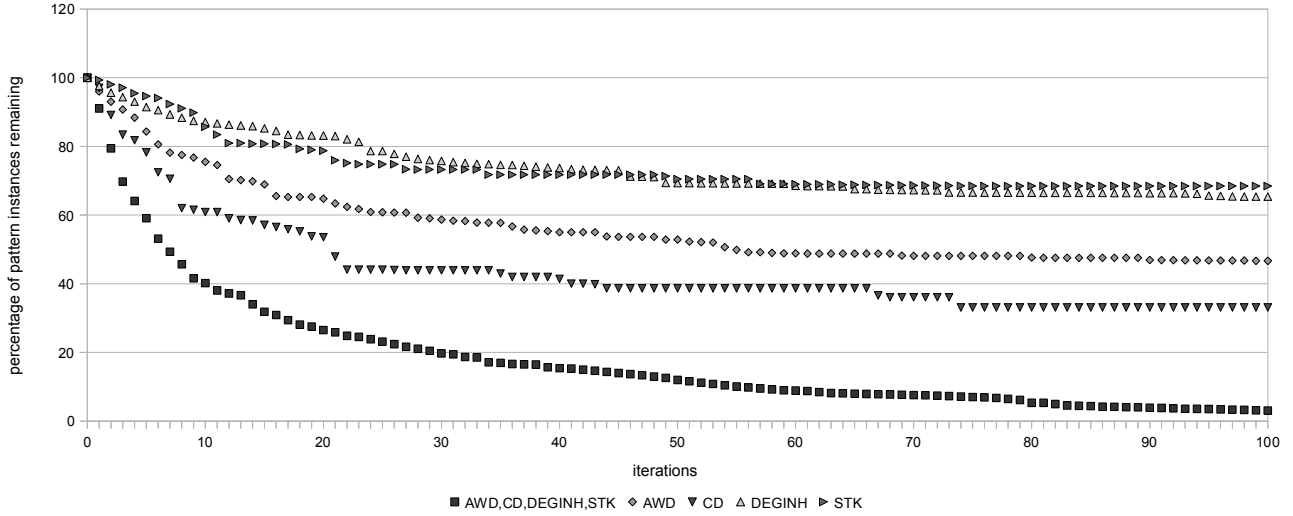


Figure 9: Comparison of antipattern instance removal using analysis based on single and combined antipatterns

only this one pattern is used to compute the edge to be removed. We have then measured how the total number of pattern instances found for all patterns drops. Figure 9 shows the results for the first 50 iterations — the values are the means of pattern instances remaining after the respective number of edge removals. This figure shows that by using the combined strategy (the data series with the label “AWD,CD,DEGINH,STK”) better results can be obtained. The curves representing the single pattern scoring strategies flatten out — indicating that all instances of the respective patterns are eventually removed, but that a significant number of instances of other patterns remain.

### 5.3 Dependency on Program Size

The question arises of whether the trend depends on program size. To address this issue, we have divided the set of programs into two new sets, consisting of relatively small and relatively large programs. The difference between the larger and smaller halves of the programs is relatively small (to remove 50% of the initial number of antipatterns, the mean of edges to be removed is 8 for larger programs and 6 for smaller programs). That indicates that our approach may be particularly useful to guide the refactoring of larger programs: the effort (to apply refactorings corresponding to the removal of edges in the dependency graph) only increases slowly with program size.

This is surprising, since the number of antipattern instances increases significantly with program size. The av-

erage number of instances in the smaller (larger) half is 335 (15423) for AWD, 2748 (140254) for CD, 153 (1605) for DEGINH and 27 (356) for STK, respectively. The large numbers for some antipatterns are caused by the combinatorial explosion of the number of paths defined by references (edges in the dependency graph). It turns out that many of these antipattern instances can be considered as variations of a much smaller number of “significant” instances [8].

Figure 10 shows the number of iterations that are necessary to remove 50% of antipattern instances, depending on program size measured by the number of vertices in the dependency graph. This chart shows that, for most programs, only few edge removals are necessary to achieve the goal. However, there are a few programs that require a very large number of edge removals.

One of these programs is the spring framework, a well known dependency injection container. Here, the fact that there are no high-impact refactorings can be seen as an indicator of good design — those refactoring opportunities have already been detected and the respective refactorings have been performed by moving dependencies into configuration files. Those dependencies are not part of the dependency graph.

### 5.4 Scalability

We have found that for average size programs our implementation of the algorithm scales very well. Anal-

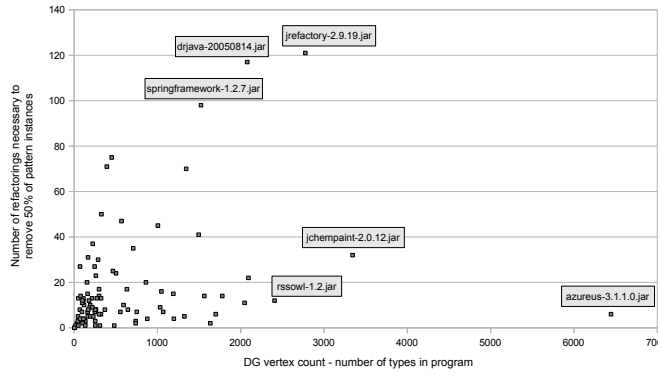


Figure 10: Number of refactorings necessary to remove 50% of antipattern instances

program	vertices	edges	iter. 1	iter. 10	iter. 100
azureus-3.1.1.0	6444	35392	717718	275599	73288
jrubby-1.0.1.jar	2093	11016	90713	89101	31226
derby-10.1.1.0	1198	11174	27882	8468	3898
xerces-2.8.0.jar	878	4782	3448	1533	738
ant-1.7.0	752	3326	6553	1838	659
lucene-1.4.3.jar	231	930	479	456	430
junit-4.5.jar	188	648	439	432	426

Table 1: Performance Data (times in ms)

ysis typically finishes within a few seconds or minutes. We have used a MacBook Pro with a 2.8 GHz Intel Core 2 Duo with 4GHz of memory. We have used the Java(TM) SE Runtime Environment (JRE build 1.6.0\_17-b04-248-10M3025) with the Java HotSpot(TM) 64-Bit Server VM, and a multithreaded solver running on two threads for analysis. For the largest programs in the data set, *azureus-3.1.1.0*, the time needed to finish the initial iteration was about 12min (717718ms). Table 1 shows performance data for some selected, widely-used programs. The time to run an iteration decreases significantly as more edges are removed, in particular for larger programs. As more and more edges are removed, the dependency graph becomes more and more disconnected and the solver has to iterate over fewer sets of paths.

We have also tried to analyse the JRE itself, consisting of the three libraries *rt.jar*, *jce.jar* and *jsse.jar*. The dependency graph extracted from these libraries is large, consisting of 17253 vertices and 173911 edges. The algorithm can still be applied, but computing the first iteration alone took approximately 4.5 hours. Note that the solver algorithm takes full advantage of multi-core processors and can be easily distributed on grids. We therefore think that it is still possible to use our approach for exceptionally large programs by utilising distributed computing environments such as Amazon's EC2.

## 5.5 Classifying Edge Removals

An edge in the dependency graph represents a dependency from a source type to a target type in the program. Dependencies arise in a number of ways from the source code. The edge removal we have performed corresponds to an actual refactoring that has to be applied to the original program. We expect that a template based approach can be used for this purpose, based on the kind of dependency. For this purpose, we have classified the edges according to the source code pattern detected that has caused this dependency.

We classify the edges into eight categories as follows:

1. **Variable Declaration (VD):** The target type is used

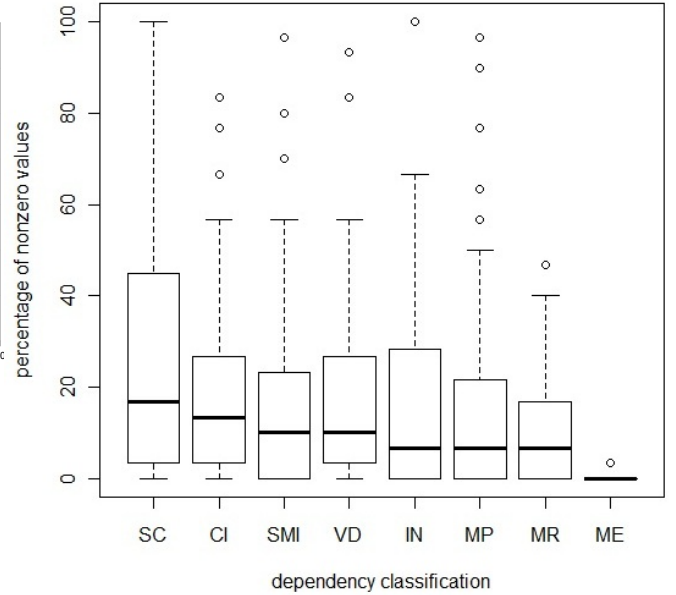


Figure 11: Dependency classification results

to declare a field or a temporary variable.

2. **Constructor Invocation (CI):** A target type constructor is invoked with the keyword *new*.
3. **Static Member Invocation (SMI):** Invocation of a static member (method or field) of the target type.
4. **Method Return Type (MR):** The target type is used as a method return type.
5. **Method Parameter Type (MP):** The target type is used as a parameter type in the method signature.
6. **Method Exception Type (ME):** The target type is used as an exception type with *throws* keyword.
7. **Superclass (SC):** The target type is used as a superclass by using *extends* keyword.
8. **Interface (IN):** The target type is used as an interface by using the *implements* keyword.

We have analyzed a high-scoring subset of the removed edges in order to classify them according to the dependencies giving rise to those edges. The edges in the dependency graph contain one of the three different labels i.e. *uses*, *extends* and *implements*. A *uses* edge can be involved in multiple dependency categories. This is because a source type can use the target type in a number of above-mentioned ways.

Figure 11 shows the distribution of the percentage of non-zero values in every dependency category. We analysed all 95 programs and in every program the first 30 removed edges, with a few exceptions where the total number of edges removed was less than 30. We scaled the non-zero values of every category to 100% with respect to the number of edges analysed. For example, if, in the top 30 relationships (edges) SMI is encountered 15 times, then, for the given program the usage of SMI would be 50%. We can see from figure 11 that most of the dependencies are caused by inheritance relationships, while the lowest number of dependencies comes from the method exception types.

In order to see how often we have multiple reference types, we calculated the participation of the first removed edge for every program in different dependency categories. We found that 41% of the programs have edges



that participate in multiple dependency categories. This suggests that refactoring of these programs will be more challenging.

## 5.6 Implementing Edge Removals

The algorithm presented here operates only on *models* (the dependency graphs), not *programs*. The refactorings recommended by the algorithm are operations to remove artefacts from the model rather than from the program itself. The question arises as to how these refactorings can be implemented so that the actual program can be transformed.

In general, implementing refactorings of the program corresponding to edge removals in the dependency graph is a difficult problem as it requires a very detailed understanding of the design of the respective program. There are situations when nobody has this understanding, for instance if projects evolve, many people are involved, participants change, and design is neither documented nor planned. However, there are some edge removals that can easily be interpreted. The first edge tagged for removal from the Java Runtime Environment (OpenJDK version 1.6.0 b.14 for Windows) is the reference from `java.lang.Object` to `java.lang.Class`. This is caused by the fact that all classes reference `Object` and `Class` has outgoing edges as well. It is probably very difficult and not necessarily desirable to refactor the JRE in order to get rid of this particular edge. However, the second and third targeted edges are references from AWT to Swing: `java.awt.Component` uses `javax.swing.JComponent` and `java.awt.Container` uses `javax.swing.JInternalFrame`. These references point to a real problem. While it is understandable that Swing references the older AWT toolkit, it is hard to see why AWT has to reference the newer Swing toolkit. This makes it impossible to deploy AWT applications without the more resource-demanding Swing. There are several use cases for this: AWT uses the more efficient platform widget toolkits, and AWT applets are at least partially compatible with Microsoft Internet Explorer.

It is interesting to see that those two references are not present in the alternative Apache Harmony [3] implementation of the Java development kit (version 6.0, r917296-snapshot). This implies that it is really possible to “break” the respective edges in the model without compromising the behavioural integrity of the respective system. In this case, a comprehensive set of test suites is used to ensure compatibility between Apache Harmony and the OpenJDK, which is the reference implementation of the Java Development kit.

Another interesting example is `azureus-3.1.1.0`, the largest program in the data set. It has a large initial number of pattern instances in the model (846147) that suddenly drops to 271166 (32.05% of the initial count) after only 5 edge removals. The first five edges removed are:

The first edge is a reference from the plugin manager interface `org.gudy.azureus2.plugins.PluginManager` that orchestrates the application to its concrete subclass `org.gudy.azureus2.pluginsimpl.local.PluginManagerImpl`. There are five references in the compilation unit, all sharing the same structure: static method calls are delegated to the implementation class. These dependencies can be easily removed through the use of a service registry: the plugin manager can obtain the name of the implementation class from the registry, load this class and invoke the respective method using reflection. The next four edges are similar, and can be removed using the same strategy.

We believe that it may not be possible to automate, or even always implement, the refactorings recommended by the proposed algorithm. Actual refactoring is about manipulating program source code or mod-

els close to source code (such as abstract syntax trees), and is therefore programming language dependent. However, we can observe certain patterns causing dependencies between classes which occur in all mainstream programming languages. These are the categories discussed in section 5.5. For some of these categories, there are common refactoring techniques that can be applied. These include the use of design patterns and dynamic programming techniques that have been developed to avoid or reduce dependencies, such as factories, proxies, service registries and dependency injection containers. These techniques are particularly useful to remove dependencies between client classes and service implementation classes. Examples include general-purpose frameworks such as the Spring framework, Guice, the `java.util.ServiceLoader` class, OSGi and its extensions such as declarative services and the Eclipse extension registry, and specialist solutions such as the JDBC driver manager and the JAXP Document Builder Factory.

Often, referenced types can be replaced by their supertypes if those supertypes define the part of the interface of the type that is being referenced by the client. This is possible in all modern mainstream programming languages that use dynamic method lookup. For instance, if a (Java) method references a method with a parameter type `java.util.ArrayList`, the parameter type can usually safely be replaced by `java.util.List`.

There are limitations to this approach that make it unlikely that this can be completely automated. In particular, the use of dynamic programming techniques such as reflection makes it sometimes difficult to predict the behavioural changes caused by these transformations. This implies that firstly, refactoring activities must be safeguarded by verification techniques, such as post refactoring testing; and secondly, that it is an empirical question to find out to what extent these refactorings can be automated in real world systems.

A comprehensive study to determine to what extent refactorings corresponding to our edge removal operations can be automated is subject to further research.

## 6 Conclusion

We have presented an algorithm that can be used to detect potential high-impact refactorings based on the participation of references in sets of antipatterns that are seen as design flaws. We have validated our approach by using a set of four antipatterns that are known to compromise modularisation of programs, applied to a set of 95 programs. The main result presented in this paper is that, in most cases, the algorithm will be able to detect high-impact refactoring opportunities.

We have demonstrated that the respective refactorings can be applied without changing the program behaviour, for some examples, using the largest and most complex programs in our data set. We did not discuss an actual algorithm to automatically perform refactorings corresponding to edge removal. This question has to be explored in future investigations. We believe that the classification of dependency types in section 5.5 is a good starting point for such a study. We have realistic expectations here — while we expect that in many cases the refactorings are easy to describe and can be automated (for instance, by introducing dependency injection or replacing concrete type references by references to interfaces), this will not always be the case. The research challenge is to define refactorings that can be automated in restricted situations where certain prerequisites are fulfilled, and then to find the *weakest prerequisites*. The difficulty of performing dependency-breaking refactorings represents a cost that could be taken into account when defining the scoring functions used in our approach.

Investigating alternative combinations of antipattern sets and scoring functions is an interesting and promising field. There is no evidence that the combination we have used is optimal. Unfortunately, the validation for each set of parameters against the corpus is computationally expensive and takes several hours to complete, this makes a trial and error approach difficult.

There are several interesting theoretical aspects related to this work that can be explored further. For instance, how does the pattern density found in the dependency graphs of typical Java programs compare to that for randomised graphs? For the simpler notion of motifs used in bio-informatics, a study of this kind has been done by Milo et al. to detect the Z-score [21].

## References

- [1] OSGi™— the dynamic module system for Java. <http://www.osgi.org/>.
- [2] Project jigsaw. <http://openjdk.java.net/projects/jigsaw/>.
- [3] Apache Harmony, 2010. <http://harmony.apache.org/>.
- [4] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149, 2005.
- [5] F. Bourqun and R. K. Keller. High-impact refactoring based on architecture violations. In *Proceedings CSMR '07*.
- [6] W. J. Brown, R. C. Malveau, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, March 1998.
- [7] T. Copeland. *PMD Applied*. Centennial Books, 2005.
- [8] J. Dietrich, C. McCartin, E. Temero, and S. M. A. Shah. Barriers to Modularity — An empirical study to assess the potential for modularisation of Java programs. In *Proceedings QoSA'10*, 2010.
- [9] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, and M. Duchrow. Cluster analysis of Java dependency graphs. In *Proceedings SoftVis'08*, pages 91–94, 2008.
- [10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
- [11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [12] M. Fowler. Inversion of control containers and the dependency injection pattern, 2004. <http://martinfowler.com/articles/injection.html>.
- [13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, 1995.
- [14] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. Identifying architectural bad smells. In *Proceedings CSMR'09*, pages 255–258, Los Alamitos, CA, USA, 2009. IEEE Computer Society.
- [15] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proc Natl Acad Sci U S A*, 99(12):7821–7826, June 2002.
- [16] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings OOPSLA '04*, pages 132–136, New York, NY, USA, 2004. ACM.
- [17] C. Humble. IBM, BEA and JBoss adopting OSGi. <http://www.infoq.com/news/2008/02/osgi-jee>.
- [18] M. Lippert and S. Roock. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 2006.
- [19] R. Martin. OO Design Quality Metrics: An Analysis of Dependencies. <http://www.objectmentor.com/resources/articles/oodmetrc.pdf>, May 1994.
- [20] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, 6(3):269–285, September 2007.
- [21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, October 2002.
- [22] M. O’Keeffe and M. O’Cinneide. Search-based software maintenance. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 249–260, Washington, DC, USA, 2006. IEEE Computer Society.
- [23] J. O’Madadhain, D. Fisher, S. White, and Y.-B. Boey. The JUNG (Java universal network/graph) framework. Technical Report UCI-ICS 03-17, University of California, Irvine, 2003.
- [24] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings ICSE '78*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.
- [25] Qualitas Research Group. Qualitas corpus version 20090202, 2009.
- [26] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
- [27] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.
- [28] M. Sakkinen. Disciplined inheritance. In *Proceedings ECOOP'89*, pages 39–56, 1989.
- [29] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916, New York, NY, USA, 2006. ACM.
- [30] F. Simon, F. Steinbrueckner, and C. Lewerentz. Metrics based refactoring. In *Proceedings CSMR'01*, page 30. IEEE Computer Society, 2001.
- [31] G. B. Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.
- [32] W. Stevens, G. Myers, and L. Constantine. Structured design. pages 205–232, 1979.
- [33] T. Taibi, editor. *Design Patterns Formalization Techniques*. Idea Group Inc., Hershey, USA, 2007.



- [34] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.
- [35] J. Tessier. Dependency finder. <http://depfind.sourceforge.net/>.
- [36] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 99(RapidPosts):347–367, 2009.



# Declarative Diagnosis of Floundering in Prolog

Lee Naish

Department of Computing and Information Systems  
The University of Melbourne, Victoria 3010, Australia  
lee@cs.mu.oz.au

## Abstract

Many logic programming languages have delay primitives which allow coroutining. This introduces a class of bug symptoms — computations can *flounder* when they are intended to succeed or finitely fail. For concurrent logic programs this is normally called *deadlock*. Similarly, constraint logic programs can fail to invoke certain constraint solvers because variables are insufficiently instantiated or constrained. Diagnosing such faults has received relatively little attention to date. Since delay primitives affect the procedural but not the declarative view of programs, it may be expected that debugging would have to consider the often complex details of interleaved execution. However, recent work on semantics has suggested an alternative approach. In this paper we show how the declarative debugging paradigm can be used to diagnose unexpected floundering, insulating the user from the complexities of the execution.

Keywords: logic programming, coroutining, delay, debugging, floundering, deadlock, constraints

## 1 Introduction

The first Prolog systems used a strict left to right evaluation strategy, or computation rule. However, since the first few years of logic programming there have been systems which support coroutining between different sub-goals (Clark & McCabe 1979). Although the default order is normally left to right, individual calls can *delay* if certain arguments are insufficiently instantiated, and later *resume*, after other parts of the computation have further instantiated them. Such facilities are now widely supported in Prolog systems. They also gave rise to the class of *concurrent* logic programming languages, such as Parlog (Gregory 1987), where the default evaluation strategy is parallel execution and similar delay mechanisms are used for synchronisation and prevention of unwanted nondeterminism. Delay mechanisms have also been influential for the development of *constraint logic programming* (Jaffar & Lassez 1987). Delays are often used when constraints are “too hard” to be handled by efficient constraint solvers, for example, non-linear constraints over real numbers.

Of course, more features means more classes of bugs. In theory, delays don’t affect soundness of Prolog<sup>1</sup> (see (Lloyd 1984)) — they can be seen as affect-

ing the “control” of the program without affecting the logic (Kowalski 1979). However, they do introduce a new class of bug symptoms. A call can delay and never be resumed (because it is never sufficiently instantiated); the computation is said to *flounder*. Most Prolog systems with delays still print variable bindings for floundered derivations in the same way as successful derivations (in this paper we refer to these as “floundered answers”), and may also print some indication that the computation floundered. Floundered answers are not necessarily valid, or even satisfiable, according to the declarative reading of the program, and generally indicate the presence of a bug. In concurrent logic programs the equivalent of floundering is normally called *deadlock* — the computation terminates with no “process” (call) sufficiently instantiated to proceed. In constraint logic programming systems, the analogue is a computation which terminates with some insufficiently instantiated constraints not solved (or even checked for satisfiability). Alternatively, if some constraints are insufficiently instantiated they may end up being solved by less efficient means than expected, such as exhaustive search over all possible instances.

There is a clear need for tools and techniques to help diagnose floundering in Prolog (and analogous bug symptoms in other logic programming languages), yet there has been very little research in this area to date. There has been some work on showing floundering is impossible using syntactic restrictions on goals and programs (particularly logic databases), or static analysis methods (for example, (Marriott, Søndergaard & Dart 1990)(Marriott, García de la Banda & Hermenegildo 1994)). However, this is a far cry from general purpose methods for diagnosing floundering. In this paper we present such a method. Furthermore, it is a surprisingly attractive method, being based on the declarative debugging paradigm (Shapiro 1983) which is able to hide many of the procedural details of a computation. Declarative debugging has been widely used for diagnosing wrong answers in programming languages based on (some combination of) the logic, functional and constraint paradigms (Pope & Naish 2003, Caballero, Rodríguez-Artalejo & del Vado Vírveda 2006) and there has been some work on diagnosing missing answers (Naish 1992) (which mentions some problems caused by coroutining) (Caballero, Rodríguez-Artalejo & del Vado Vírveda 2007), pattern match failure (Naish & Barbour 1995) and some other bug symptoms (Naish 1997). However, floundering is not among the symptoms previously diagnosed using this approach.

The paper is structured as follows. We first give some examples of how various classes of bugs can lead to floundering. We then present our method of diagnosing floundering, give examples, and discuss how our simple prototype could be improved. Next we

Copyright ©2011, Australian Computer Society, Inc. This paper appeared at 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology, Vol. 122. Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

<sup>1</sup>In practice, floundering within negation can cause unsoundness.

```

% perm(As0, As): As = permutation of
% list As0
% As0 or As should be input
perm([], []).
perm([A0|As0], [A|As]) :-
    when((nonvar(As1) ; nonvar(As)),
        inserted(A0, As1, [A|As])),
    when((nonvar(As0) ; nonvar(As1)),
        perm(As0, As1)).

% inserted(A, As0, As): As = list As0
% with element A inserted
% As0 or As should be input
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(As)),
        inserted(A, As0, As)).

```

Figure 1: A reversible permutation program

```

% Bug 1: wrong variable AS0 in recursive
% call
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(As)),
        inserted(A, AS0, As)). % XXX

% Bug 2: wrong variable A in when/2
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when((nonvar(As0) ; nonvar(A)), % XXX
        inserted(A, As0, As)).

% "Bug" 3: assumes As0 is input XXX
% (perm/2 intended modes incompatible)
inserted(A, As0, [A|As0]).
inserted(A, [A1|As0], [A1|As]) :-
    when(nonvar(As0), % XXX
        inserted(A, As0, As)).

```

Figure 2: Buggy versions of inserted/3

briefly consider some more theoretical aspects, then conclude. Basic familiarity of Prolog with delays and declarative debugging is assumed.

## 2 Example

Figure 1 gives a permutation program which has simple logic but is made reversible by use of delaying primitives and careful ordering of sub-goals in `perm/2` (see (Naish 1986) for further discussion). The delay primitive used is the “when meta-call”: a call `when(Cond,A)` delays until condition `Cond` is satisfied, then calls `A`. For example, the recursive call to `perm/2` will delay until at least one of its arguments are non-variables. Generally there are other features supported, such as delaying until a variable is ground; we don’t discuss them here, though our method and prototype support them. A great number of delay primitives have been proposed (Clark & McCabe 1979, Naish 1986). Some, like the when meta-call, are based on calls. Others are based on procedures (affecting all calls to the procedure), which is often more convenient and tends to clutter the source code less. Our general approach to diagnosis is not affected by the style of delay primitive. The when meta-call is by far the most portable of the more flexible delay primitives, which is our main reason for choosing it. We have developed the code in this paper using SWI-Prolog.

We consider three separate possible bugs which

```

?- perm([1,2,3],A).
Call: perm([1,2,3],_G0)
Call: when(...,inserted(1,_G1,[_G2|_G3]))
Exit: when(...,inserted(1,_G1,[_G2|_G3]))
Call: when(...,perm([2,3],_G1))
Call: perm([2,3],_G1)
Call: inserted(1,[_G4|_G5],[_G2|_G3])
Exit: inserted(1,[_G4|_G5],[1,_G4|_G5])
Call: when(...,inserted(2,_G6,[_G4|_G5]))
Exit: when(...,inserted(2,_G6,[_G4|_G5]))
Call: when(...,perm([3],_G6))
Call: perm([3],_G6)
Call: inserted(2,[_G7|_G8],[_G4|_G5])
Exit: inserted(2,[_G7|_G8],[2,_G7|_G8])
Call: when(...,inserted(3,_G9,[_G7|_G8]))
Exit: when(...,inserted(3,_G9,[_G7|_G8]))
Call: when(...,perm([],_G9))
Call: perm([],_G9)
Call: inserted(3,[],[_G7|_G8])
Exit: inserted(3,[],[3])
Exit: perm([],[])
Exit: when(...,perm([],[]))
Exit: perm([3],[3])
Exit: when(...,perm([3],[3]))
Exit: perm([2,3],[2,3])
Exit: when(...,perm([2,3],[2,3]))
Exit: perm([1,2,3],[1,2,3])

```

Figure 3: Trace with delayed and resumed calls

could have been introduced, shown in Figure 2. They exemplify three classes of errors which can lead to floundering: logical errors, incorrect delay annotations and confusion over the modes of predicates. Bug 1 is a logical error in the recursive call to `inserted/3`. Such errors can cause wrong and missing answers as well as floundering. Due to an incorrect variable name, other variables remain uninstantiated and this can ultimately result in floundering. This bug can be discovered by checking for singleton variables, so in practice declarative debugging should not be required, but we use it as a simple illustration of several points. Despite the simplicity of the bug and the program, a complex array of bug symptoms results, which can be quite confusing to a programmer attempting to diagnose the problems.

The call `perm([1,2,3],A)` first succeeds with answer `A=[1,2,3]`, which is correct. Figure 3 shows the execution trace generated using SWI-Prolog (some details on each line are removed to save space). The trace is the same for all versions of the program. The first call to `inserted/3` (wrapped in a when annotation) delays, shown in the first `Exit` line of the trace. It is resumed immediately after the recursive call to `perm([2,3],_G1)` because matching with the clause head for `perm/2` instantiates `_G1`. Subsequent calls to `inserted/3` also delay and are resumed after further recursive calls to `perm/2`. In this case, all resumed subcomputations immediately succeed (they match with the fact for `inserted/3`). To find other permutations the recursive clause for `inserted/3` must be selected, and the resumed subcomputations delay again when `inserted/3` is called recursively.

On backtracking, for Bug 1, there are four other successful answers found which are satisfiable but not valid, for example, `A=[1,2,3|_]` and `A=[3,1|_]`. An atomic formula, or atom, is *satisfiable* is some instance is true according to the programmer’s intentions and *valid* if all instances are true. These answers could be diagnosed by existing wrong answer declarative debugging algorithms, though some early approaches assumed bug symptoms were unsatisfiable atoms (Naish 1997). An atom is *unsat-*

```

?- perm([A,1|B],[2,3]).
Call: perm([_G0,1|_G1],[2,3])
Call: when(...,inserted(_G0,_G2,[2,3]))
Call: inserted(_G0,_G2,[2,3])
Exit: inserted(2,[3],[2,3])
Exit: when(...,inserted(2,[3],[2,3]))
Call: when(...,perm([1|_G1],[3]))
Call: perm([1|_G1],[3])
Call: when(...,inserted(1,_G3,[3]))
Call: inserted(1,_G3,[3])
Call: when(...,inserted(1,_G7,[]))
Call: inserted(1,_G7,[])
Fail: inserted(1,_G7,[])
Fail: when(...,inserted(1,_G7,[]))
Fail: inserted(1,_G3,[3])
Fail: when(...,inserted(1,_G3,[3]))
Fail: perm([1|_G1],[3])
Fail: when(...,perm([1|_G1],[3]))
Redo: inserted(_G0,_G2,[2,3])
Call: when(...,inserted(_G0,_G4,[3]))
Call: inserted(_G0,_G4,[3])
Exit: inserted(3,[],[3])
Exit: when(...,inserted(3,[],[3]))
Exit: inserted(3,[2|_G5],[2,3])
Exit: when(...,inserted(3,[2|_G5],[2,3]))
Call: when(...,perm([1|_G1],[2|_G5]))
Call: perm([1|_G1],[2|_G5])
Call: when(...,inserted(1,_G6,[2|_G5]))
Exit: when(...,inserted(1,_G6,[2|_G5]))
Call: when(...,perm(_G1,_G6))
Exit: when(...,perm(_G1,_G6))
Exit: perm([1|_G1],[2|_G5])
Exit: when(...,perm([1|_G1],[2|_G5]))
Exit: perm([3,1|_G1],[2,3])
    
```

Figure 4: Trace for Bug 1

isfiable if no instance is true according to the programmer's intentions. These answers are interleaved with four floundered answers, such as  $A=[1,3,[_]]$ , which are also satisfiable but not valid — when `inserted/3` is called recursively, `As0` remains uninstantiated because the incorrect variable is used, and after several more calls and some backtracking, floundering results. The call `perm(A,[1,2,3])` succeeds with the answer  $A=[1,2,3]$  then has three floundered answers, also including  $A=[1,3,[_]]$ . The call `perm([A,1|B],[2,3])` is unsatisfiable and should finitely fail but returns a single floundered answer with  $A=3$ . Figure 4 gives a trace of this computation.

Diagnosing floundering using execution traces is also made more challenging by backtracking. In Figure 4, lines 13–18 are failures, which cause backtracking over previous events. Another complicating factor is that when a predicate exits, it may not have completed execution. There may be subcomputations delayed which are subsequently resumed (see Figure 3), and these resumed subcomputations may also have delayed parts, etc — there can be coroutines between multiple parts of the computation. There are typically `Exit` lines of the trace which are not valid but are correct because the subcomputation floundered rather than succeeded (often this is not immediately obvious from the trace).

With Bug 2, an incorrect delay annotation on the recursive call to `inserted/3`, several bug symptoms are also exhibited. The call `perm([X,Y,Z],A)` behaves correctly but `perm([1,2,3],A)` succeeds with the answers  $A=[1,2,3]$  and  $A=[1,3,2]$ , then loops indefinitely. We don't consider diagnosis of loops in this paper, though they are an important symptom of incorrect control. The call `perm(A,[1,2,3])` succeeds with the answer  $A=[1,2,3]$  then has three further

floundered answers,  $A=[1,2,[_]]$ ,  $A=[1,[_],[_]]$  and  $A=[[_],[_],[_]]$ , before terminating with failure.

Bug 3 is a more subtle control error. When `inserted/3` was coded we assume the intention was the second argument should always be input, and the delay annotation is correct with respect to this intention. This is a reasonable definition of `inserted/3` and we consider it is correct. However, some modes of `perm/2` require `inserted/3` to work with just the third argument input. When coding `perm/2` the programmer was either unaware of this or was confused about what modes `inserted/3` supported. Thus although we have modified the code for `inserted/3`, we consider the bug to be in `perm/2`. This version of the program behaves identically to Bug 2 for the goal `perm(A,[1,2,3])`, but the bug diagnosis will be different because the programmer intentions are different. The mistake was made in the coding of `perm/2`, and this is reflected in the diagnosis. The simplest way to fix the bug is change the intentions and code for `inserted/3`, but we only deal with diagnosis in this paper.

Because delays are the basic cause of floundering and they are inherently procedural, it is natural to assume that diagnosing unexpected floundering requires a procedural view of the execution. Even with such a simple program and goals, diagnosis using just traces of floundered executions can be extremely difficult. Subcomputations may delay and be resumed multiple times as variables incrementally become further instantiated, and this can be interleaved with backtracking. Reconstructing how a single subcomputation proceeds can be very difficult. Although some tools have been developed, such as printing the history of instantiation states for a variable, diagnosis of floundering has remained very challenging.

### 3 Declarative diagnosis of floundering

To diagnose unexpected floundering in pure Prolog programs with delays we use an instance of the three-valued declarative debugging scheme described in (Naish 2000). We describe the instance precisely in the following sections, but first introduce the general scheme and how it is applied the more familiar problem of diagnosing wrong answers. A computation is represented as a tree, with each node associated with a section of source code (a clause in this instance) and subtrees representing subcomputations. The choice of tree generally depends in the language and the bug symptom. For example, diagnosing wrong answers in Prolog generally uses proof trees (see Section 3.1) whereas diagnosing pattern match failure in a functional language requires a different kind of tree. The trees we use here are a generalisation of proof trees. The debugger searches the tree and eventually finds a node for which the associated code has a bug.

Each node has a truth value which expresses how the subcomputation compares with the intentions of the programmer. Normally the truth values of only some nodes are needed to find a bug and they are determined by asking the user questions. Three truth values are used for tree nodes: *correct*, *erroneous*, and *inadmissible*. To diagnose wrong answers, the user is asked about the atoms in proof tree nodes, which were proved in the computation. Correct nodes are those containing an atom which is valid in the intended interpretation, such as `inserted(2,[1],[1,2])`. The corresponding subcomputation returned the correct result and the subtree is eliminated from the search space, since it cannot be the cause of a wrong answer at the root of the tree. Erroneous nodes correspond to subcomputations which returned a wrong answer, such as `inserted(2,[1],[1])`. If such a

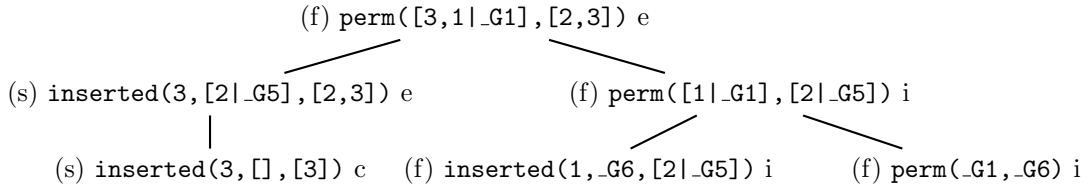


Figure 5: A Partial proof tree for Bug 1

node is found, the search space can be restricted to that subtree, since it must contain a bug. Inadmissible nodes correspond to subcomputations which should never have occurred. Inadmissibility was initially used to express the fact that a call was ill-typed (Pereira 1986). For example, `inserted` is expected to be called with a list in the second and/or third argument, so `inserted(2,a,[2|a])` would be considered inadmissible. Inadmissible means a *pre-condition* of the code has been violated, whereas erroneous means a *post-condition* has been violated. For inadmissible nodes, the subtree can be eliminated from the search space in the same way as a correct node. Here calls which flounder because they never become sufficiently instantiated are considered inadmissible.

Given a tree with truth values for each node, a node is *buggy* if it is erroneous but has no erroneous children. The computation associated with the node behaved incorrectly but none of its subcomputations behaved incorrectly, so there must be a bug in the code associated with that node. Buggy nodes can be classified into two kinds, depending on whether or not they have any inadmissible children. If there are no inadmissible children (all children are correct) the code has simply produced the wrong result, called an *e-bug* in (Naish 2000). If there are inadmissible children the code has resulted in other code being used in an unintended way, violating a pre-condition. This is called an *i-bug*. Diagnosis consists of searching the tree for a buggy node; such a node must exist if the root is erroneous and the tree is finite. Many search strategies are possible and (Naish 2000) provides very simple code for a top-down search. The code first checks that the root is erroneous. It then recursively searches for bugs in children and returns them if they exist. Otherwise the root is returned as a buggy node, along with an inadmissible child if any are found. In the next sections we first define the trees we use, then discuss how programmer intentions are formalised, give some simple diagnosis sessions and finally make some remarks about search strategy.

### 3.1 Partial proof trees

Standard wrong answer declarative diagnosis uses Prolog proof trees which correspond to successful derivations (see (Lloyd 1984)). Each node contains an atomic goal which was proved in the derivation, in its final state of instantiation, and the children of a node are the subgoals of the clause instance used to prove the goal. Leaves are atomic goals which were matched with unit clauses. We use *partial* proof trees which correspond to successful or *floundered* derivations. The only difference is they have an additional class of leaves: atomic goals which were never matched with any clause because they were delayed and never resumed.

**Definition 1 ((Callable) annotated atom)** An annotated atom is an atomic formula or a term of the form `when(C,A)`, where *A* is an atomic formula

and *C* is a condition of a *when* meta-call. It is callable if it is an atom or *C* is true according to the normal Prolog meaning (for “;”, “,” and `nonvar/1`). *atom(X)* is the atom of annotated atom *X*.

**Definition 2 (Partial proof tree)** A partial proof tree for annotated atom *A* and program *P* is either

1. a node containing *A*, where *atom(A)* is an instance of a unit clause in *P* or *A* is not callable, or
2. a node containing callable atom *A* together with partial proof (sub)trees *S<sub>i</sub>* for annotated atom *B<sub>i</sub>* and *P*, *i* = 1...*n*, where *atom(A) :- B<sub>1</sub>, ... B<sub>n</sub>* is an instance of a clause in *P*.

A partial proof tree is *floundered* if it contains any annotated atoms which are not callable, otherwise it is successful.

Figure 5 gives the partial proof tree corresponding to the floundering of goal `perm([A,1|B],[2,3])` with Bug 1 (which corresponds to the trace of Figure 4). The “when” annotations are not shown. The single letter prefix for each node indicates whether the subtree is successful (s) or floundered (f). The single letter postfix relates to the truth values of the nodes: correct (c), erroneous (e), or inadmissible (i). Given these assignments, the only buggy node is that containing the atom `inserted(3,[2|C],[2,3])`. In Section 3.2 we explain how these assignments reflect the intentions of the programmer.

Representing a computation as a (partial) proof tree has several advantages over representing it as a linear trace if the goal is to diagnose incorrect successful or floundered derivations. First, backtracking is eliminated entirely, avoiding an important distraction. Second, the details of any coroutining are also eliminated. It has long been known this could be done for successful computations, but the realisation it can be done for floundered computations is relatively new and is the key to our approach. We retain information on what sub-goals were never called, but the order in which other subgoals were executed is not retained. The structure of the tree reflects the static dependencies in the code rather than the dynamic execution order. Because of this, each node gives us the final instantiation state of the atom, not just the instantiation state when it exited (at that time some subcomputations may have been delayed). Finally, the tree structure allows us search efficiently for buggy nodes by checking the truth value of nodes, which can be determined from the programmer’s intentions.

Declarative debuggers use various methods for representing trees and building such representations. The declarative debugger for Mercury (Somogyi & Henderson 1999) is a relatively mature implementation. A much simpler method, which is suitable for prototypes, is a meta interpreter which constructs an explicit representation of the tree. Figure 6 is a very concise implementation which we include for completeness. Floundering is detected using the “short

```

% solve_atom(A, C0, C, AT): A is an
% atomic goal, possibly wrapped in
% when meta-call, which has succeeded
% or floundered; AT is the corresponding
% partial proof tree with floundered
% leaves having a variable as the list
% of children; C0=C iff A succeeded
solve_atom(when(Cond, A), C0, C, AT) :- !,
    AT = node(when(Cond, A), C0, C, Ts),
    when(Cond, solve_atom(A, C0, C,
        node(_,_,Ts))).
solve_atom(A, C0, C, node(A,C0,C,AsTs)) :-
    clause(A, As),
    solve_conj(As, C0, C, AsTs).

% As above for conjunction;
% returns list of trees
solve_conj(true, C, C, []) :- !.
solve_conj((A, As), C0, C, [AT|AsTs]) :- !,
    solve_atom(A, C0, C1, AT),
    solve_conj(As, C1, C, AsTs).
solve_conj(A, C0, C, [AT]) :-
    solve_atom(A, C0, C, AT).
    
```

Figure 6: Code to build partial proof trees

circuit” technique — an accumulator pair is associated with each subgoal and the two arguments are unified if and when the subgoal succeeds. Tree nodes contain an annotated atom, this accumulator pair and a list of subtrees. A subcomputation is floundered if the accumulator arguments in the root of the subtree are not identical.

### 3.2 The programmer’s intentions

The way truth values are assigned to nodes encodes the user’s intended behaviour of the program. In the classical approach to declarative debugging of wrong answers the intended behaviour is specified by partitioning the set of ground atoms into true atoms and false atoms. There can still be non-ground atoms in proof tree nodes, which are considered true if the atom is valid. A difficulty with this two-valued scheme is that most programmers make implicit assumptions about the way their code will be called, such as the “type” of arguments. Although `inserted(2,a,[2|a])` can succeed, it is counter-intuitive to consider it to be true (since it is “ill-typed”), and if it is considered false then the definition of `inserted/3` must be regarded as having a logical error. The solution to this problem is to be more explicit about how predicates should be called, allowing pre-conditions (Drabent, Nadjm-Tehrani & Maluszynski 1988) or saying that certain things are inadmissible (Pereira 1986) or having a three-way partitioning of the set of ground atoms (Naish 2006).

In the case of floundering the intended behaviour of non-ground atoms must be considered explicitly. As well as assumptions about types of arguments, we inevitably make assumptions about how instantiated arguments are. For example, `perm/2` is not designed to generate all solutions to calls where neither argument is a (nil-terminated) list and even if it was, such usage would most likely cause an infinite loop if used as part of a larger computation. It is reasonable to say that calls to `perm/2` where neither argument is ever instantiated to a list should not occur, and hence should be considered inadmissible. An important heuristic for generating control information is that calls which have an infinite number of solutions should be avoided (Naish 1986). Instead, such a call is better delayed, in the hope that other parts of the

computation will further instantiate it and make the number of solutions finite. If the number of solutions remains infinite the result is floundering, but this is still preferable to an infinite loop.

We specify the intended behaviour of a program as follows:

**Definition 3 (Interpretation)** *An interpretation is a three-way partitioning of the set of all atoms into those which are inadmissible, valid and erroneous. The set of admissible (valid or erroneous) atoms is closed under instantiation (if an atom is admissible then any instance of it is admissible), as is the set of valid atoms.*

The following interpretation precisely defines our intentions for `perm/2`: `perm(As0,As)` is admissible if and only if either `As0` or `As` are (nil-terminated) lists, and valid if and only if `As` is a permutation of `As0`. This expresses the fact that either of the arguments can be input, and only the list skeleton (not the elements) is required. For example, `perm([X],[X])` is valid (as are all its instances), `perm([X],[2|Y])` is admissible (as are all its instances) but erroneous (though an instance is valid) and `perm([2|X],[2|Y])` is inadmissible (as are all atoms with this as an instance). For diagnosing Bugs 1 and 2, we say `inserted(A,As0,As)` is admissible if and only if `As0` or `As` are lists. For diagnosing Bug 3, `As0` must be a list, expressing the different intended modes in this case.

Note we do not have different admissibility criteria for different sub-goals in the program — the intended semantics is predicate-based. Delay primitives based on predicates thus have an advantage of being natural from this perspective. Note also that atoms in partial proof tree nodes are in their final state of instantiation in the computation. It may be that in the first call to `inserted/3` from `perm/2`, no argument is instantiated to a list (it may delay initially), but as long as it is eventually sufficiently instantiated (due to the execution of the recursive `perm/2` call, for example) it is considered admissible. However, since admissibility is closed under instantiation, an atom which is inadmissible in a partial proof tree could not have been admissible at any stage of the computation. The debugger only deals with whether a call flounders — the lower level procedural details of when it is called, delayed, resumed *et cetera* are hidden.

Truth values of partial proof tree nodes are defined in terms of the user’s intentions:

**Definition 4 (Truth of nodes)** *Given an interpretation  $I$ , a partial proof tree node is*

1. correct, if the atom in the node is valid in  $I$  and the subtree is successful,
2. inadmissible, if the atom in the node is inadmissible in  $I$ , and
3. erroneous, otherwise.

Note that floundered subcomputations are never correct. If the atom is insufficiently instantiated (or “ill-typed”) it is inadmissible, otherwise it is erroneous.

### 3.3 Diagnosis examples

In our examples we use a top-down search for a buggy node, which gives a relatively clear picture of the partial proof tree. They are copied from actual runs of our prototype<sup>2</sup> except that repeated identical questions are removed and some white-space is changed.

<sup>2</sup>Available from <http://www.cs.mu.oz.au/~lee/papers/ddf/>

```
?- wrong(perm([A,1|B],[2,3])).
(floundered) perm([3,1|A],[2,3])...? e
(floundered) perm([1|A],[2|B])...? i
(succeeded) inserted(3,[2|A],[2,3])...? e
(succeeded) inserted(3,[],[3])...? v
BUG - incorrect clause instance:
inserted(3,[2|A],[2,3]) :-
    when((nonvar(A);nonvar([3])),
        inserted(3,[],[3])).
```

Figure 7: Bug 1 diagnosis for `perm([A,1|B],[2,3])`

```
...
(floundered) perm([1,2,3],[1,3,A|B])...? e
(floundered) perm([2,3],[3,A|B])...? e
(floundered) inserted(2,[3],[3,A|B])...? e
(floundered) inserted(2,[A|B],[A|C])...? i
BUG - incorrect modes in clause instance:
inserted(2,[3],[3,A|B]) :-
    when((nonvar([]);nonvar([A|B]))
        inserted(2,[A|_],[A|B])).
```

Figure 8: Bug 1 diagnosis for `perm([1,2,3],A)`

In section 3.4 we discuss strategies which can reduce the number of questions; the way diagnoses are printed could also be improved. The debugger defines a top-level predicate `wrong/1` which takes an atomic goal, builds a partial proof tree for an instance of the goal then searches the tree. The truth value of nodes is determined from the user. The debugger prints whether the node succeeded or floundered, then the atom in the node is printed and the user is expected to say if it is valid (v), inadmissible (i) or erroneous (e).

Figure 7 shows how Bug 1 is diagnosed for the goal `perm([A,1|B],[2,3])`. The root of the tree (shown in Figure 5) is erroneous, so the debugger proceeds to recursively search for bugs in the children. In this case it first considers the right child, which is inadmissible (so the recursive search fails), then the left child, which is erroneous (and the search continues in this subtree). Note that the call to `perm/2` in the root calls itself in an inadmissible way but this, in itself, does not indicate a bug. The cause of the inadmissible call is the other child, which is erroneous, and the root is not a buggy node. The recursive search in the left subtree determines the left-most leaf is correct and hence the buggy node is found. The diagnosis is a logical error in the `inserted/3` clause: the body of the clause is valid but the head is not.

Figure 8 shows how Bug 1 is diagnosed for the goal `perm([1,2,3],A)`. We assume the user decides to diagnose a floundered answer, backtracking over the previous answers. The diagnosis is ultimately a control error: the arguments of the clause head are sufficiently instantiated but the arguments of the recursive call are not. Both are diagnoses are legitimate. Even without delays, logical bugs can lead to both missing and wrong answers, which typically result in different diagnoses in declarative debuggers.

