Pedro C. Diniz Eduardo Marques
Koen Bertels Marcio Merino Fernandes
João M.P. Cardoso (Eds.)

Reconfigurable Computing: Architectures, Tools and Applications

Third International Workshop, ARC 2007
Mangaratiba, Brazil, March 27-29, 2007
Proceedings

Springer

Volume Editors

Pedro C. Diniz
Instituto Superior Técnico (IST)/INESC-ID
Departamento de Engenharia Informática (DEI)
Tagus Park, 2780-990 Porto Salvo, Portugal
E-mail: pedro.diniz@tagus.istl.utl.pt

Eduardo Marques
Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação (ICMC)
P.O. Box 668, 13560-970 São Carlos, Brazil
E-mail: emarques@icmc.usp.br

Koen Bertels
Delft University of Technology
Computer Engineering Lab
Mekelweg 4, 2628 CD Delft, The Netherlands
E-mail: k.l.m.bertels@ewi.tudelft.nl

Marcio Merino Fernandes
Universidade Metodista de Priacicaba
Programa de Mestrado em Ciência da Computação Campus Taquaral
13400-911 Piracicaba-SP, Brazil
E-mail: mmfernan@unimep.br

João M.P. Cardoso
INESC-ID, Instituto Superior Técnico (IST)
Av. Alves Redol 9, 1000-029, Lisbon, Portugal
E-mail: jmpc@acm.org

Library of Congress Control Number: 2007922940

CR Subject Classification (1998): C, B, I.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-71430-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71430-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12035847 06/3142 5 4 3 2 1 0

Preface

Reconfigurable computing platforms have been gaining wide acceptance, spanning a wide spectrum from highly specialized custom controllers to general-purpose high-end computing systems. They offer the promise of increasing performance gains by exploiting coarse-grain as well as fine-grain instruction level parallelism opportunities given their ability to implement custom functional, storage and interconnect structures. With the continuous increase in technology integration, leading to devices with millions of logic gates, ready to be programmed according to the (run-time) application needs, it is now possible to implement very sophisticated and reconfigurable systems. Configurability is seen as a key technology for substantial product life-cycle savings in the presence of evolving product requirements and/or interfaces or standards. The extreme configurability and flexibility also makes reconfigurable architectures the medium of choice for very rapid system prototyping or early design verification.

The relentless capacity growth of reconfigurable devices, such as FPGAs (Field-Programmable Gate Arrays), is creating a wealth of new opportunities and increasingly complex challenges. Recent generation devices have heterogeneous internal resources such as hardware multiplier units and memory blocks in addition to a vast amount of fine grain logic cells. Taking advantage of the wealth of resources in today's configurable devices is a very challenging problem. Although the inclusion of FPGAs in mainstream computing products clearly shows that this technology is maturing, many aspects still require substantial research to effectively deliver the promise of this emerging technology.

A major motivation for the International Applied Reconfigurable Computing (ARC)[1] workshop series is to provide a forum for presentation and discussion of on-going research efforts, as well as more elaborated, interesting and high-quality work, on applied reconfigurable computing. The workshop also focuses on compiler and mapping techniques, and new reconfigurable computing architectures.

The ARC series started in 2005 in the Algarve, Portugal. The second workshop (ARC 2006) took place in Delft, The Netherlands in March 2006, and the selected papers were published as a Springer LNCS (Lecture Notes in Computer Science) volume[2]. The success of previous workshops clearly reveals the growing interest of academia and industry and thus the timeliness of this forum.

This LNCS volume includes the papers selected for the third workshop (ARC 2007), held at Mangaratiba, Rio de Janeiro, Brazil, on March 27–29, 2007. The workshop attracted a large number of very good papers, describing interesting work on reconfigurable computing related subjects. A total of 72 papers

[1] http://www.arc-workshop.org

[2] Koen Bertels, João M. P. Cardoso, and Stamatis Vassiliadis (Eds.), Reconfigurable Computing: Architectures and Applications, Second International Workshop, ARC 2006, Delft, The Netherlands, March 2006, Revised Selected Papers, Springer Lecture Notes in Computer Science, LNCS 3985, August 2006.

were submitted to the workshop from 20 countries: The Netherlands (4), France (3), Germany (4), Republic of South Korea (12), Brazil (14), People's Republic of China (8), Denmark (1), Mexico (2), Portugal (2), South Africa (1), Lebanon (1), Australia (1), Republic of Ireland (2), Puerto Rico (1), Spain (5), UK (2), India (2), Japan (5), Poland (1), and Greece (1). Submitted papers were evaluated by at least three members of the Program Committee. After careful selection, 27 papers were accepted for presentation as full papers (37.5% of the total number of submitted papers) and 10 as short papers (global acceptance rate of 51.4%). This volume also includes an article from the 2006 ARC workshop, which was, by lapse, not included in the 2006 proceedings. Those accepted papers led to a very interesting workshop program, which we considered to constitute a representative overview of ongoing research efforts in reconfigurable computing, a rapidly evolving and maturing field.