Figure 9 shows how Bug 2 is diagnosed. The first question relates to the first answer returned by the goal. It is valid, so the diagnosis code fails and the computation backtracks, building a new partial proof tree for the next answer, which is floundered. The root of this tree is determined to be erroneous and after a few more questions a buggy node is found. It is a floundered leaf node so the appropriate diagnosis is an incorrect delay annotation, which causes `inserted(A,B,[])` to delay indefinitely (rather than fail). Ideally we should also display the instance of the

```
?- wrong(perm(A,[1,2,3])).
(succeeded) perm([1,2,3],[1,2,3])...? v
(floundered) perm([1,2,A,B|C],[1,2,3])...? e
(floundered) perm([2,A,B|C],[2,3])...? e
(floundered) perm([A,B|C],[3])...? e
(floundered) inserted(A,[3|B],[3])...? e
(floundered) inserted(A,B,[])...? e
BUG - incorrect delay annotation:
when((nonvar(A);nonvar(B)),inserted(B,A,[]))
```

Figure 9: Diagnosis of bug 2

```
?- wrong(perm(A,[1,2,3])).
(succeeded) perm([1,2,3],[1,2,3])...? v
(floundered) perm([1,2,A,B|C],[1,2,3])...? e
(floundered) perm([2,A,B|C],[2,3])...? e
(floundered) perm([A,B|C],[3])...? e
(floundered) inserted(A,[3|B],[3])...? i
(floundered) perm([A|B],[3|C])...? i
BUG - incorrect modes in clause instance:
perm([A,C|D],[3]) :-
    when((nonvar([3|B]);nonvar([])),
        inserted(A,[3|B],[3])),
    when((nonvar([C|D]);nonvar([3|B])),
        perm([C|D],[3|B])).
```

Figure 10: Diagnosis of bug 3

clause which contained the call (the debugger code in (Naish 2000) could be modified to return the buggy node and its parent), and the source code location.

Figure 10 shows how Bug 3 is diagnosed, using the same goal. It proceeds in a similar way to the previous example (the partial proof trees are identical), but due to the different programmer intentions (the mode for `inserted/3`) the floundering call `inserted(A,[3|B],[3])` is considered inadmissible rather than erroneous, eventually leading to a different diagnosis. Both calls in the buggy clause instance are inadmissible. The debugger of (Naish 2000) returns both these inadmissible calls as separate diagnoses. For diagnosing floundering it is preferable to return a single diagnosis, since the floundering of one can result in the floundering of another and its not clear which are the actual culprit(s).

### 3.4 Search strategy

A top-down left to right search is the simplest search strategy to implement. In our prototype we have a slightly more complex strategy, searching floundered subtrees before successful ones (this is done by adjusting the order in which the children of a node are returned — see Figure 11). More complex strategies for diagnosing some forms of abnormal termination

```
% returns children of a node,
% floundered ones first
child(node(_,_,_,Ts),T):-
    nonvar(Ts), % not a floundered leaf
    (
        member(T,Ts),
        T = node(_,C0,C,_),
        C0 \== C % T is floundered
    );
    member(T,Ts),
    T = node(_,C0,C,_),
    C0 == C % T is not floundered
).
```

Figure 11: Finding children in partial proof trees



are given in (Naish 2000), and these can be adapted to floundering. From our definition of truth values for nodes, we know no floundered node is correct. We also know that floundering is caused by (at least one) floundered leaf node. Thus we have (at least one) path of nodes which are not correct between the root node and a leaf. It makes sense to initially restrict our search to such a path. Our prototype does a top-down search of such a path. There must be an erroneous node on the path with no erroneous children on the path. Both bottom-up and binary search strategies are likely to find this node significantly more quickly than a top-down search. Once this node is found, its other children must also be checked. If there are no erroneous children the node is buggy. Otherwise, an erroneous child can be diagnosed recursively, if it is floundered, or by established wrong answer diagnosis algorithms.

#### 4 Theoretical considerations

We first make some remarks about the soundness and completeness of this method of diagnosis, then discuss related theoretical work. An admissible atomic formula which flounders has a finite partial proof tree with an erroneous root and clearly this must have a buggy node. Since the search space is finite, completeness is easily achieved. Soundness criteria come from the definition of buggy nodes (erroneous nodes with no erroneous children). The three classes of bugs mentioned in Section 2 give a complete categorisation of bugs which cause floundering. Incorrect delay annotations cause floundered buggy leaf nodes: they are admissible but delay. Confusions over modes cause floundered buggy internal nodes: they are admissible but have one or more floundered inadmissible children. Logical errors can also cause such nodes and can cause successful buggy nodes. Note that ancestors of a successful buggy node may also be floundered, as in Figure 5.

Our current work on diagnosis arose out of more theoretical work on floundering (Naish 2008). Nearly all delay primitives have the property that if a certain call can proceed (rather than delay), any more instantiated version of the call can also proceed (the set of callable annotated atoms is closed under instantiation). Our diagnosis method can be effectively applied to other delay primitives for which this property holds simply by changing the definition of callable annotated atoms. An important result which follows from this property is similar to the result concerning success: whether a computation flounders, and the final instantiation of variables, depends on the delay annotations but not on the order in which sufficiently instantiated call are selected. Non-floundering is also closed under instantiation, so it is natural for admissibility to inherit this restriction and partial proof trees provide a basis for intuitive diagnoses. Furthermore, there is a very close correspondence between floundered and successful derivations, and this is what enables our approach to diagnosis. A floundered derivation  $D$  for program  $P$  can be transformed into a successful derivation  $D'$  for a program  $P'$  which is identical to  $P$  except for the delay annotations. We briefly explain (a simplified version of) the mapping next.

The key idea is that a floundered derivation (or partial proof tree) using  $P$  will correspond to a successful derivation (or proof tree) in  $P'$  where the variables which caused floundering in  $P$  are instantiated to special terms which could not be constructed by the rest of the computation — the variables are *encoded* using special terms. This encoding has no effect on successful subcomputations; any subcomputation which succeeds with an answer containing vari-

ables will also succeed if any of those variables are further instantiated. Because Prolog uses most general unifiers, the only terms constructed in a Prolog derivation contain function symbols which appear in the program. Thus “extraneous” function symbols, which do not occur in  $P$ , can be used to encode variables. For simplicity, we will just use \$, and assume it does not appear in  $P$  (in (Naish 2008) multiple encodings are used, which makes the mapping between derivations more precise).

Each annotated atom  $when(C, A)$  in  $P$  is transformed so that the corresponding code in  $P'$  just calls  $A$  when  $C$  is satisfied but can also succeed when  $C$  is not satisfied, encoding the appropriate variables. Calling  $A$  is achieved by having  $A$  as a disjunct in the transformed code. The other disjunct implements the *negation* of  $C$ , using the encoding. The negation of  $nonvar(V)$  ensures  $V$  is an encoded variable, \$, and De Morgan’s laws handle conjunction and disjunction in delay conditions.

**Definition 5** *The transformation  $T$  applied to when annotations is:*

$$\begin{aligned} T(when(C, A)) &= (T(C); A) \\ T((C_1, C_2)) &= (T(C_1); T(C_2)) \\ T((C_1; C_2)) &= (T(C_1), T(C_2)) \\ T(nonvar(V)) &= (V = \$) \end{aligned}$$

For example, the recursive clause for the correct version of `inserted/3` is transformed to:

```
inserted(A, [A1|As0], [A1|As]) :-
    (As0 = $, As = $ ; inserted(A, As0, As)).
```

The goal `inserted(1,[2,3|A],[2,3|B])` has a derivation/partial proof tree in  $P$  which is floundered due to an annotated recursive call to `inserted/3` with variables  $A$  and  $B$  as the second and third arguments, respectively. There is a corresponding successful derivation/proof tree in  $P'$  where  $A$  and  $B$  are instantiated to \$. This correspondence between floundered and successful computations means we can use the properties of successful derivations in the diagnosis of floundering — in particular, abstracting away backtracking and the order in which sub-goals are executed.

In section 3.2 we defined interpretations by partitioning the set of all atoms, rather than just the ground atoms. This is what allows us to say an insufficiently instantiated floundered atom is inadmissible. The ground encoded versions of such atoms would normally be considered ill-typed, hence it is reasonable to consider them inadmissible also. For example, `inserted(1,[2,3|A],[2,3|B])` is inadmissible and the encoded atom `inserted(1,[2,3|$],[2,3|$])` has non-lists in the last two arguments. Thus, by encoding atoms and using ill-typedness in place of under-instantiation, it is possible to define interpretations over just ground atoms. The way we described our intended interpretation in section 3.2 can remain unchanged. Encoding our example from that section, `perm([$],[\$])` is valid, `perm([$],[2|$])` is erroneous and `perm([2|$],[2|$])` is inadmissible. By only using ground atoms, the three-valued semantics proposed in (Naish 2006) can be used unchanged (and our approach can indeed be considered “declarative”). Diagnosis of floundering becomes almost identical to diagnosis of wrong answers in the three-valued scheme. The only difference is the rare case of a valid ground atom which flounders rather than succeeds: when floundering is converted to success it appears there is no bug. For this case it is necessary to distinguish success from floundering, for example, with extra information in each node of the proof tree, as we have done in our implementation.

## 5 Conclusion

There has long been a need for tools and techniques to diagnose unexpected floundering in Prolog with delay primitives, and related classes of bug symptoms in other logic programming languages. The philosophy behind delay primitives in logic programming languages is largely based on Kowalski's equation:  $\text{Algorithm} = \text{Logic} + \text{Control}$  (Kowalski 1979). By using more complex control, the logic can be simpler. This allows simpler reasoning about correctness of answers from successful derivations — we can use a purely declarative view, ignoring the control because it only affects the procedural semantics. When there are bugs related to control it is not clear the trade-off is such a good one. The control and logic can no longer be separated. Since the normal declarative view cannot be used, the only obvious option is to use the procedural view. Unfortunately, even simple programs can exhibit very complex procedural behaviour, making it very difficult to diagnose and correct bugs using this view of the program.

In the case of floundering, a much simpler high level approach turns out to be possible. The combination of the logic and control can be viewed as just slightly different logic, allowing declarative diagnosis techniques to be used. The procedural details of backtracking, calls delaying and the interleaving of subcomputations can be ignored. The user can simply put each atomic formula into one of three categories. The first is inadmissible: atoms which should not be called because they are insufficiently instantiated and expected to flounder (or are “ill-typed” or violate some pre-condition of the procedure). The second is valid: atoms for which all instances are true and are expected to succeed. The third is erroneous: atoms which are legitimate to call but which should not succeed without being further instantiated (they are not valid, though an instance may be). A floundered derivation can be viewed as a tree and this three-valued intended semantics used to locate a bug in an instance of a single clause or a call with a delay annotation.

## References

- Caballero, R., Rodríguez-Artalejo, M. & del Vado Vírveda, R. (2006), Declarative diagnosis of wrong answers in constraint functional-logic programming, in S. Etalle & M. Truszczynski, eds, 'ICLP', Vol. 4079 of *Lecture Notes in Computer Science*, Springer, pp. 421–422.
- Caballero, R., Rodríguez-Artalejo, M. & del Vado Vírveda, R. (2007), Declarative debugging of missing answers in constraint functional-logic programming, in V. Dahl & I. Niemelä, eds, 'ICLP', Vol. 4670 of *Lecture Notes in Computer Science*, Springer, pp. 425–427.
- Clark, K. & McCabe, F. (1979), The control facilities of IC-Prolog, in D. Michie, ed., 'Expert systems in the microelectronic age', Edinburgh University Press, pp. 122–149.
- Drabent, W., Nadjm-Tehrani, S. & Maluszynski, J. (1988), The use of assertions in algorithmic debugging, in 'Proceedings of the 1988 International Conference on Fifth Generation Computer Systems', Tokyo, Japan, pp. 573–581.
- Gregory, S. (1987), *Design, application and implementation of a parallel logic programming language*, Addison-Wesley.
- Jaffar, J. & Lassez, J.-L. (1987), From unification to constraints, in K. Furukawa, H. Tanaka & T. Fujisaki, eds, 'Proceedings of the Sixth Logic Programming Conference', Tokyo, Japan, pp. 1–18. published as Lecture Notes in Computer Science 315 by Springer-Verlag.
- Kowalski, R. (1979), 'Algorithm = Logic + Control', *CACM* **22**(7), 424–435.
- Lloyd, J. W. (1984), *Foundations of logic programming*, Springer series in symbolic computation, Springer-Verlag, New York.
- Marriott, K., García de la Banda, M. & Hermenegildo, M. (1994), Analyzing Logic Programs with Dynamic Scheduling, in '20th. Annual ACM Conf. on Principles of Programming Languages', ACM, pp. 240–254.
- Marriott, K., Søndergaard, H. & Dart, P. (1990), A characterization of non-floundering logic programs, in S. Debray & M. Hermenegildo, eds, 'Proceedings of the North American Conference on Logic Programming', The MIT Press, Austin, Texas, pp. 661–680.
- Naish, L. (1986), *Negation and control in Prolog*, number 238 in 'Lecture Notes in Computer Science', Springer-Verlag, New York.
- Naish, L. (1992), 'Declarative diagnosis of missing answers', *New Generation Computing* **10**(3), 255–285.
- Naish, L. (1997), 'A declarative debugging scheme', *Journal of Functional and Logic Programming* **1997**(3).
- Naish, L. (2000), 'A three-valued declarative debugging scheme', *Australian Computer Science Communications* **22**(1), 166–173.
- Naish, L. (2006), 'A three-valued semantics for logic programmers', *Theory and Practice of Logic Programming* **6**(5), 509–538.
- Naish, L. (2008), 'Transforming floundering into success'.  
**URL:** [www2.cs.mu.oz.au/~lee/papers/flounder](http://www2.cs.mu.oz.au/~lee/papers/flounder)
- Naish, L. & Barbour, T. (1995), A declarative debugger for a logical-functional language, in G. Forsyth & M. Ali, eds, 'Eighth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems — Invited and Additional Papers', Vol. 2, DSTO General Document 51, Melbourne, pp. 91–99.
- Pereira, L. M. (1986), Rational debugging in logic programming, in E. Shapiro, ed., 'Proceedings of the Third International Conference on Logic Programming', London, England, pp. 203–210. published as Lecture Notes in Computer Science 225 by Springer-Verlag.
- Pope, B. & Naish, L. (2003), Practical aspects of declarative debugging in Haskell-98, in 'Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming', pp. 230–240. ISBN:1-58113-705-2.
- Shapiro, E. Y. (1983), *Algorithmic program debugging*, MIT Press, Cambridge, Massachusetts.
- Somogyi, Z. & Henderson, F. J. (1999), The implementation technology of the Mercury debugger, in 'Proceedings of the Tenth Workshop on Logic Programming Environments', Las Cruces, New Mexico, pp. 35–49.

# Learning Time Series Patterns by Genetic Programming

Feng Xie

Andy Song

Vic Ciesielski

School of Computer Science and Information Technology  
RMIT University  
Melbourne, VIC 3001, Australia,  
Email: feng.xie@rmit.edu.au

School of Computer Science and Information Technology  
RMIT University  
Melbourne, VIC 3001, Australia,  
Email: andy.song@rmit.edu.au

School of Computer Science and Information Technology  
RMIT University  
Melbourne, VIC 3001, Australia,  
Email: vic.ciesielski@rmit.edu.au

## Abstract

Finding patterns such as increasing or decreasing trends, abrupt changes and periodically repeating sequences is a necessary task in many real world situations. We have shown how genetic programming can be used to detect increasingly complex patterns in time series data. Most classification methods require a hand-crafted feature extraction preprocessing step to accurately perform such tasks. In contrast, the evolved programs operate on the raw time series data. On the more difficult problems the evolved classifiers outperform the OneR, J48, Naive Bayes, IB1 and Adaboost classifiers by a large margin. Furthermore this method can handle noisy data. Our results suggest that the genetic programming approach could be used for detecting a wide range of patterns in time series data without extra processing or feature extraction.

*Keywords:* Genetic Programming, Pattern Recognition, Time Series

## 1 Introduction

Time series patterns describe how one variable or a group of variables change over a certain period of time. They are a critical factor in many real world problems where temporal information is essential. For example, any single point on an electrocardiogram is meaningless by itself, but a repeating sequence can reveal whether the rhythm of a heart is normal or abnormal. Similarly tasks like understanding patterns of climate change, recognizing words in audio waveforms and detecting target objects in videos all rely on finding patterns in the time domain. It is clear that a method which can capture regularities in temporal data is of great importance.

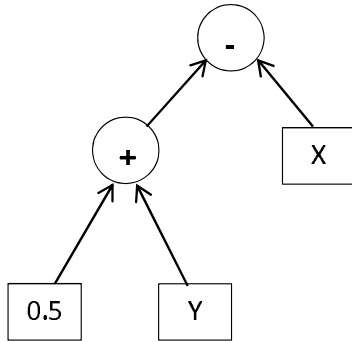
Handling temporal data is not a new area. Many methods have been proposed in the past. However they are significantly different as they were introduced for different tasks. For instance, temporal logic

uses symbols and rules to represent time flow whereas a temporal database marks records with time stamps. Temporal databases introduced the concepts of transaction time and valid period for preserving the time information. Learning methods have been proposed for learning temporal rules or patterns as well. Sutton described the temporal difference (TD) learning method for prediction and proved its convergence and performance on a number of problems(1). Unlike traditional supervised machine learning approaches, the TD method learns from the differences between successive predictions to improve the final outcome. However this algorithm relies on the availability of an optimal set of features for the learning process. Boots and Gordon addressed this problem by adding a feature selection component to retain features which only contain predictive information(3). Neural networks have been used for handling temporal data as well. Dorffner summarized different types of artificial neural networks (ANNs) for this purpose(8). Furthermore ANNs has been combined with evolutionary strategies and a greedy randomised adaptive search procedure for time series forecasting(9).

These aforementioned existing methods do not constitute a generalized approach for learning time series patterns. Instead of operating on raw data, they often rely on some kind of additional processes to extract features from the time series data for learning, such as converting input signals from the time domain into the frequency domain, calculating variations at different points or transforming numeric data into symbolic data. The process for determining relevant features requires experience and domain knowledge from human experts. Manual interventions are usually essential here. Additionally the parameters and configurations are hand-crafted to suit only a certain problem. Therefore it is difficult to generalize a method for various problems. A generalized method for handling temporal data and learning underlying patterns still remain a big challenge.

In this study we propose genetic programming (GP) as a method for learning time series patterns without any extra processes such as data preprocessing or feature extraction. The aim of our investigation is to establish such a generalized method which can deal with different kind of scenarios such as detecting abrupt changes and recognizing various periodical patterns under one framework. Note the focus of this study is on recognizing patterns, not on time series prediction.

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Figure 1: A Typical GP Program Tree  $(y + 0.5) - x$ 

The main reason for using genetic programming is that GP has proved itself as a powerful and creative problem solving method. Even on some difficult problems such as image processing and computer vision, GP can work with unprocessed raw data to learn useful models. Essentially GP is performing two tasks here, extracting useful features and building a classification model accordingly. Typically these two tasks are treated as separate components in the conventional methods, but considered as one by GP-based method. This characteristic makes GP particularly suitable for problems of which the optimal features are unclear. Learning patterns in the time domain can be considered as such a problem. Because it is difficult to foresee what kind of features would be the most suitable if the nature of the underlying temporal patterns is not clear.

The rest of this paper is organized as such: Section 2 gives a brief background of Genetic Programming and its applications. Section 3 presents a collection of problems used in this study. Their difficulties increase gradually. The GP methodology and experiments are discussed in Section 4, while the discussions of our experiments are in Section 5. Section 6 concludes this study and discusses future investigations.

## 2 Background

Genetic Programming is a kind of evolutionary computation methods inspired by the survival-of-the-fittest principal. It was pioneered by Koza(2) who has successfully applied GP in many areas and even patented several solutions found by GP. A solution in GP is typically represented as a program tree on which the internal nodes are functions (operators) and the leaf nodes are terminals (operands). Initially a population of program trees are generated randomly as the first generation. These programs are evaluated on the problem to be solved. Based on the performance, each one of them is assigned with a fitness value. Solutions with higher fitness are more likely to be selected as the parents to generate a new population of solutions, namely the next generation. Programs in the new generation may be created by mutation (applying random change on a parent), crossover (exchanging tree branches between parents), or elitism (directly copying the best solutions from the previous generation). Figures 1 shows a simple GP tree.

There are existing studies using GP techniques to handle problems involving time series information. Kaboudan applied both neural networks and GP to the problem of forecasting housing prices based on both spatial and temporal information and suggested GP could produce more reliable and logically more ac-

$t_5$	$t_4$	$t_3$	$t_2$	$t_1$	$t_0$	CLASS
1	0	0	0	0	0	Positive
1	1	0	0	0	0	Positive
0	0	0	1	1	1	Positive
0	0	0	0	0	1	Positive
0	0	0	0	0	0	Negative
1	1	1	1	1	1	Negative

Figure 2: Examples of Binary Pattern

ceptable forecasts(4). Song and Pinto(5) evolved programs to detect motion on live videos. GP was used to evolve programs to recognize interesting motions from background and uninteresting motions based on pixel values over a sequence of video frames.

Some researchers have investigated hybrid methods. Hetland and Sætrum presented a new algorithm combining GP and a pattern matching chip to discover temporal rules(6). The outcome of this experiment was comparable to some existing work. Another hybrid method is liquid state genetic programming proposed by Oltean(7). The core idea is dividing the whole system into two parts, the dynamic memory component and GP component. The former is kept by the liquid state machine while GP acts as a problem solver. These hybrid methods are computationally expensive because they require large memory and have long run times. It should be noted that the GP component here does not handle temporal rules directly.

As the aim of our work is to provide a GP-based method which does not require any extra components, we will take raw time series data as input and learn to recognize different patterns.

## 3 Time Series Patterns

This section presents a collection of tasks investigated in this study. They are a group of artificial problems which are to represent tasks increasing level of difficulties: sequence of binary numbers, integers, floating point numbers, linear functions and periodic functions. Real-world applications will be studied in our future work.

### 3.1 Binary Patterns

This is the simplest problem where all the data points are either 0 or 1. One input sequence consists of six time-units worth of data, from  $t_0$  the current value, to  $t_5$  the value recorded 5 time-units ago. There are two types of patterns to be separated here. Negative means no change occurred during a period of six time-units. Positive means there is a change either from 0 to 1, or from 1 to 0 at any time within that period.

Note, in a positive sample, the point of change is not important, because detecting a change should not rely on a particular sampling position. For example, using one second as the time unit, a change occurring at 100th second should be captured by a 6-unit sampling window at multiple positions, from the 101st second to the 105th second. The direction of such change (either increase or decrease) is also not important. Multiple changes within one period such as 001100 are not considered here. Some examples are illustrated in Figure 2.

$t_5$	$t_4$	$t_3$	$t_2$	$t_1$	$t_0$	CLASS
5	5	6	6	6	6	Positive
7	100	100	100	100	100	Positive
1000	1000	1000	1000	100	100	Positive
6	6	6	6	6	6	Negative
100	100	100	100	100	100	Negative

Figure 3: Examples of Binary Pattern

$t_1$	$t_0$	CLASS
-12	-11.49	Positive
5.4	8.6	Positive
-5.63	-5.947	Negative
2233.2	2233	Negative

Figure 4: Examples of Floating Point Pattern (Threshold = 0.5)

### 3.2 Integer Patterns

The task here is very similar to the binary pattern, but the data points are integers with no restriction on the value. The length of a window is again 6 time-units. Any single change in values is considered as Positive while Negative means no changes. The total numbers of possible negatives and positives are enormous. Therefore a generalized rule to differentiate these two patterns is highly desirable. Examples are shown in Figures 3.

### 3.3 Floating-point Numbers with Threshold

The data points here are floating point numbers. Additionally a threshold is introduced. In real world applications, values which are close enough are often considered identical. Ignoring minor differences would be an advantage under this kind of circumstances. A hyper-sensitive detector would be equally bad as an insensitive one if not worse. Therefore data points with variations below a threshold are considered negatives. Otherwise they are positives.

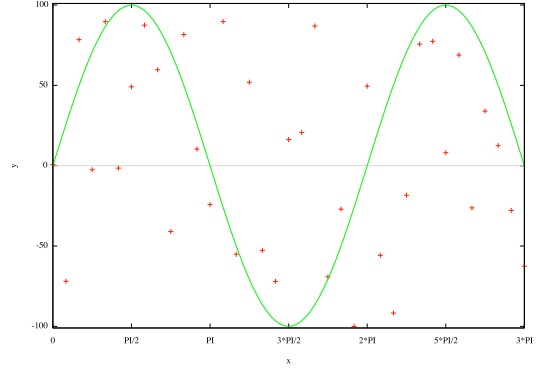
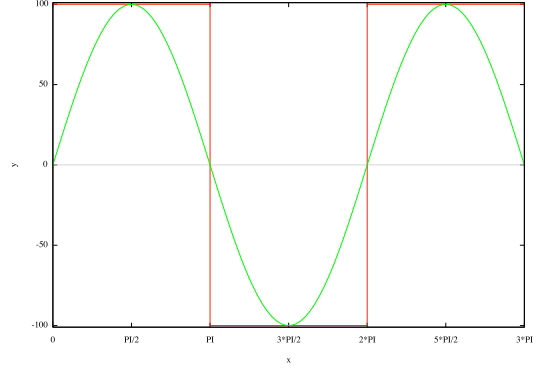
Two types of tasks are studied. The simple version uses a window of length 2. Variations below 0.5 are considered no change. The other version uses a window of length 6, which is the same as the one for binary and integer patterns. The threshold here is bigger as well, which is 5. Examples are shown in Figures 4 and 5.

### 3.4 Sine Waves and Random Numbers

In many real world scenarios no changes does not necessarily mean a constant value. Regular variations can be considered normal as well, for example the electric charge of alternating current. Under such circumstances, simply finding the existence of variation is not enough. Here a sine wave with an amplitude of 100 is used to generate negative samples, while posi-

$t_5$	$t_4$	$t_3$	$t_2$	$t_1$	$t_0$	CLASS
7	1000	239	1000	43.9372	1000	Positive
4	9.21	4.3	6.23	5.4	7.32	Positive
1	-0.3	0.94	2.953	0.32	2.04	Negative
2232.2	2233	2231	2232	2231.3	2333	Negative

Figure 5: Examples of Floating Point Pattern (Threshold = 5)


 Figure 6: A Sine Wave:  $y = 100 \times \sin(x)$  and A Random Sequence

 Figure 7: A Sine Wave:  $y = 100 \times \sin(x)$  and A Step Function

tive samples are randomly generated numbers in the range of  $[-100, 100]$ .

These data points can be visualized in Figure 6. For sine waves, values are taken at intervals of 15 degrees. To enable the learning process to capture the characteristics of a sine wave,  $3\pi$  e.g. one and a half periods of data are included. This means that each sample contains 37 consecutive points sampled along the time line. This is much bigger than that in the previous tasks.

Note that only one sine wave is shown in Figure 6. Sine waves could start from different phases. Therefore negative samples consist of a collection of sine waves with 15 degree shift. So all the negative samples are different. A good model should consider them as the same class, but report random sequences as anomalies.

### 3.5 Sine Waves and Other Periodical Functions

The previous task might be not challenging enough as the random sequence has no regularities at all while the sine waves do. This could provide hints for a learning process. So other periodical functions are introduced as positives here. They are shown in Figures 7 and 8. The first is a step function which has oscillating values from 100 to -100. The second is a triangle wave of which the value varies from 100 to -100 as well. Furthermore all these functions have an identical frequency. Samples of both negatives and positives are taken with 15 degree shifts. Hence all samples for the step function and the triangle function are different.

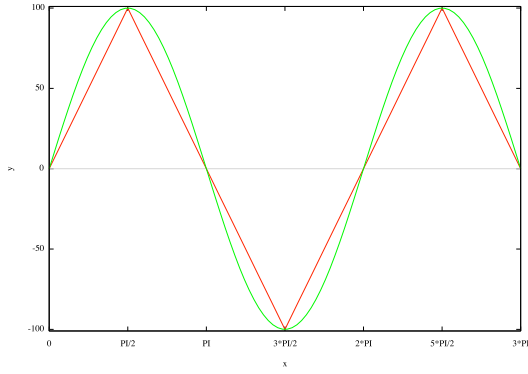


Figure 8: A Sine Wave:  $y = 100 \times \sin(x)$  and A Triangle Function

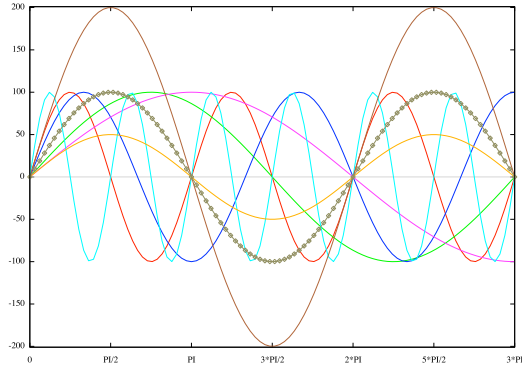


Figure 9: Target Sine Wave:  $y = 100 \times \sin(x)$  (in dotted curve) and Other Sine Waves

### 3.6 Different Sine Waves

To make the task even more challenging, different sine waves are mixed in this dataset. As shown in Figure 9, one particular sine wave is set as the target while other Sine waves with different frequencies and amplitudes are marked as negatives.

### 3.7 Patterns with Noise

Often signals in application would contain noise. To investigate how noisy data are handled by different learning methods, we add random noise to the patterns described in Section 3.5 and Section 3.6. The range of noise is in between  $[-1, 1]$ .

## 4 Methodology and Experiments

The GP method is briefly described in this section. Table 1 shows the function set we used. In addition to basic arithmetic operators, conditional operators are included to perform value comparison. The terminal set simply contains the input variables and random constants.

Table 3 shows the runtime parameters of our experiments. One objective of this study is to obtain solutions which are human comprehensible, so we could understand the learned models. Therefore a relatively small tree depth is used. Furthermore the population size is rather small because we aim to use as few evaluations as possible.

For comparison purposes a number of classical classifiers were used for all the tasks described above. OneR is the simplest classifier which builds rules based on one attribute(10). IBk is an instance based

Table 1: GP Function Set

Function	Return Type	Arguments
+	Double	Double, Double
-	Double	Double, Double
×	Double	Double, Double
/	Double	Double, Double
if	Double	Boolean, Double, Double
>	Boolean	Double, Double
<	Boolean	Double, Double

Table 3: GP Runtime Parameters

Maximum Depth of Program	10
Minimum Depth of Program	2
Number of Generations	100
Population Size	10
Mutation Rate	5%
Crossover Rate	85%
Elitism Rate	10%

algorithm which classifies the target according to its closest neighbour in feature space(13). The  $k$  value is 1 in all experiments. NaiveBayes is a probability based classification method(12). J48 generates decision trees based on the information gain of each attribute(11). Instead of using one classifier, multiple classifiers could be combined as an ensemble to improve the performance. Therefore AdaBoost was also used (14). For each task, the best performer, either OneR, or J48, or NaiveBayes or IB1, is selected as the base classifier in AdaBoost.

For each task the same set of examples are supplied to GP and to other methods for training and test. All data sets include both positive and negative cases. The number of both cases for each task is listed in Table 2. Two thirds of data were for training and one third for test. Table 4 lists the test accuracies achieved by all these methods on various tasks, numbered from No.1 to No.11. Each row in the table represents the results obtained by different methods for one particular task. GP solutions were evolved at least ten times. The test accuracies under GP are results from the best individuals.

As shown in Table 4, GP consistently outperformed the classical methods. There were only two cases that these methods could match GP: AdaBoost for binary patterns and instance-based learning (IB1) for differentiating sine wave and sequences of random numbers. All these methods performed poorly on handling floating-point numbers especially when there are 6 consecutive values (No. 4), while GP still achieved reasonably high accuracy. These methods also performed poorly on another rather difficult task, distinguishing different sine waves (No. 8) while GP achieved 100% accuracy. Even after adding noise to patterns, GP was still able to achieve better results compared to these classical classifiers.

## 5 Discussion

Most of the GP runs terminated around the 30th to 50th generations because a perfect solution was found. This suggests that the representation described earlier is appropriate for recognizing these patterns, so solutions could be found quickly.

To understand the behavior of evolved programs we examined some of the best individuals. Although most of these programs are not quite comprehensible, such analysis does provide some insights. For the simple version of floating-point numbers (No.3, 2 units,

Table 2: Number of Positive and Negative Instances for Each Task

	Positive Instances	Negative Instances
1. Binary Pattern	10	2
2. Integer Pattern	52	10
3. Floating-Point (2 Units)	37	25
4. Floating-Point (6 Units)	37	25
5. Sine Wave vs. Random Numbers	101	24
6. Sine Wave vs. Step Function	127	24
7. Sine Wave vs. Triangle Wave	144	24
8. Different Sine Waves	134	24
9. Sine Wave vs. Step Function(With Noise)	127	24
10. Sine Wave vs. Triangle Wave(With Noise)	144	24
11. Different Sine Waves(With Noise)	134	24

Table 4: Test Accuracies in Percentages(%)

	OneR	J48	NBayes	IB1	AdaBoost	GP
1. Binary Pattern	83.3	83.3	83.3	83.3	100	100
2. Integer Pattern	85.71	85.71	85.71	90.48	90.48	100
3. Floating-Point (2 Units)	76.19	61.9	57.14	66.67	76.19	100
4. Floating-Point (6 Units)	69.23	61.54	60.97	53.85	53.85	92.68
5. Sine Wave vs. Random Numbers	86.05	79.07	81.4	100	95.35	100
6. Sine Wave vs. Step Function	88.68	88.68	50.94	92.45	92.45	100
7. Sine Wave vs. Triangle Wave	86.2	81.03	56.9	89.66	89.66	100
8. Different Sine Waves	11.76	41.18	43.53	78.82	82.35	100
9. Sine Wave vs. Step Function(With Noise)	52.83	88.68	50.94	92.45	92.45	98.11
10. Sine Wave vs. Triangle Wave(With Noise)	79.31	87.93	56.9	89.66	89.66	94.34
11. Different Sine Waves(With Noise)	17.65	40	43.53	78.82	82.35	92.94

threshold 0.5), one program behaves like this:

$$(t_1 - t_0 < 1)? \text{Negative} : \text{Positive} \quad (1)$$

The decision is simply based on the difference between the two values. For separating sine waves and random numbers (No. 5), one best program is effectively

$$(t_4 + t_{16} == 0)? \text{Negative} : \text{Positive} \quad (2)$$

The distance between  $t_4$  and  $t_{16}$  is exactly  $\pi$ , half of the period. Therefore the sum of these two values on a sine wave should be always zero regardless the phase of the wave. This suggests that the evolved GP program did capture a defining characteristic of the periodic function.

One might argue that given appropriate features such as calculating difference between consecutive points, finding the frequency by Fourier transform and so on, the other methods could perform equally well. Certainly such processes would be helpful for learning. However as discussed before, such a process requires domain knowledge from human experts who understand the problem itself. Automatically generating optimal features for the task in hand is often difficult. Additionally it can not be generalized for other problems. There is no universal feature set which is suitable for all kinds of patterns. GP combines the feature finding and classification process together.

## 6 Conclusion and Future Work

A Genetic Programming based method is presented in this study for learning time series patterns. Eleven groups of patterns with increasing difficulties were used to evaluate this GP method. In comparison with five well known machine learning methods: a rule based classifier, a decision tree classifier, a Naive Bayes classifier, an instance based classifier and Ada Boosting, the evolved programs achieved perfect accuracies on most of the tasks and consistently outperformed the other classifiers. We conclude that the

presented GP method is suitable for learning time series patterns. This method has clear advantages, as it is capable of finding characteristics directly on raw input to differentiate various patterns rather than on manually defined features. No extra process is required by this method. Additionally it is capable of handling noisy data input.

Our future work will go beyond a single variable because in many scenarios such as climate change and video analysis, one must be able to handle multiple variables which may or may not be independent to each other. Another extension is treating monotonicity as a pattern, so a “normal” variable should always be stable or change in one direction and never oscillate. Mixtures of multiple patterns is another area to explore, such as a step function on top of a sine wave. This method will be applied on real world applications in the near future.

## References

- [1] Sutton, R.S.: Learning to predict by the methods of temporal differences, Machine learning, vol. 3, pp.9–44 (1988)
- [2] Koza, J.R.: Genetic programming I: On the programming of computers by means of natural selection, vol. 1, MIT press(1996)
- [3] Byron B. and Geoffrey J. G.: Predictive State Temporal Difference Learning, Proceedings of Advances in Neural Information Processing Systems 24(2010)
- [4] M. A. Kaboudan: Spatiotemporal Forecasting of Housing Price By Use of Genetic Programming, the 16th Annual Meeting of the Association of Global Business, 2004
- [5] Andy S. and Brian P.: Study of GP representations for motion detection with unstable back-

ground, IEEE Congress on Evolutionary Computation (2010)

- [6] Magnus L. H. and Pal S.: Temporal Rule Discovery using Genetic Programming and Specialized Hardware, Proceedings of the 4th International Conference on Recent Advances in Soft Computing (2002)
- [7] Oltean, Mihai: Liquid State Genetic Programming, Proceedings of the 8th international conference on Adaptive and Natural Computing Algorithms (2007)
- [8] Georg D.: Neural Networks for Time Series Processing, vol. 6, pp.447-468, Neural Network World, 1996
- [9] Aranildo R. L. J. and Tiago A. E. F.: A hybrid method for tuning neural network for time series forecasting, pp.531-532, Proceedings of the 10th annual conference on Genetic and evolutionary computation, 2008
- [10] R.C. Holte: Very simple classification rules perform well on most commonly used datasets, vol. 11, pp.63-91, Machine Learning, 1993
- [11] Ross Quinlan: C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers, 1993
- [12] George H. J, Pat L.: Estimating Continuous Distributions in Bayesian Classifiers, Eleventh Conference on Uncertainty in Artificial Intelligence, pp.338-345, 1995
- [13] D. Aha, D. Kibler: Instance-based learning algorithms, vol. 6, pp.37-66, Machine Learning 1991
- [14] Yoav Freund, Robert E. S.: Experiments with a new boosting algorithm, pp.148-156, Thirteenth International Conference on Machine Learning, 1996



# ERA Challenges for Australian University ICT

**Paul A. Bailes**

School of Information Technology and Electrical Engineering The University of Queensland QLD 4072 Australia

Chair, Computer Systems and Software Engineering Technical Board, Australian Computer Society

paul@itee.uq.edu.au

## Abstract

The ERA 2010 rankings reflect badly on Australian university ICT performance, especially by comparison with some likely benchmark disciplines, for reasons not exclusively under the control of ICT researchers, groups or schools. As a result nevertheless, the future of Australian academic ICT is under threat of reduced government funding, from diminished international reputation and from university reaction. Changes to ERA for 2012 may ameliorate the position somewhat, but continued complementary and concerted action by individuals, institutions and professional organisations such as ACS, CORE and others remains imperative.

**Keywords:** ACS, ARC, CORE, ERA, NICTA, RAE, REF, RQF

## 1 Introduction

The purpose of this paper is to raise awareness among the Australian Computer Science academic and research community about the threats posed to Australia's higher education capabilities in ICT research and teaching by the Excellence in Research for Australia (ERA) scheme and especially the results of the 2010 exercise. In particular, we consider how these threats may conceivably originate in ERA's and universities' treatment of ICT, and what kind of cooperative action could lead to win-win outcomes for all concerned.

It is not our purpose to rehash or to engage in the controversies surrounding ERA, but some history is useful. ERA may be thought to have its origins in the UK Research Assessment Exercise (RAE) and its proposed successor Research Excellence Framework (REF) as currently documented at the REF website <http://www.hefce.ac.uk/Research/ref/>. RAE has been conducted approximately 5-yearly since 1986, and REF has been envisaged to commence in 2014. Academic disciplines at UK universities have been ranked on a number of measures, primarily by the outputs of continuing academics, and the published rankings impact variously upon government funding and institutional reputation. Some of the improvements reflected by REF can be identified in ERA, including the use of

bibliometric quality indicators.

Australia's ERA was preceded by the abortive Research Quality Framework (RQF), and (like REF over RAE in the UK) ERA promised more accurate measures of research quality, especially metrics (Department of Innovation, Industry, Science and Research, 2008). Implementation of ERA in 2010 was preceded by a trial in 2009 involving research grouped into two "discipline clusters": Physical and Chemical Sciences; and Humanities and Creative Arts.

## 2 ERA 2010 Processes

The full ERA was consequently conducted during 2010 according to the following broad parameters, sourced from ARC's ERA 2010 Evaluation Guidelines (Australian Research Council, 2010).

### 2.1 Terms of assessment

A wide range of objects of assessment (see below) were assessed across eight discipline clusters (Cluster Five—Mathematical, Information and Computing Sciences being most relevant to ICT as we shall see). Within each cluster, research was classified and assessed in terms of "disciplines" as per the four-digit and two-digit Fields of Research (FoRs) as identified in the Australian and New Zealand Standard Research Classification (ANZSRC) (Australian Bureau of Statistics, 2008). The "four digit" disciplines represent specialisations of the "two digit" disciplines; in particular the latter subsumed the former and additionally material that did not meet quantity thresholds for assessment under the former was nevertheless eligible for assessment under the latter. The two-digit FoR code 08 Information and Computing Sciences was most applicable to ICT, though several others were more or less applicable. The following lists the most relevant:

- 08 Information and Computing Sciences
  - 0801 Artificial Intelligence and Image Processing
  - 0802 Computation Theory and Mathematics
  - 0803 Computer Software
  - 0804 Data Format
  - 0805 Distributed Computing
  - 0806 Information Systems
  - 0807 Library and Information Studies
  - 0899 Other Information and Computing Sciences
- 10 Technology
  - 1005 Communications Technologies
  - 1006 Computer Hardware

Some ICT research may also have been classified under 0906 Electrical and Electronic Engineering.

---

Copyright © 2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122. M. Reynolds and B. Thomas, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

It must be noted that it was outputs that were classified not researchers; and some outputs from ICT researchers would have been classified under other codes, especially interdisciplinary work where ICT was an enabling technology. This was because outlets (not publications) were assigned FoR codes, so that an essentially computer science publication in an application domain journal or conference would have been recognised under the FoR code(s) of the application domain, not 08 etc.

Each university's submission under the above framework (at the two- and four-digit levels) constituted a "Unit of Evaluation" (UoE) against which a university's performance was assessed and published. In order however to be assessed, a unit's output had to meet a threshold over the six-year period of evaluation – for all of the ICT disciplines listed above, (both two- and four-digit codes) this was 50 articles in journals (*not* conferences) recognised under the ranking scheme (see below) over the period of assessment (see below). Thus, while some four-digit UoEs might not have been assessed for not meeting the threshold, the research outputs were still eligible for consideration under the covering two-digit unit.

## 2.2 Objects of assessment

Even though ranked journal publication outputs provided the exclusive qualification for meeting thresholds, a rich set of objects were considered when assessing research performance in a UoE.

- Research Outputs (primarily books/chapters, journal articles, conference papers in selected disciplines including 08 Information and Computing Sciences) for a six-year period: 1 January 2003–31 December 2008
- Research Income (in terms of HERDC categories: Australian Competitive Grants; other public sector research income; CRC income) for a three-year period: 1 January 2006–31 December 2008
- Applied Measures (research commercialisation income, patents, registered designs, plant breeder's rights and NHMRC endorsed guidelines) for a three-year period: 1 January 2006–31 December 2008
- Esteem Measures (fellowships, board/committee memberships) for a three-year period: 1 January 2006–31 December 2008.

## 2.3 Subjects of assessment

Even though assessments of research outputs were not organised around individual producers of research, a credible affiliation with a university for the period of assessment needed to be demonstrated for an individual's research (objects as summarised above) to be considered:

- on the census date of 31 March 2009, to have been a paid employee or in some other relationship, including as a visiting academic; and
- if not a paid employee, to have that affiliation substantiated by a publication association.

A researcher's affiliation on the census date determined the university to which credit for the objects of assessment was allocated; but for non-paid employees, only publications that explicitly cited that university were included in the assessment.

## 2.4 Criteria of research output assessment

Research outputs (publications) were included for assessment only if in recognised outlets. Recognised outlets were ranked according to criteria as follows:

A\* (journals only): one of the best in its field or subfield; typically covering the entire field/subfield; all or nearly all papers will be of a very high quality where most of the work shapes the field; acceptance rates will typically be low and the editorial board will be dominated by field leaders, including many from top institutions (leaving aside the question of circularity raised by the last point).

A: Tier A journal (or conference) will mostly be of very high quality and would enhance the author's standing, showing they are engaged with the global research community; journal acceptance rates will be lowish and editorial boards will include a reasonable fraction of well known researchers from top institutions; conference acceptance rates will be low and program committee and speaker lists will include a reasonable fraction of well known researchers from top institutions, with a high level of scrutiny by the program committee to discern significance of submissions.

B: Tier B journals (or conferences) will have a solid, though not outstanding, reputation; for journals, only a few papers of very high quality would be expected; they are often important outlets for the work of PhD students and early career researchers; journals are further typified by regional outlets with high acceptance rates, and editorial boards that have few leading researchers from top international institutions; conferences are typically regional conferences or international conferences with high acceptance rates.

C: Tier C includes quality, peer reviewed, journals (or conferences) that do not meet the criteria of the higher tiers; but are nevertheless worthy of consideration in ERA. Outputs that do not achieve a C ranking are not considered at all.

As well as the ranking of the outlet, publications were assessed by citation count. Verbal testimony to the author was that citations took priority over journal rankings – one may speculate that for more recent publications where citations might as yet be unlikely, outlet rankings served as a proxy.

While the ranking scheme commands admiration as an open and accountable basis for assessing the quality of research outputs, and importantly as a target for institutional- and self-improvement, it has attracted fair criticism for the errors and omissions that risk inducing distorted behaviours by academics and institutions seeking to maximise their ERA outcomes. In her report of the demise of rankings for ERA 2012 (see further discussion below), Rowbotham (2011) gives some typical examples of these.

## 2.5 Results of assessment

Results for each UoE were expressed in a six-point rating scale.

5 : The UoE profile is characterised by evidence of outstanding performance well above world standard presented by the suite of indicators used for evaluation.

4 : The UoE profile is characterised by evidence of

performance above world standard presented by the suite of indicators used for evaluation.

3 : The UoE profile is characterised by evidence of average performance at world standard presented by the suite of indicators used for evaluation.

2 : The UoE profile is characterised by evidence of performance below world standard presented by the suite of indicators used for evaluation.

1 : The UoE profile is characterised by evidence of performance well below world standard presented by the suite of indicators used for evaluation.

n/a: Not assessed due to low volume. The number of research outputs did not meet the volume threshold standard for evaluation in ERA.

Assessments were undertaken for each four- and two-digit disciplinary (FoR code) unit for which the output threshold (50 journals) was achieved.

### 3 ERA 2010 Outcomes for ICT

Outcomes for Australian university ICT in ERA 2010 were not without successes, but it is arguable that the overall performance was less than desirable. The source for all our data is the ERA 2010 outcomes summary published by *The Australian* (Hare, 2011).

#### 3.1 ICT research performance

Raw data of ICT research scores in 2010 ERA is summarised in Table 1. It includes the results that contributed to Cluster Five - Mathematical, Information and Computing Sciences, i.e. FoR code 08 (Information and Computing Sciences) and its sub-disciplines, and the subdisciplines of FoR code 10 (Technology) most applicable to ICT.

#### 3.2 Benchmark analysis of ICT research performance

Because of the evident dominance of FoR code 08 Information and Computing Sciences (ICS) and its constituent (sub-) disciplines in evaluated (i.e., above-threshold) ICT research, these data will be the focus of our comparison between ICT other disciplines. Table 2 compares overall ERA outcomes in ICS (2-digit level 08) with those of some benchmarks in science and engineering with which ICT is likely to be compared by interested parties.

The message sent by table 2 is that as measures of increasing comparable quality are taken into account, ICT research performs increasingly badly against its likely benchmarks:

- Generally somewhat fewer universities managed to meet ERA performance thresholds in ICT
- The discrepancy becomes ever more-pronounced in more specialised areas (4-digit FoR codes)
- The discrepancy is likewise more-pronounced as higher levels of achievement against world benchmarks are reflected. Thus: while ICT research is at least of world standard at approximately 66% of the number of universities at which science/engineering is, when we progress to a level of at least above world standard, the relative percentage for ICT drops to 50%; and when we progress to a level of well above world standard, the

relative percentage drops to close to 5%.

- This impression of ICT's relative poor performance to likely benchmarks is corroborated by the comparison of ICT scores with university averages: ICT research scored at or above average results at approximately only 20% of universities, whereas science/engineering performed at or above average results in approximately 45% of universities. It is clear that in many cases, poor ICT scores contributed to poorer-than-otherwise university outcomes. Moreover, in the relatively few cases where ICT performed at or above average, that was largely in the context of overall poor performance by the relevant university. There were only three at-or-above-average results for ICT where the university as a whole performed at or above world standard.

#### 3.3 Bases for Poor ERA Outcomes for ICT

A number of factors may have combined to give these disappointing results.

- The relative fine granularity of FoR codes pertaining to the ICS discipline would generally have diminished the relative performance of ICS compared to other broad disciplines. For example, the entire field of Engineering was also covered by a single two-digit code, whereas substantial sub-disciplines (e.g. Chemical, Civil, Electrical Engineering) had only four-digit codes. Further, this fine granularity for ICS would have militated against the achievement of thresholds especially at four-digit level, though failure to register at four-digit level did not necessarily preclude a good result, e.g. RMIT achieved a two-digit score of 3 without having met any four-digit thresholds.
- ICT is particularly at risk of being submerged into other disciplines, especially in view of its role as a fundamental enabling technology for contemporary scholarship in life sciences and the humanities, and not just in the engineering and physical sciences. Because FoR codes were assigned to outlets (journals or conferences) rather than specific publications, publishing a computer science breakthrough in its application context would have led to no recognition of the publication as a computer science contribution (08 FoR-coded). For example, a data mining breakthrough published in a life sciences journal would have been recorded under FoR codes 03, 05 or 06. Moreover as noted, some ICT fields were classified under "Engineering" or "Technology" (communications or hardware).
- The citation coverage service used by the 2010 ERA (Scopus) is frequently claimed not to cover ICT well, nor conferences.
- The exclusion of conferences from threshold counts is inconsistent with including conferences in the overall ICT research outputs.
- Research performance was measured against international benchmarks of university quality and quantity output, and was not pro-rated for small units (viz. the firm threshold of 50 ranked journal articles).

	O'all Avg.*	08	0801	0802	0803	0804	0805	0806	0807	0899	1005	1006
ACU	1.91	-	-	-	-	-	-	-	-	-	-	-
ANU	4.38	5	5	3	-	-	-	-	-	-	-	-
Batchelor	-	-	-	-	-	-	-	-	-	-	-	-
Bond	1.91	-	-	-	-	-	-	-	-	-	-	-
CQU	1.54	-	-	-	-	-	-	-	-	-	-	-
CDU	2.33	-	-	-	-	-	-	-	-	-	-	-
CSU	1.88	1	-	-	-	-	-	-	-	2	-	-
Curtin	2.50	2	-	-	-	-	-	-	-	-	-	-
Deakin	2.41	1	-	-	-	-	-	-	-	-	-	-
ECU	2.06	2	-	-	-	-	-	-	-	-	-	-
Flinders	2.44	-	-	-	-	-	-	-	-	-	-	-
Griffith	3.00	3	3	-	-	-	-	-	-	-	-	-
JCU	2.47	-	-	-	-	-	-	-	-	-	-	-
Latrobe	2.62	1	-	-	-	-	-	-	-	-	-	-
MQ	3.24	3	-	3	-	-	-	-	-	-	-	-
MCD	3.00	-	-	-	-	-	-	-	-	-	-	-
Monash	3.45	3	3	-	-	-	-	3	-	-	-	-
Murdoch	2.65	-	-	-	-	-	-	-	-	-	-	-
QUT	3.09	4	-	-	-	-	-	5	4	-	-	-
RMIT	2.61	3	-	-	-	-	-	-	-	-	-	-
SCU	1.85	-	-	-	-	-	-	-	-	-	-	-
Swinburne	2.24	3	-	-	-	-	-	-	-	-	-	-
U.Adelaide	3.55	3	4	-	-	-	-	-	-	-	-	-
U.Ballarat	1.56	-	-	-	-	-	-	-	-	-	-	-
U.Canberra	2.14	-	-	-	-	-	-	-	-	-	-	-
U.Melb	4.33	4	3	-	-	-	-	4	-	-	5	-
UNE	2.31	-	-	-	-	-	-	-	-	-	-	-
UNSW	4.04	4	3	-	4	-	4	3	4	-	-	-
U.N'cle.	2.71	2	-	3	-	-	-	-	-	-	-	-
U.ND	1.33	-	-	-	-	-	-	-	-	-	-	-
UQ	4.17	3	3	-	-	-	-	4	-	-	-	-
UniSA	2.61	2	-	-	-	-	-	-	-	-	-	-
USQ	2.00	-	-	-	-	-	-	-	-	-	-	-
U.Sydney	3.83	4	4	-	-	-	-	-	-	-	3	-
UTas	2.81	-	-	-	-	-	-	-	-	-	-	-
UTS	2.95	3	2	-	-	-	-	3	-	-	-	-
USC	1.44	-	-	-	-	-	-	-	-	-	-	-
UWA	3.64	3	-	-	-	-	-	-	-	-	-	-
UWS	2.48	1	-	-	-	-	-	-	-	-	-	-
U.W'gong	2.71	2	-	-	-	-	-	-	-	-	-	-
VU	1.71	2	-	-	-	-	-	-	-	-	-	-

\* "O'all Avg." refers to the overall institutional average.

**Table 1: ERA 2010 results for ICT disciplines**

	ICS	Biology	Engineering	Maths	Physics
scored at all (2-digit FoR at least)	24	34	31	25	24
also scored one or more 4-digit FoR	13	27	24	21	19
also scored two or more 4-digit FoR	8	24	19	17	15
also scored three or more 4-digit FoR	2	21	17	11	12
scored at or above world std (score 3 or greater)	14	23	22	18	20
scored above world std (score 4 or greater)	5	11	9	8	12
scored well above world std (score 5)	1	8	3	2	8
scored at or above uni. average	8	21	16	16	18

**Table 2: ERA 2010 outcomes for ICT disciplines c.f. other science and engineering**

- f) The significant decline in Australian university ICT academic staff numbers since 2000 must therefore have impacted significantly on the evaluation, as successive rounds of redundancies have seen the departure of numerous productive research personnel, recalling that individuals needed to have a demonstrable association with a university by the 31 March 2009 census date for their research output for the 2003-2008 period to be attributed to that university. Some university ICT organisational units lost of the order of 50% of their academic staff- at that scale of retrenchment, it is difficult not to lose a number of research-capable staff, and anecdotal evidence supports the contention that a significant number of high-achievers took the opportunity to take effective early retirement.
- g) The significant national investment in university-derived ICT research represented by NICTA goes unrecognised because NICTA-funded staff are employed directly by NICTA rather than by universities funded from NICTA. This contradicts the normal pattern of Australian research funding.

While the past downsizing is beyond the control of the ARC, the above catalogue (items a-e) suggests that the current ERA system includes pitfalls that need to be taken into account when attempting to modify organisational and individual behaviours to optimise future ERA outcomes.

## 4 Impact of Poor ERA Outcomes for ICT

These poor outcomes place university ICT at risk in various ways.

### 4.1 Funding threats

ERA is part of the wider Sustainable Research Excellence in Universities (SRE) initiative which aims to compensate for the gap in funding for the indirect costs of university research, including hitherto uncoded items such as proportions of academic staff salaries, and more realistic costing of technical and administrative research infrastructure (for broad information see the SRE Website <http://www.innovation.gov.au/Research/ResearchBlockGrants/Pages/SustainableResearchExcellence.aspx>).

Following a trial in 2010 for funding allocation in 2011, SRE funding will in future be contingent upon ERA performance: approximately 67% of the funding available under the scheme ("Threshold 2", worth approx. \$81M across the sector in 2011) is at stake in future (Department of Innovation, Industry, Science and Research, 2011). SRE extends the existing RIBG (Research Infrastructure Block Grants) and it is not inconceivable that RIBG may be rolled into SRE in future.

\$81M distributed across 41 universities on the basis of ERA outcomes is barely significant and probably approaches the overall university- and government-side costs of administering the exercise. However, now that

the principle of “excellence”-based distribution of public funds has been established, influential forces will be at work to increase its impact in future. It can be assumed that whatever revenue is earned by universities as a result of ERA will largely return to the research-successful disciplines that “earned” same.

## 4.2 Reputational threats

While direct ERA-based public funding remains relatively insignificant, the early impact of ERA on other sources of funding and other enablers of university effectiveness could be significant.

- In the absence of a national teaching assessment, ERA is likely to serve as a proxy for overall rankings and thus become critical to Australian universities’ profiles as destinations for international fee-paying students at all levels. For example, <http://www.australian-universities.com/rankings/> lists a number of rankings of Australian universities, with ERA-derived rankings prominently displayed (and by virtue of the level of detail supplied, apparently most authoritative as well). In view of the dependence of Australian universities on international student numbers (some over 50% - see [http://www.students.idp.com/study/australian\\_universities.aspx](http://www.students.idp.com/study/australian_universities.aspx)), the budget impact of same being heightened by the excess of international student fees charged by many universities over government subsidies for domestic students.
- In view of the emphasis placed on research achievement in academics’ career development and prospects, the poor research rating of Australian university ICT will act a strong disincentive for ICT academics to consider Australian universities: top international researchers will be less likely to consider Australia; and top Australian researchers will be tempted increasingly to pursue their careers overseas. The threat to Australian ICT research capacity is obvious, but the threat also applies to the quality of ICT education: while the research-teaching nexus may in some cases be exaggerated, it is undoubtable that a significant number of advanced-level Australian ICT courses benefit from being taught by active researchers in the relevant fields; and while there may be some truth to the stereotype of the brilliant but inarticulate researcher, it is often the case that excellent academics excel in both dimensions of endeavour – teaching as well as research. Any threat to the attraction and retention of excellent ICT researchers in Australian universities is a real threat to Australian ICT education.
- In similar vein, any detraction from the appeal of Australian universities to prospective research students (masters and PhD) will further detract from the appeal of Australian universities to research academics and will in itself substantially impact upon Australian universities’ research capacity. In view that the global market for PhD students is becoming one in which top students are awarded fee waivers or equivalent scholarships, these threats are probably more significant in the long run than any impact on fee income from this class of student.

## 4.3 Internal threats

The financial and reputational pressures upon universities (see above) to improve their ERA scores could conceivably have beneficial impact upon ICT, as universities seek to remedy deficiencies made apparent by ERA.

On the other hand, one may not unreasonably fear that universities may be tempted consider other options to improve their ERA scores, not necessarily to ICT’s advantage. For example, a reasonable strategy that might be adopted would be for a university to invest in areas that have demonstrated their potential to perform by relatively good ERA results, but which have room for improvement. Under such a scenario, below-average performances in the ICS disciplines might very well not meet universities’ criteria for development investment. Indeed, the temptation to remove resources from “losers” in order to maximise the further prospects of “winners” might see a catastrophic decline in ICT’s position in a number of universities.

Moreover, for universities at which ICT was unrated (over 40%!), a choice decision now confronts them. Any effort to put ICT “on the map” runs a considerable risk in that it will be difficult for universities to determine with confidence that the result will actually be creditable. An unrated performance is the result of not meeting the publication output threshold, and might be transformable into a rated one by transforming academics’ behaviours to pursue ranked outlets in future. That is however no guarantee that the resulting ERA assessment would be one that the university desires. At the very least, any future below-average ERA result for ICT is one that is likely to attract a university’s disapproval.

Finally, it must be emphasised that it is not just Australia’s ICT research capacity which is under threat in this way. As well as the broad risk to the quality of Australian ICT academic staff, the very existence of ICT as an academic endeavour at some universities may be under question.

## 4.4 A vicious cycle threatens

To summarise, Australian university ICT is threatened by a vicious cycle of poor ERA evaluations leading to reduced resourcing and reputation leading to reduced performance leading to poorer ERA evaluations etc.

## 5 What to Do?

While it may be the case that Australian ICT suffers from systemic deficiencies, it is essential in the first place that the picture revealed by the ERA microscope is an accurate one.

Even though the 2012 ERA exercise has effectively already begun (reference period for publications was six years until 31.12.10; census date for staff is 31.3.11 - see [http://www.arc.gov.au/era/era\\_2012/important\\_dates.htm](http://www.arc.gov.au/era/era_2012/important_dates.htm) for ERA 2012 Important Dates), various stakeholders could still usefully engage in (initiate or maintain) activities that could lead to improved outcomes if not in 2012 then in likely subsequent ERA exercises.

### 5.1 Changes to ERA 2012

First, reactions in word or deed need to be cognizant of



how the ERA rules are evolving.

### 5.1.1 ICT-specific changes

At the ICT disciplinary level, representations led by CORE have resulted in two major developments.

- Conferences will be included in the count of outputs required to meet the publication threshold for a UoE to be evaluated.
- The quality of ICT publications will be measured by peer review rather than citations.

While the removal of the current (2010) debatable basis for citation analyses has to be welcomed, it needs to be recognised that peer review is not the only option. Other analyses such as CiteSeer's (Lawrence et al., 1999) or Google Scholar <http://scholar.google.com.au/> could conceivably be demonstrated to the ARC as having an authority for ICT comparable to Scopus for natural science.

Potentially more risky is the inclusion of conference outputs in UoE thresholds: at present, unranked UoEs at least "fly below the radar" and may well attract less adverse attention than those which are ranked but badly. Clearly some institutions have perceived this change to be advantageous, but it may very well be not universally so. Judgment will however have to be suspended until ERA 2012 results are released and compared to ERA 2010 outcomes.

### 5.1.2 Overall changes

Changes across the entire ERA 2012 process are outlined in the ERA FAQ <http://www.arc.gov.au/era/faq.htm>, especially:

- abolition of the controversial publication rankings (Rowbotham, 2011) in favour of a "refined journal indicator";
- conferences are not *a priori* assigned FoR codes;
- more flexible FoR coding will better reflect interdisciplinary research achievement;
- output thresholds for peer reviewed disciplines will be aligned with citation-analysed disciplines (conveniently for ICT)

Instead of recording UoEs' publications according to outlet rank, outlets will be listed in order of frequency of occurrence of publication in the UoE (absolute and relative), the idea being that assessment panels will be able thereby to discern if the UoE's publication profile represents quality (or not). While it is understandable that the ARC has resiled from the hitherto inflexible ranking scheme, it is not clear that that new system will be without its drawbacks. For example, for UoEs trying to improve their performance (which is what ERA is all about, surely), the ranking system at least provided guidance.

The ARC will continue to maintain a list of admissible journals and their default FoR codes, but conferences will be unclassified (and so the idea of a conferences list becomes somewhat redundant). For ICT, peer review will apparently be the means by which the quality of conferences publications is measured.

The obstacle in ERA 2010 to the reflection of interdisciplinary research achievement (so important for

ICT in its enabling role) was the inflexibility of FoR-coding of publications. Publication of significant ICT research in an application area's journal lead inevitably to that research being classified according to one of the application area FoR codes. Under ERA 2012, following a trial for mathematical sciences in ERA 2010, individual journal articles will be able to be reassigned to FoR codes other than that of the journal in which they appear provided that at least 66% of the paper's content lies in the "new" area. For conferences the lack of *a priori* FoR-coding reinforces that degree of flexibility.

There may also be an increase in the amount of information to be submitted for peer review. For ERA 2010 peer reviews, UoEs were required to nominate 20% of their outputs for submission to the peer review process, as specified in the ERA 2010 Discipline Matrices see - [http://www.arc.gov.au/xls/ERA2010\\_discipline\\_matrices.xls](http://www.arc.gov.au/xls/ERA2010_discipline_matrices.xls). For the ERA 2012 consultation process, a figure of 30% is said to have been proposed but the submission deadline against the Draft ERA 2012 Submission Guidelines and Discipline Matrix has passed (1 August), and "Page not found" is the result of attempting to access [http://arc.gov.au/era/era\\_2012/era\\_2012\\_documents.htm](http://arc.gov.au/era/era_2012/era_2012_documents.htm).

## 5.2 What can ICT researchers and groups do?

For the immediate future (i.e. ERA 2012), there is little that individual researchers can do. For example, the survey period for publication data has long ago closed (31.12.10), as have the various ERA 2012 consultation cutoff dates.

For subsequent ERA-style exercises in the medium/long term however, a number of lines of development suggest themselves for attention; some more useful than others.

### 5.2.1 Focus

It may be tempting for universities especially with relatively small numbers of ICT researchers to improve outcomes, especially at the four-digit FoR level, by concentrating on a very few, maybe even one, research areas. As well as potentially improved ERA scores, this is a means by which the issue of critical mass may be addressed. However there are means by which small numbers of effective researchers can manage to establish effective connections (with PhD students, with collaborators at other institutions, through NICTA). Further, lack of recognition at ERA four-digit level does not appear to exclude a good result at two-digits (e.g. the "world standard" evaluation achieved by RMIT for two-digit ICS without any four-digit result). Thus, imposition of a tighter research focus does not in itself seem to be a priority (as opposed to any which may emerge as a result of differentiation – see below).

### 5.2.2 Select

It would be advantageous for selection of publication outlets needs in future to be much more attuned to ERA requirements (i.e. listed journals and conferences), noting that the journal-only threshold for ICT will be relaxed to include conferences for 2012. For the future, the abolition of explicit ranks makes it impossible to offer counsel about the trade-off between pursuing one outlet vs another. e.g. in terms of chance of acceptance vs. ERA

kudos. Had the journal ranking scheme been retained, it might have been possible at least to have given advice to colleagues based on extremes of quality (e.g. A\* vs. C). It should however be noted that it is not clear how much weight was attached in 2010 to varying kinds of performance, i.e. higher vs lower ranked outlets, ranking vs citations, conferences vs journal (other than meeting the threshold).

Hopefully the refined journal indicators of different institutions' UoEs will be published. As well as giving guidance to the community about what is regarded as quality publication patterns, this would offer an important measure of accountability of assessors' performance.

Other kinds of quality-reflecting behaviours should not be neglected, e.g. competitive grants and industry contracts. Senior staff would do well also to pursue esteem measures such as prestigious fellowships and memberships of boards. The celebrated unsociability of ICT experts will need to be overcome.

### 5.3 What can universities do?

Universities have options other than to wind-down apparent poorly-performing ERA organisational units.

For the longer-term, a number of lines of development suggest themselves for attention in parallel with those that can be undertaken by individuals and research groups.

#### 5.3.1 Differentiate

In the light of possible significant levels of actual research capability among staff, it does not seem wise to maintain an even distribution of teaching load in some cases at relatively high-levels compared to some of the other more successful ERA disciplines.

#### 5.3.2 Invest

Differentiated workloads for staff on the basis of research capability represents a significant HR investment, and would well be matched by other kinds of investment in ICT research capability. In particular, the prevalence in the global market for PhD students of fee waivers and stipends indicates that any perception of (high-quality) PhD students as a significant source of revenue will have diminishing validity. Rather, PhD students should be thought of as relatively high return on investment research personnel, in a sense as amplifiers of their supervisors' capabilities. Needless to say, the more capable the supervisor, the more effective the amplification. Other more expensive HR investments could be considered (such as hiring research "stars"), but this author is of the opinion that quality PhD students are the royal road to research productivity. Fee waivers and living stipends for top-quality PhD students should be greatly encouraged as a general rule across all disciplines.

For ICT specifically however, it is well past time for (some) universities to stop treating ICT as a cash cow to be sent to the slaughterhouse once it's stopped giving cream. During the ICT boom of the 1990s, universities as a whole did very well from the overheads charged on ICT student income (government subsidies and mostly international fees), but during the following decade of downturn ICT organisational units were drastically pruned proportionate to the drop in revenue. This makes

for an interesting contrast with the treatment of some other disciplines the research capability of which was preserved by internal subsidies from universities until their enrolments problems corrected themselves.

### 5.4 What can the ICT community do?

Australian ICT leadership groups (particularly ACDICT, ACS, CORE, ACPHIS and ALIA - see Glossary) have collaborated to an encouraging degree to improve ICT's position for 2012. In particular, the opportunity was taken jointly to advocate a consensus position to the ARC on key problem areas cited above (i.e. of citation analyses, inclusion of conferences in thresholds and flexible treatment of interdisciplinary research outputs). CORE undertook to lead the follow-through with evident success: in each case ERA 2012 will proceed with the ICT community's submissions reflected in the changed procedures also documented above.

It will be important to build upon this unity, and better to include other key organisations (such as AIIA, EA's ITEE College and NICTA), in pursuing some further ERA-related reforms, such as the following:

- while peer reviewing may yet prove to be to the satisfaction of all concerned, the viability of alternative bibliometrics (such as Google Scholar and CiteSeer) should be explored thoroughly;
- NICTA's own research personnel should routinely be found adjunct or honorary appointments in the university labs with which they are associated;
- the new refined journal indicators for each UoE that will replace journal rankings in ERA 2012 should be published for reasons of (i) openness and accountability, and (ii) exemplifying desirable patterns of publication behaviour for future performance improvement across the sector;
- in similar vein, the relative weightings attached by ERA assessment panels to the various objects of evaluation should be explicated.

Not all future community action needs however to be explicitly focussed on ERA processes. In particular, universities need to receive loud and clear messages from the ICT community not to adopt hasty and punitive responses to the flawed assessment of ICT in ERA 2010. In this regard, it will be essential that the lead is taken by organisations other than those which may be thought to have the strongest vested interests, in other words by professional and industrial bodies such as ACS, AIIA, ALIA and EA/ITEE rather than academic groups such as ACDICT, CORE and ACPHIS. More generally, the industry/professional groups need to be in a position to advocate for the continued vitality of the research sector of the Australian ICT scene, which can only be the case if continued closer contact with them is maintained by academe.

## 6 Conclusions

Some process such as ERA is inevitable at this stage in the development of Australian higher education policy; it may well not be the case that ERA will be a long-standing, let alone a permanent feature of the Australian higher education policy landscape. Coming as it does however when Australian university ICT is struggling to



recover from a severe period of retrenchment, it is essential that any misunderstandings of ICT research that it creates and perpetuates do not go unchallenged.

Our main message however is that opportunities for proactive response to the challenges posed by ERA 2010 are both rich and rewarding:

- rich, in the sense that there is a wide range of opportunity for individuals, groups, institutions and the entire ICT community (industry/professional as well as academe);
- rewarding, in the sense that concerted action to date has been evidently fruitful.

In particular, the ICT community needs to organise and communicate better among its constituent parts in order to secure better outcomes for the whole. If there are indeed systemic problems with Australian ICT research, this would likely feature among the means of addressing them. It is as if the notorious unsociability of the stereotypical ICT researcher or professional needs as much remedying at the community level as well as the individual. If the ERA 2010 “lemon” becomes the “lemonade” that helps achieve this, we will jointly have accomplished much more than having scored a victory (however important) in the government funding game.

## 7 Disclaimer

The views expressed in this paper are those of the author alone, and neither of The University of Queensland nor of the Australian Computer Society.

## 8 Acknowledgements

Numerous colleagues in the Australian university and ICT professional community have offered the benefit of their insights on ERA in the development of the above, especially CORE President Prof. Tom Gedeon. Anonymous referees’ suggestions have improved the presentation greatly.

## 9 References

- Australian Bureau of Statistics (2008): Australian and New Zealand Standard Research Classification, <http://www.abs.gov.au/ausstats/abs@.nsf/0/4AE1B46AE2048A28CA2574180004422?opendocument>, accessed 23 August 2011.
- Australian Research Council (2010): ERA 2010 Evaluation Guidelines, [http://www.arc.gov.au/pdf/ERA2010\\_eval\\_guide.pdf](http://www.arc.gov.au/pdf/ERA2010_eval_guide.pdf), accessed 23 August 2011.
- Australian-Universities.com (undated): Rankings of Australian Universities, <http://www.australian-universities.com/rankings/>, accessed 24 August 2011.
- Department of Innovation, Industry, Science and Research (2008): New Era for Research Quality, <http://minister.innovation.gov.au/Carr/MediaReleases/Pages/NEWERAFORRESEARCHQUALITY.aspx>, accessed 23 August 2011.
- Department of Innovation, Industry, Science and Research (2011): Consultation Paper on Options for the Inclusion of ERA in SRE Funding Allocation Model, <http://www.innovation.gov.au/Research/ResearchBlockGrants/Documents/SREConsultationPaperInclusionofE>

[RA.pdf](#), accessed 24 August 2011.

Department of Innovation, Industry, Science and Research (undated): ERA 2010 Discipline Matrices [http://www.arc.gov.au/xls/ERA2010\\_discipline\\_matrices.xls](http://www.arc.gov.au/xls/ERA2010_discipline_matrices.xls), accessed 26 August 2011

Department of Innovation, Industry, Science and Research (undated): Sustainable Research Excellence <http://www.innovation.gov.au/Research/ResearchBlockGrants/Pages/SustainableResearchExcellence.aspx>, accessed 23 August 2011

Department of Innovation, Industry, Science and Research (undated): ERA 2012 Important Dates [http://www.arc.gov.au/era/era\\_2012/important\\_dates.htm](http://www.arc.gov.au/era/era_2012/important_dates.htm), accessed 26 August 2011

Department of Innovation, Industry, Science and Research (undated): Frequently Asked Questions <http://www.arc.gov.au/era/faq.htm>, accessed 26 August 2011

Google Corp (undated): Google Scholar <http://scholar.google.com.au/>, accessed 26 August 2011

Hare, J. (2011): Elite eight head university research ratings. *The Australian* Jan. 31 2011, <http://www.theaustralian.com.au/higher-education/elite-eight-head-university-research-ratings/story-e6frgcjx-1225997293930>, accessed 23 August 2011.

Higher Education Funding Council for England (undated): Research Excellence Framework <http://www.hefce.ac.uk/Research/ref/>, accessed 23 August 2011.

IDP (undated): Australian universities, [http://www.students.idp.com/study/australian\\_universities.aspx](http://www.students.idp.com/study/australian_universities.aspx), accessed 24 August 2011.

Lawrence, S., Giles, C.L. and Bollacker, K. (1999): Digital libraries and autonomous citation indexing. *IEEE Computer* **32**(6):67-71.

Rowbotham, J. (2011): Kim Carr bows to rank rebellion over journal rankings. *The Australian* June 7 2011, <http://www.theaustralian.com.au/higher-education/kim-carr-bows-to-rank-rebellion/story-e6frgcjx-1226066727078>, accessed 28 August 2011.

## 10 Glossary

- ACDICT: Australian Council of Dean of ICT
- ACPHIS: Australian Council of Professors and Heads of Information Systems
- ACS: Australian Computer Society
- AIIA: Australian Information Industry Association
- ALIA: Australian Library and Information Association
- CORE: Computing Research and Education Association of Australasia
- EA: Engineers Australia
- HERDC: Higher Education Research Data Collection
- ITEE: Information, Telecommunications, and Electronics Engineering College (of EA)
- NHMRC: National Health and Medical Research Council
- NICTA: National ICT Australia



# Evolutionary Design of Optical Waveguide with Multiple Objectives

Qiao Shi<sup>1</sup>

Andy Song<sup>1</sup>

Thach Nguyen<sup>2</sup>

Arnan Mitchell<sup>2</sup>

<sup>1</sup> School of Computer Science & Information Technology  
RMIT University,  
GPO Box 2476, Melbourne, Victoria 3001,  
Email: qiao.shi@student.rmit.edu.au, andy.song@rmit.edu.au

<sup>2</sup> School of Electrical & Computer Engineering  
RMIT University,  
GPO Box 2476, Melbourne, Victoria 3001,  
Email: {thach.nguyen, arnan.mitchell}@rmit.edu.au

## Abstract

In this paper, we investigate a real-world problem, constructing optical waveguide structures using evolutionary search strategies. Optical waveguide is the most basic component in optical communication and integrated optical circuits. The structure of a waveguide is of great importance as it would significantly impact on the quality of light transmission. The aim of this paper is to find a set of potential structures which satisfy multiple waveguide design objectives including minimum group velocity dispersion and minimum propagation loss. Therefore, evolutionary algorithms which are populate-based search techniques are more suitable for this type of tasks. As a part of this investigation, a GP-based parametric optimization methodology called Parameter Mapping Approach (PMA) is introduced. This method together with traditional GA have been adapted into this study. The experiment results demonstrate both PMA and GA can produce multiple waveguide structures that meet the design criteria. Furthermore these evolved structures have very low dispersion and loss compared to those reported in the current literature.

**Keywords:** Optical Waveguide, Genetic Programming, Genetic Algorithms, Parameter Optimization, Structural Optimization

## 1 Introduction

Optical signal transmission is a foundation of our modern communication networks. Comparing with traditional electronic signal transmission, optical methods have more advantages such as more energy efficient less interference and higher capacity in carrying information. In these optical transmission networks the user data are aggregated and converted into optical signal to be transmitted in optical fibers. At the central office or switches, the optical signal is converted back to electrical domain so that the data can be regenerated, buffered or switched. The data stream are then converted back to optical signal for next transmission. Although optical fiber can transmit data at very high rate, processing data in the current approach in the electrical domain is the main bottle-neck of the current optical transmission

systems. As the demand for bit-rate continues to growth, it is desirable to process the optical signal directly in the optical domain. All optical signal processing is possible by utilising the nonlinear properties of the optical transmission medium such as optical fibers or optical waveguides. Optical waveguides are the preferred medium for all optical signal processing since they allow for compact devices and the possibility of integrating many functions into a single device to create an integrated photonic chip. By using waveguide materials with strong optical nonlinearity such as Chalcogenide, many optical signal processing functions can be realized on a short optical waveguide (Eggleton et al. 2011). In order to achieve highly efficient optical signal processing functions in a waveguide, it is important to minimize the propagation loss and the dispersion of the waveguide mode. By optimizing the waveguide structure, it is possible to achieve both low loss and low dispersion waveguides.

Structural optimization is an important area itself. In many circumstances, the complexity of the problem is high because there are a large number of factors to be optimized such as shape, quality and dimension. Moreover often there are many constraints to be taken into consideration, such as weight, size, cost and so on. This type of optimization tasks can be divided into two categories as suggested by (Rasheed 1998), pure structural optimization and parametric optimization. The former one involves making high level decision about geometric properties of the structure while the latter mainly focuses on the numeric aspects of structures, that is finding more suitable combinations of parameters for a given shape. Waveguide structure design can be addressed at both levels. The study presented here only focuses on the latter, finding better parameters for single-ridge waveguides.

One prominent approach in optimization is evolutionary search methods which are inspired by Darwin's natural selection principle. Among a population of potential solutions, the better ones are selected to create a new generation of solutions which presumably will be better than the previous generation. This iterative process usually stops at a point that a good solution is found or no improvement can be achieved. The main methods under this category include Genetic Algorithm (GA) (Holland 1975), Genetic Programming (GP) (Koza 1992), Differential Evolution (DE) and Particle Swarm Optimization (PSO). Because evolutionary methods are population based, they are capable of provide not only one solution but a set of solutions. Finding multiple solutions is one the aims of our waveguide design task.

GA has been successfully used in a wide range

of parametric optimization problems (Pujol & Poli 2004a) and remains as a main choice for this kind of tasks. Therefore GA is selected for this study. In addition GP is also introduced here although it is not strong in parameter optimization but structural optimization such as designing circuits and satellite antenna (Lohn et al. 2004). Because our long term goal is to include structural optimization for waveguide design as well. The conventional symmetric single ridge waveguide is not necessarily the best structure. Ideally one method, presumably a GP-based technique, can optimize both the structure as well as its parameters. Therefore GP is used and compared with GA. As proposed and studied by Pujol & Poli (2008), GP could be adapted into parametric optimization and even outperform others in some cases when applying to some benchmark functions (Ingber & Rosen 1992).

Another aspect of this study is its multiple objectives. The quality of a waveguide structure is not measured by just one criterion. Multiple objectives such as minimum dispersion and minimum propagation loss are all highly desirable in waveguide applications. This is another reason of using evolutionary methods as they can be naturally adapted into multi-objective optimization (Deb et al. 2002, Zitzler et al. 2001).

The rest of this paper is organized as such: Section 2 explains the basics of waveguide structure and the meanings of dispersion and propagation loss which we mentioned above. This section also covers a brief introduction to GP/GA and the related work. Section 3 presents the methodology used in this study. Section 4 describes the experiments with the results. Section 5 concludes this study and discusses our future investigations.

## 2 Background

The first two parts of this section introduces the basics of optical waveguide and GP, GA briefly. Additionally the related work for structural design, parametric optimization as well as waveguide structure optimization are reviewed.

### 2.1 Optical Waveguide

Optical waveguide is a medium to guide light wave propagation. In this paper a single-ridge waveguide structure is considered. Figure 1 shows a modeled structure of optical ridge waveguide, which is the basis of this study. The middle core layer is one kind of chalcogenide glasses  $As_2S_3$ , a highly nonlinear crystalline material. The properties of the waveguide mode are determined by the waveguide cross-section parameters, including the ridge width, the height of the core layer and the etch depth.

The measurement of waveguide structure is a key point in our experiment. In industry, the most accurate way is to produce a real waveguide and test it through some devices. However, this operation often time consuming and costly. The alternative is using simulators. The simulator in this study is from RMIT's Microplatform Research Group and has been used for a series of waveguide projects (Nguyen et al. 2009a,b).

There are a set of properties which are related to the quality of waveguides, such as single-mode or multi-mode, dispersion, nonlinearity, loss and etc. This study concerns two properties: the dispersion and the loss. The followings are their descriptions.

- *Dispersion:* Group velocity dispersion, or simply dispersion in this study, of a waveguide mode is a

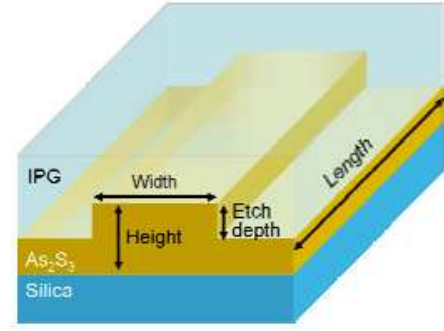


Figure 1: A modelled structure of  $As_2S_3$  waveguide. (Courtesy to M. R. E. Lamont, etc.)

parameter that measures how the group velocity of the waveguide mode depends on wavelength of frequency. Group velocity dispersion is caused by material dispersion and waveguide dispersion. The material dispersion comes from material. Waveguide dispersion is caused by the geometric (structural) reasons. As indicated in prior literature, in some certain situations these two kinds of dispersion can compensate each other and result in a zero-dispersion waveguide (Lamont et al. 2007).

- *Propagation loss:* The propagation loss of a waveguide mode can be caused by material absorption, scattering loss and leakage loss. At telecommunication wavelength of  $1.55 \mu m$ , the materials used in the considered Chalcogenide waveguide have negligible absorption loss. The scattering loss is mainly determined by the fabrication process. Leakage loss is caused by the coupling between the guided mode and radiation modes of a waveguide. Leakage loss can be effectively reduced to zero by optimizing the waveguide structure (Nguyen et al. 2009c). In this study, material loss and scattering loss are ignored when consider waveguide propagation loss.

It should be noted that the loss is measured as  $dB/cm$ , decibel per centimeter of the waveguide. The aim is to find waveguide structures with both zero-dispersion and low loss. In fact absolute zero-dispersion is hardly achievable at telecommunication wavelength of  $1.55 \mu m$ . Instead the following formula is used to define zero-dispersion.

$$Dispersion = \left| \frac{\partial^2 \beta}{\partial \omega^2} \right| < 1.0 \times 10^{-26} (nm^2/m)$$

The partial second derivative part  $\left| \frac{\partial^2 \beta}{\partial \omega^2} \right|$  is the wave equation in which  $\beta$  is a function of  $\omega$ , angular frequency. Note for legibility, zero dispersion can be also expressed as:

$$\left| \frac{\partial^2 \beta}{\partial \omega^2} \times 10^{24} \right| < 0.01 (nm^2/m).$$

Additionally, there is another criterion which is used to determine zero-dispersion: whether the waveguide is able to achieve absolute zero dispersion at around  $1.55 \mu m$  wavelength. It is believed in such case we can shift the zero-dispersion wavelength to telecom wavelength (Lamont et al. 2007). In order to validate this criterion, we should plot all the dispersion value from  $1.4 \mu m$  to  $1.7 \mu m$  wavelength. Figure 2

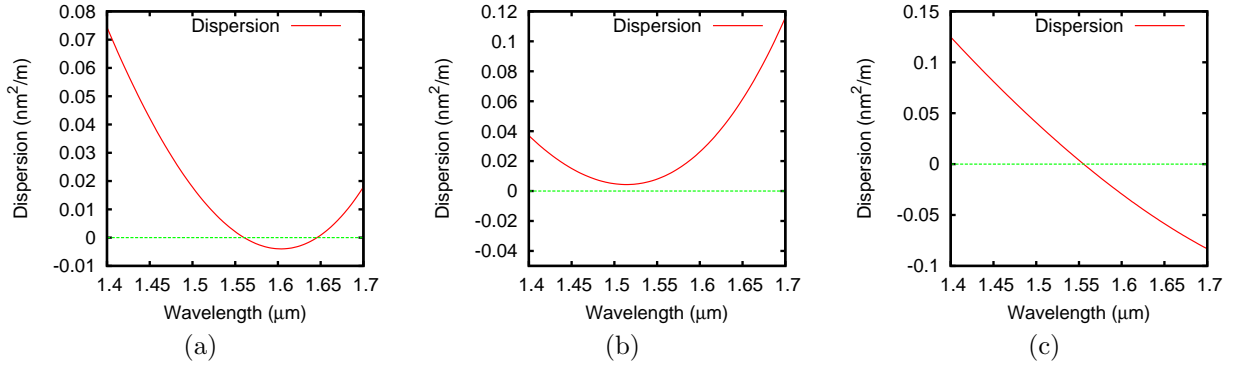


Figure 2: Three different kinds of distribution for dispersion property

shows three possible dispersion distributions. All of them meet the first requirement which can have dispersion value less than  $0.01 \text{ nm}^2/\text{m}$ . However, the case in Figure 2 (b) is not valid as it is not able to achieve absolute zero-dispersion at  $1.55 \mu\text{m}$  wavelength.

In terms of propagation loss, recent research reported structures with loss down to  $0.2 \text{ dB/cm}$  (Madden et al. 2007, Cardenas et al. 2009, Ruan et al. 2005) as that is considered good quality in the field. However we aim to further reduce the loss below  $0.1 \text{ dB/cm}$ . Propagation loss is calculated as following formula:

$$\text{Loss} = 8.868 \cdot \alpha \cdot \frac{2\pi}{\lambda \cdot 100} < 0.1 (\text{dB/cm})$$

The  $\lambda$  value in the formula is the wavelength of optical transmission. It is set as  $1.55 \mu\text{m}$  which is mentioned before, the commercial telecommunication wavelength. The  $\alpha$  value is a parameter to indicate reduction in light density. It can be produced by waveguide simulator. Thus the aim of reducing loss below  $0.1 \text{ dB/cm}$  is equivalent to bringing down the  $\alpha$  value to a low level:  $\alpha < 2.8 \times 10^{-7}$ .

## 2.2 Genetic Algorithm and Genetic Programming

Genetic Algorithm (GA) and Genetic Programming (GP) are two typical members in the area of Evolutionary Computing. GP could be considered as a variation of GA as they do share large amount of similarities.

Both GA and GP randomly generate a population of solutions as the initial generation. These solutions, also called individuals, are then evaluated in terms of their capability in solving a particular problem. The better ones have higher probabilities of being selected for creating the new generation. Therefore the next generation is likely better than the previous one. Majority of individuals in the new generation are created by exchanging genetic materials among parents, the individuals selected from the previous generation. This process is known as crossover. Some individuals are created by randomly changing one selected parent. This process is named mutation. Some individuals are just straight copied from the previous generation. This is called elitism.

An evolutionary process will continue from generation to generation until a perfect solution is found or one of other criteria is met such as no improvement for a number of generations, or a maximum number of generations is reached. The driving force of this

evolutionary process is the fitness measure which determines the survivability of individuals. The fitness tends to improve over the generations. In our case, the fitness is the quality of a waveguide structure, in terms of both zero dispersion and low propagation loss.

The main difference between GA and GP is how to represent an individual. In GA, an individual is often a fixed-length binary string (chromosome) which can naturally be used to express a list of numeric parameters. By crossing over between different individuals or mutating one individual, various combinations of parameters can be created. The better ones will survive and eventually emerge as the final solution. Due to this linear representation GA is suitable for parametric optimization of which the number of parameters are given.

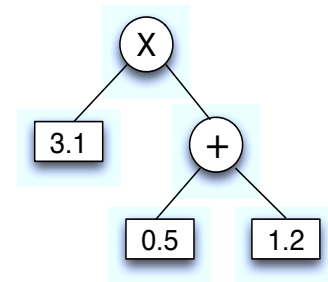


Figure 3: GP tree representation

In contrast an individual in GP is often represented as a tree as shown in Figure 3. The internal nodes on a tree are called functions which are often some kind of operators and the leaf nodes are called terminals which often are input values or parameters for the function nodes connected to them. The tree shape is usually very flexible as long as the number of levels on the tree is within a limit and the tree is syntactically sound, meaning it can be evaluated without any error. Crossover in GP is done by two parents swapping tree branches. Mutation in GP is randomly replacing one branch on an individual with an external branch. It can be seen that new individuals would be very different to their parents topologically. Due to this tree representation GP is suitable for exploring different structures. GP has shown its ability and effectiveness in designing structures and solving wide range of real-world problems (Poli et al. 2008).

### 2.3 Related Work

In the literature waveguide structures are optimized by domain experts from the area of photonics. Lamont et al. (Lamont et al. 2007) proposed a dispersion engineering method of  $As_2S_3$  waveguide structure with dispersion  $\approx 0.24nm^2/m$  and loss value  $\approx 0.25dB/cm$  in 2007. Cardenas et al. demonstrated an optical waveguide structure with propagation loss of  $0.3dB/cm$  at  $1.55\mu m$  using Silicon as the medium material (Cardenas et al. 2009). Madden et al. discovered a structure with loss as low as  $0.05dB/cm$  at telecom wavelength (Madden et al. 2007). All these structures were optimized manually with domain knowledge. Our aim is to generate a set of better waveguides without requiring domain experience and human intervention during the process.

Structural optimization is an important area in GA. (Rasheed 1998) attempted to optimize the aircraft structure using GA. In their work, the structural optimization is actually treated as optimizing several parameters. The task is to determine the dimensions of the aircraft, the length of wings and so on, while the basic shape of the aircraft does not change. (Chafekar et al. 2003) continued the previous work and improved its performance in addressing constrained multi-objective optimization problem. In the aircraft design, the structural optimization problem is converted into a parametric optimization problem. This strategy is applied in this study.

GP, as a strong problem solving method, has demonstrated its capability in designing structures topologically. NASA uses GP to design satellite antennas. Their investigation produced an antenna with higher ratio of signal gain to self weight. It is more effective than the designs from NASA engineers. Also, it was stated that the GP design schema significantly reduced the design life cycle (Lohn et al. 2004).

### 3 Methodology

The waveguide design problem is addressed by two evolutionary methods: using the relative new PMA technique and using the traditional GA. Firstly we explain the PMA methodology and then give a brief introduction of Non-dominated Sorting for multi-objective optimization.

#### 3.1 GA and PMA methodology

For a single-ridge symmetric waveguide, there are three variables, the width, the height and the etch depth as illustrated in Figure 1. As the basic shape does not change, the task is actually a parametric optimization problem. The representation for GA is straightforward, three double numbers expressed as one binary string. The first part of the string corresponds to width, the middle part to the height and the last part for the etch depth.

For GP the representation is a little more involved as parametric optimization is not native for GP. In order to achieve this, Parameter Mapping Approach (PMA) is used (Pujol & Poli 2004b). The basic idea is not directly finding a good combination of parameters, but searching for GP individuals as a mapping function which can accept a set of raw inputs to produce another set of adaptive parameters. The optimal combination of parameters are searched indirectly.

Figure 4 shows the procedure to evaluate an individual in PMA. A mapping function receives a set of raw parameters as input and transforms them to three parameters which are interpreted as the width,

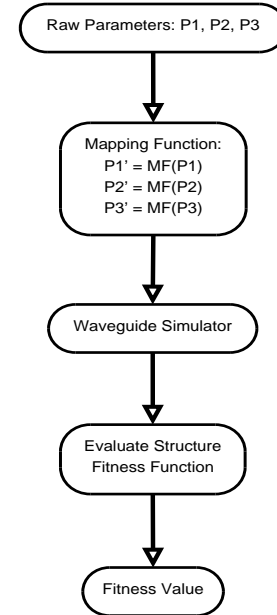


Figure 4: Fitness evaluation for a GP individual

the height and the etch depth of a waveguide. Note that, during the evolutionary process, all individuals share the same raw parameters. That is different to the original PMA proposed in (Pujol & Poli 2004b), in which the initial raw parameters are randomly generated. By introducing a constant initial parameter, these generated mapping functions could share some of the components or building blocks. Moreover the step of generating random numbers for each raw parameter can be removed.

The evaluation of a combination of parameters is done by a waveguide simulator developed by Microplatform Research Group in RMIT. It can simulate the transmission process in a given waveguide structure and therefore calculate its properties such as dispersion and propagation loss. These output from the simulator are used to assign fitness values of individuals either in GA or GP.

PMA uses 4-arity operators as the GP functions, because they can break the symmetry of the addition and multiplication arithmetic operators (Pujol & Poli 2004b), thus reducing the possibility of the so-called *permutation* or *competing convention* problem (Radcliffe 1991, Hancock 1992).

Table 1: GP Functions used in PMA

Plus	$x \times y + u \times v$
Minus	$x \times y - u \times v$
Multiply	$(x + y) \times (u + v)$
Divide	$PDV(x + y, u + v)$

The function set of GP is listed in Table 1: where  $PDV(num, den)$  is the protected division, which returns  $num$  if the denominator  $den$  is zero. These four are the only operators used. The terminal set is simply random numbers and raw input. Tournament selection strategy is employed in this model. The GA and GP runtime parameters are described along with the experiments.

To evaluate the robustness of PMA, a series of test functions were introduced. The first step of optimizing waveguide structures is using only one objective, dispersion which is the most important measure for waveguides. Both GA and GP were used for this sin-



gle objective task. Followed the single objective experiments, both dispersion and loss were introduced as the objectives for GA and GP.

### 3.2 Non-dominated Sorting/NSGA-II

Non-dominated Sorting Genetic Algorithm (NSGA) was first proposed by (Deb et al. 2002). Since the time it was developed, this algorithm has been criticized due to its high computational complexity of nondominated sorting, lack of elitism. Further investigation on this approach leads to an improved version of NSGA, namely *Non-dominated Sorting Genetic Algorithm II*. Instead of only one optimal solution, the NSGA-II provides a set of optimal solutions. The multiple solutions are those none of which that can be considered to be better than any others with respect to all objectives. This set of optimal solutions is known as a *Pareto Optimal Set* or a *Pareto Frontier*.

For a multiple objective optimization problem, a feasible solution can be represented as a vector  $X$ :

$$X(\text{Objective1}, \text{Objective2}, \dots, \text{ObjectiveN})$$

This solution is considered to be non-dominated if and only if,

- For any other vector  $Y$ , each objective determined by vector  $X$  is better or at least equal to that one determined by vector  $Y$ .
- For any other vector  $Y$ , at least one of the objectives determined by vector  $X$  is strictly better than the corresponding objective determined by vector  $Y$ .

For a given number of solutions, there is only one vector that can satisfy both the above criteria. It cannot be improved without worsening at least another objective. The *Pareto Optimal Set* is composed of such kind of solutions.

In terms of the implementation of NSGA-II algorithm, the basic idea is to divide the population into a number of sub-populations referred as fronts which are ranked in terms of levels (Nguyen & Yousefi 2010). For each front, there is one non-dominated solution which satisfies the above two criteria. In this way, for the entire population, there is a set of non-dominated solutions derived from the individual frontier.

In the second generation starting from the initial population, these ranked points are then reproduced through genetic operators. Individual elements with a higher rank are more likely to be selected for reproduction. The solutions in the first level front are assigned the highest priority, and then are those in the second level and so forth. Eventually the Pareto frontier is formed as the rank can no longer be improved.

## 4 Experiments

Three groups of experiments mentioned above are presented in this section. The difficulty gradually increases.

### 4.1 Optimizing Test Formulae

The task there is to find the minimum of the following four formulae of which the number of parameters differs. The fitness measure is the lower output the better.

- $f_1(x, y) = x^2 + y^2$

- $f_2(x, y) = 100 \times (x^2 - y)^2 + (1 - x)^2$
- $f_3(x, y, z) = (x - 5)^2 + (y - 15)^2 + (z - 40)^2$
- $f_4(a, b, c, d, e) = (a + 0.5)^2 + (b - 55)^2 + (c - 0.5)^2 + (d - 15)^2 + (e - 99)^2$

Table 2: Test Formulae for PMA

Formula	No. of Parameters	Success Rate
$f_1(x, y)$	2	99/100
$f_2(x, y)$	2	99/100
$f_3(x, y, z)$	3	100/100
$f_4(a, b, c, d, e)$	5	97/100

The population size here was set to 200 and the total number of generations is 100. The GP runtime configuration was that 80% crossover rate, 10% mutation rate and 10% elitism.

Table 2 shows the results: the number of runs (out of 100 runs) can find a solution of which the output is lower than 0.001. Absolute zero was not required here as it was the case in the waveguide design. The success rate was very high in all the experiments. The modified PMA method is capable in parametric optimization on simple tasks.

### 4.2 Optimization with Single Objective: Dispersion

The goal of this set of experiments is to search for structures with zero-dispersion. There is only one objective. For each of the parameter, there is a reasonable range of values. Parameters outside of this range are not practical therefore should not be explored. The ranges are

$$\text{waveguide width} \in [0.1\mu\text{m}, 2.0\mu\text{m}] \quad (1)$$

$$\text{waveguide height} \in [0.1\mu\text{m}, 1.5\mu\text{m}] \quad (2)$$

$$\text{etch depth} \in [0.0\mu\text{m}, \text{waveguide height}] \quad (3)$$

As GA represents numerical values of these parameters directly, so the range can be easily imposed on these parameters in GA. However, the PMA methodology does not deal with values directly. Thus the value for an individual parameter can not be guaranteed to situate in that range setting. Therefore a parameter normalization procedure is introduced here. It can be expressed as:

$$\text{Parameter} = \text{LOW} + \frac{\text{UP} - \text{LOW}}{1 + |\text{OUTPUT}|}$$

where:

LOW is the lower limit of the parameter;

UP is the upper limit of the parameter;

OUTPUT is the output value from a GP tree.

Note that the third parameter etch depth is can not be larger than the second parameter waveguide height. To fulfil this constrain, a *swap* strategy is introduced which examines all individuals before fitness evaluation, both for GA and for PMA. If one individual violates that constrain, then its third parameter will be swapped with the second parameter. If its third parameter is beyond the up limit, 1.5 $\mu\text{m}$ , then it will be trimmed to 1.5.

In terms of the fitness measure for GA and PMA, they are the same. As there is only one objective, the fitness evaluation can be simply expressed as:

$$\text{fitness} = |\text{Dispersion}|$$

Table 3: Optimizing Dispersion in TM mode by PMA and GA

Solutions	Width( $\mu\text{m}$ )	Height( $\mu\text{m}$ )	Etch Depth( $\mu\text{m}$ )	Dispersion ( $\text{nm}^2/\text{m}$ )	Approach
1	1.563	1.092	0.442	0	PMA
2	1.059	1.043	0.991	0	PMA
3	1.042	1.065	1.011	0	PMA
4	1.089	1.083	1.066	0	PMA
5	1.386	1.240	1.180	0	PMA
6	0.482	1.172	1.122	0	PMA
7	0.842	0.828	0.742	0	PMA
8	1.253	1.087	0.847	0	PMA
9	0.584	0.717	0.657	0	PMA
10	0.732	0.732	0.632	0	PMA
11	1.576	1.091	0.439	0	GA
12	1.670	1.422	0.699	0	GA
13	1.576	1.077	0.759	0	GA
14	1.338	1.177	0.873	0	GA
15	1.144	1.018	0.393	0	GA
16	1.803	1.132	1.032	0	GA
17	1.555	0.988	0.343	0	GA
18	1.576	1.091	0.439	0	GA
19	1.103	1.091	1.028	0	GA
20	0.937	1.091	0.747	0	GA

Table 4: Optimizing Dispersion in TE mode by PMA and GA

Solutions	Width( $\mu\text{m}$ )	Height( $\mu\text{m}$ )	Etch Depth( $\mu\text{m}$ )	Dispersion ( $\text{nm}^2/\text{m}$ )	Approach
1	0.943	0.943	0.843	0	PMA
2	1.339	0.807	0.798	0	PMA
3	0.946	0.946	0.846	0	PMA
4	0.965	1.247	1.185	0	PMA
5	1.355	1.331	1.260	0	PMA
6	0.778	0.931	0.874	0	PMA
7	0.935	0.935	0.835	0	PMA
8	1.311	1.383	1.329	0	PMA
9	0.500	0.647	0.569	0	PMA
10	1.320	1.320	1.220	0	PMA
11	1.978	1.266	1.010	0	GA
12	1.250	1.341	1.224	0	GA
13	1.250	1.175	1.021	0	GA
14	1.250	1.233	1.224	0	GA
15	1.250	0.998	0.992	0	GA
16	0.620	1.335	1.224	0	GA
17	1.978	1.266	1.010	0	GA
18	1.250	1.175	1.021	0	GA
19	0.620	1.335	1.224	0	GA
20	1.250	1.175	1.021	0	GA

There is another issue related to the optimization process, that is the multi-modes waveguide dispersion measurement. As in this set of experiment, the *single mode* condition was not take into consideration, it is possible to work out waveguide structures with multiple modes. For each mode in waveguide, it has its corresponding dispersion and loss value. Regarding to this situation, average value is usually expected. However, this strategy is not satisfactory here, because in real-world optical applications, only the mode with the lowest dispersion value will be used. Thus, we selected the lowest dispersion value for the fitness measurement of multi-modes waveguide.

The run time configuration for GA and PMA were the same: 80% of crossover, 10% of mutation and 10% of elitism. The population size and maximum number of generations were also same, which were both 200. In PMA, the maximum depth of the GP tree is set to 5 to avoid solutions being too complex.

There are two main transverse modes need to be considered in waveguide design, TE (Transverse Elec-

tric) mode where there is no electric field along the propagation path, and TM (Transverse Magnetic) mode where there is no magnetic field along the propagation path. In this study we investigate both TM mode and TE mode. In optical applications, the waveguide will be applied either in TM environment or TE environment. The results for TM mode and TE mode are shown in Table 3 and Table 4 respectively.

The solutions listed in Tables 3 and 4 are the best individuals found by these evolutionary search processes. It can be seen that most of them are different. That means both GA and GP are capable of finding multiple solutions. This feature is important in practice as engineers would have choices in deciding which one to use under different circumstances, such as manufacturing cost or size limits for different applications.

All these solutions found by GA and PMA have zero dispersion at telecom wavelength, much lower than the required zero dispersion threshold,  $0.01\text{nm}^2/\text{m}$ . In general, PMA appeared to have the equal performance as GA as both of them find zero



dispersion. In terms of search speed, GA performs slightly better than PMA because PMA needs to construct GP tree which is a more complex data structure than just numeric values. However, such difference has small impact on the overall speed as the majority of computation is consumed by the waveguide simulator. A single run for PMA in this set of experiments took approximate 28 hours and it was around 26 hours for GA. Both of them ran under the same environment: a quad-core 2.3 GHz machine with 12 GB memory.

### 4.3 Optimization with Multiple Objectives: Dispersion & Loss

The requirement for dispersion here is exactly the same as for the previous experiments. Propagation loss is added as the second objective. We have evaluated the solutions obtained from the previous experiments in term of loss. None of them could meet that criterion although their dispersion were very low.

The typical multi-objective optimization method, Non-dominated Sorting, could be used to incorporate these two criteria. NSGA-II which has native support for optimizing multiple objectives was introduced into this set of experiments. The fitness measurement for PMA is different from NSGA-II, instead we used the *Weighted Sum Approach* as the dispersion has higher preference than loss in waveguide design. The fitness evaluation is the following:

$$fitness = |Dispersion| \times 100 + |Loss|$$

The weights of dispersion and loss in this fitness measurement were determined empirically. Since the requirement of dispersion is much higher than that of loss, a high weight is assigned to the dispersion property. In this case, it was 100. The choosing of this weight is not limited to just 100. It could be even larger or smaller. The purpose of this weight value is only to specify that dispersion is a much more important feature than loss in waveguide design. Empirically 100 is a more suitable value.

The runtime configuration such as crossover rate, mutation rate, elitism, population size, number of generations were identical to those in the previous set of experiments. The corresponding results for TM mode are shown in Table 5. Similar experiments were conducted in TE mode, and there was no results that met both the dispersion and loss criteria. However, such results are not surprising, since the literature has already stated that it is less likely to form zero-dispersion at telecom wavelength in TE mode when considering more properties (Lamont et al. 2007). Our findings is consistent with the discovery from researchers in optical engineering.

The results presented in Table 5 show that GP is able to find solutions which satisfy both objectives. The results are extracted from the last generation of PMA and NSGA-II. The top 20 results are given here and sorted by the dispersion value. Eleven of the results are from PMA while nine of them are from NSGA-II. It should be noted that the reported loss values are the approximations. The “0” loss values are the direct output from the waveguide simulator which is not able to generate high precision loss value when the loss is very small. However what is certain is that the loss values are much smaller than 0.1 dB/cm.

Table 5 shows that the PMA approach and NSGA-II have very similar performance in terms of minimizing dispersion and loss. However, the average dispersion value from PMA ( $\approx 0.00086231nm^2/m$ ) is slightly better than that from NSGA-II ( $\approx$

$0.0010853nm^2/m$ ). The best solution was found by PMA. The dispersion value of it is only  $0.000173335nm^2/m$ . The difference between these approaches may be due to the multi-objective fitness measurement. In the Non-dominated Sorting in NSGA-II, all the objectives are treated equally, while in the weighted sum approach in PMA, dispersion is give a much higher weight than than loss. Due to the priority setting, PMA could focus on finding solutions with smaller dispersion value. In our further study, we will investigate the effectiveness of weighted sum approach compared with the standard non-dominated sorting in handling multiple objectives.

## 5 Conclusions and Future Work

In this paper we studied designing optimal waveguide structures with multiple objectives using evolutionary techniques, namely GA and PMA, a GP-based optimization method. The aim here is not to just find one solution but a set of different solutions. Based on the investigation we conclude the followings: evolutionary search methods GA and GP are suitable for solving this real world problem. They are capable of finding multiple waveguide structures which meet multiple design objectives. By the combination of PMA and weighted sum fitness, and by NSGA-II, we were able to find waveguide structures with dispersion as well as propagation loss much lower than what in the current literature. Additionally little human intervention is required by these methods to generate these satisfactory optical waveguides.

The GP-based PMA method is a suitable method for parametric optimization. As stated in (Pujol & Poli 2008), PMA should be investigated on real-world problems. To our knowledge this is the first time PMA was tested on a real-world application. It is arguably better than or at least equivalent to classical GA in this problem.

In the near future we will combine parametric optimization and geometric design of waveguide into a single framework by using GP, so the structure may contain multiple ridges and may not be rectangular or symmetric. Furthermore we will incorporate more objectives such as maximizing nonlinearity and maintaining a single mode in the evolutionary waveguide design.

## References

- Cardenas, J., Poitras, C. B., Robinson, J. T., Preston, K., Chen, L. & Lipson, M. (2009), ‘Low loss etchless silicon photonic waveguides’, *Opt. Express* **17**(6), 4752–4757.
- Chafekar, D., Xuan, J. & Rasheed, K. (2003), Constrained multi-objective optimization using steady state genetic algorithms, in ‘Proceedings of the 2003 international conference on Genetic and evolutionary computation’, Springer-Verlag, Berlin, Heidelberg, pp. 813–824.
- Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002), A fast and elitist multiobjective genetic algorithm: Nsga-ii, in ‘IEEE Transactions on Evolutionary Computation’, Vol. 6, pp. 182–197.
- Eggleton, B. J., Luther-Davies, B. & Richardson, K. (2011), ‘Chalcogenide photonics’, *Nat Photon* **5**(3), 141–148. 10.1038/nphoton.2011.309.

Table 5: Optimizing Dispersion &amp; Loss in TM mode using PMA and NSGA-II

Solutions	Width ( $\mu\text{m}$ )	Height ( $\mu\text{m}$ )	Etch Depth ( $\mu\text{m}$ )	Dispersion (/W/km)	Loss (dB/cm)	Approach
1	1.820	1.029	0.621	0.000173335	0.0	PMA
2	1.629	1.073	0.774	0.000196365	0.0	PMA
3	1.401	1.130	0.940	0.000222729	0.0	NSGA-II
4	1.410	1.069	0.625	0.000224547	0.0	NSGA-II
5	1.301	1.138	0.916	0.000263336	0.0	PMA
6	0.848	1.222	1.218	0.000547884	0.0	PMA
7	0.986	1.198	1.090	0.000647582	0.0	PMA
8	1.998	1.029	0.745	0.00079728	0.0	NSGA-II
9	1.876	1.015	0.544	0.000811825	0.0	PMA
10	1.293	1.120	0.831	0.000911523	0.0	PMA
11	1.361	1.067	0.580	0.000951221	0.0	NSGA-II
12	1.167	1.177	1.041	0.00107001	0.0	PMA
13	1.249	1.082	0.626	0.00110486	0.0	NSGA-II
14	1.841	1.073	0.985	0.00114456	0.0	NSGA-II
15	1.020	0.630	0.452	0.0016485	0.0	PMA
16	0.898	1.134	0.929	0.00171092	0.0	PMA
17	1.049	1.116	0.773	0.00171183	0.0	PMA
18	1.650	1.020	0.458	0.0017688	0.0	NSGA-II
19	1.783	0.938	0.303	0.00177244	0.0	NSGA-II
20	0.782	1.135	1.027	0.00179699	0.0	NSGA-II

- Hancock, P. (1992), Genetic algorithms and permutation problems: a comparison of recombination operators for neural structure specification, in 'Proceedings of COGANN Workshop', IEEE Computer Society Press.
- Holland, J. H. (1975), *Adaption in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*, University of Michigan Press.
- Ingber, L. & Rosen, B. (1992), Genetic algorithms and very fast simulated reannealing: a comparison, in 'Mathematical and Computer Modelling', Vol. 16, pp. 87–100.
- Koza, J. R. (1992), *Genetic Programming: On the programming of computers by means of natural selection*, The MIT Press.
- Lamont, M. R. E., Sterke, C. M. & Eggleton, B. J. (2007), 'Dispersion engineering of highly nonlinear  $AS_2S_3$  waveguides for parametric gain and wavelength conversion', *Optics Express*.
- Lohn, J. D., Hornby, G. S. & Linden, D. S. (2004), An evolved antenna for deployment on nasa's space technology 5 mission, in 'Genetic Programming Theory Practice 2004 Workshop'.
- Madden, S. J., Choi, D.-Y., Bulla, D. A., Rode, A. V., Luther-Davies, B., Ta'eed, V. G., Pelusi, M. D. & Eggleton, B. J. (2007), 'Long, low loss etched  $as_2s_3$  chalcogenide waveguides for all-optical signal regeneration', *Opt. Express* **15**(22), 14414–14421.
- Nguyen, T. G., Tummidi, R. S., Koch, T. L. & Mitchell, A. (2009a), 'Lateral leakage in tm-like whispering gallery mode of thin-ridge silicon-on-insulator disk resonators', *Optics Letters* **34**(7), 980–982.
- Nguyen, T. G., Tummidi, R. S., Koch, T. L. & Mitchell, A. (2009b), 'Rigorous modeling of lateral leakage loss in soi thin-ridge waveguides and couplers', *IEEE Photonics Technology Letters* **21**(7), 486–488.
- Nguyen, T. G., Tummidi, R. S., Koch, T. L. & Mitchell, A. (2009c), 'Rigorous modeling of lateral leakage loss in SOI thin-ridge waveguides and couplers', *IEEE Photon. Technol. Lett.* **21**(7), 486.
- Nguyen, T. T. & Yousefi, A. (2010), Multi-objective approach for optimal locaiton of tsc using nsga ii, in 'International Conference on Power System Technology'.
- Poli, R., Langdon, W. B. & McPhee, N. F. (2008), *A field Guide to Genetic Programming*.
- Pujol, J. & Poli, R. (2004a), A highly efficient function optimization with genetic programming, in 'Late-breaking papers of Genetic and Evolutionary Computation Conference', pp. 26–30.
- Pujol, J. & Poli, R. (2004b), Optimization via parameter optimization with genetic programming, in 'Proceedings of the 8th International Conference on Parallel Problem Solving from Nature', pp. 18–22.
- Pujol, J. & Poli, R. (2008), 'Parameter mapping: A genetic programming approach to function optimization', *International Journal of Knowledge-based and Intelligent Engineering Systems* **12**(1), 29–45.
- Radcliffe, N. (1991), Genetic set recombination and its application to neural networks topology optimization, Technical report, Scotland.
- Rasheed, K. (1998), GADO: A Genetic Algorithm for continuous design optimization, PhD thesis, The State University of New Jersey.
- Ruan, Y. L., Luther-Davies, B., Li, W. T., Rode, A., Kolev, V. & Madden, S. (2005), 'Large phase shifts in  $as_2s_3$  waveguides for all-optical processing devices', *Opt. Lett.* **30**(19), 2605–2607.
- Zitzler, E., Laumanns, M. & Thiele, L. (2001), Spea2: Improving the strength pareto evolutionary algorithm, Technical report, Swiss Federal Institute of Technology.

# Real-time Evolutionary Learning of Cooperative Predator-Prey Strategies

Mark Wittkamp<sup>1</sup>

Luigi Barone<sup>1</sup>

Phil Hingston<sup>2</sup>

Lyndon While<sup>1</sup>

<sup>1</sup> School of Computer Science and Software Engineering  
University of Western Australia,  
Crawley, Western Australia  
Email: {wittkamp, luigi, lyndon}@csse.uwa.edu.au

<sup>2</sup> School of Computer and Security Science  
Edith Cowan University,  
Mount Lawley, Western Australia  
Email: p.hingston@ecu.edu.au

## Abstract

Despite games often being used as a testbed for new computational intelligence techniques, the majority of artificial intelligence in commercial games is scripted. This means that the computer agents are non-adaptive and often inherently exploitable because of it. In this paper, we describe a learning system designed for team strategy development in a real time multi-agent domain. We test our system in a prey and predators domain, evolving adaptive team strategies for the predators in real time against a single prey opponent.

Our learning system works by continually training and updating the predator strategies, one at a time for a designated length of time while the game is being played. We test the performance of the system for real-time learning of strategies in the prey and predators domain against a hand-coded prey opponent. We show that the resulting real-time team strategies are able to capture hand-coded prey of varying degrees of difficulty without any prior learning. The system is highly adaptive to change, capable of handling many different situations, and quickly learning to function in situations that it has never seen before.

*Keywords:* evolution, learning, multi-agent, predator-prey

## 1 Introduction

Games are often used as test-beds to further the development of computational intelligence techniques. They are suitable for this task because they involve similar problems to those encountered in real life, but are simpler and more clearly defined, generally with a well understood goal. Video games present a particularly interesting problem domain in that they typically have a far greater number of actions available for players to make and these actions have temporal significance. The development of adaptive behaviour using opponent modeling with evolutionary algorithms has been demonstrated before (Wittkamp 2006), (Wittkamp 2006), but the problem becomes

much more difficult when we require the learning to occur in real-time, as the game itself is being played.

Artificial players that train offline (generally by playing the game) can have a near limitless amount of training time available to them. The learning and fine tuning of artificial players could run continuously for many days or weeks until desirable behaviours have been found. Contrast this with real-time learning, where there is very little time to run simulations and the processor must also be shared with the game engine itself. Computational intelligence techniques require many iterations and many more test cases for the evolution process to yield desirable results. In order for a real-time approach to be feasible, standard computational intelligence techniques will need to be sped up.

### 1.1 The Case For Real-Time Learning

Despite a large amount of research in the field of video game AI, the majority of AI strategy in commercial games is still in the form of scripted behaviour (Berger 2002). Developers turn to scripts for a number of reasons; they are understandable, predictable, easy to modify and extend, and are usable by non-programmers (Tozour 2002). Scripts often have parameters that may be optimised using computational intelligence techniques offline, but the learning aspect is rarely a component in the released product (Charles 2007).

While scripts can respond to the actions of human players, artificial agents (or “bots”) are often inherently exploitable due to their inability to adapt. Once an agent’s weakness has been discovered it can be exploited time and time again and soon the game fails to remain challenging or realistic and human players may lose interest. No matter how thorough the training process, in many modern games there are too many possible scenarios to expect that a hand-coded player will be able to handle them all equally well.

Scripted bots and their predetermined behaviour are susceptible to being overly repetitive or unrealistic, especially if the bots find themselves in a situation that the developers did not foresee. Stochastic systems can be used to introduce some variety into the behaviour of artificial players, but they may offer only slight variation to some predetermined strategy. Too much variation has the potential for creating seemingly random or irrational behaviour which adversely affects a human player’s sense of immersion in the game environment.

Another common limitation of current game AI is

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

that teams of agents tend to be overly self-interested. While many good agents may be useful for a team, this is very different from team-interested agents who can understand and prioritise the good of the team over individual gain. Without team based learning, artificial players run the risk of being overly “greedy” to the detriment of the team. No matter how well the individual parts may be tuned, certain team strategies may never arise — a self-interested individual would not sacrifice itself to draw fire away from teammates or to lead opponents into an ambush, for example. Team based learning is useful where the goal to be accomplished is too complex to be achieved by individuals without team coordination, RoboCup soccer (Kitano 1997) is a good example.

The real-time learning and continuous adaptation of a team of artificial agents is desirable for a number of reasons. An agent capable of real-time learning would be inherently robust just as strategies learnt offline are inherently exploitable. Ideally, an adapting agent could be expected to perform in situations never considered by the game developers. Quinn et al demonstrated the use of a real-time evolutionary learning system for the task of cooperative and coordinated team behaviour for robots (Quinn 2002). The aim was for the team to move to a new location while remaining within sensor range of each other all times. Despite being a relatively simple task, it is an encouraging result.

Our previous work in the domain of Pacman (Wittkamp 2008) was a proof-of-concept study in “simulated” real-time — that is, the learning was continuous and took place in parallel with the agents acting in the environment, but the environmental simulation was paused to allow the learning system to explore strategies. This paper takes the next step and investigates to what extent sufficient learning is possible in real-time. We explore the use of computational intelligence techniques for real-time learning in a simple prey and predators domain. Focusing on team-work development, we examine how these techniques can be used to evolve strategies for a team of predators aiming to capture a single prey opponent.

The real-time system we propose makes use of continuous short-term learning to regularly update predator strategies. Our approach aims to parallelise offline learning through lookaheads and simulations with actual game play. Constant adaptation over short time periods means the predators need not learn complex general strategies, but rather focus all attention on current the state of the game.

## 2 The Iterative Real-time Team Learning System

Our real-time learning system is a novel implementation of an Evolutionary Algorithm, designed to run in parallel with the game environment and to iteratively evolve a team of agents via an analogy of Darwinian selection. Learning takes place continuously within discretised time slices; during each time slice, a role is selected for training.

The system first looks ahead to the predicted state at the start of the next time-slice ( $ES_{t+1}$ ). This state is used to determine which role to train and from which population (each role maintains its own population). Each time-slice, a single role is trained in a round-robin fashion. How these roles map to the agents is up to the implementation, but for this study we use a direct one-to-one mapping of each role to a unique predator. It may be advantageous to organise the mapping of roles to predators in a more meaning-

ful way (such as by distance to the prey) and then automatically switch the strategies used by predators as their circumstances change, but we plan to address these considerations in future work.

The lookahead state ( $ES_{t+1}$ ) contains the expected state of the environment and all agents one time-slice into the future. The learning system has access to the predator strategies, and also the prey strategy — that is, simulations run have accurate models of how the environment and all agents contained will behave. When training a particular role, the role is replaced in the lookahead state and then a simulation from this state ( $ES_{t+1}$ ) is completed. Even though only a single predator is training during any given time slice, the fitness measure used evaluates the team as a whole rather than sanctioning the individual directly. The individuals in the population are each evaluated by their contribution to the predator team’s predicted performance at the end of the next time slice. The evolutionary algorithm uses this performance data to create the next generation of strategies.

The evolutionary process takes place in real-time, in parallel with actual events in the game environment. As many generations as possible are completed during the time slice, with the fittest individual from the evolving population being used to replace the role for play in the next time slice. We use the same fitness function as that of (Yong 2001) as described below where  $d_0$  is the sum of all predator’s starting distances to the prey, and  $d_e$  is the sum of the ending distances. The system is depicted visually in Figure 1 and written up as pseudo-code in Algorithm 2.1.

$$f = \begin{cases} d_0 - d_e/10 & \text{if prey not caught} \\ 200 - d_e/10 & \text{if prey caught} \end{cases}$$

We use an elitist selection scheme where the top half of the population reproduces by one-point crossover and mutation to replace the bottom half of the population. Mutation is applied randomly to a single weight of the individual, with 0.4 strength. We cap our simulation time at 600 game ticks (roughly 40 seconds). Though we are interested in completing a capture far sooner than that, we allow the simulations to run up to 600 game ticks for data collection purposes.

We allow our learning system to have access to a perfect simulation model. While playing, the predators do not explicitly communicate. For the lookahead and training simulations the predator currently undergoing a learning cycle has access to every other predator’s agent model. That is, the learning system will have a perfect understanding of what each of its team mates will do in any given scenario and these are used to train a predator. This is possible due to predators being completely deterministic given any scenario.

Given the learning system’s intimate knowledge to all agents’ strategies and that the game environment is completely deterministic, the prey opponent model is the only remaining uncertainty in the lookahead and simulation process. In this paper, in order to completely remove noise in our simulations, we assume access to a perfect model of the prey opponent. Having a perfect opponent model is no small assumption, but the aim of this paper is to demonstrate the effectiveness of our real-time learning system compared to an offline approach. If our predators were learning offline by training against a particular prey, then the offline learning system would also have access to a perfect prey model. In a real-time scenario this may be infeasible because the opponent may be

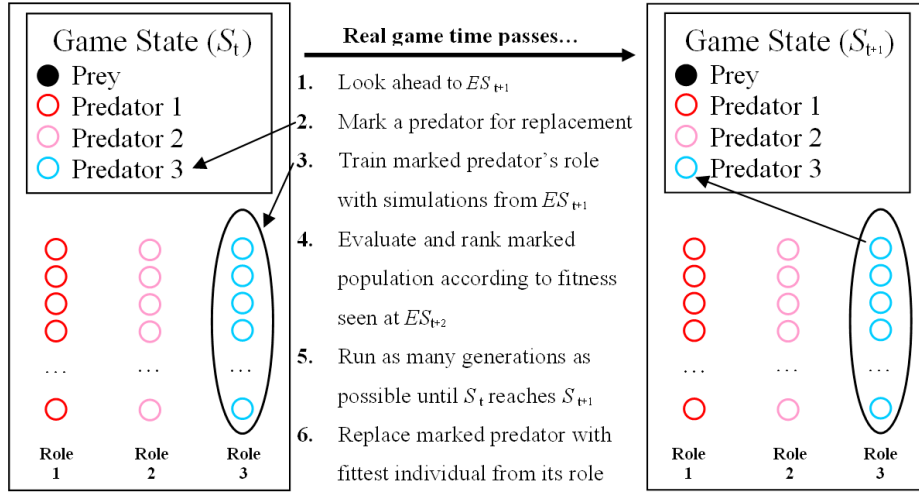


Figure 1: Pictorial representation of the real-time learning system

“black box” or simply not available for use in simulations — consider the case when playing against a human opponent in real-time. Section 6 discusses our intended future work with respect to inaccurate opponent models and other sources of simulation noise.

### 3 Experimental Domain

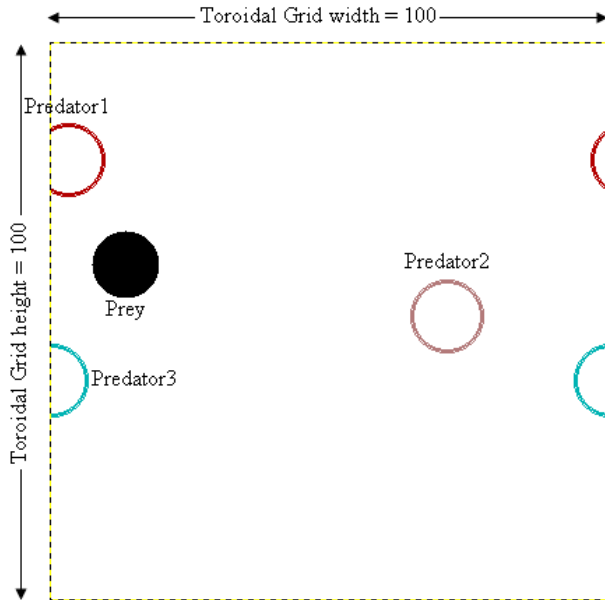


Figure 2: The prey and predators environment

We have developed a system for learning effective team strategies in real-time as a game is being played. We allow for no prior offline learning; all learning takes place while the game is being played. To test our system, we use the prey and predators domain studied in (Rawal 2010, Yong 2001). We are interested in evolving a team of predator strategies to

coordinate their movements to trap and capture the prey in real-time.

#### 3.1 Prey and Predators Environment

The game environment we use is closely modelled from that of (Yong 2001). In this predators-prey environment, we have a single prey and a team of 3 predators. The goal of the predators is to catch (making contact with) the prey. The prey’s aim is simple; avoid being caught by the predators.

We are interested in training the team of predators in real-time to cooperate with each other towards the goal of catching the prey. In all but one experiment the predators and prey move at the same speed, thus making the task of capturing any competent prey impossible without some degree of cooperation — in the remaining experiment, the prey is given a powerful advantage by being able to move at 3 times the speed of the predators.

The environment for all experiments is a  $100 \times 100$  toroidal grid without obstacles where agents (prey and predators) are represented by circles of radius 6. In this environment a simple hand-coded prey could quite easily evade 2 predators indefinitely, thus the task of capturing the prey will need the cooperative actions of all 3 predators working together. The initial setup places the 3 predators in a corner of the toroid grid (being a toroid, they are all one and the same) and the prey is randomly positioned. The speed of all agents is fixed — each is either moving at this speed or stationary; there is no in between.

A predator travelling across the toroid diagonally from corner to corner (the longest straight-line path across the toroid) takes 150 *game ticks*, which takes 10 seconds in real-time. This time was chosen as this seemed a realistic speed for the game if it were made to be playable by a human. What this means is that in the time taken for a predator to cover this distance, there are 150 decision points for every agent. The number 150 was chosen to match that of (Yong 2001) for which we aim to compare results.

**Algorithm****2.1: REAL-TIME EVOLUTIONARY TEAM LEARNING SYSTEM()**

```

comment: Initialise a population ( $P_r$ ) of individuals for each identified role ( $r$ )
for each  $r \in \text{Environment.Roles}$ 
  do  $\{P_r \leftarrow \text{CREATEPOPULATIONOFINDIVIDUALS}(\ )$ 

for each  $t \in \text{Time} - \text{slices}$ 
  comment: Capture the current state of Environment to  $S_t$ 
   $S_t \leftarrow \text{Environment.GETSTATE}(\ )$ 

  comment: Look ahead from the captured state to the next expected state  $ES_{t+1}$ 
   $ES_{t+1} \leftarrow \text{LOOKAHEAD}(S_t, \text{OpponentModel})$ 

   $\text{MarkedRole} \leftarrow \text{CHOOSEROLE}(ES_{t+1})$ 

  for  $g \leftarrow 1$  to  $\text{NumGenerations}$ 
    in parallel for each individual  $i \in P_{\text{MarkedRole}}$ 
      do  $\left\{ \begin{array}{l} \text{StartStates}[i] \leftarrow ES_{t+1} \\ \text{StartStates}[i].\text{REPLACEROLE}(\text{Environment.Roles}[\text{MarkedRole}], \\ P_{\text{MarkedRole}}[i]) \\ ES_{t+2}[i] \leftarrow \text{RUNSIMULATION}(\text{StartStates}[i]) \end{array} \right.$ 

    do  $\left\{ \begin{array}{l} \text{comment: Evaluate } P_{\text{MarkedRole}} \text{ by inspecting expected end states } (ES_{t+2}) \\ \text{Fittest} \leftarrow \text{EVALUATE}(P_{\text{MarkedRole}}, ES_{t+2}) \\ \text{comment: Evolve the next generation of individuals for } P_r \\ \text{EPOCH}(P_r) \end{array} \right.$ 

   $\text{Environment.REPLACEROLE}(\text{Environment.Roles}[\text{MarkedRole}], \text{Fittest})$ 

```

**3.2 Hand-coded Prey Controllers**

In order to test our domain we have created some hand-coded opponents capable of evading the predators to varying degrees. The 3 different prey strategies we have created are *Simple*, *Repelled*, and *Fast*. These are listed in increasing order of how difficult they are to capture, as confirmed by the results in Section 4.1.

1. **Simple:** our most basic of preys; its strategy is to always head directly away from the predator closest to it. This prey always travels at the same speed as the predators. The Simple opponent is based on the description of the prey opponent used in Yong and Miikkulainen's work (Yong 2001); we use this prey as a simple starting point and to allow more meaningful comparisons between our approaches.
2. **Repelled:** a more complex prey that aims to avoid predators proportionate to their proximity. For all predators, the prey applies a force of repulsion equal to  $1/d^2$  in the direction of the predator, where  $d$  is the minimum toroidal distance from the prey to that predator. This prey moves at the same speed as the predators, heading in a direction determined by the sum of the repulsive forces. Our aim in creating the Repelled prey was to create a strong training partner for the bulk of our experiments, after initial experiments seemed to indicate that capturing the *Simple* prey did not sufficiently challenge our system.

3. **Fast:** a prey that employs the same strategy as the Repelled prey, but one that travels at 3 times the speed of the predators rather than at the same speed. This prey provides a very difficult capture task intended to push our learning system beyond its limits.

**3.3 Predator Controller**

A predator takes the form of a randomly initialised feed-forward neural network with 2 inputs, 5 outputs, and a hidden layer of size 10. The only inputs to the predators are their x and y toroidal distances to the prey. The predator's x and y coordinates on the toroidal grid do not factor into its decision making process. The outputs are *North*, *South*, *East*, *West* and *Stationary*.

The predator will remain still if the *Stationary* output exceeds that of all other outputs. Otherwise, the difference between the East and West outputs determines the x component of the predator's direction vector and the difference between North and South determines y. The predator travels at a fixed speed, equal to that of the Simple and Repelled prey types (and one third the speed of the Fast prey). While this network representation could be used to define an agent that is capable of varying its speed, here we are only using it to describe the predator's direction, not magnitude. Like the prey, predators will always be either motionless or travelling at their predefined maximum speed.

## 4 Experiments and Results

### 4.1 Prey Strategy Evaluation

We have designed the Simple, Repelled, and Fast prey to be used as training partners to our real-time system. These strategies are described in Section 3.2. In this experiment we aim to confirm that the 3 prey strategies have a range of skill levels that make the problem increasingly difficult. We trial the prey against our real-time system with a fixed configuration. The experimental setup uses a population of 200, running for as many generations as real-time will allow — on average, the system made it through roughly 33 generations per time-slice.

Elapsed time $n$ (game ticks)	Simple prey	Repelled prey	Fast prey
100	14	1	0
200	88	24	3
300	100	58	6
400	100	76	11
500	100	81	16
600	100	86	22

Table 1: Percentage of runs resulting in capture against various prey strategies by  $n$  game ticks.

Table 1 shows the results of each prey performing against our real-time adaptive predator team in an identical experimental setup averaged over 100 runs. The rates of capture are reported for various points of elapsed time and are therefore cumulative.

As expected, the Simple prey is the easiest strategy to capture. By 260 game ticks the real-time system managed captured the Simple prey in all 100 runs. Even at the time the simulations were capped at 600 game ticks, 100% was not achievable for this experiment against either the Repelled or the Fast prey, indicating that the Simple prey is clearly the least formidable opponent.

The real-time system took much longer to form an effective counter strategy to the Repelled prey than it took against the Simple prey. In the time that the Simple prey was completely dominated, the Repelled prey was only being captured 46% of the time, and reached an ultimate capture rate of 86% after 600 game ticks.

The Fast prey, employing the same strategy as the Repelled prey but at triple the speed, is clearly the most difficult prey to capture. This prey has the unfair advantage of being able to travel at 3 times the speed of the predators. The real-time system only manages to achieve capture in 22% of games after 600 game ticks — far lower than that achieved against the other hand-coded prey opponents. The aim in designing this opponent was to purposely create an extremely difficult task for our system; the results suggest that we have succeeded; this is indeed a very difficult prey to capture.

The results show that the real-time system is certainly very capable of producing effective predator team strategies in order to catch the prey without any prior learning. Within the time taken to move from one corner to the other (150 game ticks), the real-time controlled predator team manages to capture the Simple prey 60% of the time; this is a good result. Recall that 2 predators are not capable of capturing even the Simple prey and, due to the iterative learning construct of our system, it is not until after 3 time-slices (120 game ticks) that the real-time system

has been given an opportunity to learn a strategy for each of the 3 predators.

This experiment’s configuration was arbitrarily selected as a means of comparing the hand-coded prey strategies and to confirm that they are increasingly difficult prey to capture as intended. To observe a 100% capture rate being achieved after 260 ticks against the Simple prey is a most encouraging result.

The real-time evolved predator team manages to capture the more advanced Repelled prey strategy in 86% of cases and even manages to capture the Fast prey (a prey moving at 3 times the speed of the predators) 22% of the time. As previously mentioned, this experiment was not geared towards testing our hand-coded prey strategies and establishing a baseline, but rather towards achieving the most optimal configuration for learning. We expect our system’s performance to improve in Section 4.2, when we aim to determine how the length of our time-slices affects learning performance.

### 4.2 Time-slice Experiment

In this experiment, we investigate the effect of varying the length of the time-slice. The time-slice length affects both the rate at which new strategies are “plugged in” to the game as well how long the fitness evaluations are run. As the game progresses, learning takes place continuously across time slices, with each slice marking an insertion point for learnt strategies into the game.

Which length we use for the time slice has the potential to substantially impact the learning system. The length of the lookahead and the time taken until all predators have been given an opportunity to train are very significant factors both implied by the selection of a time-slice length.

Consider the case where we train using a time-slice of length 20. Random strategies are plugged in for all 3 predators and the training begins at 0 game ticks. The learning system looks ahead to the expected state of the game at 20 ticks, from here simulations begin in parallel for as many generations as time permits until the game reaches 20. At this point, the first predator strategy is inserted into the game and learning continues for the next predator which will be inserted into play at 40 game ticks, and then the final predator at 60. If our time-slice length was 40, then our learnt strategies would be more likely to see past myopic optima and be able to develop more effective long term strategies. However, we would be forced to wait until 120 game ticks until all predators had been given an opportunity to learn; an inherent tradeoff is seen.

In Figure 3 we see that all time-slice lengths except for the extremes of 20 and 200 managed to achieve a 100% capture rate after 600 game ticks against the Simple prey. When running our real-time system using a time-slice length of 20, 95% capture is achieved, and 96% for a time-slice length of 200. The fact that the system does not reach 100% capture after 600 game ticks under a time-slice length of 200 is not surprising at all. The 3rd predator strategy is only plugged in at 600 game ticks, meaning that only 2 of the 3 predators have had an opportunity to learn at the game’s end. From 400 game ticks onwards the system is running with 2 learned predators and 1 predator still unchanged from its original random initialisation. Impressively, the high performance result shows that 2 trained predator strategies are able to make use of their randomly initialised team-mate.

The rate of learning for the real-time system appears to be the same against the Simple prey across all time-slice lengths. The general pattern we see is a



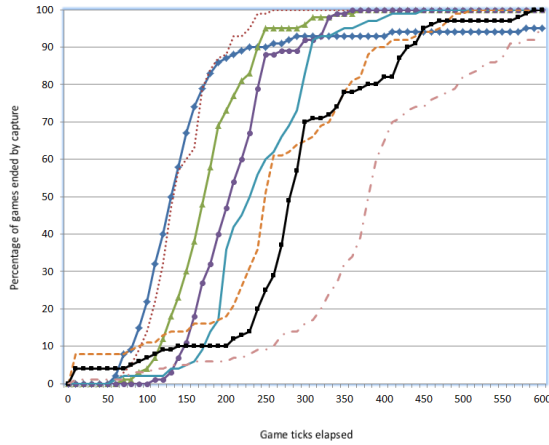


Figure 3: Effect of time-slice length on performance against Simple prey

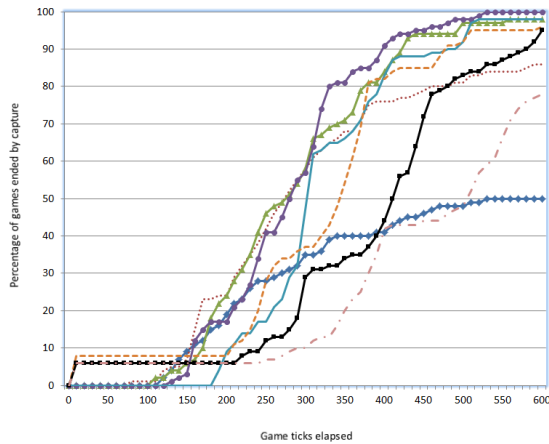


Figure 4: Effect of time-slice length on performance against Repelled prey

steep rise at about 3 times the time-slice length (once all predators have been given an opportunity to learn a strategy). The longer time-slice length runs suffer inherently because the time between insertions is longer, and the time until all 3 predators have learnt a strategy is also longer.

Against the Repelled prey, the most effective capture strategy at 150 game ticks was trained using a time-slice of 20, despite it not performing as well as the others, ultimately. A shorter time-slice length allows all predators to get through a learning cycle much sooner, but they will be limited in their understanding of the environment due to the short time-slice length. This almost always manifests itself in all 3 predators being bunched together and chasing the prey around the toroid over and over again as seen against the Repelled prey using a time-slice length of 20 in Figure 6. With an identical starting point (all predators beginning at the same location) but instead using a time-slice length of 80, after a few time-slices the predators learn to surround the prey and eventually capture as seen in Figure 7. While not as stark a relationship as we had expected, the tradeoff between forming effective strategies and the delay of actually being able to utilise them in the actual game is apparent.

If the time-slice is too short then the rewards of certain strategies are too distant to be recognised by the fitness function and thus will never be used as

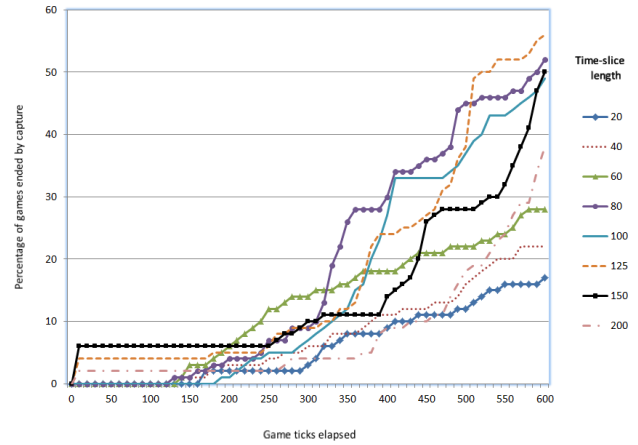


Figure 5: Effect of time-slice length on performance against Fast prey

predator strategies in actual play — i.e. the learning becomes trapped within local optima. For example, consider the case where all predators have converged upon chasing the prey and effectively acting as a single predator. If we now wish to evaluate a candidate predator strategy that breaks away from the other predators and head in the opposite direction in order to cut off the prey from the other side of the toroid. If the time-slice is too short then the fitness would be evaluated at a time where this predator had broken away from the others but had not yet caught up with the prey from the other side. The fitness function would then find this strategy to be ineffective because it is unable to see past the temporary hit to the fitness function required to make an improvement.

We observe that a steep improvement, relative to time-slice length, occurs earlier when the time-slice length is longer. One would expect to see a strong performance increase after the third time-slice because this is when all predators have had a chance to learn. With longer time-slices, we see that this increase occurs *before* the final predator strategy is plugged into the game. This suggests that with a longer time-slice to train under, the 2 predators are able to formulate strategies that are highly effective and able to make use of the randomly initialised strategy still being used by the third predator.

The time-slice is so long that at most only 2 predators have had a chance to learn. By the time the final predator strategy has been trained and is ready to be plugged into the game at 600, the simulation is over. The trend seems to be that longer the time-slice, the better the resulting team-strategy. Also, the more difficult an opponent is to capture, the more benefit can be expected from increasing time-slice length. From Figure 5, against the Fast prey, we see a far more varied performance result at the end of 600 game ticks.

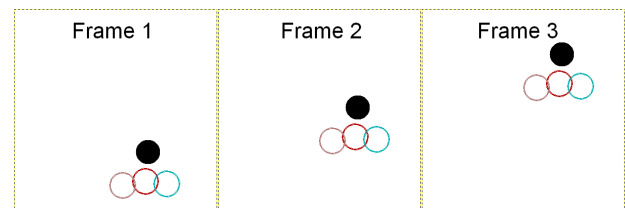


Figure 6: Short-sighted predator strategies fail to capture the Repelled prey using a time-slice length of 20 game ticks.



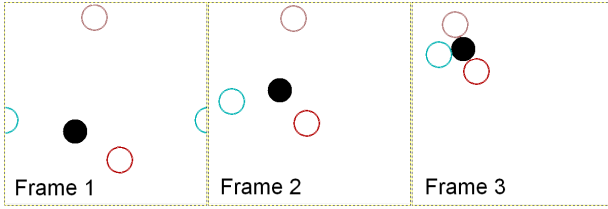


Figure 7: Predator strategies converge and capture the Repelled prey using a time-slice length of 80 game ticks.

For a counter strategy to the Simple prey, Figure 3, we observe that 20 ticks is not enough to produce an effective counter strategy. At 40 ticks, the predators are doing a lot better — reaching 100% capture rate after 260 game-ticks when training in real-time. Against the more difficult Repelled prey, 40 ticks was no longer enough to achieve a high capture percentage.

These results are in tune with what we would expect; with a more competent prey there would be more benefit to having a longer time-slice length. This is because a longer time-slice length means that predators are evaluated based on relatively long-term performance, encouraging and avoiding short term strategies that may be trapped within local optima. Indeed, this is exactly what we observe in the graph against the Simple opponent — we saw that a time-slice length of 20 game ticks was insufficiently short, and that a length of 40 seemed about right. Against the Repelled prey, we now observe that 20 ticks is far too short, achieving only a capture rate of 50% by the end of the game.

### 4.3 Real-time vs Paused

In this experiment, we compare our real-time system to a “paused” version of the same system. Our real-time system follows Algorithm 2.1, completing as many generations as it is able to within the time available.

The paused version works almost exactly the same, except it is guaranteed to complete a predetermined number of generations. This is possible because the game is paused in its execution at the end of each time-slice and waits for the learning to complete its desired number of generations. The paused version is able to tell us how well we can expect our real-time version to perform given more powerful hardware that is capable of running more simulations in a given time.

Elapsed time (ticks)	Percentage captured
40	0.44
80	4.11
120	26.67
160	62.44
200	82.33
240	91.22
280	95.89
320	98.44
360	99.22
400	99.78
440	99.89

Table 2: Real-time adaptive prey’s performance in the benchmark cases from (Yong 2001).

Tables 4.3 and 4.3 show the result of both the paused and real-time versions of our system playing against the Repelled and Fast prey strategy, respectively. What we found was that, there was no statis-

tically significant difference between the paused and real-time systems in performance. The only statistically significant difference is in Table 4.3 against the Fast prey and for a population size of 10. In this instance, the paused version performs better.

These results are extremely encouraging; the real-time system manages to do just as well as the paused system against our best prey agents. What we found with the real-time version was that simulations were not very computationally expensive at all for this game, but the operations of creating a new generation from the previous one is. This explains why the real-time system manages to get through so many total simulations when the population size is high.

### 4.4 Comparison with Yong and Miikkulainen

Here we compare our real-time learning system with the offline approach of (Yong 2001). We train in real-time but then freeze our learnt strategies and pit them against 9 fixed starting states to assess how applicable the team strategies are to general situations. We model our game environment as close as possible to that of this study in order to make comparisons as meaningful as possible.

Yong and Miikkulainen experimented with a distributed and central control system for the prey. One conclusion made was that a distributed system was more effective than a centralised approach. Also, communication between predators was deemed unnecessary and that it overburdened the learning system. Without communication between the evolving predators, they learned faster and performed better with the emergence of more distinctive roles. The best predator strategies from (Yong 2001) were trained in the non-communicating, distributed system which we will refer to as the Yong-Miikkulainen system, from this point on.

Yong and Miikkulainen trained their predator teams using 1000 trials per generation, with each trial being 6 simulations against prey beginning in a randomly determined position. A layered learning approach was used by Yong and Miikkulainen, which saw predator strategies being evolved in 6 stages: from a stationary prey, to incrementally faster prey until reaching the same speed as the predators. Our real-time system does not have the time to implement a layered learning approach such as this, so our system must tackle the full speed prey immediately. Once learnt, the strategies were tested in a set of 9 benchmark cases to determine how effective the predator team is at capturing the prey. These benchmark cases involve the predator teams all beginning in the corner of the toroid, with the prey beginning at 9 evenly spaced starting positions on the toroid. The 9 positions the centre-positions of each sub-square when the toroid is split up into a  $3 \times 3$  grid.

Table 4.3 shows the result of our real-time adaptive predator team when thrown into the same benchmarking as used in (Yong 2001). In these benchmark cases, the Yong-Miikkulainen system took an average of 87 generations to be able to solve 7 out of 9 benchmark cases, and this was done within 150 game ticks. At 150 ticks our system averaged 54% capture. Our real-time system falls slightly behind in this result, managing to achieve 7 out of 9 a bit later at 190 game ticks.

When compared in this way the real-time system falls behind. At each time-slice of 40 ticks one predator is given the opportunity to train so it is not until 120 game ticks that the real-time system is operating with a full set of learnt predator strategies. The team trained under the Yong-Miikkulainen approach

	Population size	Generations	Total simulations	Average capture time
Paused	10	500	5000	348.63
Real-time	10	48.81	488.61	336.21
Paused	50	100	5000	275.61
Real-time	50	45.64	2281.88	309.09
Paused	100	50	5000	315.72
Real-time	100	42.31	3287.38	311.84
Paused	200	25	5000	293.53
Real-time	200	32.87	8461.38	307.91
Paused	500	10	5000	296.41
Real-time	500	25.23	12612.86	307.88
Paused	1000	5	5000	274.03
Real-time	1000	11.36	11358.83	272.84
Paused	2500	2	5000	271.29
Real-time	2500	5.91	14781.84	281.39
Paused	5000	1	5000	269.76
Real-time	5000	2.24	11204.0	274.86

Table 3: Real-time vs Paused, against Repelled prey

	Population size	Generations	Total simulations	Average capture time
Paused	10	500	5000	510.97
Real-time	10	49.94	499.41	557.95
Paused	50	100	5000	536.61
Real-time	50	46.09	2302.43	518.63
Paused	100	50	5000	503.53
Real-time	100	43.13	4312.98	515.86
Paused	200	25	5000	503.23
Real-time	200	36.97	7393.77	521.36
Paused	500	10	5000	525.86
Real-time	500	23.68	11840.86	498.61
Paused	1000	5	5000	467.13
Real-time	1000	14.33	14326.31	500.19
Paused	2500	2	5000	498.14
Real-time	2500	5.62	14043.09	470.78
Paused	5000	1	5000	499.63
Real-time	5000	2.11	10557.44	532.60

Table 4: Real-time vs Paused, against Fast prey

has all 3 predators plugged in from the start because of the offline learning it has already undertaken. If we consider the elapsed time of 270 ticks ( $120 + 150$ ) so that our real-time system has had the opportunity to play the same number of game ticks with a full team of predator strategies, then it achieves a 95.67% capture rate.

The Yong-Miikulainen predator team had its population of 1000 individuals run through an average of 87 generations 6 times (each evaluation consists of play in 6 random games). The total number of simulations to achieve 7 out of 9 capture in the benchmarks is 522000. The moment when our real-time system achieves 7 out of 9 in the bench mark cases is at 190 game ticks. At this time, the system has run our population of 2500 individuals through an average of 6.1 generations 4 times (once for each time-slice that has completed). This is a total of about 61100 total simulations, fewer than one eighth that required by the best predator team of (Yong 2001).

Our system uses the same neural network inputs and outputs as (Yong 2001) and the same training partner (the Simple prey), and achieves the same performance in less than one eighth of the simulations, and in real-time. The one weakness of our system is that the initial learning takes time to slot predator strategies into the game. While (Yong 2001) finished their simulation at 150 game ticks, their predators were all employing their strategies from the onset.

At 150 ticks, our system averaged 54% captures but, of course, has spent a considerable portion of these ticks learning strategies for its predators.

#### 4.5 Generalisation

In Section 4.3 we demonstrated that our real-time system was able to evolve team strategies to play with no prior learning.

In this experiment, we allow the predators to evolve in real time (one after another in their allocated time-slices) and then “freeze” the predators’ learning once the game is over. The “frozen” (no longer learning) predator strategies are then tested in a number of different starting configurations. This experiment is to determine how effective the frozen predator strategies are for the game in general.

We take the strategies learnt in real-time from Section 4.2 and freeze their learning after the game has come to an end. The predators are placed in 9 new game environments, with only the starting position of the prey being varied in each in order to assess the team’s ability for general play in the prey and predators domain against an identical prey. The 9 prey starting positions are such that if the toroid were to be divided into a  $3 \times 3$  grid as described in Section 4.3, the prey begins in the center of each cell. We run trials against all 3 prey types, each time training against that particular prey in real-time and then testing for

general ability against that same prey in each of the starting positions.

Averaged over 100 runs of the real-time training, we recorded the total capture rate at various points in the game's play time. The results across all 3 prey types are similar — the strategies learnt in real-time completely fall apart when frozen and placed in the 9 new game environments. Not a single capture was achieved in any experiment, even in those where real-time learning routinely achieved perfect or near-perfect capture. While this may at first seem like a negative result, it is exactly what we expected to see.

When training in real-time, the predators vary their strategies to restrict the prey, and slowly surround and close in on the prey. At the time of capture, at least 2 of the prey (and most often, all 3) converge on the strategy of heading directly towards the prey. The reason this strategy is effective at the end is due to the higher level strategies of surrounding and restricting the prey's movement that were learnt earlier and have since been discarded. When these predators' highly specialised end-game strategies are then frozen and placed in new game environments, what typically results is the same endless traversal of the toroid that we observed in Section 4.2.

The benefit of our real-time system is that the strategies formed are not necessarily robust or sound strategies for the game in general and can thus be simple to learn. A general strategy must be complex enough to deal with every game situation it may encounter; learning in real-time through continuous adaptation allows the system to learn highly situation-specific strategies without being overburdened by being required to learn how to play in all other situations. The predator strategies that are evolved in real-time are constantly changing to match the current state of the game environment. To develop this level of specialisation offline for all possible game scenarios would require far more learning and a more complex predator representation. The team strategies learnt in real-time always perform badly in the 9 scenarios due to the final strategy that the predators had at the time they were frozen.

The resultant strategies are bound to the specific game scenario at the time it is encountered. The team strategies become so specifically tailored to the task at hand that any hope for generality is lost. Depending on the domain, a system that behaves well in general may be very difficult to create and may not even be possible. A real-time learning system that can change itself to new game conditions alleviates the need to solve such a difficult task, when all that is required is for the predators to be able to focus on what it needs to do, when it needs to do it.

## 5 Conclusion

The results of this study are extremely encouraging. We have shown that the real-time team strategy is able to learn, in a reasonable amount of time to capture hand-coded prey of varying degrees of difficulty. It is capable of achieving competitive results with the paused version.

Not surprisingly, the strategies formed do not make for very robust, general strategies. The strategies formed by the real-time system are extremely specialised to whichever situation currently presented to it. The strength of this system is in its ability to adapt. This real-time learning system simulates a higher level strategy capable of handling many different situations, and indeed, situations that it has never seen before.

## 6 Future Work

For this paper, we have shown that our system is capable of discovering real-time cooperative strategies for the task of controlling predators in the Prey and Predators domain. The results of these experiments depend on the learning system's access to a perfect opponent model. This is ultimately an unrealistic assumption for the problem that we wish to extend our approach into — play against a human-controlled prey opponent. How accurate must our opponent model be before our learning system is able to form effective team counter-strategies? How (and when?) do we try to improve our opponent model in real-time? In future work, we intend to investigate the answers to these questions among others.

For ease of implementation, one thing we have done is to run simulations for the length of a time-slice. This means that the predator strategies are trained to play for the length of one time-slice but because there are 3 predators, these strategies are actually in play for 3 time-slices. Since a long time-slice length hinders the initial learning curve so much a possible solution would be to have initially short slices which could grow once all predators have had the opportunity to learn at least some sort of strategy. Or, rather than being discretised to these time-slices, it may be beneficial to have an "any time" approach to when new strategies could be plugged into the game.

Intuitively, it seems to make sense to train in the environment that one wishes to play in, however there are several reasons why this may not be the best or only solution. For real-time learning we are greatly restricted in how much training we can get through; increasing the simulation length will put even more strain on our system and limit how many simulations we can get through. When combined with our goal to account for noise in the opponent model, a longer simulation length may be far too noisy to provide any benefit at all. A successful approach may be one that has a variable time-slice length depending on the perceived accuracy of the opponent model.

## References

- M. Wittkamp and L. Barone: Evolving adaptive play for the game of spooof using genetic programming, in Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games, IEEE Publications.
- M. Wittkamp, L. Barone, and L. While: A comparison of genetic programming and look-up table learning for the game of spooof, in Proceedings of the 2006 IEEE Symposium on Computational Intelligence and Games, IEEE Publications.
- Wittkamp, M. and Barone, L. and Hingston, P.: Using NEAT for continuous adaptation and teamwork formation in Pacman. In: Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On, pp. 234–242, Perth.
- Quinn, L. Smith, G. Mayley, and P. Husband: Evolving teamwork and role allocation with real robots, In Proceedings of the 8th International Conference on The Simulation and Synthesis of Living Systems (Artificial Life VIII), 2002.
- ML. Berger: Scripting: overview and code generation, in AI Game Programming Wisdom. MIT Press, 2002, vol. 1, pp. 505510.

- H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa: RoboCup: the robot world cup initiative, in Proceedings of the First International Conference on Autonomous Agents (Agents'97). ACM Press, 58, 1997, pp. 340347.
- P. Tozour: The Perils of AI Scripting, Charles River Media, Inc.
- Yong C. and Miikkulainen, R.: Cooperative Coevolution of Multi-Agent systems, University of Texas, Technical Report, 2001.
- Yong C. and Miikkulainen, R.: Coevolution of role-based cooperation in Multi-Agent systems, IEEE Transactions on Autonomous Mental Development, 2010.
- Rawal, A. and Rajagopalan, P. and Miikkulainen, R.: Constructing Competitive and Cooperative Agent Behavior Using Coevolution, 2010.
- Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences, 1981. J. Mol. Biol. 147, 195–197.
- May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg.
- Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure, 1999. Morgan Kaufmann, San Francisco.
- Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, 2001, pp. 181–184. IEEE Press, New York.
- Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration, 2002. Technical report, Global Grid Forum.
- D. Charles, C. Fyfe, D. Livingstone, and S. McGlinchey: Biologically Inspired Artificial Intelligence for Computer Games, Medical Information Science Reference, 2007.
- National Center for Biotechnology Information: <http://www.ncbi.nlm.nih.gov>

# Trends in Suffix Sorting: A Survey of Low Memory Algorithms

Jasbir Dhaliwal<sup>1</sup>

Simon J. Puglisi<sup>1</sup>

Andrew Turpin<sup>2</sup>

<sup>1</sup> School of Computer Science and Information Technology  
RMIT University,  
Melbourne, Australia,  
Email: {jasbir.dhaliwal,simon.puglisi}@rmit.edu.au

<sup>2</sup> Department of Computing and Information Systems  
University of Melbourne,  
Melbourne, Australia,  
Email: aturpin@unimelb.edu.au

## Abstract

The suffix array is a sorted array of all the suffixes in a string. This remarkably simple data structure is fundamental for string processing and lies at the heart of efficient algorithms for pattern matching, pattern mining, and data compression. In many applications suffix array construction, or equivalently suffix sorting, is a computational bottleneck and so has been the focus of intense research in the last 20 years. This paper outlines several suffix array construction algorithms that have emerged since the survey due to Puglisi, Smyth and Turpin [ACM Computing Surveys 39, 2007]. These algorithms have tended to strive for small working space (RAM), often at the cost of runtime, and make use of compressed data structures or secondary memory (disk) to achieve this goal. We provide a high-level description of each algorithm, avoiding implementation details as much as possible, and outline directions that could benefit from further research.

**Keywords:** suffix array, suffix sorting, Burrows-Wheeler transform, suffix tree, data compression

## 1 Introduction

Strings are one of the most basic and useful data representations, and algorithms for their efficient processing pervade computer science. A fundamental data structure for string processing is the suffix array [Manber and Myers, 1993]. It provides efficient – often optimal – solutions for pattern matching (counting or finding all the occurrences of a specific pattern), pattern discovery and mining (counting or finding generic, previously unknown, repeated patterns in data), and related problems, such as data compression. The suffix array is widely used in bioinformatics and computational biology [Gusfield, 1997, Abouelhoda et al., 2004, Flicek and Birney, 2009], and as a tool for compression in database systems [Chen et al., 2008, Ferragina and Manzini, 2010]. More recently it is beginning to move from a theory to practice as an index in information retrieval [Culpepper et al., 2010, Patil et al., 2011].

In all these applications the construction of the suffix array — a process also known as suffix sort-

ing — is one of the main computational bottlenecks. Suffix array construction algorithms (SACAs) therefore have been the focus of intense research effort in the last 15 years or so. The survey by Puglisi et al. [2007] counts 19 different SACAs, and in the last five years even more methods have emerged. The trend in these more recent algorithms has been to use as little memory as possible, either by finding a clever way to trade runtime, or by using compressed data structures, or by using disk, or some combination of these techniques. It is these recent “low memory” SACAs which are our focus in this paper.

In the next section we set notation and introduce basic concepts to be used throughout. This overview can be safely skimmed by readers already familiar with suffix sorting, but may serve as a useful tutorial for those new to the problem. Section 3 describes the new algorithms in turn, illustrating each with a worked example. A snapshot experimental comparison is offered in Section 4. We then outline some directions future work might take.

## 2 Background

Throughout we consider a string  $x$  of  $n$  characters (or symbols),  $x = x[1..n] = x[1]x[2]...x[n]$ , drawn from a fixed, ordered alphabet  $\Sigma$  of size  $\sigma$ . The final character,  $x[n]$ , is a special end-of-string character,  $\$$ , which occurs nowhere else in  $x$  and is lexicographically (alphabetically) smaller than any other character in  $\Sigma$ . The string  $x$  requires  $n \log \sigma$  bits of storage without compression.

For  $i = 1, \dots, n$  we write  $x[i..j]$  to represent the substring  $x[i]x[i+1]...x[j]$  of  $x$  that starts at position  $i$  and ends at position  $j$ . We write  $x[i..n]$  to denote the suffix of  $x$  of length  $n-i$ , that is  $x[i..n] = x[i]x[i+1]...x[n]$ , which we will frequently refer to as ‘suffix  $i$ ’ for simplicity. Similarly a *prefix* is a substring of the form  $x[1..i]$ .

The *suffix array* of a string  $x$ , which we write as SA, is an array containing all the suffixes of  $x$  sorted into lexicographical order. Suffixes are represented as indices into the original string, and thus, the suffix array requires only space sufficient to store  $n$  integers, or  $n \log n$  bits. More formally, SA is an array  $SA[1..n]$  that contains the permutation of the integers  $1..n$  such that  $x[SA[1]..n] < x[SA[2]..n] < \dots < x[SA[n]..n]$ . Figure 1 shows an example SA for the string *werribbe\$*, where  $x[9..9] = \$$  is the lexicographically least suffix,  $x[6..9] = bbe\$$  is the second least and so on.

Some of the algorithms we describe do not produce the SA directly, but instead produce the *Burrows-*

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the Thirty-Fifth Australasian Computer Science Conference (ACSC2012), Melbourne, Australia, January 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

	1	2	3	4	5	6	7	8	9
$x$	$w$	$e$	$r$	$r$	$i$	$b$	$b$	$e$	$\$$
SA	9	6	7	8	2	5	4	3	1
BWT	$e$	$i$	$b$	$b$	$w$	$r$	$r$	$e$	$\$$

Figure 1: The Suffix Array and Burrows-Wheeler transform for the string *werribbe\$*.

All Rotations		F	L	$i$
werribbe\$		\$ werribb e		1
erribbe\$w		b be\$werr i		2
rribbe\$we		b e\$werri b		3
ribbe\$wer		e \$werrib b		4
ibbe\$werr	Sorted	e rribbe\$ w		5
bbe\$werri	→	i bbe\$wer r		6
be\$werrib		r ibbe\$we r		7
e\$werribb		r ribbe\$w e		8
\$werribbe		w erribbe \$		9

Figure 2: The left column shows all rotations of the string *werribbe\$*. When sorted (right column) this gives the BWT as column L. The F column contains characters of the input sorted lexicographically.

*Wheeler transform (BWT)* of the input string [Burrows and Wheeler, 1994]. The BWT is a reversible transformation of a string that allows the string to be easily and efficiently compressed [Manzini, 2001]. The BWT was discovered independently of the suffix array, but it is now known that the two data structures are equivalent. In the last decade the relationship between the BWT and the SA has been heavily investigated and has led to the very active field of compressed full-text indexing: the study of data structures that allow fast pattern matching, but require space close to that of the compressed text (see Navarro and Mäkinen [2007] and references therein). As we shall see, suffix sorting is the computational bottleneck for performing the BWT.

The Burrows-Wheeler transform (BWT) transforms the string  $x$  by sorting its  $n$  cyclic rotations as in Figure 2. For the full properties of the BWT matrix, we refer the reader to Burrows and Wheeler [1994], Manzini [2001] and Ferragina and Manzini [2005]. We only discuss properties of the matrix that will aid in the explanation of the algorithms. The two main columns of the matrix are: F, the first column which is obtained by lexicographically sorting the characters in  $x$ ; and L, the last column that represents the BWT. Observe the relationship between the SA and BWT: in the matrix of sorted rotations, the prefixes of each rotation up to the  $\$$  are precisely the suffixes of  $x$  in the same order in which they appear in SA. Formally, the BWT of  $x$  is an array  $\text{BWT}[1..n]$  such that:

$$\text{BWT}[i] = \begin{cases} x[\text{SA}[i] - 1] & \iff \text{SA}[i] \neq 1 \\ \$ & \iff \text{SA}[i] = 1 \end{cases} \quad (1)$$

The notation used to describe the three main properties of the BWT is as follows [Ferragina and Manzini, 2005].

- $C[c]$  where  $c \in \Sigma$  denotes the number of characters that are smaller than  $c$  in the BWT.
- $\text{Occ}(c, q)$  denotes the number of occurrences of character  $c$  in prefix  $L[1..q]$ .

Combining  $C$  and  $\text{Occ}$  gives the *Last to First mapping*, LF. This function allows one to locate the character  $c = L[i]$  in F.

$$\text{LF}(i) = C[c] + \text{Occ}(c, i) \quad (2)$$

Observe how the last character in  $L[i]$  precedes the character in  $F[i]$  at any given row in the string (as each row is a cyclic rotation). For example, character  $F[6] = 'i'$  comes after  $L[6] = 'r'$  in  $x$ . The LF function allows one to locate the character  $L[i]$  in  $F[i]$ . Continuing with our example, we want to find the character  $L[6] = 'r'$  in F. So,  $\text{LF}(6) = C[L[6]] + \text{Occ}(L[6], 6) = C['r'] + \text{Occ}('r', 6)$  where  $C['r'] = 6$ , as there are six characters that are lexicographically smaller than the character  $'r'$ , and  $\text{Occ}('r', 6) = 1$ , as it occurs once in  $L[1..6]$ . Thus,  $\text{LF}(6) = 6 + 1 = 7$ , indicating that the character  $'r'$  is at position 7 in F,  $F[\text{LF}(6)] = F[7]$ .

### 3 The Algorithms

#### 3.1 Algorithm K [Kärkkäinen, 2007]

Figure 3 shows a high level description of Kärkkäinen's algorithm. The algorithm begins by computing the difference cover sample [Burkhardt and Kärkkäinen, 2003], a special set of suffixes, the order of which allows the relative order of two arbitrary suffixes to be determined easily. In particular the difference cover sample allows us to determine the relative order of two suffixes,  $i$  and  $j$ , by first comparing their  $v$  character prefixes. If these prefixes are not equal, the order of the suffixes has been determined. However, if a tie occurs, the order of the suffixes is given by the order of suffixes  $i + v$  and  $j + v$ , which are guaranteed to be in the difference cover sample. Here,  $v$  is a parameter, and controls a space-time tradeoff. For more information on difference covers, we refer the reader to Burkhardt and Kärkkäinen [2003].

**Input:**  $x$

- 1: Compute difference covers.
- 2: Select suffixes to become splitters.
- 3: **while** SA not fully computed
- 4:     Collect suffixes based on splitters.
- 5:     Sort suffixes.
- 6:     Write suffixes to disk.

**Output:** SA.

Figure 3: Pseudocode for Algorithm K, the suffix sorting algorithm due to Kärkkäinen [2007].

The next step in this algorithm is to select and sort a set of *splitters*. For instance, using the example string *ababaacaa\$*, we could select splitters at positions 1, 4, and 8 as indicated by the arrows shown below. The last suffix, suffix 10, is also taken as it is the smallest suffix in the string.

	1	2	3	4	5	6	7	8	9	10
$x$	a	b	a	b	a	a	c	a	a	\$
	↑			↑				↑		↑
$j$	S		Suffixes							
1	10		\$							
2	8		aa\$							
3	1		ababaacaa\$							
$m = 4$	4		baacaa\$							

These  $m$  splitters,  $S[1..m]$  are then sorted and stored in memory. Notice that adjacent pairs of splitters divide the suffix array into blocks. In the next phase of the algorithm these splitters will be used to compute the suffix array one block at a time, where a block of suffixes can fit in RAM.

	1	2	3	4	5	6	7	8	9	10
SA <sub>1</sub>	10	9								
SA <sub>2</sub>			8	5	3					
SA <sub>3</sub>						1	6			
SA <sub>4</sub>								4	2	7

Figure 4: SA<sub>*i*</sub> represents the portion of the SA constructed using a lower bound splitter  $j = i$  and an upper bound splitter  $j = i + 1$ . Note the final pass using splitter  $m$  does not have an upper bound.

With the splitters sorted, Algorithm K proceeds by taking the first two splitters  $S[1] = 10(\$)$  and  $S[2] = 8$  ( $aa\$$ ) and using them as lower bound and upper bound values. A left to right scan of the string is then made. During the scan, any suffix larger than or equal to the lower bound and smaller than the upper bound is collected. In our example, this means on the first scan suffix 1 is ignored as it is lexicographically larger than the upper bound value. However, suffix 9 ( $a\$$ ) is picked as it is lexicographically larger than suffix  $S[1]$  but lexicographically smaller than suffix  $S[2]$ . In order to determine efficiently whether a suffix falls between two splitters we make use of the difference cover sample, so at most  $v$  character comparisons are required. At the end of the pass, there are two suffixes in the first block: suffix 9 and suffix 10. These suffixes are ordered to depth  $v$  using multikey quicksort [Bentley and Sedgewick, 1997] and if there is a tie, the difference cover sample is used again to order them. The suffixes are then written to disk, and the memory that was used to hold them (during collection) is reclaimed.

Subsequent passes work similarly. The upper bound splitter from the last round becomes the lower bound, and the next splitter in the set becomes the new upper bound. In our example,  $S[2] = 8$  becomes the lower bound and  $S[3] = 1$  becomes the upper bound. Continuing with our example, suffix 1 is compared with suffix 8 and is lexicographically larger than suffix 8 but it is not lexicographically smaller than itself, thus it is ignored. On the other hand, suffix 3 does fall between the splitters and so it is collected. This process continues until all the suffixes in the range  $[x[8..n], x[1..n]]$  have been collected. The suffixes are then sorted and written to disk. The process continues, deriving the SA as shown in Figure 4.

As splitters are randomly chosen (from a lexicographic point of view), there is no guarantee they divide the SA evenly, and so some blocks may be much larger than others. In particular, a block may exceed RAM limits. Kärkkäinen provides a clever method for dealing with this problem. If, while collecting suffixes for the current block, the amount of available memory is reached, the scan is halted and the contents of the current block is sorted. The lexicographically larger half of the block is then discarded, the median suffix in the block becomes the new upper bound splitter, and the scan resumes. This trick does not (asymptotically) increase the number of scans.

Algorithm K requires  $O(n \log n + vn)$  time and  $O(n \log n / \sqrt{v})$  bits of space in addition to the text, where  $v$  is the period of the difference cover used. The algorithm also allows for different space-time trade-offs, depending on the amount of available memory. Setting  $v = \log^2 n$  gives a runtime of  $O(n \log^2 n)$  and requires  $O(n)$  bits of working space.

### 3.2 Algorithm NZC [Nong et al., 2009]

The algorithm of Nong et al. begins by selecting and sorting a subset of suffixes, and then using the order of those suffixes to induce the sort of the remaining suffixes. The algorithm uses only space to hold the input string and the resulting suffix array. In this sense it is “lightweight” [Manzini and Ferragina, 2004], and ranks among the most space efficient algorithms at the time of the 2007 survey, however it is the most space consuming algorithm we discuss in the survey. Algorithm NZC also runs in linear time in the length of the string, settling an open problem posed by [Puglisi et al., 2007] by showing it possible to be simultaneously lightweight in space usage and linear in runtime.

The idea of differentiating the suffixes into subsets is similar to Ko and Aluru [2005]. Suffixes are split into *Larger* and *Smaller* types (L and S respectively) depending on their lexicographic order relative to their righthand neighbouring suffix. Then a group of leftmost S suffixes (LMS) can be used to derive the sort of the L suffixes, which in turn is used to induce the sorted order of the S suffixes.

**Input:**  $x$

- 1: Label each position S or L.
- 2: Identify LMS substrings and place in SA<sub>1</sub>.
- 3: Scan SA<sub>1</sub> left to right: move type Ls to get SA<sub>2</sub>.
- 4: Scan SA<sub>2</sub> right to left: move type Ss to get SA.

**Output:** SA.

Figure 5: Pseudocode for the copying algorithm due to Nong et al. [2009].

The algorithm begins by making a pass over the string,  $x[1..n]$  assigning a type of either L or S to the suffix beginning at each position depending if it is Larger or Smaller than its righthand neighbour suffix. Thus a suffix  $x[i..n]$  is type S if  $x[i..n] < x[i+1..n]$ , or type L if  $x[i..n] > x[i+1..n]$ . These definitions are then used to define the Left Most S-type (LMS) positions, which are the positions  $i$  such that suffix  $i$  is of type S and suffix  $i-1$  is of type L for  $x > 1$ .

The end-of-string symbol,  $x[n] = \$$ , is defined to be S, and hence is also an LMS suffix (as its left neighbour must be larger than it). For example, types are assigned when a right to left scan is made over our example string, *ababaacaa\$* as follows.

	1	2	3	4	5	6	7	8	9	10
$x$	a	b	a	b	a	a	c	a	a	\$
type	S	L	S	L	S	S	L	L	L	S
LMS			*		*					*

Suffix 1 is categorised as type S (*ababaacaa\$*) as it is lexicographically smaller than suffix 2 (*babaacaa\$*). Suffix 2 is of type L as it is lexicographically larger than suffix 3 (*abaacaa\$*). This process is repeated until the types are assigned to all the suffixes of the string. While making the scan, we also identify the LMS types, which are marked with an \*.

After identifying the types, the indices of the LMS positions are placed in buckets in the SA, processing the LMS left to right, to get SA<sub>1</sub>. Each suffix beginning with an LMS character is placed at the rightmost end of its character’s bucket in SA<sub>1</sub>, and the bucket counter decremented. Let  $C[j]$  be the cumulative count of all characters in the string that are lexicographically less than or equal to  $j$ . Thus, for our example string:

	\$	a	b	c
$j$	0	1	2	3
$C[j]$	1	7	9	10

The first LMS character encountered is 'a', the beginning of suffix 3, which is placed in position 7, as  $C[a] = 7$ , and  $C[a]$  is decremented. Next we encounter 'a' at the beginning of suffix 5, and so it is inserted at  $C[a] = 6$ . Finally, suffix 10 beginning with \$ is inserted at  $C[\$] = 1$ . The end result is shown here, with the boundaries of each bucket denoted by brackets.

	\$	(	$a$						)	(	$b$		)	(	$c$		)
$SA_1$	10	(	2	3	4	5	6	7	)	(	8	9	)	(	10	)	)
$SA_2$	10	(	9	8			5	3	)	(	4	2	)	(	7	)	)
$SA$	10	(	9	8	5	3	1	6	)	(	4	2	)	(	7	)	)

For the next pass, we make a left to right ( $j = 1..n$ ) scan over  $SA_1$  to derive the order of the L suffixes. Within a bucket, type L suffixes always come before type S suffixes since the latter is lexicographically larger than the former. During the scan, if we find a suffix  $i$  ( $i = SA_1[j]$ ), we look for suffix  $i - 1$  and if that suffix is of type L and has not been placed in  $SA_1$ , we place this suffix in the first empty position of the group. Hence our bucket counters start off as  $C = \{1, 2, 8, 10\}$ , one more than the number of characters in the string less than the index. In our case, when  $j = 1$  and  $i = SA_1[1] = 10$ , we find suffix  $10-1 = 9$  which is a suffix of type L that is yet to be sorted. Therefore, we place it at the start of its 'a' bucket ( $C[a] = 2$ ) and increment the bucket count. Continuing the scan,  $i = SA_1[2] = 9$ , suffix  $9-1=8$  is a suffix of type L. So, it is placed at the next available position in bucket 'a', which is  $C[a] = 3$ . This process repeats itself until all type L suffixes have been placed in the array to get  $SA_2$ .

After placing the type L suffixes, we now collect the type S suffixes. We make a right to left ( $j = n..1$ ) scan over  $SA_2$  and for each  $i$  ( $i = SA_2[j]$ ), we look for suffix  $i - 1$  and if the suffix of type S and has yet to be sorted, we place this suffix at the end of its bucket indicated by its first character. (The bucket counters have again been reset.) For example, when  $j = n$ ,  $i = 7$ , we find a suffix  $7-1=6$  and it is a type S suffix from the group 'a'. So, we overwrite  $SA_2[7]$  with 6, and decrement the bucket counter for 'a'. Continuing with our example, when  $j = 9$ ,  $i = 2$ , we find a suffix  $2-1=1$  and it is a suffix of type S. So, we overwrite the suffix 5 in  $SA_2[6]$  with suffix 1. This process is repeated until all the type S have derived (as indicated by SA).

### 3.3 Algorithm OS [Okanohara and Sadakane, 2009]

The algorithm of Okanohara and Sadakane uses the same framework as the algorithm by Nong et al. [2009], described in the previous section, but derives the BWT string, rather than the SA. Like NZC, OS runs in linear time, but via a careful implementation of the required data structures, it is able to reduce memory overheads to  $O(n \log \sigma \log \log n)$  bits.

Like NZC, suffixes are first classified into types L and S. Then, the LMS substrings are identified. The difference comes in that OS explicitly stores the LMS substrings in queue-like data structures.

Figure 6 illustrates the operation of OS on the example string *ababaacaa\$*. The LMS substrings are stored in queues and their actual positions in  $x$  are stored in F. When these LMS substrings are moved among queues the L suffixes are induced. Their movement is indicated by the arrows and captured in P.

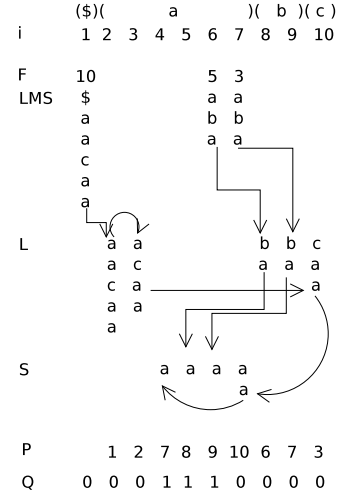


Figure 6: An example of the modified algorithm that induces the sort of the string *ababaacaa\$* using LMS substrings that are moved in queue-like data structures.

For example, since the LMS substring, *\$aaciaa* was moved from position  $i = 1$  to  $i = 2$ ,  $P[2] = 1$ . Furthermore, inducing suffix 9 of type L and so on.

Lastly,  $Q[i]$  shows the location of the LMS substrings in the queues that are further indicated with '1's. Moreover, the order of two adjacent LMS substrings can be determined by tracing this Q data structure together with P where  $i = P[i]$  is computed repeatedly when  $Q[i] = 1$  until the head of the substring is reached (stored in F). For example, when  $i = 4$  and  $Q[i] = 1$ ,  $i = \{P[4] = 7, P[7] = 10, P[10] = 3, P[3] = 2, P[2] = 1\}$ . In fact, the order of two adjacent LMS substrings can be determined in time proportional to their lengths.

### 3.4 Algorithm FGM [Ferragina et al., 2010]

**Input:**  $x$

- 1: Compute BWT for the rightmost block.
- 2: Store BWT on external disk,  $BWT_{ext}$ .
- 3: **while** BWT not fully computed
- 4:     Compute  $BWT_{int}$  for the next block.
- 5:     Merge  $BWT_{ext}$  and  $BWT_{int}$ .

**Output:** BWT.

Figure 7: Pseudocode for computing the BWT by Ferragina et al. [2010].

Figure 7 shows the high level description of Ferragina et al.'s algorithm. In a nutshell, the BWT is computed for the first block of certain size and stored on disk. For the next subsequent blocks, the BWT is computed in internal memory which we will call as  $BWT_{int}$  and merged with the external BWT that is on disk which we will refer as  $BWT_{ext}$ . This process is repeated until the entire BWT for the text has been computed.

#### 3.4.1 Compute BWT for the first block

The algorithm begins by dividing the text into blocks of size  $m$ . For example, with  $m = 3$ :

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
B		3			2			1		



The first block,  $B_1 = x[7..9]$ , is brought into memory, its BWT is derived and stored on disk as,  $BWT_{ext}$ .

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
$BWT_{ext}$							b	a	a	
SA							7	8	9	
B		3			2			1		

### 3.4.2 Compute BWT for subsequent blocks

The BWT computation for the next blocks differs from the first block. In our example, text for the blocks  $B_1 = x[7..9]$  and  $B_2 = x[4..6]$  are brought into memory. Suffixes are compared naïvely, character by character. For example, when comparing suffix 4 ( $aaaaab\$$ ) with suffix 5 ( $aaaab\$$ ), the suffix was ordered when a mismatch occurred at character 5, position 9 in the string. Having ordered the suffixes (built the SA) for block  $B_2$ , the BWT is computed and stored in memory as  $BWT_{int}$ .

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
$BWT_{int}$				a	a	a				
$BWT_{ext}$							b	a	a	
SA				4	5	6	7	8	9	
B		3			2			1		

The next step is find the number of suffixes of the already processed string (covered by previous blocks) that fall between the suffixes of the current block. Let  $B_i = x[j..j+m]$  be the current block, and let  $SA_B$  be it's suffix array. We compute an array  $G$  such that  $G[i]$  is the number of suffixes of  $x[j+m..n]$  that are (lexicographically) between  $SA_B[i]$  and  $SA_B[i+1]$ .  $G$  can be computed efficiently using the “backward search algorithm” [Navarro and Mäkinen, 2007] on a suitably preprocessed  $BWT_{int}$ .

In our case, the current block is  $B_2$ .

$k$	$x[k]$	G				
		0	1	2	3	4
10	\$	1				
9	b				1	
8	a					1
7	a					1

Using the  $G$  array for the SA of the first block,  $BWT_{ext}$  and  $BWT_{int}$  are merged. For instance,  $G[0]$  means there is one suffix that is smaller than suffix 4 (that's suffix 10 '\$'). However, since the BWT for this suffix was never computed, it can safely be ignored. Since  $G[0..2]=0$ , we can copy  $BWT_{int}[0..2]$  to disk. Then, the  $BWT_{ext}[0..2]$  is copied (as indicated by  $G[3]$  and  $G[4]$ ) which shows that there are three suffixes that are larger than the suffix 6.

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
$BWT_{ext}$				b	a	a	a	a	a	
SA				4	5	6	7	8	9	
B		3			2			1		

Lastly, we compute the BWT for  $B = 3$ . Similar to the previous pass,  $x[1..6]$  is brought into memory and the SA for it is computed naïvely, via character comparisons and having a tie, a data structure that orders the suffixes in  $O(m)$  time is used (see Ferragina et al. [2010] for implementation details).

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
$BWT_{int}$	b	a	#							
$BWT_{ext}$				b	a	a	a	a	a	
SA	2	3	1	4	5	6	7	8	9	
B		3			2			1		

$k$	$x[k]$	G				
		0	1	2	3	4
10	\$	1				
9	b				1	
8	a					1
7	a					1
6	a					1
5	a					1
4	a					1

$G[2] = 6$  shows there are six old suffixes that exist between  $SA[2]$  and  $SA[3]$ . Therefore, the new  $BWT_{ext}$  is merged as below.

	1	2	3	4	5	6	7	8	9	10
$x$	(b	a	a)	(a	a	a)	(a	a	b)	\$
$BWT_{int}$										
$BWT_{ext}$	b	a	b	a	a	a	a	a	#	
SA	2	3	1	4	5	6	7	8	9	
B		3			2			1		

## 4 Experiments

In this section we provide a brief experimental comparison of the recent algorithms for which we have efficient implementations. The experiments were run on an otherwise idle 3.16GHz Intel Core (TM) 2 Duo CPU E8500 of 4 GB of RAM and a cache size of 6144 KB. The operating system is Ubuntu 10.04.3. All the code was compiled with gcc/g++ version of 4.4.3 and the -O3 optimisation flag. The memory usage was measured using the memusage tool and times reported are the minimum of three runs, measured with the C time function.

We measured implementations of Algorithms K, S, NZC and FGM. From personal communication with Okanohara [2010], we understand that there is no publicly available code for Algorithm OS. For Algorithm NZC, we downloaded a publicly available implementation by Yuta Mori [SAIS, 2011]. Likewise, we downloaded a publicly available implementation for Algorithm S by the author himself. We implemented Algorithm K ourselves. Of these algorithms, Algorithm NZC is an in-memory algorithm; Algorithm FGM is an external memory algorithm (using disk as working space) and Algorithm K is a semi-external algorithm (writing only the BWT to disk).

Table 1 shows our test data: 200MB of ENGLISH, DNA and SOURCES from the Pizza and Chili [2009] Corpus. We equated the amount of memory allocated by Algorithm K and Algorithm FGM (to 481MB). Algorithm NZC requires both the SA and input string to be resident in RAM and so consumes 1,000MB of memory.

Running times are reported in Table 2. Algorithm NZC is clearly fastest, but uses twice the memory of the other two algorithms. This result is expected as NZC uses an induced copying heuristic that is used in all fast algorithms [Puglisi et al., 2007]. Also, NZC is an in-memory algorithm, whereas Algorithm K requires only the text to be in memory and for FGM, all the data including the text resides in external memory. The size of the available RAM on the test machine is big enough to hold both the input string and

Files	$\Sigma$	Min LCP	Max LCP
ENGLISH	226	9,390	987,770
DNA	17	59	97,979
SOURCES	231	373	307,871

Table 1: LCP is the longest common prefix between adjacent suffixes in the suffix array. A higher LCP generally increases the cost of suffix sorting.

Algorithms	ENGLISH	DNA	SOURCES
K	244	246	293
NZC	<b>58</b>	<b>59</b>	<b>41</b>
FGM	405	421	324

Table 2: Total wall clock time taken in seconds by the algorithms to run on the 200MB dataset. The minimum time is shown in bold.

Algorithms	Peak Memory
K	<b>481</b>
NZC	1000
FGM	<b>481</b>

Table 3: The peak memory usage (MB) by the algorithms for the 200MB dataset. The minimum is shown in bold.

the SA in memory, and so Algorithm FGM and Algorithm K might get faster if they were tuned for RAM.

## 5 Concluding Remarks

Algorithms for suffix sorting continue to mature. In this paper we have surveyed a number of recent techniques, all of which aim to reduce the amount of main memory required during sorting. Algorithms also continue to emerge for related problems; for example Sirén [2009] describes a method for directly building the compressed suffix array of a collection of strings, such as documents or genomes. Bauer et al. [2011] consider a similar problem, but where each string in the collection is relatively short (in particular the length of a sequenced DNA fragment).

A somewhat neglected aspect of many of the new algorithms is the alphabet size. In particular the external memory algorithm of Ferragina et al. [2010] assumes a small, constant alphabet. The development of an efficient external memory algorithm free of this assumption is an important open problem. Some theoretical progress has been made [Hon et al., 2003, Na, 2005], but we are aware of no practical approaches.

Another area to explore, at least for obtaining practical algorithms, is the blending of older suffix sorting heuristics, surveyed in Puglisi et al. [2007], with the low memory algorithms examined here. Some initial attempts to introduce pointer copying heuristics to Kärkkäinen’s algorithm are described by Dhaliwal and Puglisi [2011].

Finally, a related problem called *suffix selection*, where one seeks the suffix of a given rank, without resorting to sorting all suffixes, has received some attention recently [Franceschini and Muthukrishnan, 2007a,b, Franceschini et al., 2009, Franceschini and Hagerup, 2011]. Efficient suffix selection algorithms have the potential to aid suffix sorting algorithms – for example they could be used to choose good splitters in Kärkkäinen’s algorithm – however suffix selection algorithms discovered to date have high constant factors (both on time and space bounds) and do not seem practical.

## 6 Acknowledgements

Our thanks goes to the authors that made their code available to us. This work was supported by the Australian Research Council.

## References

- M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms (JDA)*, 2(1):53–86, 2004.
- M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *22nd Annual Combinatorial Pattern Matching (CPM) symposium*, volume 6661 of *Lecture Notes in Computer Science*, pages 219–231. Springer, 2011.
- J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. 8th Annual Symposium on Discrete Algorithms*, pages 360–369. ACM, 1997.
- S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium Combinatorial Pattern Matching (CPM)*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003.
- M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital Equipment Corporation, 1994.
- G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time and space. *Mathematics in Computer Science*, 1(4):605–623, 2008.
- J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top- $k$  ranked document search in general text databases. In U. Meyer and M. de Berg, editors, *Proc. 18th Annual European Symposium on Algorithms (ESA)*, volume 6347 of *Lecture Notes in Computer Science*, pages 194–205. Springer, 2010.
- J. Dhaliwal and S. J. Puglisi. Fast semi-external suffix sorting. Technical Report TR-11-1, RMIT University, School of Computer Science and Information Technology, 2011.
- P. Ferragina and G. Manzini. Indexing compressed text. *Journal of ACM (JACM)*, 52(4):552–581, 2005.
- P. Ferragina and G. Manzini. On compressing the textual web. In *WSDM ’10: Proceedings of the third ACM international conference on Web search and data mining*, pages 391–400, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-889-6. doi: <http://doi.acm.org/10.1145/1718487.1718536>.
- P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In A. López-Ortiz, editor, *Proc. of the 9th Latin American Theoretical Informatics Symposium (LATIN)*, volume 6034 of *Lecture Notes in Computer Science*, pages 697–710. Springer, 2010.
- P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods (Suppl)*, 6(11):S6–S12, 2009.
- G. Franceschini and T. Hagerup. Finding the maximum suffix with fewer comparisons. *Journal of Discrete Algorithms (JDA)*, 9(3):279–286, 2011.

- G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *34th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4596 of *Lecture Notes in Computer Science*, pages 533–545. Springer, 2007a.
- G. Franceschini and S. Muthukrishnan. Optimal suffix selection. In D. S. Johnson and U. Feige, editors, *Proc. of the 39th ACM Symposium on Theory of Computing (STOC)*, pages 328–337. ACM, 2007b.
- G. Franceschini, R. Grossi, and S. Muthukrishnan. Optimal cache-aware suffix selection. In *26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 3 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 457–468. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.
- D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, New York, 1997.
- W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:251, 2003. ISSN 0272-5428.
- J. Kärkkäinen. Fast bwt in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
- P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3:143–156, 2005.
- U. Manber and G. W. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
- G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
- G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40:33–50, 2004.
- J. C. Na. Linear-time construction of compressed suffix arrays using  $o(n \log n)$ -bit working space for large alphabets. In A. Apostolico, M. Crochemore, and K. Park, editors, *16th Annual Combinatorial Pattern Matching (CPM) Symposium*, volume 3537 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2005.
- G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.
- G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference (DCC)*, pages 193–202. Snowbird, UT, USA, 2009. IEEE Computer Society.
- D. Okanohara. Personal communication, 2010.
- D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler Transform using induced sorting. In J. Karlgren, J. Tarhio, and H. Hyrö, editors, *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2009.
- M. Patil, S. V. Thankachan, R. Shah, W.-K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted indexes for phrases and strings. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information*, SIGIR '11, pages 555–564, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0757-4. doi: <http://doi.acm.org/10.1145/2009916.2009992>.
- Pizza and Chili. Pizza and chili corpus, compressed indexes and their testbeds. May 2009. URL <http://pizzachili.dcc.uchile.cl/>.
- S. J. Puglisi, W. F. Smyth, and A. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2), 2007.
- SAIS. Sais: An implementation of the induced sorting algorithm. Aug. 2011. URL <http://sites.google.com/site/yuta256/sais>.
- J. Sirén. Compressed suffix arrays for massive data. In J. Karlgren, J. Tarhio, and H. Hyrö, editors, *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 5721 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2009.



# Spectral debugging: How much better can we do?

Lee Naish, Hua Jie Lee and Kotagiri Ramamohanarao  
 Department of Computing and Information Systems  
 The University of Melbourne, Victoria 3010, Australia  
 lee,kotagiri@unimelb.edu.au, hua jie.lee@gmail.com

## Abstract

This paper investigates software fault localization methods which are based on program spectra – data on execution profiles from passed and failed tests. We examine a standard method of spectral fault localization: for each statement we determine the number of passed and failed tests in which the statement was/wasn't executed and a function, or *metric*, of these four values is used to rank statements according to how likely they are to be buggy. Many different metrics have been used. Here our main focus is to determine how much improvement in performance could be achieved by finding better metrics. We define the cost of fault localization using a given metric and the *unavoidable cost*, which is independent of the choice of metric. We define a class of *strictly rational* metrics and argue that is reasonable to restrict attention to these metrics. We show that every *single bug optimal* metric performs as well as any strictly rational metric for single bug programs, and the resulting cost is the unavoidable cost. We also show how any metric can be adapted so it is single bug optimal, and give results of empirical experiments using single- and two-bug programs.

## 1 Introduction

Bugs are pervasive in software under development and tracking them down contributes greatly to the cost of software development. One of many useful sources of data to help diagnosis is the dynamic behaviour of software as it is executed over a set of test cases where it can be determined if each result is correct or not; each test case is said to *pass* or *fail*. Software can be instrumented automatically to gather data known as program spectra (Reps, Ball, Das & Larus 1997), such as the statements that are executed, for each test case. If a certain statement is executed in many failed tests but few passed tests we may conclude it is likely to be buggy. Typically the raw data is aggregated to get the numbers of passed and failed tests for which each statement is/isn't executed. Some function is applied to this aggregated data to rank the statements, from those most likely to be buggy to those least likely. We refer to such functions as *metrics*. A programmer can then use the ranking to help find a bug.

We make the following contributions:

- We define a class of metrics we call *strictly rational* metrics and argue why restricting attention to such metrics is reasonable.

- We define the *unavoidable cost* of bug localization, and show it is the minimum cost for any strictly rational metric.
- We show that a class of previously proposed metrics lead to the lowest possible cost of any strictly rational metrics for programs with single bugs.
- We show how any metric can be easily adapted so it is optimal for single bug programs.
- We evaluate several metrics for benchmark sets with one and two bugs.
- We perform additional experiments to help further understand the best known metric for two-bug programs.
- We suggest how test selection strategies can be improved to help performance of bug localization.

The rest of this paper is structured as follows. We first describe spectral fault localization and define the metrics we evaluate in this paper. Section 3 revisits metrics and defines (strictly) rational metrics. Section 4 describes how the cost of fault localization is measured in this paper and also introduces the idea of “unavoidable” cost. Section 5 discusses previous work on optimal metrics for single bug-programs and significantly extends those results. Section 6 shows how any metric can be adapted so it is optimal for single bug programs. Section 7 describes empirical experiments and their results and Section 9 concludes.

## 2 Background — Spectral fault localization

All spectral methods use a set of tests, each classified as failed or passed; this can be represented as a binary vector, where 1 indicates failed and 0 indicates passed. For statement spectra (Jones & Harrold 2005, Abreu, Zoetewij & van Gemund 2006, Wong, Qi, Zhao & Cai 2007, Xie, Chen & Xu 2010, Naish, Lee & Kotagiri 2011, Lee 2011), which we use here, we gather data on whether each statement is executed or not for each test. This can be represented as a binary matrix with a row for each statement and a column for each test; 1 means executed and 0 means not executed. For each statement, four numbers are ultimately produced. They are the number of passed/failed test cases in which the statement was/wasn't executed. We adapt the notation from Abreu et al. (Abreu et al. 2006) —  $\langle a_{np}, a_{nf}, a_{ep}, a_{ef} \rangle$ . The first part of the subscript indicates whether the statement was executed (*e*) or not (*n*) and the second indicates whether the test passed (*p*) or failed (*f*). We use superscripts to indicate the statement number where appropriate. For example,  $a_{ep}^1$  is the number of passed tests that executed statement 1. We use  $F$  and  $P$  to denote the total number of tests which

Table 1: Statement spectra with tests  $T_1 \dots T_5$ [illegible]

fail and pass, respectively. Clearly,  $a_{nf} = F - a_{ef}$  and  $a_{np} = P - a_{ep}$ . In most of this paper we avoid explicit use of  $a_{nf}$  and  $a_{np}$ ; making  $F$  and  $P$  explicit suits our purposes better. Table 1 gives an example binary matrix of execution data and binary vector containing the test results. This data allows us to compute  $F$ ,  $P$  and the  $a_{ij}$  values,  $i \in \{n, e\}$  and  $j \in \{p, f\}$ .

Metrics, which are numeric functions, can be used to rank the statements. Most commonly they are defined in terms of the four  $a_{ij}$  values. Statements with the highest metric values are considered the most likely to be buggy. We would expect buggy statements to generally have relatively high  $a_{ef}$  values and relatively low  $a_{ep}$ . In the example in Table 1, Statement 2 ( $S_2$ ) is executed in both failed tests and only one passed test, which is the minimum for all statements, and thus would typically be ranked highest. The relative rank of statements 1 and 3 is not so clear cut, since statement 3 is executed in more failed tests but also more passed tests. One way of viewing the ranking is that rows of the matrix are ranked according to how “similar” they are to the vector of test results, or how similar the set of failed tests is to the set of tests which execute the statement. Measures of similarity are important in classification and machine learning, not just fault localization, and many different metrics have been proposed; Lee (2011) evaluates the fault localization performance of 50 metrics.

Programmers searching for a bug are expected to examine statements, starting from the highest-ranked statement, until a buggy statement is found. In reality, programmers are likely to modify the ranking due to their own understanding of whether the code is likely to be buggy, based on other information such as static analysis, the history of software changes, *et cetera*. Also, checking correctness generally cannot be done by a single statement at a time, or even one basic block at a time. Evaluation of different ranking methods, which we discuss in Section 4, generally ignores such refinement and just depends on where the bug(s) appear in the ranking.

Table 2 gives definitions of the metrics used here, all of which have been evaluated for fault localization in Naish et al. (2011) and their origins are discussed more there. Space prevents us from including all proposed metrics. Several were originally proposed for spectral fault localization: Tarantula (Jones & Harrold 2005), which is generally credited as being the first spectral fault localization system, Zoltar (Gonzalez 2007), Wong3, the best of several metrics proposed in Wong et al. (2007), and  $O$  and  $O^p$  (Naish et al. 2011). We also use a metric we refer to as Wong4, which was devised for fault localization. It is Heuristic III of Wong, Debroy & Choi (2010), with  $\alpha = 0.0001$ ; we do not provide its definition here due to its complexity. Also, Ample is an adaptation, from Abreu et al. (2006), of a metric developed for the AMPLE system (Dallmeier, Lindig & Zeller 2005) and CBIlog is an adaptation, from Naish et al. (2011), of a metric developed for the CBI system (Liblit, Naik, Zheng, Aiken & Jordan 2005).

Jaccard (Jaccard 1901), the oldest metric here, originally used for classification of plants, has been used in the Pinpoint system (Chen, Kiciman, Fratkín, Fox & Brewer 2002). Ochiai (Ochiai 1957) and Russell (Russell & Rao 1940), both developed for other domains, were first evaluated for fault localization in Abreu et al. (2006) and Kulczynski2 (see Lourenco, Lobo & Bação (2004)) was first evaluated for fault localization in Naish et al. (2011). Kulczynski2 is the best metric we know of for two and three bug programs (see Naish, Lee & Kotagiri (2009), for example). We discuss some of these metrics in more detail later.

### 3 Ranking metrics, revisited

The formulas for ranking metrics are only used in constrained ways. We know that  $a_{ef}$ ,  $a_{nf}$ ,  $a_{ep}$  and  $a_{np}$  are all natural numbers. We can also assume there is at least one test case. Furthermore, for any given program and set of test cases  $F$  and  $P$  are fixed, and  $a_{ef} \leq F$  and  $a_{ep} \leq P$ . Therefore it is possible to define metrics as follows (in Table 2 we can assume  $a_{nf}$  and  $a_{np}$  are defined in terms of the other values):

**Definition 1 (Metric)** A metric is a partial function from four natural numbers,  $a_{ef}$ ,  $a_{ep}$ ,  $F$  and  $P$  to a real number. It is undefined if  $a_{ef} > F$  or  $a_{ep} > P$  or  $F = P = 0$ .

Metrics are intended to measure *similarity* between the set of failed tests and the set of tests which execute a statement. The whole idea behind the approach is that *in most cases* there is a positive correlation between execution of buggy statements and failure, and a negative correlation between execution of correct statements and failure. Some metrics measure *dis-similarity* and produce very poor rankings. Typically the bugs are ranked towards the bottom rather than the top; we have observed such behaviour when we have incorrectly translated metrics which are described using different terminology from different application areas. Thus we can put additional constraints on what functions can sensibly, or rationally, be used as metrics.

**Definition 2 ((strictly)rational metric)** *A metric  $M$  is rational if it is monotonically increasing in  $a_{ef}$  and monotonically decreasing in  $a_{ep}$ : if  $a'_{ef} > a_{ef}$  then  $M(a'_{ef}, a_{ep}, F, P) \geq M(a_{ef}, a_{ep}, F, P)$  and if  $a'_{ep} > a_{ep}$  then  $M(a_{ef}, a'_{ep}, F, P) \leq M(a_{ef}, a_{ep}, F, P)$ , for points where  $M$  is defined.*

A metric  $M$  is strictly rational if  $a'_{ef} > a_{ef}$  implies  $M(a'_{ef}, a_{ep}, F, P) > M(a_{ef}, a_{ep}, F, P)$  and  $a'_{ep} > a_{ep}$  implies  $M(a_{ef}, a'_{ep}, F, P) < M(a_{ef}, a_{ep}, F, P)$ , for points where  $M$  is defined.

There are cases where the correlations are the opposite of what is expected, and some metric which is not rational will perform better than rational metrics. However, there is no way of knowing *a priori* whether we have such a case and *overall* rational metrics perform better. Thus we consider it reasonable to restrict attention to rational metrics in the search for good metrics and when assessing “ideal” performance. Almost all metrics previously used for fault localization are strictly rational. Ample is the only metric which is not rational, because “absolute value” is used. In Naish et al. (2011), a variation of this metric is defined (called “Ample2”) which does not use absolute value, and it performs significantly better than Ample.

The Russell metric does not strictly decrease as  $a_{ep}$  increases, so it is rational but not strictly rational, and similarly for  $O$  whenever  $a_{ef} < F$ . Metrics

Table 2: Definitions of ranking metrics used

Name	Formula	Name	Formula	Name	Formula
Kulczynski2	$\frac{1}{2} \left( \frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ef}}{a_{ef}+a_{ep}} \right)$	Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000a_{nf}a_{ep}}{a_{ef}}}$	Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$	Russell	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	Ample	$\left  \frac{a_{ef}}{a_{ef}+a_{nf}} - \frac{a_{ep}}{a_{ep}+a_{np}} \right $
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}}$	$O$	$\begin{cases} -1 & \text{if } a_{nf} > 0 \\ a_{np} & \text{otherwise} \end{cases}$	$O^p$	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$
CBILog	$\frac{2}{\frac{1}{c} + \frac{\log(a_{ef}+a_{nf})}{\log a_{ef}}}$ , where $c = \frac{a_{ef}}{a_{ef}+a_{ep}} - \frac{a_{ef}+a_{nf}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$				
Wong3	$a_{ef} - h$ , where $h = \begin{cases} a_{ep} & \text{if } a_{ep} \leq 2 \\ 2 + 0.1(a_{ep} - 2) & \text{if } 2 < a_{ep} \leq 10 \\ 2 \cdot 8 + 0.001(a_{ep} - 10) & \text{if } a_{ep} > 10 \end{cases}$				

which are rational but not strictly rational can generally be tweaked so they are strictly rational with no loss of performance in typical cases.  $O$  was designed specifically for the case of single bugs, where we know the bug is executed in all failed tests ( $a_{ef} = F$  for the bug; we discuss this further in Section 5). If we modify  $O$  so it gives a small negative weight to  $a_{ep}$  when  $a_{ef} < F$  it becomes strictly rational (it produced the same rankings as  $O^p$ ). This does not affect performance at all for single bug programs and generally improves performance for multiple bug programs. Similarly, if we give a small negative weight to  $a_{ep}$  in the Russell metric it also becomes equivalent to  $O^p$ , which performs better in nearly all cases (see Naish et al. (2011)). Even if non-strict rational metrics remain of some practical benefit, considering only strict rational metrics leads us to additional theoretical insights.

#### 4 Measuring performance

The most common way of performance measure for spectral fault localization is the rank of the highest-ranked bug, as a percentage of the total number statements; typically, only statements which are executed in at least one test case are counted (Abreu et al. 2006, Wong et al. 2007, Naish et al. 2011). This is often called the *rank percentage*. If a bug is ranked highest, which is the best case, the rank is 1. Here we give a slightly different definition, which we call *rank cost* to avoid confusion, where the best case is zero. This is more convenient for our work, and also when averaging the performance over several programs with different numbers of statements, which is normally done. When bugs and non-bugs are tied in the ranking we assume the bugs are ranked in the middle of all these equally ranked statements. We discuss this more in Section 7; there is some variation in how ties are handled in the literature.

**Definition 3 (rank cost)** *Given a ranking of  $S$  statements, the rank cost is*

$$\frac{GT + EQ/2}{S}$$

where  $GT$  is the number of correct statements ranked strictly higher than all bugs and  $EQ$  is the number of correct statements ranked equal to the highest ranked bug.

For most programs and sets of tests, we cannot expect a buggy statement to be ranked strictly higher than all correct statements, whatever metric is used to produce the ranking. For example, all statements in

the same basic block as a bug will be tied with the bug in the ranking, since they will have the same  $a_{ef}$  and  $a_{ep}$  values as the bug. Furthermore, there may be other statements with higher  $a_{ef}$  and lower  $a_{ep}$  values, which must be ranked higher than the bug for all strictly rational metrics. By explicitly considering such statements, we can determine how much of the cost of bug localization could potentially be avoided by choosing a different strictly rational metric, and how much is unavoidable. We define the *unavoidable cost* in a way which makes it easy to compare with the rank cost. We first introduce some additional notation concerning a partial order of statements based on their associated spectra.

**Definition 4** ( $=^s, \leq^s, <^s$ ) *For two statements,  $x$  and  $y$ , with associated spectra:*

- $x =^s y$  if  $a_{ef}^x = a_{ef}^y \wedge a_{ep}^x = a_{ep}^y$
- $x \leq^s y$  if  $a_{ef}^x \leq a_{ef}^y \wedge a_{ep}^x \geq a_{ep}^y$
- $x <^s y$  if  $x \leq^s y \wedge \neg(a_{ef}^x = a_{ef}^y)$

**Definition 5 (unavoidable cost)** *Given a set of  $S$  statements and corresponding spectra, the unavoidable cost is the minimum of  $UC_b$ , for all bugs  $b$ , where*

- $UC_b = \frac{GT'_b + EQ'_b/2}{S}$
- $GT'_b$  is the number of correct statements  $c$ , such that  $b <^s c$ ,
- $EQ'_b$ , the number of correct statements  $c$ , such that  $b =^s c$ .

**Proposition 1** *If  $x \leq^s y$ , any rational metric will rank  $x$  below or equal to  $y$ . If  $x <^s y$ , any strictly rational metric will rank  $x$  below  $y$ .*

**Proof** Follows from definitions.  $\square$

**Proposition 2** *For any set of statements, associated spectra and strictly rational ranking metric, the rank cost of the resulting ranking is at least the unavoidable cost.*

**Proof** If  $b$  is the highest ranked bug, the rank cost is  $UC_b$ , which is at least the unavoidable cost.  $\square$

**Proposition 3** *For any set of statements and associated spectra there exists a strictly rational ranking metric such that the rank cost of the resulting ranking is the unavoidable cost.*

**Proof** Let  $b$  be a bug which minimises  $UC_b$ . There is no buggy statement  $b'$  s.t.  $b <^s b'$ , otherwise the unavoidable cost would be lower. Consider the following definitions:

$$M(a_{ef}, a_{ep}, F, P) = f(a_{ef} - a_{ef}^b) + f(a_{ep}^b - a_{ep})$$

$$f(x) = \begin{cases} \epsilon x & \text{if } x < 0 \\ 1 + \epsilon x & \text{otherwise,} \end{cases}$$

$$\epsilon = 1/(F + P + 1)$$

$M$  is a strictly rational metric. For the bug  $b$  (and all statements  $c$  s.t.  $c =^s b$ ) it has value 2. The metric value of a statement  $c$  is greater than 2 if and only if  $b <^s c$ , and all such statements are correct. Thus the rank cost using  $M$  is the unavoidable cost.  $\square$

**Proposition 4** *For any set of statements and associated spectra, the unavoidable cost is the minimum rank cost for any strictly rational metric.*

**Proof** Follows from Propositions 2 and 3.  $\square$

If we do not restrict the class of metrics, there will always be *some* metric which ranks the bug highest and the minimum cost, ignoring ties, will always be zero. Note that a metric which is rational but not *strictly* rational can also have a cost lower than the unavoidable cost. For example, with one passed and one failed test, all statements could be executed in the failed test but the passed test could execute only the buggy statement(s). The unavoidable cost is the highest possible cost, which is close to 1, whereas a rational metric could have all statements tied in the ranking, with a cost of around 0.5. Although theoretically interesting, such examples do not seem to provide strong practical motivation for using metrics which are not strictly rational.

We cannot necessarily expect to achieve a cost as low as the unavoidable cost in practice — it simply gives a lower bound on what we can reasonably expect using this approach to bug localization. If we can achieve the unavoidable cost or very close to it in all cases, we know there is no point in searching for better metrics. If there is a wide gap between the cost we achieve and the unavoidable cost for some buggy programs and sets of test cases we might be able to close the gap with different metrics, but we risk making the situation worse for other programs. There is no single metric which achieves the unavoidable cost for all programs — we cannot swap the order of quantifiers in Proposition 3. Also, the unavoidable cost does not give a lower bound on what can be achieved by other fault localization methods. However, the same methodology could potentially be applied. If, for example, a richer form of spectral data was used, we may be able to find an appropriate unavoidable cost definition for that method.

## 5 Optimality for single bug programs

In Naish et al. (2011), optimality of metrics is introduced and “single bug” programs are the focus. In order to establish any technical results, we must be clear as to what constitutes a bug, so it is clear if a program has a single bug. In Naish et al. (2011) a bug is defined to be “a statement that, when executed, has unintended behaviour”. A programmer may make a single mistake which leads to multiple bugs according to this definition. For example, when coding various formulas which use logarithms, the programmer may use the wrong base for all the logarithms. Also a mistake in a single `#define` directive in a C program can lead to multiple bugs. The `#define` directive is not a statement which is executed and no spectral data is generated for it, but several statements which use the macro may behave incorrectly.

In order to understand the fault localization problem better, a very simple model program, with just two if-then-else statements and a single bug is proposed in Naish et al. (2011), along with a very simple way of measuring performance of a metric with a given set of test cases, based on whether the bug is ranked top, or equal top. A set of test cases corresponds to a multiset of execution paths through the program. Performance depends on the multiset, but overall performance for  $T$  tests is determined by the average performance over all possible multisets of  $T$  execution paths. Using a combinatorial argument, the  $O$  metric is shown to be “optimal”: its overall performance is at least as good as any other metric, for any number of tests. Although  $O$  is not strictly rational, there are strictly rational metrics such as  $O^P$  which are also optimal, so restricting attention to strictly rational metrics does not reduce potential performance, at least in this case. There are two conditions for a metric to be optimal for this simple model and performance measure (here we show the definition has much wider utility):

**Definition 6 (Single bug optimality)** *A metric  $M$  is single bug optimal if*

1. *when  $a_{ef} < F$ , the value returned is always less than any value returned when  $a_{ef} = F$ , that is,  $\forall F \forall P \forall a_{ep} \forall a'_{ep}$  if  $a_{ef} < F$  then  $M(a_{ef}, a_{ep}, F, P) < M(F, a'_{ep}, F, P)$ , and*
2. *when  $a_{ef} = F$ ,  $M$  is strictly decreasing in  $a_{ep}$ , that is, if  $a'_{ep} > a_{ep}$  then  $M(F, a'_{ep}, F, P) < M(F, a_{ep}, F, P)$ .*

The first condition is motivated by the fact that for single bug programs, the bug must be executed in all failed tests. Since  $a_{ef} = F$  for the bug, statements for which  $a_{ef} < F$  are best ranked strictly lower. The second condition is motivated by the fact that the bug tends to have a lower  $a_{ep}$  value than correct statements, because some executions of the bug lead to failure, the model program is symmetric with respect to buggy and correct statements, and all possible multisets of executions paths are used to evaluate overall performance. In addition to proving optimality under these very artificial conditions, Naish et al. (2011) conjectured such metrics were optimal for a wider class of models. In addition, empirical experiments were conducted with real programs and the optimal metrics performed better than other proposed metrics when measured using rank percentages.

Here we give an optimality result for single bug programs which does not constrain us to a simplistic program structure or performance measure or a particular distribution of sets of test cases. Indeed, we prove optimal performance for every set of test cases, not just overall performance. We obtain this *much* more widely applicable technical result, which also has a much simpler proof, by restricting attention to strictly rational metrics.

**Proposition 5** *Given any program with a single bug, any set of test cases and any single bug optimal metric  $M$  used to rank the statements, the rank cost equals the unavoidable cost.*

**Proof** The rank cost is  $\frac{GT+EQ/2}{S}$ . Since there is a single bug,  $b$ , the unavoidable cost is  $\frac{GT'_b+EQ'_b/2}{S}$  and  $a_{ef}^b = F$ .  $M$  is single bug optimal so any statement  $c$  ranked strictly higher than the bug must have the same  $a_{ef}$  value and a strictly lower  $a_{ep}$  value (so  $b <^s c$ ) and any statement ranked equal to the bug must have the same  $a_{ef}$  value and the same  $a_{ep}$  value as the



bug (so  $b =^s c$ ). Thus  $EQ = EQ'_b$  and  $GT = GT'_b$ .  
 $\square$

**Proposition 6** *Given any program with a single bug, any set of test cases and any single bug optimal metric  $M$  used to rank the statement, the rank cost using  $M$  is no more than the rank cost using any other strictly rational metric.*

**Proof** Follows from Propositions 2 and 5.  $\square$

The rank cost is not necessarily the best measure of performance. However, a consequence of this proposition is that for any cost measure which is monotonic in the rank cost, single bug optimal metrics have a lower or equal cost than any other rational metric. For example, optimality applies with respect to rank percentages and the simple cost measure of Naish et al. (2011) which only examines the statement(s) ranked (equal) top. Alternatively, a more complex non-linear cost function could be considered desirable, since the time spent finding a bug typically grows more than linearly in the number of different lines of code examined.

Drawing definitive conclusions from empirical experiments such as those of Naish et al. (2011) is normally impossible because the results may be dependent on the set of benchmark programs used, or the sets of test cases, or the details of the performance evaluation method. However, in this case we can use Proposition 6 to remove any doubt that the “optimal” metrics are indeed better than other metrics for single bug programs. Interestingly, there was one case found in Naish et al. (2011) where the optimal metrics did not perform the best overall — when sets of test cases were selected so that the buggy statement in the model program was executed in nearly every test. The only better metric was Russell, which benefits from a large number of ties in such cases, as discussed earlier. It was also noted that Russell performed better than the optimal metrics for some of the empirical benchmark programs, though its overall performance was worse. From Proposition 6 we know such behaviour can only occur for metrics which are not strictly rational.

## 6 Optimizing metrics for single bugs

$O^p$  was proposed as a metric which was single bug optimal and also expected to perform rather better than  $O$  for multiple bug programs. While this is true, experiments have shown that  $O^p$  does not perform particularly well for multiple bug programs (Naish et al. 2009). The two conditions for single bug optimality of a metric place no constraint on the relative ordering of statements for which  $a_{ef} < F$ . Any metric can thus be adapted so it becomes optimal for single bug by adding a special case for  $a_{ef} = F$  — we just need to ensure it is decreasing in  $a_{ep}$  and larger than any other value possible with the same  $F$  and  $P$ .

**Definition 7 (Optimal single bug version)** *The optimal single bug version of a metric  $M$ , denoted  $O1(M)$  is defined as follows.*

$$O1(M)(a_{ef}, a_{ep}, F, P) = \begin{cases} K + 1 + P - a_{ep} & \text{if } a_{ef} = F \\ M(a_{ef}, a_{ep}, F, P) & \text{otherwise,} \end{cases}$$

where  $K$  is the maximum of  $\{M(x, y, F, P) | x < F \wedge y \leq P\}$ .

**Proposition 7**  *$O1(M)$  is single bug optimal for all metrics  $M$ .*

Table 3: Description of Siemens + Unix benchmarks

Program	1 Bug	2 Bugs	LOC	Tests
<i>tcas</i>	37	604	173	1608
<i>schedule</i>	8	—	410	2650
<i>schedule2</i>	9	27	307	2710
<i>print_tok</i>	6	—	563	4130
<i>print_tok2</i>	10	10	508	4115
<i>tot_info</i>	23	245	406	1052
<i>replace</i>	29	34	563	5542
<i>Col</i>	28	147	308	156
<i>Cal</i>	18	115	202	162
<i>Uniq</i>	14	14	143	431
<i>Spline</i>	13	20	338	700
<i>Checkeq</i>	18	56	102	332
<i>Tr</i>	11	17	137	870

**Proof** When  $a_{ef} = F$ ,  $O1(M)$  is clearly strictly decreasing in  $a_{ep}$  and the value returned is at least  $K+1$ , since  $a_{ep} \leq P$ , so it is greater than any value returned when  $a_{ef} < F$ .  $\square$

In practice, many metrics range between 0 and 1 so we could just choose  $K = 1$  in all cases for these metrics. A larger fixed value, such as  $K = 999999$  is sufficient for all metrics proposed to date unless there are a very large number of test cases.

If  $a_{ef} < F$  for a large proportion of statements,  $O1(M)$  will produce a similar ranking to  $M$  and if  $M$  works very well for multiple bug programs, we would expect  $O1(M)$  to also work well. Of course,  $O1(M)$  will also work as well as any other rational metric for single bug programs.

## 7 Experimental results

We performed empirical evaluation using a collection of small C programs: the Siemens Test Suite (STS), from the Software Information Repository (Do, Elbaum & Rothermel 2005), plus several small Unix utilities, from Wong, Horgan, London & Mathur (1998). These, particularly STS, are widely used for evaluating spectral ranking methods. Table 3 gives the names of the programs (the first seven are from STS), and the numbers of single bug and two-bug versions, lines of code (LOC) and test cases. A small number of programs in the repository were not used because there was more than one bug according to our definition (for example, a `#define` was incorrect) or we could not extract programs spectra. We used the `gcov` tool, part of the `gcc` compiler suite, and it cannot extract spectra from programs with runtime errors. We generated the two-bug versions from pairs of single-bug versions, eliminating resulting programs if they encountered runtime errors, as in Naish et al. (2009). This collection of two-bug programs is far from ideal. However, most collections of buggy programs have a very strong bias towards single bugs or have only a relatively small number of program versions. Obtaining better benchmarks is clearly a priority.

We conducted experiments to compute the average unavoidable cost and the rank cost for each metric and the single bug optimal version of the metric, for both one and two bug benchmark sets. Table 4 gives the results. The unavoidable costs are given in the second row. The last column of figures uses the optimal single bug version of the metrics. For the single bug benchmark, the optimal single bug version of each metric gives a rank cost of 16.87, which is the unavoidable cost, so we omit these from the table. The original versions of the metrics range in perfor-

Table 4: Unavoidable and rank costs

Benchmark	1 Bug	2 Bug	2 Bug
Unavoidable	16.87	11.72	$O1(\dots)$
O	16.87	23.75	23.75
$O^p$	16.87	21.64	21.64
Wong3	17.20	21.34	21.56
Zoltar	17.24	19.32	21.42
Kulczynski2	18.07	18.32	21.24
Ochiai	20.63	18.95	21.18
Wong4	21.23	21.51	21.60
Jaccard	22.65	19.87	21.20
CBILog	25.23	21.04	21.56
Tarantula	26.10	21.91	21.54
Ample	29.17	23.26	21.75
Russell	29.02	30.88	21.82

Table 5: Two bug rank cost wrt  $a_{ef} = F$  category

$a_{ef} = F$ for ...	2 Bugs	1 Bug same	1 Bug inverted	No Bug
% of Cases	44	35	11	10
% $a_{ef} = F$	67	51	44	37
Unavoidable	17.55	9.53	2.43	3.25
Kul2	18.62	21.91	4.52	17.80
$O1(Kul2)$	17.55	20.88	17.51	42.43
$O^p$	17.55	20.88	17.51	46.50
O	17.55	20.88	17.51	68.04
Russell	32.96	25.40	21.83	48.40

mance for the single bug benchmark set, with  $O$  and  $O^p$  being the best, and equal to the unavoidable cost, as expected.

For the two bug benchmark set, the unavoidable cost, 11.72, is significantly lower. This is to be expected since it is essentially the minimum of the unavoidable costs of two bugs. However, the rank costs are higher for most metrics, and the best rank cost, 18.32, for Kulczynski2, is significantly higher than the unavoidable cost. Although it cannot be guaranteed, it seems likely better metrics to exist. The optimal single bug versions of the metrics show much less variation in rank cost and, unfortunately, perform significantly worse than the best metrics. We conducted additional experiments to better understand the performance of Kulczynski2 and the single bug optimal metrics.

Table 5 summarises the results. It breaks down the 2-bug benchmark set into four categories: the programs for which  $a_{ef} = F$  for both bugs, the programs for which  $a_{ef} = F$  for just one bug where  $O^p$  and Kulczynski2 rank the bugs in the same order, the programs for which  $a_{ef} = F$  for just one bug where  $O^p$  and Kulczynski2 rank the bugs in the opposite (inverted) order, and the programs for which  $a_{ef} = F$  for no bug. The first row of figures gives the percentages of cases for these four categories. Overall, in 90% of cases at least one bug is executed in all failed tests, which is generally helpful for the single bug optimal metrics. The second row gives the percentages of correct statements for which  $a_{ef} = F$  in the four categories. On average, 57% of correct statements are used in all failed tests, so the “special case” in  $O1$  actually applies to most statements, and  $O1$  affects the ranking more than expected for this benchmark set. This is the main reason why  $O1$  performs more poorly than anticipated. The second row gives the average unavoidable cost for each category. It shows significant variation in unavoidable cost and we discuss this further below. The following lines of Table 5 give the average rank cost (percentage) for the dif-

Table 6: Two bug rank cost wrt  $\%a_{ef} = F$ 

$\%a_{ef} = F$	<20	20–40	40–60	60–80	$\geq 80$
%of Cases	10.6	8.1	19.0	57.9	4.2
Kul2	7.05	10.99	17.67	21.33	19.65
$O1(Kul2)$	5.47	16.08	21.51	24.51	21.89
$O^p$	6.24	19.17	21.64	24.58	21.75
O	16.70	23.39	23.57	24.96	22.37
Russell	8.35	22.59	26.11	36.26	43.96

ferent metrics; Kulczynski2 is abbreviated to Kul2.

The three single bug optimal metrics have equal rank cost for the first three categories since the top-ranked bug has  $a_{ef} = F$  and the ranking of all such statements is the same with these metrics. The difference between these metrics and Russell indicates the usefulness of  $a_{ep}$  — Russell ranks according to  $a_{ef}$  and essentially ignores  $a_{ep}$ . In the first category, our treatment of ties is (arguably) unfair to Russell — 67% of statements, including both bugs, are tied at the top of the ranking. Some researchers report the “worst case” (67%) in such situations and assume the bugs are ranked at the bottom of this range (Chilimbi, Liblit, Mehra, Nori & Vaswani 2009); this over-estimation of cost is discussed in Naish, Lee & Kotagiri (2010). Some researchers report both the “best case” (0%) and the “worst case” (Wong et al. 2007). Here we assume both bugs are ranked in the middle of the range, but even this leads to some over-estimation. If both bugs were to appear at a random point amongst these ties, the top-most bug would have an average cost of around 22% rather than 33%. Similarly, in the fourth category, the  $O$  metric has both bugs tied with 63% of the correct statements, at the bottom of the ranking, and a fairer treatment of ties would give an average cost of 59% rather than 68%. With the better metrics there are far fewer ties and thus the over-estimation of cost is much less and we doubt it affects any overall conclusions. Our treatment of ties was motivated by much simpler analysis and could be refined further.

Kulczynski2 performs slightly worse (around 1.1%) than  $O1(Kulczynski2)$  for the first two categories, which cover the majority (79%) of cases. This is because around 1.1% of correct statements have lower  $a_{ef}$  and significantly lower  $a_{ep}$  values than the bugs, and they overtake both bugs in the ranking. However, in the third category, Kulczynski2 performs extremely well. Almost half the statements have  $a_{ef} = F$  and when a bug with a lower  $a_{ef}$  and  $a_{ep}$  is placed higher in the ranking, it overtakes nearly all these statements. Although this category account for only 11% of cases, it more than compensates for the cases when correct statements overtake the bugs. Kulczynski2 also performs significantly better than the other metrics in the last category. Russell and all the single bug optimal metrics rank the statements with  $a_{ef} = F$  highest, so the rank cost must be at least 37%, whereas Kulczynski2 does even better than its overall performance.

The unavoidable cost figures also underscore the importance of the number of statements which are executed in all failed tests. In the first category, where both bugs are executed in all failed tests, we may intuitively expect bug localization to be easiest and the good metrics should achieve their best performance. However, it actually has the highest unavoidable cost, by a large margin. As well as both bugs being executed in all failed tests, on average, two thirds of correct statements are also executed in all failed tests. Table 6 gives an alternative breakdown of the two-bug performance figures, based on

the percentage of statements which are executed in all failed tests. Most cases fall into the 60–80% range for this benchmark set. Performance for all metrics drops as the percentage increases, except when the percentage is very high. When the percentage is less than 20%, O1(Kulczynski2) performs better than all other metrics.

The good overall performance of Kulczynski2 compared to O1(Kulczynski2) and other single bug optimal metrics is thus strongly linked to the number of statements with  $a_{ef} = F$ . For larger programs we would expect this proportion to be significantly smaller. We know from Naish et al. (2011) that for the Space benchmark (around 9000 LOC) the rank percentages for the better metrics is around one tenth that of the Siemens Test Suite; this is partly due to a smaller percentage of statements with  $a_{ef} = F$ . We could also improve overall performance by initially computing this percentage using the spectra for all statements, then using it to select either Kulczynski2, if it is relatively large, or O1(Kulczynski2), for example.

We also note that the tests suites are designed primarily to detect the *existence* of bugs, not find the *location* of bugs. There is a desire for a large coverage of statements. For example the STS tests were designed with the aim of having every statement executed by at least 30 tests. Tests which execute a large percentage of the code are generally better for detecting the existence of bugs but are worse for locating bugs. We are hopeful that with better test selection strategies and larger programs, single bug optimal metrics can be of significant practical benefit.

## 8 Other related work

In Section 2 we referred to several papers which introduced new metrics for spectral fault localization, or evaluated metrics which had previously been introduced for other domains. Here we briefly review other related work. There are a couple of approaches which post-process the ranking produced which are equivalent to adjusting the metric, similar to our O1 function. The post-ranking method of Xie et al. (2010) essentially drops any statement which is not executed in any failed test to the bottom of the ranking. That of Debroy, Wong, Xu & Choi (2010) ranks primarily on the  $a_{ef}$  value and secondarily on the original rank. Thus if the original ranking is done with a strictly rational metric, the resulting ranking is the same as that produced by  $O^p$ .

Other variations on the statement spectra ranking method described in this paper attempt to use additional and/or different information from the program executions. Execution frequency counts for statements, rather than binary numbers, are used in Lee, Naish & Kotagiri (2010) to weight the different  $a_{ij}$  values and in Naish et al. (2009) aggregates of the columns of the matrix are used to adjust the weights of different failed tests. The RAPID system (Hsu, Jones & Orso 2008) uses the Tarantula metric but uses branch spectra rather than statement spectra.

The CBI (Liblit et al. 2005) and SOBER (Liu, Yan, Fei, Han & Midkiff 2005) systems use predicate spectra: predicates such as conditions of if-then-else statements are instrumented and data is gathered on whether control flow ever reaches that point and, if it does, whether the predicate is ever true. CBI uses sampling to reduce overheads but aggregates the data so there are four numbers for each predicate, which are ranked in a similar way to how statements are ranked using statement spectra. SOBER uses frequency counts and a different form of statistical ranking method. The Holmes system (Chilimbi

et al. 2009) uses path spectra: data is collected on which acyclic paths through single functions are executed or “reached”, meaning the first statement is executed but not the whole path, and the paths are ranked in a similar way to predicate ranking in CBI. Statement and predicate spectra are compared in Naish et al. (2010), and it is shown that the aggregate data used in predicate spectra methods is more expressive than that used for statements spectra and modest gains in theoretical performance are demonstrated. The data collected for path spectra contains even more information and thus could potentially be used to improve performance further.

## 9 Conclusion

Spectra-based techniques are a promising approach to software fault localization. Here we have used one of the simplest and most popular variants: ranking statements according to some metric, a function of the numbers of passed and failed tests in which the statement is/isn’t executed. We have identified the class of *strictly rational* metrics, which are strictly increasing in the number of failed test executions and strictly decreasing in the number of passed test executions. We have argued that it is reasonable to restrict attention to this class of metrics, and there is no apparent evidence that doing so reduces fault localization performance. Having made this restriction, we can put a lower bound on the cost of fault localization — the “unavoidable cost”. No strictly rational metric can achieve a lower cost.

We have shown that *single bug optimal* metrics perform at least as well as any other strictly rational metric, for various reasonable measures of performance, for all programs with a single bug and all sets of test cases. This significantly extends a previous theoretical result and shows that we cannot do any better with this variant of spectral fault localization for single-bug programs. We also showed how any metric can be adapted so it becomes single bug optimal.

Performance of spectral fault localization on multiple-bug programs is much less well understood. We have performed empirical experiments with a variety of metrics on a benchmark set of small two-bug programs. All metrics resulted in costs significantly greater than the unavoidable cost on average. Also, the single bug optimal metrics had significantly greater cost than the best metrics overall. We have identified one reason for this: typically a large proportion of statements are executed in every failed test. We know that this proportion is smaller in benchmarks with larger programs. Also, it may be practical to reduce it further by careful creation and selection of test cases. For the subset of our two-bug benchmark set where less than 20% of statements were executed in all failed tests, the best performance was achieved by the single bug optimal version of a metric known to perform well on multiple bug programs. Overall, there seem reasonable prospects for improving performance when there is a mixture of single- and multiple-bug programs, which is what real fault localization tools are faced with.

## References

- Abreu, R., Zoetewij, P. & van Gemund, A. (2006), ‘An evaluation of similarity coefficients for software fault localization’, *PRDC’06* pp. 39–46.
- Chen, M., Kiciman, E., Fratkin, E., Fox, A. & Brewer, E. (2002), ‘Pinpoint: Problem determination in

- large, dynamic internet services', *Proceedings of the DSN* pp. 595–604.
- Chilimbi, T., Liblit, B., Mehra, K., Nori, A. & Vaswani, K. (2009), HOLMES: Effective statistical debugging via efficient path profiling, in 'Proceedings of the 2009 IEEE 31st International Conference on Software Engineering', IEEE Computer Society, pp. 34–44.
- Dallmeier, V., Lindig, C. & Zeller, A. (2005), Lightweight bug localization with AMPLE, in 'Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging', ACM, pp. 99–104.
- Debroy, V., Wong, W., Xu, X. & Choi, B. (2010), A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques, in '10th International Conference on Quality Software', 2010. QSIC 2010'.
- Do, H., Elbaum, S. & Rothermel, G. (2005), 'Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact', *Empirical Software Engineering* **10**(4), 405–435.
- Gonzalez, A. (2007), Automatic Error Detection Techniques based on Dynamic Invariants, Master's thesis, Delft University of Technology, The Netherlands.
- Hsu, H., Jones, J. & Orso, A. (2008), RAPID: Identifying bug signatures to support debugging activities, in '23rd IEEE/ACM International Conference on Automated Software Engineering, 2008. ASE 2008', pp. 439–442.
- Jaccard, P. (1901), 'Étude comparative de la distribution florale dans une portion des Alpes et des Jura', *Bull. Soc. Vaudoise Sci. Nat* **37**, 547–579.
- Jones, J. & Harrold, M. (2005), 'Empirical evaluation of the tarantula automatic fault-localization technique', *Proceedings of the 20th ASE* pp. 273–282.
- Lee, H. J. (2011), Software Debugging Using Program Spectra, PhD thesis, University of Melbourne.
- Lee, H. J., Naish, L. & Kotagiri, R. (2010), Effective Software Bug Localization Using Spectral Frequency Weighting Function, in 'Proceedings of the 2010 34th Annual IEEE Computer Software and Applications Conference', IEEE Computer Society, pp. 218–227.
- Liblit, B., Naik, M., Zheng, A., Aiken, A. & Jordan, M. (2005), 'Scalable statistical bug isolation', *Proceedings of the 2005 ACM SIGPLAN* **40**(6), 15–26.
- Liu, C., Yan, X., Fei, L., Han, J. & Midkiff, S. P. (2005), 'Sober: statistical model-based bug localization', *SIGSOFT Softw. Eng. Notes* **30**(5), 286–295.
- Lourenco, F., Lobo, V. & Bação, F. (2004), 'Binary-based similarity measures for categorical data and their application in Self-Organizing Maps', *JOCLAD*.
- Naish, L., Lee, H. J. & Kotagiri, R. (2009), Spectral debugging with weights and incremental ranking, in '16th Asia-Pacific Software Engineering Conference, APSEC 2009', IEEE, pp. 168–175.
- Naish, L., Lee, H. J. & Kotagiri, R. (2010), Statements versus predicates in spectral bug localization, in 'Proceedings of the 2010 Asia Pacific Software Engineering Conference', IEEE, pp. 375–384.
- Naish, L., Lee, H. J. & Kotagiri, R. (2011), 'A model for spectra-based software diagnosis', *ACM Transactions on software engineering and methodology (TOSEM)* **20**(3).
- Ochiai, A. (1957), 'Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions', *Bull. Jpn. Soc. Sci. Fish* **22**, 526–530.
- Reps, T., Ball, T., Das, M. & Larus, J. (1997), The use of program profiling for software maintenance with applications to the year 2000 problem, in 'Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT', Springer-Verlag New York, Inc. New York, New York, USA, pp. 432–449.
- Russel, P. & Rao, T. (1940), 'On habitat and association of species of Anopheline larvae in south-eastern Madras', *J. Malar. Inst. India* **3**, 153–178.
- Wong, W. E., Debroy, V. & Choi, B. (2010), 'A family of code coverage-based heuristics for effective fault localization', *Journal of Systems and Software* **83**(2).
- Wong, W. E., Qi, Y., Zhao, L. & Cai, K. (2007), 'Effective Fault Localization using Code Coverage', *Proceedings of the 31st Annual IEEE Computer Software and Applications Conference* pp. 449–456.
- Wong, W., Horgan, J., London, S. & Mathur, A. (1998), 'Effect of Test Set Minimization on Fault Detection Effectiveness', *Software-Practice and Experience* **28**(4), 347–369.
- Xie, X., Chen, T. Y. & Xu, B. (2010), Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques, in '10th International Conference on Quality Software', 2010. QSIC 2010'.

# Importance of Single-Core Performance in the Multicore Era

Toshinori Sato      Hideki Mori      Rikiya Yano      Takanori Hayashida

Department of Electronics Engineering and Computer Science

Fukuoka University

8-19-1 Nanakuma, Jonan-ku, Fukuoka 814-0180, Japan

toshinori.sato@computer.org

## Abstract

This paper first investigates what the best multicore configuration will be in the future, when the number of usable transistors further increases. Comparing five multicore models: single-core, many-core, heterogeneous multicore, scalable homogeneous multicore, and dynamically configurable multicore, surprisingly unveils that single-core performance is a key to improve multicore performance. Based on the findings, this paper secondly proposes a technique to improve single-core performance. It is based on Intel's Turbo Boost technology. From the detailed simulations, it is found that the technique achieves single-core performance improvement.

**Keywords:** Multicore, Amdahl's law, Pollack's rule, Turbo Boost technology.

## 1 Introduction

Multicore processors have already been popular to improve the total performance (Howard, et al. 2010). Considering the power and temperature constraints, they might be the sole practical solution. A lot of studies to determine the best multicore configuration is conducted (Annavaram, et al. 2005, Balakrishnan, et al. 2005, Ekman and Stenstrom 2003, Hill and Marty 2008, Kumar, et al. 2005 and Morad, et al. 2006) and it is believed that the heterogeneous multicore is the best in power and performance trade-off. However, it is not clear whether this answer is still correct in the future. As Amdahl's law states, performance of parallel computing is limited by that of its serial computing portion inside. Hence, to utilize the increasing number of transistors for increasing the number of cores on a chip might not be the best choice.

This paper has two contributions. The one is that it investigates which configuration is the best multicore processor and unveils that single-core processor performance should be still improved. The other is that it proposes a technique that improves single-core performance, which we name Cool Turbo Boost technique.

The rest of the paper is organized as follows. The next section summarizes the related works. Section 3 investigates the best multicore configuration. Section 4 proposes Cool Turbo Boost technique. Section 5 concludes.

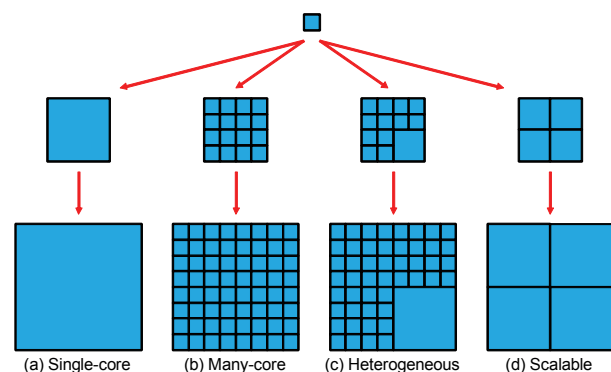
## 2 Related Works

There are a lot of studies investigating configurations of multicore processors (Annavaram, et al. 2005, Balakrishnan, et al. 2005, Ekman and Stenstrom 2003, Hill and Marty 2008, Kumar, et al. 2005 and Morad, et al. 2006). Most of them assume the number of usable transistors is fixed and then search the best processor configuration. Early studies mostly conclude that integrating a lot of simple cores is better in the power-performance trade-off than integrating a single complex core (Ekman and Stenstrom 2003). Later, heterogeneity and dynamic configurability are also considered. They might be keys to overcome Amdahl's law (Hill and Marty 2008).

On the top of the above studies, this paper further investigates what the best multicore configuration is. We also consider the advance in semiconductor technologies. We guess the best choice is different when the number of transistors increases.

## 3 Searching for Best Multicore

As the number of transistors on a chip increases, the flexibility to determine a processor configuration also increases. The current trend is to use them to integrate multiple cores on a chip and we have almost 50 cores (Howard, et al. 2010). With such a large flexibility, we are confused what the best configuration is. How many cores should be integrated on a chip? Should each core have simple in-order pipelines or complex out-of-order ones? Should all cores are the same? These questions have to be answered.



**Figure 1: Variations of Multicore Processors**

Figure 1 shows some variations of multicore processors. When chip integration is advanced, there are two choices. One is to increase the size of a core, and the other is to increase the number of cores. Figure 1a explains the former choice. All transistors on a chip are utilized by a

single core. Figure 1b explains the latter choice. The core microarchitecture is fixed and multiple copies of the core are integrated on the chip. Figures 1c and 1d consider hybrids of the two choices. In Figure 1c, only one core becomes large and the other cores remain small, and hence the multicore is heterogeneous. In Figure 1d, all cores become large and thus the multicore is homogeneous.

The best configuration must be different according to applications executed on the processor. Hence, we analyse theoretically and use simple model to compare these variations in the following subsections.

### 3.1 Single-core vs. Many-core

First, a single-core (Figure 1a) and a many-core (Figure 1b) processors are compared. As the single-core processor becomes larger and larger, its area-performance ratio meets a diminishing return as explained by Pollack's rule (Borkar 2007). It says processor performance is proportional to the square-root of its area. Performance improvement rate  $I_{Pollack}$  is expressed as:

$$I_{Pollack} = \sqrt{N}$$

where  $N$  explains that the processor is  $N$  times larger in area than the baseline one. On the other hand, multiprocessor performance is dominated by Amdahl's law. It says if a portion  $p$  of a program is executed in parallel by  $N$  baseline processors, the speedup  $I_{Amdahl}$  is:

$$I_{Amdahl} = \frac{1}{\frac{p}{N} + (1-p)}$$

Hence,  $p$  is an important factor that determines how efficient in performance the many-core processor is. Note that we use  $N$  both for area of the single-core processor and for the number of cores in the many-core processor. It is not confusing because the area of  $N$  cores equals that of  $N$ -times larger core.  $N$  is interchangingly used in this paper as the size of core, the number of cores, and the chip area.

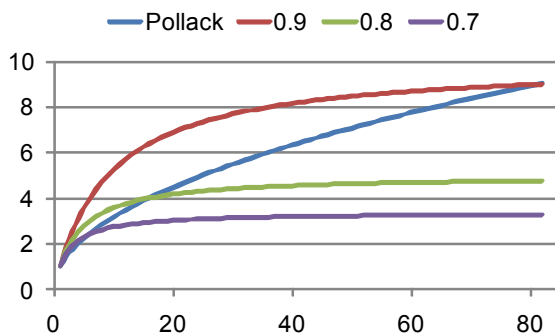


Figure 2: Pollack's Rule vs. Amdahl's Law

Figure 2 compares the single-core processors and the many-core processors. The horizontal axis indicates  $N$  and the vertical one indicates performance improvement rate. There are four lines. The blue line labelled with Pollack presents the performance improvement rate of the single-core,  $I_{Pollack}$ . The other three lines present the rate of the many-core,  $I_{Amdahl}$ . The labels indicate the parallelized portion,  $p$ .

When  $p$  is as large as 0.9, the many-core processor is always better in performance until  $N$  reaches 81. This

matches with the investigation in (Ekman and Stenstrom 2003): multiple simple cores are better in power-performance ratio than a big core. Please note that power consumption is proportional to the total area,  $N$ , in the model of Pollack's rule. Unfortunately, only large scale scientific computing enjoys such a large  $p$ . The conventional computing such as desktop and mobile cannot be easily parallelized.  $p$  is small (Wang, et al. 2009). As  $p$  becomes smaller and smaller, many-core performance is seriously limited. When  $p$  equals 0.7, the 8-core processor is poorer in performance than the 8-times larger single-core processor. We already have commercial 8-core processors such as Intel's Xeon 7500 series and AMD's FX series. Now is the time when single-core processors would be more beneficial for desktop and mobile applications than many-core processors, if single-core processor performance were ideally scalable to Pollack's rule.

### 3.2 Single-core vs. Heterogeneous Multicore

Next, the winner single-core processor (Figure 1a) is compared with a heterogeneous multicore processor (Figure 1c) (Kumar, et al. 2005). The heterogeneous or asymmetric multicore processors are widely studied for improving energy efficiency (Annaram et al. 2005, Balakrishnan et al. 2005 and Morad et al. 2006). They can be utilized to attack Amdahl's law (Hill and Marty 2008). Parallelized portions are executed by multiple small cores and hard-to-parallelize portions are executed by a big and strong core.

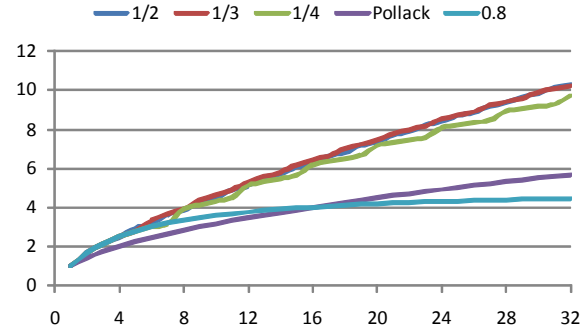


Figure 3: Single-core vs. Heterogeneous Multicore

Figure 3 compares the heterogeneous multicore processors with the single-core processors when  $p$  equals 0.8. Each heterogeneous multicore has only one big core. The figure has the same layout to Figure 2. The vertical axis additionally includes performance improvement rate of the heterogeneous multicore processors,  $I_{Hetero}$ . The lines labelled with Pollack and 0.8 are for  $I_{Pollack}$  and  $I_{Amdahl}$ . The other three lines are for  $I_{Hetero}$ . The labels 1/2, 1/3, and 1/4 mean the big core occupies half, one third, and one fourth of the chip area, respectively. Pollack's rule models the big core's performance. The rest of chip area is used by the baseline cores. Figure 1c presents the case labelled with 1/4.

Interestingly, the heterogeneous multicore processors have equivalent performance regardless of the big core's size and their performance is much scalable to the chip area. This confirms that hard-to-parallelize portion



dominates the speedup. If that portion can be executed by the big core, the speedup is significantly improved.

### 3.3 Heterogeneous vs. Scalable Homogeneous

Next, the winner heterogeneous (Figure 1c) and a scalable homogeneous multicore (Figure 1d) are compared with each other. The scalable multicore is different from the many-core investigated in Section 3.1 in that the former has the smaller number of large cores. The number of large cores is determined as follows. Guess when utilizing the half number of double-size cores become desirable in terms of performance. That is expressed as:

$$\frac{1}{(1-p) + \frac{p}{N}} < \frac{1}{\frac{1-p}{\sqrt{2}} + \frac{p}{\sqrt{2} \times \frac{N}{2}}}$$

$$N > \frac{\sqrt{2} \times p}{1-p}$$

Hence, when  $p$  equals 0.8 for example, 3 double-size cores have better performance than 6 small cores do.

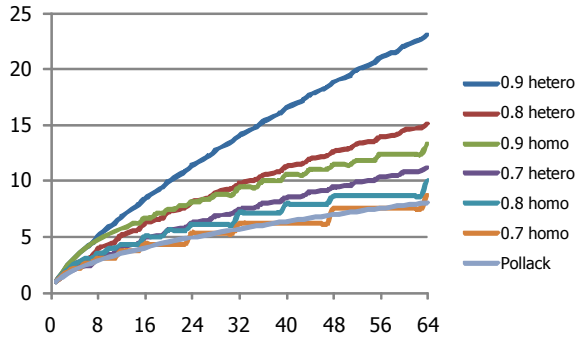


Figure 4: Heterogeneous vs. Scalable Homogeneous

Figure 4 compares two kinds of multicore processors: heterogeneous and scalable homogeneous. The figure has the same layout to Figures 2 and 3. The vertical axis additionally includes performance improvement rate of the scalable homogeneous multicore processors,  $I_{Homo}$ . The grey line labelled with Pollack presents  $I_{Pollack}$ . The other six lines consist of three for the heterogeneous and three for the scalable homogeneous. Each of their labels is the combination of  $p$  and the multicore type. For example, “0.9 hetero” and “0.7 homo” present  $I_{Hetero}$  when  $p$  equals 0.9 and  $I_{Homo}$  when  $p$  equals 0.7, respectively. The heterogeneous multicore utilizes one fourth of its chip area as its big core.

As for the heterogeneous multicore, performance is still improved when  $p$  is increased. When  $p$  is increased from 0.8 to 0.9, performance is improved by approximately 50%. In addition, different from the case of many-core, which we have already seen in Figure 2, its scalability is not diminished regardless of  $p$ . In contrast, the scalable homogeneous one shows poor performance. Even though  $p$  equals 0.9, its performance improvement rate is smaller than that of the heterogeneous multicore when  $p$  equals 0.8. This means that the heterogeneous multicore exploits performance even when hard-to-parallelize portions are large.

### 3.4 Hetero vs. Dynamically configurable

Up to now, the heterogeneous multicore processor is the best choice. However, we only investigated the statically configured multicores. We have not yet considered dynamically configurable multicores such as Core-fusion (Ipek, et al. 2007) and CoreSymphony (Wakasugi, et al. 2010). They dynamically configure the number of cores and the size of each core, as shown in Figure 5. When the currently executing portion of a program is easy to parallelize, the dynamically configurable multicore processor increases the number of cores. In contrast, otherwise, it combines some cores to a large core. The adaptability will improve the performance.

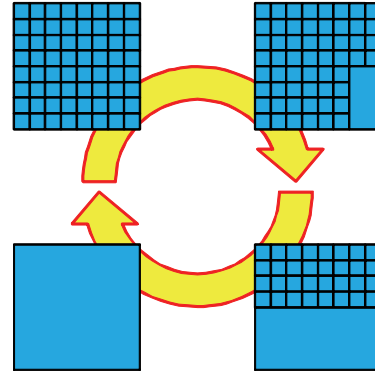
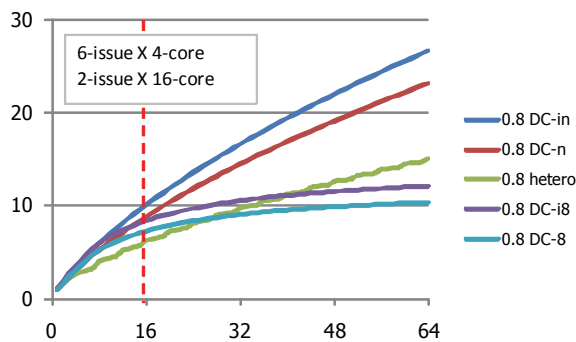


Figure 5: Dynamically configurable Multicore

Figure 6 compares the dynamically configurable multicore processor (Figure 5) with the current winner heterogeneous one (Figure 1c) when  $p$  is 0.8. Since we do not perform a simulation of an application, the configuration does not change dynamically but can be determined statically. The parallelizable part is executed by all small cores and the not-parallelizable part is executed by a single core, which is a combination of all small cores. The figure has the same layout to Figures 2-4. There are five lines. The green one labelled with “0.8 hetero” presents  $I_{Hetero}$ . The other four lines present performance improvement rate of the configurable multicore,  $I_{DC}$ . We consider four models of the reconfigurable multicore. The dynamic reconfigurability suffers a penalty in performance. It is approximately 25% of performance loss in comparison with a monolithic core (Ipek, et al. 2007 and Wakasugi, et al. 2010). The models labelled with “0.8 DC-n” and “0.8 DC-8” consider the penalty. The other models labelled with “0.8 DC-in” and “0.8 DC-i8” do not consider it and thus has an ideal single-core performance. As the number of cores increases, it becomes difficult to combine all cores due to the increasing complexity of interconnects. Hence, we consider a limit in the number of combinable cores. In this investigation, we assume the number is 8. The models labelled with “0.8 DC-i8” and “0.8 DC-8” consider the limit. The other models labelled with “0.8 DC-in” and “0.8 DC-n” do not consider it and thus are ideally scalable. The model “0.8 DC-8” is the most practical one.

When there are not any limits in the number of combinable cores, the performance improvement is very significant.  $I_{DC}$  is almost twice better than  $I_{Hetero}$ . The penalty slightly diminishes performance, but the

improvement rate is still fascinating. Comparing with  $I_{\text{Amdahl}}$ ,  $I_{\text{DC}}$  is increased by approximately 400% even if the penalty is considered. However, if the number of combinable cores is limited to 8,  $I_{\text{DC}}$  is seriously degraded. When  $N$  is around 30, the heterogeneous multicore becomes better in performance than the dynamically configurable one. As mentioned earlier, the model “0.8 DC-8” is the most practical one. Comparing it with the heterogeneous multicore unveils that the latter is better when  $N$  is larger than 27. The red dashed line in Figure 6 presents the current technology, where four 6-instruction-issue cores or sixteen 2-instruction-issue cores can be integrated on a chip. Remember that the integration is doubled generation by generation.  $N$  will be 32 very soon. Considering the above, the heterogeneous multicore processor is the best choice in the near future.



**Figure 6: Hetero vs. Dynamically configurable**

In order to enjoy the continuously improving performance of the heterogeneous multicore, one serious problem should be solved. It consists of a single big core and a lot of small cores and its configuration is statically determined on the design phase. Both size and performance of the big core have to be increased with the same pace of  $N$ . This means that single-core performance is still important.

#### 4 Single-Core Performance Improvement

This section presents a preliminary study that aims to improve single-core performance. Increasing clock frequency is the easiest way to improve single-core performance. However, as widely known, it also increases the power supply voltage, resulting in serious power and temperature problems. This section proposes a technique that increases clock frequency without the increase in the supply voltage.

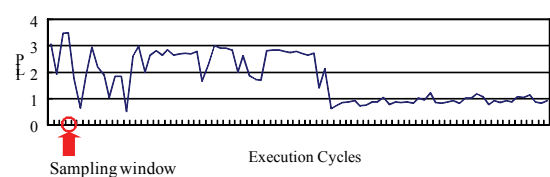
##### 4.1 Cool Turbo Boost Technique

Intel’s Turbo Boost technology (Intel Corporation 2008) has a unique feature that increases the supply voltage and thus the clock frequency when the number of active cores is small. This is possible because TPD (Thermal Design Power) is determined by considering the case when all cores are active and thus it has a large margin in that case. We extend it and further increase the core clock frequency. Different from the baseline Turbo Boost technology, our technique will not require the increase in the supply voltage, and hence we name it Cool Turbo Boost technology.

Cool Turbo Boost exploits the small critical path delay of a small core. If the hardware size and complexity become small, its critical path delay is reduced. Hence, there is an opportunity to increase its clock frequency. Intel’s ATOM processor (Thakkar 2008) is a good example that shows a simple and small core improves energy efficiency. In Cool Turbo Boost, processors datapath, where data flow and are processed, dynamically becomes small to boost clock frequency.

When the datapath becomes small, its computing performance is degraded. If the performance loss is not compensated by the clock frequency boost, the total processor performance is diminished. This is not our goal. Hence, we consider the following observation. When instruction level parallelism (ILP) is small, the small datapath is enough. Otherwise, the datapath should not become small. Hence, the datapath is dynamically configured according to ILP in each program phases. In order to realize the idea, we utilize Multiple Clustered-Core Processor (MCCP) (Sato and Funaki 2008). It is shown in Figure 7. MCCP configures its datapath according to ILP and thread level parallelism (TLP) in the program. We extend MCCP so that its clock frequency is increased when it configures its datapath small.

The amount of ILP varies between application programs and by more than a factor of two even within a single application program (Bahar and Manne 2001). Figure 8 shows an example of the issue rate for SPECint2000 benchmark gcc. The horizontal axis indicates the execution cycles and the vertical one represents the average number of instructions issued per cycles (issue IPC) over a window of 10,000 execution cycles. The issue IPC varies by more than a factor of two over a million cycles of execution. These variations can be exploited to determine when the datapath should become small. We manage MCCP to utilize wide datapath only when issue IPC is high and similarly to utilize narrow datapath only when issue IPC is low. When issue IPC is low, there are idle execution resources and thus the narrow datapath provides dependability without serious performance loss. We assume that past program behaviour indicates future behaviour. Hence, based on past issue IPC, future issue IPC could be predicted. We measure the number of instructions issued over a fixed sampling window. We predict future issue IPC based on the past number of issued instructions rather than on past issue IPC. We use predicted issue IPC to determine when the datapath should become small. If it is smaller than a predetermined threshold value, MCCP switches to use the narrow datapath. Similarly, if predicted issue IPC is larger than another predefined threshold value, MCCP switches to use the wide datapath.



**Figure 8: Issue IPC Variation (gcc)**



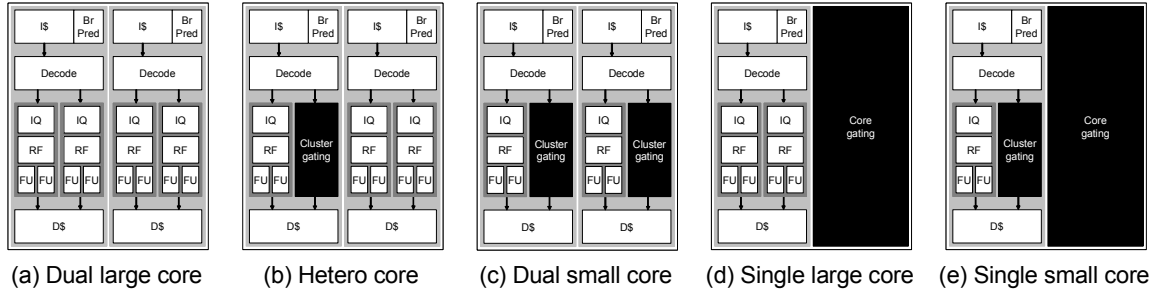


Figure 7: Multiple Clustered-Core Processor

## 4.2 Evaluation Methodology

SimpleScalar tool set (Austin, et al. 2002) is used for evaluation. Table 1 summarizes the processor configuration. The frontend and the L2 cache do not change. When issue IPC is larger than 2.0, the wide datapath is used. On contrary, when issue IPC is smaller than 1.6, the narrow datapath is used. These threshold values are not optimally determined and thus further study to determine the optimal values is required. Six programs: gzip, vpr, gcc, parser, vortex and bzip2 from SPECint2000 are used. 1 billion instructions are skipped before actual simulation begins. After that each program is executed for 2 billion instructions.

	Narrow	Wide
Fetch width	16 instructions	
L1 I cache	16KB,2-way	
Branch predictor	1K-gshare,512-BTB	
Dispatch width	4 instructions	
Scheduling queue	64 instructions	128 instructions
Issue width	2 instructions	4 instructions
Integer ALUs	2	4
Integer MULs	2	4
Floating ALUs	2	4
Floating MULs	2	4
L1 D cache	16KB,2-way,1-port	16KB,2-way,2-port
L2 cache	512KB,2-way	

Table 1: Processor Configurations

Boosting ratio is defined as the clock frequency in the narrow datapath mode divided by that in the wide mode. We vary the boosting ratio between 1.0 and 2.0 and evaluate how processor performance is improved.

## 4.3 Results

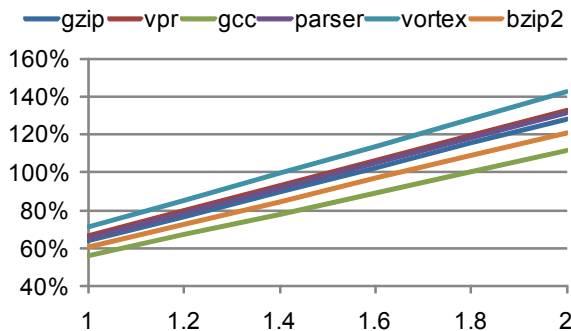


Figure 9: Narrow Datapath Results

Figure 9 presents the normalized performance when the narrow datapath is always used. The horizontal axis indicates the boosting ratio and the vertical one indicates the single-core performance normalized by the baseline performance. When the vertical value is less the 100%, processor performance is degraded. When the boosting rate is 1.0, performance is seriously diminished. It is not improved until the boosting ratio reaches 1.6. Hence, it is very difficult to improve single-core performance only by combining the narrow datapath with high clock frequency.

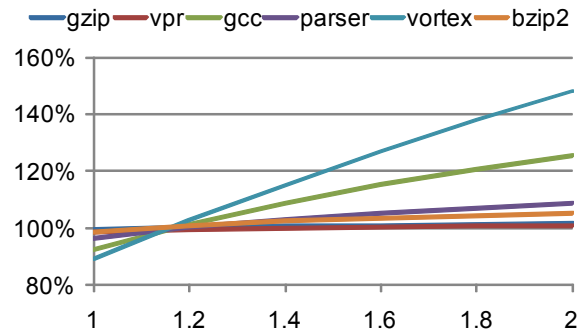


Figure 10: Cool Turbo Boosting Results

Figure 10 presents how Cool Turbo Boosting technique improves single-core performance. The figure has the same layout to Figure 9. When the boosting ratio equals 1.0, performance is degraded in all programs. However, the average performance loss is only 4.2% and is much smaller than that seen in Figure 9, which is 36.1% loss. When the boosting rate reaches 1.4 and 1.6, performance is improved by 5.0% and 8.7% on average, respectively. gzip and gcc achieve significant improvements, which are 26.7% and 15.2%, respectively.

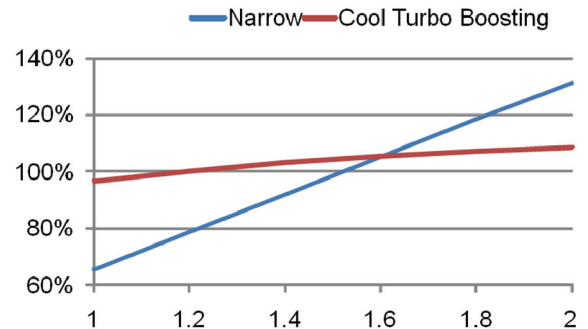
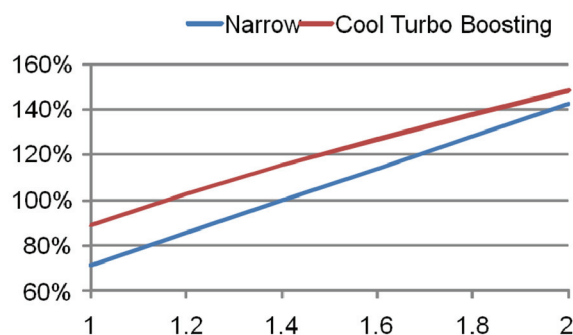


Figure 11: Comparison of 2 Techniques (parser)



**Figure 12: Comparison of 2 Techniques (vortex)**

Figures 11 and 12 compare 2 techniques. In Figure 11, parser represents the group of gzip, vpr, parser and bzip2. When the boosting ratio is large, Cool Turbo Boosting technique does not work well. In Figure 12, vortex represents the group of gcc and vortex. Cool Turbo Boosting achieves better performance regardless of the boosting ratio. While determining the boosting ratio requires future studies, the boosting ratio larger than 1.5 will be impractical. Because Cool Turbo boosting achieves single-core performance improvement in the small boosting ratio, it has the potential to further improve performance.

Achieving much single-core performance improvement requires further investigations.

## 5 Conclusions

This paper investigates what the best multicore configuration is in the future. Five models of single-core, many-core, heterogeneous multicore, scalable homogeneous multicore, and dynamically configurable multicore are compared with each other. From the investigations, it is unveiled that single-core performance is still important. Without the achievement, the heterogeneous multicore processor cannot continue to improve performance in the near future. This is the major contribution of this paper.

In the latter half of the paper, we present the preliminary case study that aims to improve single-core performance. We named it Cool Turbo Boost technique. When ILP in the program is small, the execution resources in the processor are dynamically configured to be narrow and thus its clock frequency is increased. From the detailed simulations, we found the average performance improvement of 5.0% is achieved. Unfortunately, this achievement is not enough to continue multicore performance improvement and the future studies are strongly required.

The future studies regarding the heterogeneous multicore processors include investigating the heterogeneity to enhance dependability. Hardware defects also cause heterogeneity. We are studying to utilize the cores with defects to improve dependability. For example, high performance is not always required for checking correctness. Combining the idea with the findings in this paper will explore a new horizon for dependable multicore processors.

## 6 Acknowledgments

This work was supported in part by JSPS Grant-in-Aid for Scientific Research (B) #20300019, and is supported in part by JST CREST program and by the fund from Central Research Institute of Fukuoka University.

## 7 References

- Annaram, M., Grochowski, E. and Shen, J. (2005): Mitigating Amdahl's law through EPI throttling. *Proc. International Symposium on Computer Architecture*, Madison, WI, USA:298-309, IEEE Computer Society Press.
- Austin, T., Larson, E. and Ernst, D. (2002): SimpleScalar: an infrastructure for computer system modeling. *IEEE Computer*, **35**(2):59-67.
- Bahar, R.I. and Manne, S. (2001): Power and energy reduction via pipeline balancing. *Proc. International Symposium on Computer Architecture*, Goteborg, Sweden:218-229, ACM Press.
- Balakrishnan, S., Rajwar, R. Upton, M. and Lai, K. (2005): The impact of performance asymmetry in emerging multicore architectures. *Proc. International Symposium on Computer Architecture*, Madison, WI, USA:506-517, IEEE Computer Society Press.
- Borkar, S. (2007): Thousand core chips: a technology perspective. *Proc. Design Automation Conference*, San Diego, CA, USA:746-749, ACM press.
- Ekman, M. and Stenstrom, P. (2003): Performance and power impact of issue-width in chip-multiprocessor cores. *Proc. International Conference on Parallel Processing*, Kaohsiung, Taiwan: 359-368, IEEE Computer Society Press.
- Hill, M.D. and Marty, M.R. (2008): Amdahl's law in the multicore era. *IEEE Computer* **41**(7): 33-28.
- Howard, J., et al. (2010): A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. *Digest of Technical Papers International Solid-State Circuit Conference*, San Francisco, CA, USA:19-21, IEEE Press.
- Intel Corporation (2008): Intel<sup>®</sup> Turbo Boost technology in Intel<sup>®</sup> Core<sup>™</sup> microarchitecture (Nehalem) based processors. *White Paper*.
- Ipek, E., Kirman, M., Kirman, N. and Martinez, J.F. (2007): Core fusion: accommodating software diversity in chip multiprocessors. *Proc. International Conference on Computer Architecture*, San Diego, CA, USA:186-197, ACM Press.
- Kumar, R., Tullsen, D.M., Jouppi, N.P. and Ranganathan, P. (2005): Heterogeneous chip multiprocessors. *IEEE Computer* **38**(11): 32-38.
- Morad, T.Y., Weiser, U.C., Kolodny, A., Valero, M. and Ayguade, E. (2006): Performance, power efficiency and scalability of asymmetric cluster chip multiprocessor. *IEEE Computer Architecture Letters* **5**(1): 14-17.
- Sato, T. and Funaki, T. (2008): Dependability, power, and performance trade-off on a multicore processor. *Proc. Asia and South Pacific Design Automation Conference*, Seoul, Korea:714-719, IEEE Computer Society Press.

- Thakkar, T. (2008): Intel Centrino Atom processor technology-enabling the best internet experience in your pocket. *Proc. Symposium on Low-Power and High-Speed Chips*, Yokohama, Japan:329-337.
- Wakasugi, Y., Sakaguchi, Y., Miyoshi, T. and Kise, K. (2010): An efficient physical register management scheme for CoreSymphony architecture. *IPSJ SIG Technical Report*, **2010-ARC-188**(3): 1-10 (in Japanese).
- Wang, Y., An, H., Yan, J., Li, Q., Han, W., Wang, L. and Liu, G. (2009): Investigation of factor impacting thread-level parallelism from desktop, multimedia and HPC applications. *Proc. International Conference on Frontier of Computer Science and Technology*, Shanghai, China:27-32.



# Explaining alldifferent

Nicholas Downing      Thibaut Feydy      Peter J. Stuckey

National ICT Australia\* and the University of Melbourne, Victoria, Australia  
Email: {ndowning@students., tfeydy@, pjs@}csse.unimelb.edu.au

## Abstract

Lazy clause generation is a powerful approach to reducing search in constraint programming. For use in a lazy clause generation solver, global constraints must be extended to explain themselves. Alternatively they can be decomposed into simpler constraints which already have explanation capability. In this paper we examine different propagation mechanisms for the *alldifferent* constraint, and show how they can be extended to explain themselves. We compare the different explaining implementations of *alldifferent* on a variety of problems to determine how explanation changes the trade-offs for propagation. The combination of global *alldifferent* propagators with explanation leads to a state-of-the-art constraint programming solution to problems involving *alldifferent*.

## 1 Introduction

Lazy clause generation (Ohrimenko et al. 2009) is a hybrid approach to constraint solving that combines features of finite domain propagation and Boolean satisfiability. Finite domain propagation is instrumented to record the reasons for each propagation step. This creates an implication graph like that built by a SAT solver, which may be used to create efficient nogoods that record the reasons for failure. These learnt nogoods can be propagated efficiently using SAT unit propagation technology.

The resulting hybrid system combines some of the advantages of finite domain constraint programming (CP): high level model and programmable search; with some of the advantages of SAT solvers: reduced search by nogood creation, and effective autonomous search using variable activities. Lazy clause generation provides state of the art solutions to a number of combinatorial optimization problems.

The *alldifferent* global constraint is one of the most common global constraints appearing in constraint programming models. *alldifferent*( $x_1, \dots, x_n$ ) requires that each of the variables  $x_1, \dots, x_n$  takes a different value. It is logically equivalent to  $\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$ . It succinctly encodes assignment subproblems occurring in a model. Such assignment subproblems occur frequently in real-world scheduling

and rostering problems, such as the *INSN\_SCHED*, *TAL-ENT\_SCHED* and *SOCIAL\_GOLFER* problems discussed in the experiments section of this paper.

Various implementations for the *alldifferent* constraint are available, some relying on specific propagation algorithms that enforce *VALUE*, *BOUNDS* and *DOMAIN* consistency, and some relying on decomposition of *alldifferent* into simpler constraints.

Learning changes the trade-offs for propagation. It may well be worth spending more time calculating stronger propagation, if the results can be reused elsewhere using learning, thus amortizing the cost over multiple uses. Conversely, it may be worth spending less time on propagation, if we can rely on the globality of learning to learn the stronger consequences of a constraint that are useful to the search in any case. Hence it is worthwhile studying what form of propagation of *alldifferent* is best for a learning solver.

Propagation algorithms for *alldifferent* have been quite well-studied, see the survey of van Hoeve (2001) for details. But until recently, global *alldifferent* propagators have not been used in learning solvers. Katsirelos (2008) describes a method for implementing the domain-consistent algorithm (Régin 1994) with explanations for use in a learning solver, but without experiments. We present for the first time an implementation of the method (with slight enhancements), and also we describe and implement for the first time an explained version of the bounds-consistent propagator (Lopez-Ortiz et al. 2003).

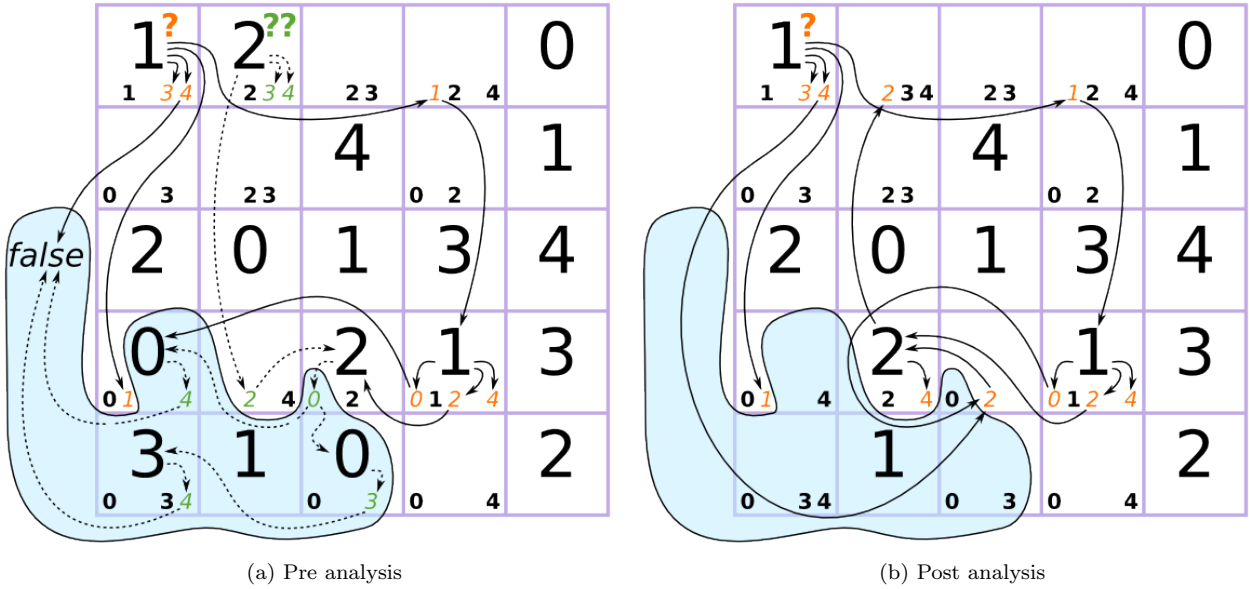
As well as the new propagators a further important contribution is a comprehensive suite of experiments using over 4000 hours of computer time to compare the learning vs. non-learning and global vs. decomposition approaches over a large set of structured problems that use *alldifferent*, and to find the best search strategy and solver combination for each problem, with comparison to previous state-of-the-art CP approaches to verify our results. We find learning to be enormously beneficial, so much so that new harder problems needed to be created to exercise our propagators, and that using the correct global or decomposed constraint is important on most models.

## 2 Lazy clause generation

We give a brief description of propagation-based solving and lazy clause generation, for more details see Ohrimenko et al. (2009). We consider constraint satisfaction problems (CSPs), consisting of constraints over integer variables  $x_1, \dots, x_n$ , each with a given finite domain  $D_{\text{orig}}(x_i)$ . A feasible solution is a valuation to the variables such that each  $x_i$  is within its allowable domain and all constraints are satisfied.

A propagation solver maintains a domain restriction  $D(x_i) \subseteq D_{\text{orig}}(x_i)$  for each variable and consid-

\*NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.  
Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

Figure 1: Conflict analysis on the implication graph of a  $5 \times 5$  QG\_COMPLETION problem

ers only solutions that lie within  $D(x_1) \times \dots \times D(x_n)$ . Solving interleaves propagation, which repeatedly applies propagators to remove unsupported values, and search which splits the domain of some variable and considers the resulting sub-problems. This continues until all variables are fixed (success) or failure is detected (backtrack and try another subproblem).

Lazy clause generation is implemented in the above framework by defining an alternative model for the domains  $D(x_i)$ , which is maintained simultaneously. Specifically, Boolean variables are introduced for each potential value of a variable, named  $[x_i = j]$  and  $[x_i \geq j]$ . Negating them gives the opposite,  $[x_i \neq j]$  and  $[x_i \leq j - 1]$ . Fixing such a *literal* modifies domains to make the corresponding fact true in  $D(x_i)$  and vice versa. Hence these literals give an alternate Boolean representation of the domain, which can support SAT reasoning.

In a lazy clause generation solver, the actions of propagators (and search) to change domains are recorded in an *implication graph* over the literals. Whenever a propagator changes a domain it must *explain* how the change occurred in terms of literals, that is, each literal  $l$  that is made true must be explained by a clause  $L \rightarrow l$  where  $L$  is a (set or) conjunction of literals. When the propagator causes failure it must explain the failure as a *nogood*,  $L \rightarrow \text{false}$ , with  $L$  a conjunction of literals which cannot hold simultaneously. Conflict analysis reduces  $L$  to a form suitable to use as a clausal propagator to avoid repeating the same search (Moskewicz et al. 2001).

**Example 2.1 (conflict analysis)** Figure 1a shows a simple  $5 \times 5$  Quasigroup Completion problem. Initially 11 of the 25 cells are filled in. The learning solver attempts to fill in the remaining 14 cells in such a way that the same digit does not appear twice in any row or column. In the (initially) blank cells is depicted a domain representation ‘1 2 3 4 5’ which shows the possible values for the cell.

A value can be removed from a domain (shown in *lightweight italics*) when that value is assigned to another cell in the same row or column. A value can be assigned to a cell when (i) the domain of the cell has been reduced to a single possibility, or (ii) it is the only cell in the same row (or column) that can take this value. Such reasoning is depicted graphically

by arrows showing, for each assignment/removal, its *preconditions* (a set of previous assignments/removals which must hold simultaneously).

Since the original problem was at fixed-point with respect to the above reasoning, search had to ‘pencil in’ the value 1 in the top-left corner, depicted ‘?’ (first decision level). Resulting implications are shown as solid arrows. Fixed-point being reached again (under this assumption), search pencilled in the value 2 in the next cell, depicted ‘??’ (second decision level). Resulting implications are shown as dotted arrows, to show they occurred at the second level.

These assumptions (1 and 2 in the initial cells), lead to a conflict because no cell in the first column can now take the value 4. The resulting conflict clause  $L \rightarrow \text{false}$  is depicted graphically.  $L$  simply expresses a rule of the puzzle and hence is useless as a learnt clause, so we have to look back in the implication graph to see the underlying causes of the conflict.

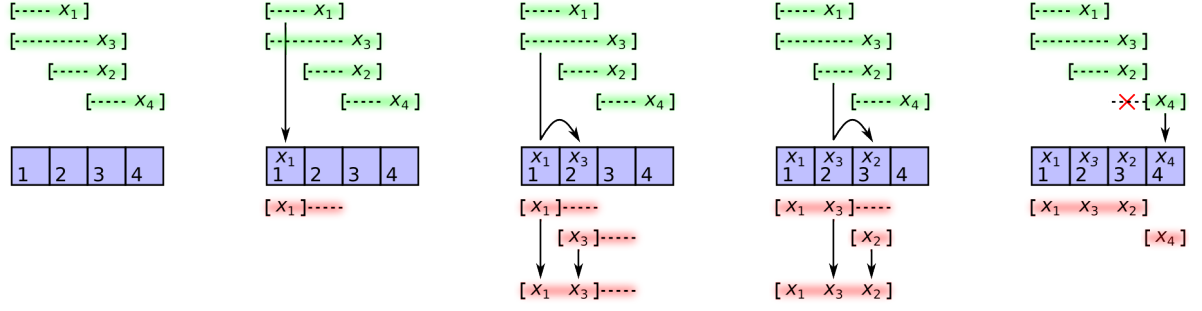
We use 1UIP conflict analysis (Moskewicz et al. 2001) to find the ‘cut’, depicted in the figures, which (i) contains the conflict, (ii) is as small as possible, and (iii) traces the conflict back to a single precondition at the current decision level, here  $[x_{43} = 2]$ . Observe that there are 3 implication arrows entering the cut (of which only one can be dotted). It is easy to see that only these preconditions need to exist simultaneously for failure to be inevitable.

The learnt nogood is simply a list of preconditions to the cut, here  $[x_{11} \neq 4] \wedge [x_{44} \neq 0] \wedge [x_{43} = 2]$ . After undoing all work at decision level 2, this new clause must propagate, as  $[x_{11} \neq 4] \wedge [x_{44} \neq 0] \rightarrow [x_{43} \neq 2]$ , which removes the immediate reason for the conflict. Inevitably this also propagates back as shown in Figure 1b, to undo the bad decision marked ‘??’. Due to this clause learning mechanism, search never makes the same mistake again.

### 3 Hall Sets

The *alldifferent* constraint requires that each argument takes a different value. The key to all propagation algorithms for *alldifferent* is the detection of *Hall sets* (Hall 1935). Given a constraint  $\text{alldifferent}(x_1, \dots, x_n)$ ,  $H \subseteq \{1, \dots, n\}$  is a Hall set if  $|H| \geq |V|$  where  $V = \cup_{h \in H} D(x_h)$ . If the inequality





(a) Initial domains (b) Singleton interval (c) Merging intervals (d) Finding Hall interval (e) Pruning lower bound  
Figure 2: Example of bounds-consistent propagator execution for pruning lower bounds

holds strictly, that is  $|H| > |V|$ , then the constraint is unsatisfiable. If it holds as an equality,  $|H| = |V|$ , then no variable  $x_i, i \notin H$  can take a value from  $V$ .

**Example 3.1 (Hall sets)** Given  $x_1 \in \{1, 2\}$ ,  $x_2 \in \{1, 3\}$  and  $x_3 \in \{1, 3\}$  are all different,  $H = \{2, 3\}$  is a Hall set with  $V = \{1, 3\}$ . Since 3 different variable-values can't fit in a domain containing only 2 values,  $x_1$  must be outside this domain, that is  $x_1 \neq 1$ .

In the next sections we examine various propagators for *alldifferent* and how they can be extended to explain their propagations. The explanation clauses are essentially descriptions of the well-known conditions for pruning. Usually these clauses also suffice to describe failure (because they wake up implicit clauses requiring domains to be non-empty) but in some cases explicit failure nogoods can also be produced.

#### 4 Global value-consistent propagator

The simplest form of *alldifferent*( $x_1, \dots, x_n$ ) is a decomposition that enforces  $x_i \neq x_j$  for all  $1 \leq i < j \leq n$ . Let  $E = \cup_{i=1}^n D_{orig}(x_i)$  be the union of the domains of all variables appearing in the *alldifferent* constraint. An equivalent decomposition based on a linear constraint is  $\sum_{i=1}^n \text{bool2int}([x_i = v]) \leq 1$  for all  $v \in E$ . Since the size of the decomposition is  $O(n|E|)$  we implement this as a single global propagator that wakes upon variable fixing, i.e. when  $D(x_h) = \{v\}$  for some  $h, v$ , it prunes all  $D(x_i), i \neq v$  with explanation

$$[x_h = v] \rightarrow [x_i \neq v].$$

The complexity of this propagator is  $O(n|E|)$ .

When  $|E| = n$ , there are no spare values and we also enforce the clauses  $\bigvee_{i=1}^n [x_i = v]$  for all  $v \in E$ , equivalent to changing the upper bound of 1 to equality with 1 in the above linear constraints. These clauses are standard in the SAT community (e.g. in the CNF output of Gomes's *lsencode* generator for QG\_COMPLETION problems) but their importance isn't widely recognised for CSPs.

#### 5 Global bounds-consistent propagator

Given the constraint *alldifferent*( $x_1, \dots, x_n$ ) over domains  $D(x_1), \dots, D(x_n)$ , bounds consistency ensures for each  $x_i$ , both  $a_i = \min(D(x_i))$  and  $b_i = \max(D(x_i))$  have a support over  $\prod_{j \neq i} a_j..b_j$ , i.e. a solution to the constraint relaxed to range domains, which uses the value  $x_i = a_i$  or  $b_i$ .

The best bounds-consistent *alldifferent* propagator is by Lopez-Ortiz et al. (2003). It rests on two key observations, (i) a solution to the constraint may be

found greedily, if one exists, by allocating each variable its minimum possible value, treating variables in the order most- to least-constrained; and (ii) a union-find data structure (Tarjan 1975) can efficiently encode the dependencies between interval domains, to build Hall intervals incrementally and inform us when a complete Hall interval has been identified.

**Example 5.1 (pruning bounds)** Suppose  $x_1 \in 1..2$ ,  $x_2 \in 2..3$ ,  $x_3 \in 1..3$  and  $x_4 \in 3..4$ . These intervals, along with a representation of the overall domain 1..4, are shown in Figure 2a. Initially, all cells of the domain representation are unoccupied. The variables are sorted in order of increasing upper bound, which is the criterion for constrainedness, since for example it would not make sense to allocate  $x_3 = 1$ ,  $x_2 = 2$  and then find all possibilities for  $x_1$  occupied.

The first variable to allocate is  $x_1 = 1$ , shown in Figure 2b. A new singleton interval is created in the union-find data structure, shown below the domain-representation. The endpoints of the interval are indicated by  $[\ ]$ , whereas the upper bound of the contained variable extends further, shown shaded and dotted. At present the new interval is not Hall; when its right-hand endpoint increases to take in the shaded region then it will become a Hall interval.

Referring to Figure 2c, we next allocate  $x_3 = 2$ . Because we had to jump over the value 1 to allocate  $x_3$ , the interval containing 1 is merged in the union-find data structure with the newly created interval. This merging preserves the invariant that for each interval in the data structure, a value in the interval can be freed up if and only if one of the variables in the interval has its bounds relaxed. The merged interval is still not a Hall interval since it contains  $x_3$  which has the highest upper bound, 3, shown.

Then, allocating  $x_2 = 3$  discovers a Hall interval (Figure 2d). Since the value 2 was passed over, the corresponding interval is merged with the new interval, including the value 1 which could only affect the new allocation indirectly. The upper bound of the newly merged interval has caught up with the upper bounds of the variables in it, so the interval is marked 'Hall'. Then when processing  $x_4$  (Figure 2e), we notice its lower bound falls into a Hall interval, and should be pruned before attempting any allocation.

Due to the order of discovering Hall intervals relative to the processing of variables, the algorithm as described above will only prune lower bounds. We use the original algorithm with minimal change, which includes a second pass of recomputing all Hall intervals to prune the upper bounds, though there is no reason in principle why the information discovered on the first pass should not be reused to save effort.

The algorithm of Lopez-Ortiz et al. (2003) is  $O(n \log n)$  to sort the variables, plus  $O(n \log n)$  to

scan variables and construct/maintain their (specialized) union-find data structures, overall  $O(n \log n)$ .

The only changes we made to the algorithm were (i) to use insertion sort for the variables at cost  $O(n^2)$ , in practice this takes only linear time since the variables are already sorted, and the algorithm is dominated by the union-find operations hence still  $O(n \log n)$ ; and (ii) to collect the set  $H$ , required for explanations. Given  $H$  with  $V = a..b$  we can explain the increased lower bound for a variable  $x_i \notin H$  as

$$[x_i \geq a] \wedge \bigwedge_{h \in H} ([x_h \geq a] \wedge [x_h \leq b]) \rightarrow [x_i \geq b + 1].$$

This requires  $O(n)$  literals per explanation.

## 6 Global domain-consistent propagator

For *alldifferent*( $x_1, \dots, x_n$ ) over domains  $D(x_1), \dots, D(x_n)$ , domain consistency ensures that for each  $x_i$ , each value in  $D(x_i)$  has a support, i.e. a solution to the entire constraint, which uses the value.

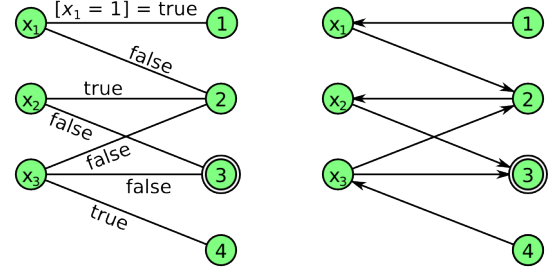
The best domain-consistent *alldifferent* propagator is by Régin (1994) with improvements by Gent et al. (2008). For *alldifferent* as bipartite graph matching problem, we can find a feasible solution (or prove that none exists) using Ford & Fulkerson's (1956) *augmenting paths* algorithm. The arcs (variable-value pairs) used in this solution are obviously supported. Support for another arc depends on whether there exists an *augmenting cycle* containing it, which we can check efficiently using Tarjan's (1972) *strongly connected components* (SCC) algorithm.

**Example 6.1 (augmenting paths)** Suppose  $x_1 \in \{1, 2\}$ ,  $x_2 \in \{2, 3\}$ ,  $x_3 \in \{2, 3, 4\}$ . Then a feasible solution is  $x_1 = 1$ ,  $x_2 = 2$ ,  $x_3 = 4$ , illustrated in Figure 3a. The corresponding residual graph, shown in Figure 3b, has a forward arc where a variable/value pair could be added to the matching or a backward arc where a variable/value pair could be removed.

Now suppose  $x_1 \neq 1$ . The *alldifferent* propagator wakes up and removes the illegal assignment from the graph as shown in Figure 3c, where unmatched nodes are double-circled. To repair the matching, a path is found in the residual graph (Figure 3d), from the unmatched variable  $x_1$  to an unmatched value 3. Augmenting along this path means adding to the matching when traversing forward arcs or removing for backward arcs (Figures 3e and 3f), so that  $x_1$  becomes 2 and  $x_2$  moves onto 3 in the proposed solution.

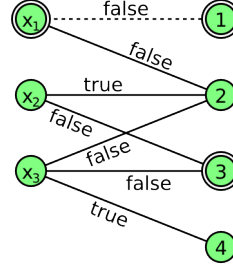
The actions of Régin's propagator may be explained by Hall sets. When Régin's propagator fails or prunes we can easily identify the failure set or Hall set which caused it. For infeasibility, the set of nodes searched for an augmenting path (the *cut*) consists of  $H \cup V$  and is a failure set as necessarily  $|H| > |V|$ . For pruning, the most recently discovered SCC consists of  $H \cup V$  and is a Hall set. We instrumented the propagator to use this knowledge to explain its failures and prunings. Note that we use SCC-splitting (Gent et al. 2008), and we generate explanations lazily.

**Example 6.2 (failure)** Continuing example 6.1, suppose  $x_1 \neq 1$  and also  $x_3 \neq 4$ . When the propagator wakes up it can repair  $x_1$  as shown previously (Figure 4a), but there is no augmenting path from  $x_3$  to an unused value, which the algorithm proves by searching the nodes indicated in Figure 4b before concluding that no further search is possible.

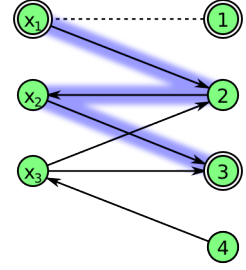


(a) Feasible solution

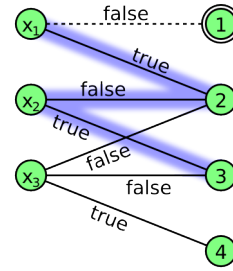
(b) Residual graph



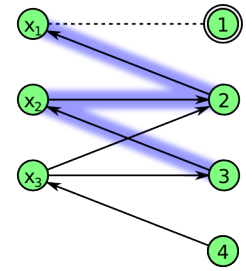
(c) Partial solution if  $x_1 \neq 1$



(d) Augmenting path

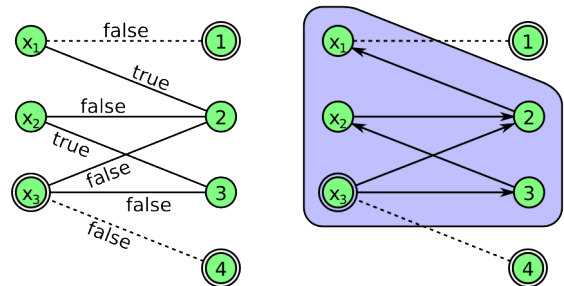


(e) Feasibility restored



(f) Flipped along the path

Figure 3: *alldifferent* as bipartite matching problem



(a) Partial solution if  $x_3 \neq 4$

(b) Set of nodes searched

Figure 4: Deriving an explanation for failure



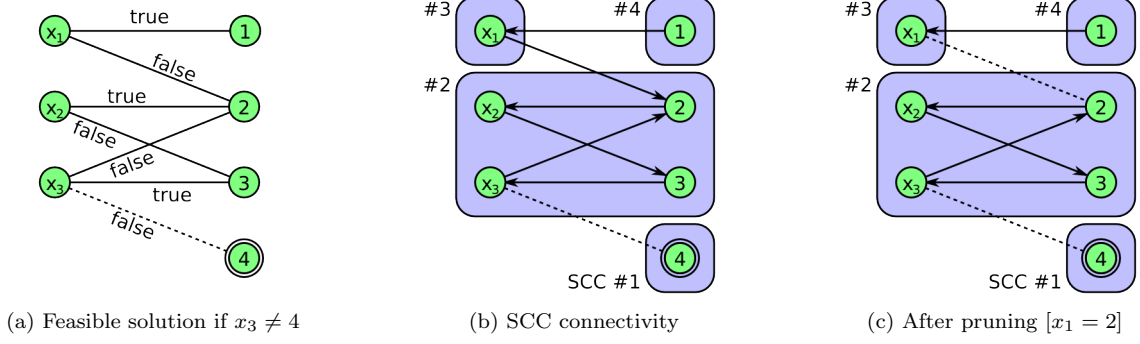


Figure 5: Isolating SCCs in depth-first manner while deriving explanations for the prunings

The resulting cut-set, partitioned into variables  $H = \{1, 2, 3\}$  and values  $V = \{2, 3\}$ , proves infeasibility since  $|H| > |V|$ , and suggests the failure nogood  $\bigwedge_{h \in \{1, 2, 3\}} (x_h \in \{2, 3\})$ . Since we do not have literals to express that  $x_h \in \{2, 3\}$ , we use an equivalent clausal representation  $x_h \neq 1 \wedge x_h \neq 4$ . By removing the literals that are false in the original domains, we obtain the nogood  $[x_1 \neq 1] \wedge [x_3 \neq 4] \rightarrow \text{false}$ . This final nogood is simply the list of dotted arcs leaving the cut; this is intuitive since the search stops precisely because those arcs are dotted.

**Example 6.3 (pruning)** Alternatively, suppose  $x_3 \neq 4$  while  $x_1 = 1$  remains possible. The propagator wakes up and repairs  $x_3$ , resulting in the feasible solution of Figure 5a. Using Tarjan’s algorithm it determines the SCCs of the resulting residual graph, shown in Figure 5b. The arc  $x_1 = 2$  crosses SCCs, so can’t be augmented (it is not part of any augmenting cycle in the residual graph), and may be removed.

The target of the arc being pruned is SCC #2 which gives the Hall set  $H = \{2, 3\}$ ,  $V = \{2, 3\}$  as evidence for the pruning, suggesting the explanation  $\bigwedge_{h \in \{2, 3\}} (x_h \in \{2, 3\}) \rightarrow [x_1 \neq 1]$ . Once again we can express this as the list of dotted arcs leaving the SCC, giving  $[x_3 \neq 4] \rightarrow [x_1 \neq 1]$ .

For the sake of simplicity we glossed over the distinction between augmenting paths and cycles. An arc may be augmented if it is part of any augmenting path, whereas the SCC-algorithm can only eliminate its appearing in an augmenting cycle. We get around this difficulty with a slight modification to Tarjan’s algorithm which makes used values reachable from unused values, so that freeing up a value by taking another value is considered to be a cycle.

In the worst case the edges are removed from the graph one by one, so there are  $n|E|$  propagator executions, each computing a single augmenting path at cost  $O(n|E|)$  and re-running the SCC-algorithm at cost  $O(n)$ , so the cost is  $O(n^2|E|^2)$  down a branch. In practice, repairing the matching is very fast (because it seldom explores the whole graph), and most time is spent in the SCC-algorithm.

Given  $H$  and  $V$  the explanation that we use for pruning values in  $j \in V$  from  $x_i \notin H$  is

$$\bigwedge_{h \in H, d \in E \setminus V} [x_h \neq d] \rightarrow [x_i \neq j] \quad (1)$$

It requires  $|H|(|E| - |V|)$ , or  $O(n|E|)$  literals per explanation in the worst case.

Our explanations are the same as Katsirelos’s (2008) except that, our explanations based on the list of dotted arcs leaving an SCC are quite general, so we naturally deduce and propagate equalities, rather than just disequalities as Katsirelos does.

**Example 6.4 (deducing equalities)** Pruning the arc from SCC #2  $\rightarrow$  #3 as described in Example 6.3 gives the residual graph of Figure 5c. Further pruning is possible:  $x_1$  may be fixed to 1, using SCC #3 as evidence, yielding explanation  $[x_1 \neq 2] \rightarrow [x_1 = 1]$ .

## 7 alldifferent by decomposition

*alldifferent* can also be implemented by decomposition into simpler constraints which already have explanation capability. The obvious decomposition is the conjunction of disequalities discussed in Section 4. A decompositions which prunes the same as the bounds-consistent global propagator is available (Bessiere et al. 2009b). There is no polynomially sized decomposition of *alldifferent* into clauses which enforces domain consistency (Bessiere et al. 2009a).

The attraction of decomposition is the ease of understanding, implementation and experimentation. We also cannot rule out that decompositions may perform better, by channelling the problem into more appropriate variables, or by making intermediate variables and implications available for conflict analysis which would otherwise have remained implicit.

A new decomposition of *alldifferent*, as a special case of *gcc* (Global Cardinality Constraint), was introduced by Feydy & Stuckey (2009). It is defined in *MiniZinc* (Nethercote et al. 2007) as follows:

```

predicate alldifferent_feydy_decomp(
    array[int] of var int: x) =
    let { int: L = lb_array(x),
          int: C = ub_array(x) + 1 - L,
          int: N = length(x),
          array[1..C] of var 0..1: c,
          array[0..C] of var 0..N: s } in
    s[0] = 0 /\ s[C] = N /\
    forall (i in 1..C) (
        s[i] = s[i - 1] + c[i] /\
        c[i] = sum (j in 1..N) (bool2int(x[j] = i)) /\
        s[i] = sum (j in 1..N) (bool2int(x[j] <= i));
    )
    
```

The new decomposition is efficient because it uses the literals  $[x_i = v]$  and  $[x_i \leq v]$ , which are native in a lazy clause generation solver, as they are part of the integer variable encoding.

To see how this works consider the simplest case when there are no spare values. Then the constraints  $s_i = s_{i-1} + c_i$  simplify to  $c_i = 1$  and  $s_i = i$ . This gives the consistency level described in Section 4, plus the detection of Hall intervals aligned to the start or end of the domain interval  $\min(E)..\max(E)$  where  $E$  is the union of the domains of the variables.

**Example 7.1** Suppose  $x_1 \in 1..2$ ,  $x_2 \in 1..3$ ,  $x_3 \in 2..3$ . No propagation is possible, e.g. considering the

constraint  $s_1 = \sum_{i=1}^3 \text{bool2int}([x_i \leq 1]) = 1$ , we find

$$\begin{aligned} & \min(\text{bool2int}([x_1 \leq 1])) \\ & + \max(\text{bool2int}([x_2 \leq 1])) \\ & + \max(\text{bool2int}([x_3 \leq 1])) = 0 + 1 + 0 \geq 1 \end{aligned}$$

which supports  $\text{bool2int}([x_1 \leq 1]) = 0$ . But if the interval for  $x_2$  becomes 2..3, then the preceding test gives  $0 + 0 + 0$  which is no longer  $\geq 1$ , therefore  $\text{bool2int}([x_1 \leq 1]) = 0$  is unsupported, and is pruned with explanation  $[x_2 \geq 2] \wedge [x_3 \geq 2] \rightarrow [x_1 \leq 1]$ .

## 8 Experiments

We implemented the *alldifferent* constraint in *Chuffed*, a state-of-the-art lazy clause generation solver. Hardware was a cluster of Dell PowerEdge 1950 with  $2 \times 2.0$  GHz Intel Quad Core Xeon E5405,  $2 \times 6$ MB Cache, and 16 GB RAM. Timeouts were 1800s, and each core was limited to 1 GB RAM in our experiments. Data files are available from <http://www.csse.unimelb.edu.au/~pjs/alldifferent>.

The *alldifferent* implementations we compare in *Chuffed* were VALUE, the global value consistent propagator described in Section 4; BOUNDS, the global bounds consistent propagator of Section 5; DOMAIN, the global domain consistent propagator of Section 6; and FEYDY, the *gcc*-based decomposition of Section 7. We also examined value-consistent *alldifferent* by decomposition to disequalities or *linear* constraints, and the bounds consistent decomposition of Bessiere et al. (2009b), but found them universally worse than the corresponding globals of equal propagation strength.

The search strategies were: IO, input order, an appropriate static search depending on the model; DWD, *dom/wdeg* search (Boussemart et al. 2004); and ACT, activity-based (VSIDS) search (Moskewicz et al. 2001). We use Luby restarts (Luby et al. 1993) for dynamic search strategies (DWD and ACT) if learning. Note that ACT is inapplicable without learning since it is the process of conflict analysis which collects activity counts, hence was only run with learning.

In the first experiment we take all benchmarks involving the global *alldifferent* constraint from the Third International CSP Solver Competition (CSP 2008) plus the CSP2008 QCP and QWH benchmarks which were only available in extensional form and had to be converted to use *alldifferent* directly. On these benchmarks the leading CSP2008 solvers were CPHYDRA, MISTRAL and SUGAR, and we compare against the published results of the competition, noting that they use an older Xeon architecture, but as they run at 3.0 GHz the performance should be comparable. We excluded the trivial PIGEONS instances, and certain BQWH instances where for some reason published results were not available. The CSP2008 solvers use their default strategy as in the published results, shown as IO in the table.

Table 1 reports the geometric mean of runtimes for the first experiment (using the timeout for timed-out instances), with the number of timeouts appearing as a superscript. For each model we show how many solved instances were unsatisfiable or satisfiable and how many were indeterminate as not solved by any solver. These latter instances aren't included in the runtime or timeouts statistics. In each block the solver with the fewest timeouts is highlighted, with ties broken by runtimes. Memory-outs were treated as timeouts (these occur on FEYDY only). Referring to Table 1 our solver is clearly far superior to the winners of the CSP2008 competition (PATAT is an exception

which arises because *MiniZinc* produced a particularly poor decomposition for the  $(x_0 \neq x_1) \vee (x_2 \neq x_3)$  constraints appearing in this model, a problem we did not address due to time constraints).

Since our runtimes were so small as to be barely measurable in most cases, we compiled a new set of much harder problems, based on the *MiniZinc* 1.1.6 benchmark suite (Nethercote et al. 2007) and the suite of Gent et al. (2008), with some additional models and additional harder instances. We compare on these models versus the state-of-the-art constraint programming system *Gecode* (*Gecode* Team 2006), running the same *MiniZinc* models as *Chuffed*. *Gecode* has won every *MiniZinc* Challenge (G12 Project 2010) run so far! The models are:

GOLOMB\_RULER (prob006 in CSPLib (Gent & Walsh 1999)),  $n = 8..11$  is a problem of placing  $n$  marks on a rule so that all the distances between the marks are distinct.

INSN\_SCHED, instruction scheduling for single-issue pipelined CPUs, similar to Lopez-Ortiz et al. (2003). Instances were obtained by compiling MediaBench benchmarks with *gcc* 4.5.2, switches *-O3 -march=barcelona -fsched-verbose=5*, and taking all sequences with 250..999 instructions. These problems are interesting as the AMD Barcelona-core CPUs have instructions with various latencies. These CPUs are multiple-issue, requiring a *gcc* constraint, so we consider a hypothetical single-issue version of the CPU requiring only *alldifferent*. We omit redundant constraints, they can improve performance, but their number grows quadratically, hurting scalability.

KAKURO, a grid-based puzzle similar to a crossword but with a numeric grid and arithmetic clues. We used the grid generator at [http://www.perlmonks.org/?node\\_id=550884](http://www.perlmonks.org/?node_id=550884) to generate 10 puzzles of size  $25 \times 25$  with coverage 50%. We use the redundant *alldifferent-sum* constraints of Simonis (2008), but not Simonis's *interact* constraints.

KNIGHTS\_TOUR, finding a cyclic knight's tour of length 56, 58, ..., 64 on an  $8 \times 8$  chessboard.

LANGFORD, Langford's number problem (prob024 in CSPLib) which is to sequence  $k$  sets of numbers  $1..n$  such that each occurrence of a number  $i$  is  $i$  numbers apart from the next  $i$  in the sequence. The selected instances are from  $k = 2.4 \times n = 3..24$ , taken from the *MiniZinc* benchmarks.

QG\_COMPLETION, Quasigroup Completion (QCP) is given an  $n \times n$  array where some cells are filled in with numbers from 1 to  $n$ , fill in the rest of the cells so that each row and each column contains the set of numbers  $1..n$ . We used Gomes's *lsencode* generator, <http://www.cs.cornell.edu/gomes/Soft/lsencode-v1.1.tar.Z>, to generate 160 problems of sizes  $30 \times 30..45 \times 45$ . An arbitrary random problem generated in this way is also usually too easy, so we used *picoSAT* 936 from <http://fmv.jku.at/picosat> to test the problems, keeping only those which were still being solved after 10 seconds on all of 10 randomized attempts. For example, on size  $30 \times 30$  we generated 27855 instances to find 10 which were hard enough.

QG\_EXISTENCE, Quasigroup Existence (prob003 in CSPLib) looks for a quasigroup of size  $n$  which satisfies various other criteria. We use sizes  $8 \times 8..13 \times 13$ , variants  $QG3..7 \times \{\text{idempotent, nonidempotent}\}$ , except that  $QG6..7$  are always idempotent. Redundant constraints are from Colton & Miguel (2001).

SOCIAL\_GOLFER (prob010 in CSPLib) is to schedule a golf tournament for  $n \times m$  golfers over  $p$  weeks playing in groups of size  $m$  so that no pair of golfers plays twice in the same group. We use instances taken from <http://>

model unsat, sat, ?	CSP2008, <i>solver</i> =				NOLEARN, <i>alldiff</i> =				LEARN, <i>alldiff</i> =			
	CPHYDRA	MISTRAL	MISTRAL'	SUGAR	VALUE	BOUNDS	DOMAIN	FEYDY	VALUE	BOUNDS	DOMAIN	FEYDY
BQWH IO DWD 0, 20, 0 ACT	0.09s	0.16s	<b>0.08s</b>	1.29s	0.24s 0.02s	0.53s 0.04s	<b>0.00s</b> 0.00s	0.01s 0.01s	0.02s 0.01s 0.02s	0.04s 0.02s 0.04s	0.01s 0.00s <b>0.00s</b>	0.02s 0.01s 0.02s
COSTAS- IO ARRAY DWD 0, 10, 1 ACT	6.28s <sup>1</sup>	5.42s <sup>2</sup>	<b>1.92s<sup>1</sup></b>	21.23s <sup>2</sup>	1.49s <sup>1</sup> <b>0.77s<sup>1</sup></b>	3.12s <sup>1</sup> 1.60s <sup>1</sup>	8.16s <sup>2</sup> 2.18s <sup>1</sup>	12.23s <sup>2</sup> 38.16s <sup>4</sup>	5.34s <sup>2</sup> <b>1.78s<sup>1</sup></b> 5.24s <sup>2</sup>	9.94s <sup>3</sup> 1.25s <sup>2</sup> 11.34s <sup>2</sup>	11.38s <sup>3</sup> 2.31s <sup>1</sup> 15.68s <sup>2</sup>	24.97s <sup>4</sup> 17.51s <sup>3</sup> 19.19s <sup>3</sup>
LATIN- IO SQUARE DWD 7, 3, 0 ACT	6.27s <sup>5</sup>	4.49s <sup>5</sup>	4.35s <sup>5</sup>	<b>2.09s<sup>1</sup></b>	0.99s <sup>5</sup> <b>0.74s<sup>4</sup></b>	1.42s <sup>5</sup> 1.09s <sup>4</sup>	1.62s <sup>5</sup> 1.32s <sup>4</sup>	1.75s <sup>5</sup> 1.78s <sup>5</sup>	0.16s 0.11s <sup>1</sup> <b>0.00s</b>	0.49s <sup>1</sup> 0.27s <sup>1</sup> 0.01s	0.55s <sup>1</sup> 0.49s 0.01s	2.18s <sup>3</sup> 0.97s <sup>2</sup> 0.03s
MAGIC- IO SQUARE DWD 0, 17, 1 ACT	53.35s <sup>9</sup>	<b>11.22s<sup>6</sup></b>	36.46s <sup>9</sup>	65.72s <sup>9</sup>	165.73s <sup>14</sup> <b>103.31s<sup>13</sup></b>	170.30s <sup>14</sup> 121.10s <sup>13</sup>	172.82s <sup>14</sup> 120.84s <sup>13</sup>	179.22s <sup>14</sup> 130.61s <sup>13</sup>	170.85s <sup>14</sup> 15.97s <sup>9</sup> 2.81s <sup>1</sup>	190.90s <sup>14</sup> 27.82s <sup>9</sup> 5.86s <sup>4</sup>	172.51s <sup>14</sup> 107.06s <sup>12</sup> <b>5.08s</b>	194.31s <sup>14</sup> 48.64s <sup>11</sup> 36.64s <sup>9</sup>
ORTHO- IO LATIN DWD 1, 3, 5 ACT	6.18s <sup>2</sup>	<b>2.72s<sup>1</sup></b>	3.91s <sup>2</sup>	25.77s <sup>1</sup>	0.28s <sup>1</sup> <b>0.34s</b>	0.82s <sup>1</sup> 1.02s <sup>2</sup>	2.07s <sup>1</sup> 1.16s <sup>2</sup>	0.88s <sup>1</sup> 1.26s <sup>2</sup>	0.58s <sup>1</sup> 0.15s 0.22s	0.94s <sup>1</sup> 0.24s 0.20s <sup>1</sup>	0.88s <sup>1</sup> <b>0.06s</b> 0.33s <sup>1</sup>	1.07s <sup>1</sup> 0.54s <sup>1</sup> 0.98s <sup>1</sup>
PATAT IO DWD 0, 42, 4 ACT	272.80s <sup>2</sup>	351.82s <sup>15</sup>	<b>59.18s<sup>1</sup></b>	375.50s <sup>18</sup>	1800.00s <sup>42</sup> 1272.96s <sup>40</sup>	1800.00s <sup>42</sup> 1377.46s <sup>41</sup>	1800.00s <sup>42</sup> 1406.26s <sup>41</sup>	1800.00s <sup>42</sup> <b>1216.18s<sup>39</sup></b>	771.11s <sup>35</sup> <b>170.75s<sup>12</sup></b> 518.98s <sup>31</sup>	845.78s <sup>35</sup> 230.99s <sup>13</sup> 632.88s <sup>29</sup>	921.16s <sup>35</sup> 429.64s <sup>20</sup> 532.62s <sup>32</sup>	1228.48s <sup>38</sup> 831.64s <sup>33</sup> 1348.48s <sup>36</sup>
QCP IO DWD 20, 40, 0 ACT	10.48s <sup>5</sup>	9.44s <sup>18</sup>	11.50s <sup>21</sup>	<b>6.08s</b>	0.05s <sup>2</sup> <b>0.01s</b>	0.06s <sup>3</sup> 0.02s	0.05s 0.02s	0.22s <sup>5</sup> 0.06s	0.02s 0.01s <b>0.01s</b>	0.03s 0.01s 0.01s	0.03s 0.01s 0.01s	0.14s 0.06s 0.05s
QUASI- IO GROUP DWD 18, 12, 5 ACT	0.86s <sup>2</sup>	0.87s <sup>3</sup>	0.53s <sup>2</sup>	<b>2.91s<sup>1</sup></b>	0.06s 0.03s	0.06s 0.03s	0.05s <b>0.03s</b>	0.07s 0.03s	0.08s <sup>1</sup> 0.03s <sup>1</sup> 0.06s <sup>2</sup>	0.08s <sup>1</sup> <b>0.03s<sup>1</sup></b> 0.06s <sup>2</sup>	0.07s <sup>1</sup> 0.03s <sup>1</sup> 0.05s <sup>2</sup>	0.09s <sup>1</sup> 0.05s <sup>1</sup> 0.08s <sup>2</sup>
QWH IO DWD 0, 40, 0 ACT	3.68s	2.10s <sup>5</sup>	2.68s <sup>10</sup>	<b>2.85s</b>	0.03s <b>0.01s</b>	0.04s 0.01s	0.03s 0.01s	0.14s 0.04s	0.02s 0.01s <b>0.01s</b>	0.02s 0.01s 0.01s	0.02s 0.01s 0.01s	0.10s 0.04s 0.05s
OTHER IO DWD 0, 3, 0 ACT	2.28s	9.75s <sup>1</sup>	<b>0.64s</b>	17.28s	<b>6.34s<sup>1</sup></b> 12.22s <sup>1</sup>	7.40s <sup>1</sup> 15.18s <sup>1</sup>	13.17s <sup>1</sup> 21.63s <sup>1</sup>	54.72s <sup>2</sup> 70.45s <sup>2</sup>	0.38s <sup>1</sup> <b>0.02s</b> 0.03s	5.00s <sup>1</sup> 0.22s 0.42s	10.38s <sup>1</sup> 0.18s 1.22s	62.23s <sup>2</sup> 81.70s <sup>2</sup> 92.14s <sup>2</sup>

Table 1: Models from the CSP2008 solver competition, against published results

<http://www.cs.brown.edu/~sello/solutions.html> and <http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html>. The model has 2 *alldifferent* constraints, referred to as *alldiff0* between players in a group, and *alldiff1* between pairs of players overall. Symmetries are broken lexicographically. We use integer variables, whereas most CSP approaches use set variables, e.g. Gange et al. (2010), and we don't claim state-of-the-art results, only that the model exercises our propagators.

TALENT\_SCHED (prob039 in CSPLib) schedules the scenes in a film to minimize the cost of the schedule in terms of actors' fees, where an actor is paid for the time from the first scene they are in until the last scene they are in. We use the three instances from CSPLib (a rehearsal problem plus *film1* and *film2* based on real data), plus the randomly generated *film1*?? instances of Smith (2005). The input-order (static) search is based on Smith's but due to time constraints we haven't yet implemented the redundant constraints described by Smith.

Since *Gecode* supports the search strategies IO and DWD we also give the geometric means for conflict counts for each benchmark under the times in Tables 2 and 3 where we comparing against *Gecode*.

Since the higher-consistency propagators can be slow to execute, our default approach is to include a VALUE propagator at the same time, at a high priority (including, when there are no spare values, the clauses described in Section 4, noting that clauses have higher priority than propagators). So the strong propagator only executes after obvious propagation has been done, and is avoided entirely if failure is obvious.

To see whether these default redundant propagators were really an improvement, we took the best solver for each model and tried removing each type of redundant constraint (NOVALUE and NOCLAUSE), at least where it made sense to do so, which resulted in the matrix of Table 4. Note that the 'sat, unsat, ?' summary numbers do not exactly match Table 2 since the set of solveable instances is recomputed for

each table based on the solvers attempted.

## 9 Discussion

In the first experiment *Chuffed* comprehensively beats the CSP2008 solver competition winners. The improvement is partly just from adding learning, but *Sugar* also has learning, and the improvement here is from lazy clause generation, because *Sugar* is based on SAT decompositions which are large on global constraints such as *element*, and also because *Sugar* has only bounds literals rather than the more advanced dual model discussed in Section 2. The new explained global propagators for *alldifferent* also played an important role in defeating the CSP2008 solvers.

In the second experiment *Chuffed* without learning is equivalent to, or slightly better than, the state-of-the-art publically available solver, *Gecode*. With learning it is comprehensively better. All problems except GOLOMB\_RULER, LANGFORD and QG\_EXISTENCE benefit from learning in our experiments. When learning, all problems except KAKURO and QG\_COMPLETION benefit from higher consistency levels (BOUNDS, DOMAIN or FEYDY). Thus, on 4 of the 9 problems selected, we demonstrate the usefulness of explained higher-consistency propagators.

Learning changes the tradeoffs for propagation. DWD allows the best comparison. For the models QG\_COMPLETION (Table 2), and SOCIAL\_GOLFER (in particular the constraint *alldiff1*; Table 3), a strong propagator (DOMAIN) was best without learning but a simpler decomposition (VALUE) was best with learning, suggesting that learning can recover some of the global knowledge lost through decomposition.

Indeed for constraint *alldiff1* of SOCIAL\_GOLFER the conflict count was *worse* with the stronger propagator, which we do not normally expect. The issue seems to be nogood reuse: The strong DOMAIN propagator produces a weaker nogood than VALUE, since the nogood involves many variables and describes a situation that might not recur often enough to pay

model unsat, sat, ?	<i>Gcode</i> <i>alldiff</i> =VALUE	BOUNDS	DOMAIN	FEYDY	NOLEARN <i>alldiff</i> =VALUE	BOUNDS	DOMAIN	FEYDY	LEARN <i>alldiff</i> =VALUE	BOUNDS	DOMAIN	FEYDY
GOLOMB_RULER 0, 3, 1	IO	30.90s 746078	14.40s 219092	22.28s 219092	147.10s <sup>1</sup> 195952	66.73s 794358	29.36s 243769	46.89s 243769	274.48s <sup>1</sup> 412471	103.95s <sup>1</sup> 170590	175.98s <sup>1</sup> 159359	146.08s <sup>1</sup> 134791
DWD		59.91s 1551794	22.58s 404733	42.88s 469285	303.10s <sup>1</sup> 402804	152.03s 1917948	49.66s 471899	98.85s 561992	514.76s <sup>2</sup> 521536	183.76s <sup>1</sup> 282786	413.46s <sup>2</sup> 299639	245.19s <sup>1</sup> 208272
ACT									1022.28s <sup>2</sup> 1299326	560.32s <sup>2</sup> 740216	976.18s <sup>2</sup> 712173	637.83s <sup>2</sup> 609045
INSN_SCHED 0, 39, 0	IO	839.72s <sup>36</sup> 23022833	159.72s <sup>29</sup> 1019960	279.15s <sup>29</sup> 951303	678.11s <sup>32</sup> 140	841.90s <sup>36</sup> 25271476	195.67s <sup>29</sup> 492194	285.44s <sup>29</sup> 554869	20.05s <sup>14</sup> 6227	5.62s <sup>7</sup> 601	11.31s <sup>9</sup> 741	30.65s <sup>12</sup> 239
DWD		1011.38s <sup>37</sup> 6176314	219.15s <sup>30</sup> 617280	368.15s <sup>30</sup> 138367	817.12s <sup>33</sup> 920	1027.00s <sup>37</sup> 14483805	287.06s <sup>30</sup> 706914	417.93s <sup>30</sup> 190637	42.48s <sup>15</sup> 19867	4.11s 574	30.67s <sup>9</sup> 2263	37.66s <sup>8</sup> 463
ACT									18.65s <sup>12</sup> 20939	5.06s 1083	7.01s <sup>1</sup> 1673	12.55s <sup>4</sup> 304
KAKURO 0, 10, 0	IO	1800.00s <sup>10</sup> 27111480	1800.00s <sup>10</sup> 22150945	1800.00s <sup>10</sup> 21166803	1800.00s <sup>10</sup> 770872	1800.00s <sup>10</sup> 82297506	1800.00s <sup>10</sup> 62950017	1800.00s <sup>10</sup> 48485283	473.32s <sup>7</sup> 580547	508.85s <sup>7</sup> 584233	514.28s <sup>7</sup> 614998	745.40s <sup>7</sup> 366761
DWD		1800.00s <sup>10</sup> 29817082	1800.00s <sup>10</sup> 23819512	1800.00s <sup>10</sup> 23343828	1800.00s <sup>10</sup> 1325438	1571.18s <sup>9</sup> 57417794	1800.00s <sup>10</sup> 52387830	1800.00s <sup>10</sup> 37955394	1.90s 21101	3.64s 32441	3.34s <sup>1</sup> 28992	8.29s <sup>1</sup> 17943
ACT									24.13s 200766	27.14s 178232	27.12s 153574	234.38s <sup>2</sup> 412243
KNIGHTS_TOUR 0, 5, 0	IO	256.59s <sup>2</sup> 2676127	271.01s <sup>2</sup> 2528878	264.78s <sup>2</sup> 2530965	511.85s <sup>3</sup> 1268024	205.73s <sup>2</sup> 3182194	218.41s <sup>2</sup> 3093521	216.60s <sup>2</sup> 3077334	244.66s <sup>3</sup> 116045	263.62s <sup>3</sup> 113638	270.15s <sup>3</sup> 112789	316.48s <sup>3</sup> 122205
DWD		554.63s <sup>4</sup> 9628143	561.59s <sup>4</sup> 7930382	655.84s <sup>4</sup> 8386774	1024.31s <sup>4</sup> 2603590	102.23s <sup>2</sup> 1616868	117.12s <sup>2</sup> 1517141	108.57s <sup>2</sup> 1560930	55.74s <sup>2</sup> 90951	84.00s <sup>2</sup> 118841	10.57s <sup>2</sup> 26154	28.26s <sup>1</sup> 42700
ACT									45.64s <sup>1</sup> 170283	20.56s <sup>1</sup> 60285	42.61s <sup>1</sup> 133384	31.79s 83068
LANGFORD 5, 16, 4	IO	0.47s <sup>3</sup> 6057	0.31s <sup>2</sup> 2601	0.32s <sup>2</sup> 2596	0.70s <sup>3</sup> 2257	0.33s <sup>2</sup> 6752	0.25s <sup>1</sup> 2679	0.26s <sup>1</sup> 2678	1.08s <sup>4</sup> 2433	0.62s <sup>3</sup> 1340	0.74s <sup>4</sup> 1341	0.88s <sup>4</sup> 1300
DWD		0.05s 379	0.06s 359	0.06s 355	0.05s <sup>1</sup> 72	0.03s 297	0.04s 288	0.04s 288	0.05s <sup>1</sup> 194	0.05s <sup>1</sup> 182	0.06s <sup>1</sup> 189	0.09s <sup>1</sup> 193
ACT									0.04s <sup>1</sup> 241	0.04s <sup>1</sup> 187	0.05s <sup>1</sup> 214	0.06s <sup>1</sup> 205
QG_COMPLETION 21, 64, 75	IO	388.57s <sup>76</sup> 3604285	385.38s <sup>76</sup> 1545976	309.90s <sup>74</sup> 996245	455.85s <sup>75</sup> 57285	364.26s <sup>74</sup> 2871125	366.36s <sup>74</sup> 1907175	390.85s <sup>74</sup> 1350303	358.64s <sup>73</sup> 138630	357.83s <sup>73</sup> 134574	389.95s <sup>73</sup> 119224	659.25s <sup>74</sup> 56068
DWD		345.13s <sup>75</sup> 2778555	342.16s <sup>75</sup> 1121675	220.43s <sup>55</sup> 608231	451.20s <sup>74</sup> 51376	273.61s <sup>54</sup> 1576137	285.31s <sup>57</sup> 1003851	270.96s <sup>46</sup> 713505	225.25s <sup>38</sup> 85790	190.24s <sup>43</sup> 73486	188.42s <sup>36</sup> 59615	478.37s <sup>61</sup> 40346
ACT									89.01s <sup>11</sup> 43014	90.49s <sup>15</sup> 41402	84.06s <sup>13</sup> 30812	357.25s <sup>38</sup> 29410
QG_EXISTENCE 12, 23, 13	IO	127.01s <sup>19</sup> 370842	93.57s <sup>17</sup> 181646	81.87s <sup>16</sup> 133805	110.09s <sup>17</sup> 125406	73.49s <sup>17</sup> 228031	72.09s <sup>17</sup> 188657	68.04s <sup>16</sup> 165335	28.08s <sup>13</sup> 16970	24.75s <sup>13</sup> 14204	24.00s <sup>13</sup> 12711	28.17s <sup>13</sup> 14281
DWD		9.25s <sup>10</sup> 20285	10.27s <sup>9</sup> 18175	9.08s <sup>8</sup> 13802	16.49s <sup>10</sup> 14731	3.21s <sup>5</sup> 7132	3.43s <sup>5</sup> 6916	3.43s <sup>5</sup> 6182	6.16s <sup>7</sup> 4621	5.08s <sup>7</sup> 3933	5.72s <sup>7</sup> 3975	4.83s <sup>6</sup> 3082
ACT									7.15s <sup>8</sup> 5086	10.05s <sup>11</sup> 6301	8.76s <sup>10</sup> 5046	13.19s <sup>11</sup> 7698
TALENT_SCHED 0, 9, 1	IO	709.06s <sup>7</sup> 13900566	708.35s <sup>7</sup> 11745423	665.89s <sup>7</sup> 11203913	715.40s <sup>7</sup> 8134702	614.28s <sup>7</sup> 21095605	631.69s <sup>7</sup> 19307512	625.50s <sup>7</sup> 20216040	521.24s <sup>7</sup> 823504	538.38s <sup>7</sup> 713159	549.09s <sup>7</sup> 714077	532.20s <sup>7</sup> 733577
DWD		716.46s <sup>7</sup> 13479876	678.88s <sup>7</sup> 10230692	685.64s <sup>7</sup> 10779862	775.07s <sup>7</sup> 7344706	623.25s <sup>7</sup> 15472903	631.93s <sup>7</sup> 15187825	623.32s <sup>7</sup> 14844274	568.22s <sup>7</sup> 605667	582.31s <sup>7</sup> 660109	585.74s <sup>7</sup> 612372	546.25s <sup>7</sup> 695941
ACT									341.45s <sup>3</sup> 576270	107.07s 185405	52.51s 104562	114.05s 190925

Table 2: Models containing one *alldifferent* constraint, per propagator and search strategy

0, 39, 7 unsat, sat, ?	Gecode <i>alldiff0</i> =VALUE	NOLEARN <i>alldiff0</i> =VALUE				LEARN <i>alldiff0</i> =VALUE			
		BOUNDS	DOMAIN	FEYDY		BOUNDS	DOMAIN	FEYDY	
IO	<i>alldiff1</i> =VALUE	8.61s <sup>14</sup>	1.26s <sup>9</sup>	0.97s <sup>9</sup>	1.62s <sup>10</sup>	3.78s <sup>13</sup>	0.93s <sup>9</sup>	0.88s <sup>9</sup>	1.09s <sup>9</sup>
	BOUNDS	13686	1066	660	722	10087	1010	848	978
	DOMAIN	9.20s <sup>14</sup>	1.39s <sup>9</sup>	1.04s <sup>9</sup>	1.71s <sup>9</sup>	4.29s <sup>13</sup>	1.08s <sup>9</sup>	0.93s <sup>9</sup>	1.20s <sup>9</sup>
	FEYDY	13060	1045	639	715	9202	983	793	952
		10.57s <sup>15</sup>	1.82s <sup>10</sup>	1.40s <sup>10</sup>	2.05s <sup>10</sup>	5.58s <sup>13</sup>	1.35s <sup>9</sup>	1.21s <sup>9</sup>	1.55s <sup>9</sup>
		7512	918	572	652	7099	867	726	856
		39.95s <sup>19</sup>	12.35s <sup>14</sup>	9.46s <sup>13</sup>	11.93s <sup>14</sup>	10.18s <sup>16</sup>	4.01s <sup>13</sup>	3.51s <sup>13</sup>	4.10s <sup>13</sup>
		494	119	77	93	490	98	80	97
DWD	<i>alldiff1</i> =VALUE	93.47s <sup>26</sup>	1.48s <sup>7</sup>	1.18s <sup>7</sup>	2.18s <sup>8</sup>	10.91s <sup>14</sup>	10.76s <sup>15</sup>	9.35s <sup>10</sup>	9.71s <sup>12</sup>
	BOUNDS	149011	1747	1327	1610	24281	22795	18686	16773
	DOMAIN	109.26s <sup>26</sup>	1.58s <sup>7</sup>	1.36s <sup>7</sup>	2.13s <sup>7</sup>	12.14s <sup>14</sup>	11.41s <sup>15</sup>	10.45s <sup>10</sup>	10.12s <sup>12</sup>
	FEYDY	154582	1622	1375	1486	23564	21319	17903	15573
		100.61s <sup>26</sup>	2.03s <sup>6</sup>	1.14s <sup>5</sup>	1.92s <sup>5</sup>	4.85s <sup>8</sup>	4.29s <sup>9</sup>	3.40s <sup>6</sup>	5.02s <sup>7</sup>
		96630	1462	819	1026	7342	6590	4916	6895
		177.35s <sup>27</sup>	20.88s <sup>13</sup>	14.64s <sup>12</sup>	22.90s <sup>13</sup>	29.74s <sup>17</sup>	47.34s <sup>20</sup>	46.13s <sup>19</sup>	47.11s <sup>20</sup>
		2173	240	154	222	2489	2841	2824	2186
ACT	<i>alldiff1</i> =VALUE								
	BOUNDS								
	DOMAIN								
	FEYDY								

 Table 3: Model SOCIAL\_GOLFER, propagator matrix for the two *alldifferent* constraints, by search strategy

model	NOVALUE		VALUE	
unsat, sat, ?				
GOLOMB_RULER NOCLAUSES	32.94s	<b>243769</b>	<b>29.67s</b>	243769
NOLEARN, BOUNDS, IO				
0, 3, 1 CLAUSES				
INSN_SCHED NOCLAUSES	<b>3.46s</b>	<b>417</b>	4.41s	573
LEARN, BOUNDS, DWD				
0, 39, 0 CLAUSES				
KAKURO NOCLAUSES			1.19s	14787 <sup>1</sup>
LEARN, VALUE, DWD				
0, 10, 0 CLAUSES			<b>1.90s</b>	<b>21101</b>
KNIGHTS_TOUR NOCLAUSES	34.46s	<b>77418</b>	32.87s	77418
LEARN, FEYDY, ACT				
0, 5, 0 CLAUSES	34.00s	83068	<b>31.79s</b>	83068
LANGFORD NOCLAUSES			0.06s	416
NOLEARN, VALUE, DWD				
5, 16, 4 CLAUSES			<b>0.03s</b>	<b>297</b>
QG_COMPLETION NOCLAUSES			341.63s	152155 <sup>62</sup>
LEARN, VALUE, ACT				
21, 53, 86 CLAUSES			<b>56.93s</b>	<b>28702</b>
QG_EXISTENCE NOCLAUSES	2.13s	3318	2.20s	3378 <sup>1</sup>
NOLEARN, FEYDY, DWD				
12, 19, 17 CLAUSES	1.68s	2892 <sup>1</sup>	<b>1.84s</b>	<b>2989</b>
TALENT_SCHED NOCLAUSES	79.24s	175839	66.76s	164128 <sup>1</sup>
LEARN, DOMAIN, ACT				
0, 9, 1 CLAUSES	<b>38.73s</b>	105665	52.51s	<b>104562</b>
SOCIAL_GOLFER NOCLAUSES	<b>0.18s</b>	170	0.23s	183 <sup>1</sup>
LEARN, DOMAIN/VALUE, ACT				
0, 38, 8 CLAUSES	0.15s	146 <sup>1</sup>	0.21s	<b>151</b>

Table 4: Removing default redundant propagators

the nogood's propagation cost. The cases where DOMAIN is beneficial, such as the constraint *alldiff0* of SOCIAL\_GOLFER, tend to be those where the domain size is very small, so the nogoods are always highly reusable and also cheap to produce.

For INSN\_SCHED where the bounds propagator is best, we see the opposite effect than for DOMAIN, the nogoods produced by BOUNDS are stronger than VALUE because the domains are large and sparse, so collisions between values (pruned by the VALUE propagator) are unlikely, whereas many different pruning opportunities are compactly described by a BOUNDS nogood. These nogoods are also symmetric between variables, since they describe a Hall interval rather than any specific pruning resulting from the existence of the Hall interval, which further promotes reuse.

For most models ACT is the best. In our experience the models for which ACT is the wrong approach, tend to be those with a good static search order (here INSN\_SCHED), where gaps in the sequence of variable assignments are difficult to resolve later on. In such cases DWD is a useful compromise between activity-based search (*weighted degree* is similar to activity) and sequential search (*domain* size causes a domino effect which makes the search ripple outwards from previously fixed variables). But where ACT works it is almost always significantly better, and allowing the use of activity based search with strong propagators is an important contribution of our work.

Our KNIGHTS\_TOUR model relies on *linear* constraints propagated to bounds consistency which creates a  $5 \times 5$  bounding box for each knight move. This causes the most propagation at the edges of the board, so it is intuitive that FEYDY, which efficiently detects Hall intervals aligned to the edges of the board, should perform best on this model. We don't claim state-of-the-art results since specialized techniques based on lookahead can solve the problem greedily (von Warnsdorff 1823), but the model is still very useful in demonstrating that the FEYDY decomposition may be best despite its (relative) simplicity.

Referring to Table 4, for most models it is important, indeed essential, to add the redundant VALUE propagator and the extra clauses if possible.

The exceptions are: `INSN_SCHED`, where `VALUE` does not prune very well as discussed above; and `SOCIAL_GOLFER` (in particular the *alldiff0* constraint) and `TALENT_SCHED`, where domains are small and the domain propagator is cheap as discussed above.

## 10 Conclusions and further work

We have shown how to extend propagators for *alldifferent* to explain their propagation, in order to use them in a lazy clause generation solver. We see that for problems involving *alldifferent*, learning is usually hugely beneficial. Each of the different propagation methods is best for some problems, so having a range of different propagators (or decompositions) that can explain their propagation is valuable. Overall combining learning and *alldifferent* leads to a highly competitive approach to these problems. The combination of global *alldifferent* constraints with explanation leads to a state-of-the-art constraint programming solution to problems involving *alldifferent*.

In further work we would like to investigate why learning isn't effective on some of the models. We conjecture it may be because the instances become too hard too quickly, and that indeed learning may be better on harder instances but this is academic when the instances are out of reach for any solver. Further work could also examine hybrid methods to see if our work can be incorporated into the specialized solvers for `TALENT_SCHED` (Garcia de la Banda et al. 2010) or `SOCIAL_GOLFER` (Gange et al. 2010).

## References

- Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G. & Walsh, T. (2009a), Circuit Complexity and Decompositions of Global Constraints, in 'Procs. of IJCAI-2009', pp. 412–418.
- Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G. & Walsh, T. (2009b), Decompositions of All Different, Global Cardinality and Related Constraints, in 'Procs. of IJCAI-2009', pp. 419–424.
- Boussemart, F., Hemery, F., Lecoutre, C. & Sais, L. (2004), Boosting Systematic Search by Weighting Constraints, in 'Procs. of ECAI04', pp. 146–150.
- Colton, S. & Miguel, I. (2001), Constraint Generation via Automated Theory Formation, in 'Procs. of CP01', pp. 575–579.
- CSP (2008), 'Third international csp solver competition'. <http://www.cril.univ-atrois.fr/CPAI08>.
- Feydy, T. & Stuckey, P. (2009), Lazy Clause Generation Reengineered, in 'Procs. of CP2009', pp. 352–366.
- Ford, L. & Fulkerson, D. (1956), 'Maximal flow through a network', *Canad. J. Math.* **8**, 399–404.
- G12 Project (2010), 'MiniZinc Challenge'. <http://www.g12.cs.mu.oz.au/minizinc/challenge2010/challenge.html>.
- Gange, G., Stuckey, P. & Lagoon, V. (2010), 'Fast set bounds propagation using a BDD-SAT hybrid', *JAIR* **38**, 307–338.
- Garcia de la Banda, M., Stuckey, P. J. & Chu, G. (2010), 'Solving Talent Scheduling with Dynamic Programming', *INFORMS J. on Computing (preprint)*.
- Gecode Team (2006), 'Gecode: Generic constraint development environment'. <http://www.gecode.org>.
- Gent, I., Miguel, I. & Nightingale, P. (2008), 'Generalised arc consistency for the AllDifferent constraint: An empirical survey', *AI* **172**(18), 1973–2000.
- Gent, I. P. & Walsh, T. (1999), CSPLIB: A Benchmark Library for Constraints, in 'Princ. and Prac. of CP', pp. 480–481. <http://www.csplib.org>.
- Hall, P. (1935), 'On Representatives of Subsets', *J. London Math. Soc.* **s1-10**(1), 26–30.
- Katsirelos, G. (2008), Nogood processing in CSPs, PhD thesis, University of Toronto, Canada.
- Lopez-Ortiz, A., Quimper, C.-G., Tromp, J. & Van Beek, P. (2003), A fast and simple algorithm for bounds consistency of the all different constraint, in 'Procs. of IJCAI-2003', pp. 245–250.
- Luby, M., Sinclair, A. & Zuckerman, D. (1993), 'Optimal speedup of Las Vegas algorithms', *Inf. Proc. Let.* **47**(4), 173–180.
- Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L. & Malik, S. (2001), Chaff: engineering an efficient SAT solver, in 'Procs. of DAC01', pp. 530–535.
- Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G. & Tack, G. (2007), MiniZinc: Towards a Standard CP Modelling Language, in 'Procs. of CP2007', Vol. 4741 of *LNCS*, Springer-Verlag, pp. 529–543.
- Ohrimenko, O., Stuckey, P. & Codish, M. (2009), 'Propagation via lazy clause generation', *Constraints* **14**, 357–391.
- Régin, J.-C. (1994), A filtering algorithm for constraints of difference in CSPs, in 'Procs. of AAAI-1994', pp. 362–367.
- Simonis, H. (2008), Kakuro as a Constraint Problem, in P. Flener & H. Simonis, eds, 'Procs. of MOD-REF08', Uppsala University, Computing Science.
- Smith, B. (2005), Caching Search States in Permutation Problems, in P. van Beek, ed., 'Procs. of CP2005', Vol. 3709 of *LNCS*, Springer Berlin / Heidelberg, pp. 637–651.
- Tarjan, R. E. (1972), 'Depth-First Search and Linear Graph Algorithms', *SIAM J. Computing* **1**(2), 146–160.
- Tarjan, R. E. (1975), 'Efficiency of a Good But Not Linear Set Union Algorithm', *J. ACM* **22**, 215–225.
- van Hoeve, W. J. (2001), The alldifferent Constraint: A Survey, in 'Procs. of the 6th ERCIM Working Group on Constraints Workshop', Vol. cs.PL/0105015.
- von Warnsdorff, H. C. (1823), *Des Rösselsprungs einfachste und allgemeinste Lösung*, Th. G. Fr. Varnhagenschen Buchhandlung, Schmalkalden.

## Author Index

- Alexander, Bradley, 11
- Bailes, Paul, 63  
Barone, Luigi, 81  
Browne, Will N., 27
- Cao, Jinli, 3  
Ciesielski, Vic, 57
- Dhaliwal, Jasbir, 91  
Dietrich, Jens, 37  
Donnellan, Sean, 11  
Downing, Nicholas, 115
- Estivill-Castro, Vlad, 21
- Feydy, Thibaut, 115
- Hayashida, Takanori, 107  
Hingston, Phil, 81
- Jeffrey, Andrew, 11
- Lee, Hua Jie, 99  
Liu, Denghui, 3
- McCartin, Catherine, 37  
Mitchell, Arnan, 73  
Mori, Hideki, 107
- Naish, Lee, 49, 99
- Nguyen, Thach, 73
- Olds, Travis, 11
- Parsa, Mahdi, 21  
Puglisi, Simon J., 91
- Ramamohanarao, Kotagiri, 99  
Reynolds, Mark, iii
- Sato, Toshinori, 107  
Shah, Syed Ali, 37  
Shi, Qiao, 73  
Sizer, Nicholas, 11  
Song, Andy, 57, 73  
Stuckey, Peter, 115
- Tempero, Ewan, 37  
Thomas, Bruce, iii  
Turpin, Andrew, 91
- While, Lyndon, 81  
Wittkamp, Mark, 81
- Xie, Feng, 57  
Xue, Bing, 27
- Yano, Rikiya, 107
- Zhang, Mengjie, 27

# Recent Volumes in the CRPIT Series

ISSN 1445-1336

Listed below are some of the latest volumes published in the ACS Series *Conferences in Research and Practice in Information Technology*. The full text of most papers (in either PDF or Postscript format) is available at the series website <http://crpit.com>.

- Volume 113 - Computer Science 2011**  
Edited by Mark Reynolds, The University of Western Australia, Australia. January 2011. 978-1-920682-93-4.  
Contains the proceedings of the Thirty-Fourth Australasian Computer Science Conference (ACSC 2011), Perth, Australia, 17-20 January 2011.
- Volume 114 - Computing Education 2011**  
Edited by John Hamer, University of Auckland, New Zealand and Michael de Raadt, University of Southern Queensland, Australia. January 2011. 978-1-920682-94-1.  
Contains the proceedings of the Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia, 17-20 January 2011.
- Volume 115 - Database Technologies 2011**  
Edited by Heng Tao Shen, The University of Queensland, Australia and Yanchun Zhang, Victoria University, Australia. January 2011. 978-1-920682-95-8.  
Contains the proceedings of the Twenty-Second Australasian Database Conference (ADC 2011), Perth, Australia, 17-20 January 2011.
- Volume 116 - Information Security 2011**  
Edited by Colin Boyd, Queensland University of Technology, Australia and Josef Pieprzyk, Macquarie University, Australia. January 2011. 978-1-920682-96-5.  
Contains the proceedings of the Ninth Australasian Information Security Conference (AISC 2011), Perth, Australia, 17-20 January 2011.
- Volume 117 - User Interfaces 2011**  
Edited by Christof Lutteroth, University of Auckland, New Zealand and Haifeng Shen, Flinders University, Australia. January 2011. 978-1-920682-97-2.  
Contains the proceedings of the Twelfth Australasian User Interface Conference (AUIC2011), Perth, Australia, 17-20 January 2011.
- Volume 118 - Parallel and Distributed Computing 2011**  
Edited by Jinjun Chen, Swinburne University of Technology, Australia and Rajiv Ranjan, University of New South Wales, Australia. January 2011. 978-1-920682-98-9.  
Contains the proceedings of the Ninth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January 2011.
- Volume 119 - Theory of Computing 2011**  
Edited by Alex Potanin, Victoria University of Wellington, New Zealand and Taso Viglas, University of Sydney, Australia. January 2011. 978-1-920682-99-6.  
Contains the proceedings of the Seventeenth Computing: The Australasian Theory Symposium (CATS 2011), Perth, Australia, 17-20 January 2011.
- Volume 120 - Health Informatics and Knowledge Management 2011**  
Edited by Kerry Butler-Henderson, Curtin University, Australia and Tony Sahama, Queensland University of Technology, Australia. January 2011. 978-1-921770-00-5.  
Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2011), Perth, Australia, 17-20 January 2011.
- Volume 121 - Data Mining and Analytics 2011**  
Edited by Peter Vamplew, University of Ballarat, Australia, Andrew Stranieri, University of Ballarat, Australia, Kok-Leong Ong, Deakin University, Australia, Peter Christen, Australian National University, Australia and Paul J. Kennedy, University of Technology, Sydney, Australia. December 2011. 978-1-921770-02-9.  
Contains the proceedings of the Ninth Australasian Data Mining Conference (AusDM'11), Ballarat, Australia, 1-2 December 2011.
- Volume 122 - Computer Science 2012**  
Edited by Mark Reynolds, The University of Western Australia, Australia and Bruce Thomas, University of South Australia. January 2012. 978-1-921770-03-6.  
Contains the proceedings of the Thirty-Fifth Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 123 - Computing Education 2012**  
Edited by Michael de Raadt, Moodle Pty Ltd and Angela Carbone, Monash University, Australia. January 2012. 978-1-921770-04-3.  
Contains the proceedings of the Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 124 - Database Technologies 2012**  
Edited by Rui Zhang, The University of Melbourne, Australia and Yanchun Zhang, Victoria University, Australia. January 2012. 978-1-920682-95-8.  
Contains the proceedings of the Twenty-Third Australasian Database Conference (ADC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 125 - Information Security 2012**  
Edited by Josef Pieprzyk, Macquarie University, Australia and Clark Thomborson, The University of Auckland, New Zealand. January 2012. 978-1-921770-06-7.  
Contains the proceedings of the Tenth Australasian Information Security Conference (AISC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 126 - User Interfaces 2012**  
Edited by Haifeng Shen, Flinders University, Australia and Ross T. Smith, University of South Australia, Australia. January 2012. 978-1-921770-07-4.  
Contains the proceedings of the Thirteenth Australasian User Interface Conference (AUIC2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 127 - Parallel and Distributed Computing 2012**  
Edited by Jinjun Chen, University of Technology, Sydney, Australia and Rajiv Ranjan, CSIRO ICT Centre, Australia. January 2012. 978-1-921770-08-1.  
Contains the proceedings of the Tenth Australasian Symposium on Parallel and Distributed Computing (AusPDC 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 128 - Theory of Computing 2012**  
Edited by Julián Mestre, University of Sydney, Australia. January 2012. 978-1-921770-09-8.  
Contains the proceedings of the Eighteenth Computing: The Australasian Theory Symposium (CATS 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 129 - Health Informatics and Knowledge Management 2012**  
Edited by Kerry Butler-Henderson, Curtin University, Australia and Kathleen Gray, University of Melbourne, Australia. January 2012. 978-1-921770-10-4.  
Contains the proceedings of the Fifth Australasian Workshop on Health Informatics and Knowledge Management (HIKM 2012), Melbourne, Australia, 30 January – 3 February 2012.
- Volume 130 - Conceptual Modelling 2012**  
Edited by Aditya Ghose, University of Wollongong, Australia and Flavio Ferrarotti, Victoria University of Wellington, New Zealand. January 2012. 978-1-921770-11-1.  
Contains the proceedings of the Eighth Asia-Pacific Conference on Conceptual Modelling (APCCM 2012), Melbourne, Australia, 31 January – 3 February 2012.
- Volume 131 - Advances in Ontologies 2010**  
Edited by Thomas Meyer, UKZN/CSIR Meraka Centre for Artificial Intelligence Research, South Africa, Mehmet Orgun, Macquarie University, Australia and Kerry Taylor, CSIRO ICT Centre, Australia. December 2010. 978-1-921770-00-5.  
Contains the proceedings of the Sixth Australasian Ontology Workshop 2010 (AOW 2010), Adelaide, Australia, 7th December 2010.