Several persons contributed to the success of the workshop. We would like to acknowledge the support of all the members of this year's workshop Steering and Program Committees in reviewing papers, in helping with the paper selection, and in giving valuable suggestions. Special thanks also to the additional researchers who contributed to the reviewing process, to all the authors who submitted papers to the workshop, and to all the workshop attendees. We also acknowledge the generous financial contribution from Altera Corp., USA and PI Componentes, Brazil. Last but not least, we are especially indebted to our colleague Jürgen Becker from the University of Karlsruhe for his strong support of this workshop.

We are grateful to Springer, particularly Mr. Alfred Hofmann and the LNCS Editorial, for their support and work in publishing this book.

January 2007

Pedro C. Diniz
Eduardo Marques
Koen Bertels
Marcio M. Fernandes
João M. P. Cardoso

Organization

The 2007 Applied Reconfigurable Computing workshop (ARC 2007) was organized by the Institute of Mathematics and Computer Science (ICMC) of the University of São Paulo (USP) in São Carlos, Brazil.

Organization Committee

General Chairs	Eduardo Marques (ICMC-USP, Brazil)
	Koen Bertels (Delft University of Technology, The Netherlands)
Program Chair	Pedro C. Diniz (IST/INESC-ID, Portugal)
Proceedings Chair	Marcio Merino Fernandes (UNIMEP, Brazil)
Finance Chair	Jorge Luiz e Silva, ICMC-USP, Brazil
Sponsorship Chair	Denis F. Wolf, ICMC-USP, Brazil
Web Chair	Carlos R. P. Almeida Jr. (ICMC-USP, Brazil)
Special Journal Edition Chair	George Constantinides (Imperial College, UK)
	João M. P. Cardoso (IST/INESC-ID, Portugal)
Local Arrangements Chairs	Marcos J. Santana, ICMC-USP, Brazil
	Regina H. C. Santana, ICMC-USP, Brazil
Leisure Chairs	Ricardo Menotti, UTFPR, Brazil
	Vanderlei Bonato, ICMC-USP, Brazil
Publicity Chair	Fernanda Lima Kastensmidt, UFRGS, Brazil

Steering Committee

George Constantinides, Imperial College, UK
João M. P. Cardoso, IST/INESC-ID, Portugal
Koen Bertels, Delft University of Technology, The Netherlands
Mladen Berekovic, IMEC vzw, Belgium
Pedro C. Diniz, IST/INESC-ID, Portugal
Stamatis Vassiliadis, Delft University of Technology, The Netherlands
Walid Najjar, University of California Riverside, USA

Program Committee

Andreas Koch, Technical University of Darmstadt (TU), Germany
Andy Pimentel, University of Amsterdam, The Netherlands

Additional Reviewers

Sponsoring Institutions

USP - Universidade de São Paulo
CAPES - Coordenação de Aperfeiçoamento de Pessoal de Nível Superior

Table of Contents

Architectures [Regular Papers]

Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array 1
Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev

A Configurable Multi-ported Register File Architecture for Soft Processor Cores 14
Mazen A.R. Saghir and Rawan Naous

MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture 26
Kehuai Wu, Andreas Kanstein, Jan Madsen, and Mladen Berekovic

Asynchronous ARM Processor Employing an Adaptive Pipeline Architecture 39
Je-Hoon Lee, Seung-Sook Lee, and Kyoung-Rok Cho

Partially Reconfigurable Point-to-Point Interconnects in Virtex-II Pro FPGAs 49
Jae Young Hur, Stephan Wong, and Stamatis Vassiliadis

Systematic Customization of On-Chip Crossbar Interconnects 61
Jae Young Hur, Todor Stefanov, Stephan Wong, and Stamatis Vassiliadis

Authentication of FPGA Bitstreams: Why and How 73
Saar Drimer

Architectures [Short Papers]

Design of a Reversible PLD Architecture 85
Jae-Jin Lee, Dong-Guk Hwang, and Gi-Yong Song

Designing Heterogeneous FPGAs with Multiple SBs 91
Kostas Siozios, Stelios Mamagkakis, Dimitrios Soudris, and Antonios Thanailakis

Mapping Techniques and Tools [Regular Papers]

Partial Data Reuse for Windowing Computations: Performance Modeling for FPGA Implementations 97
Joonseok Park and Pedro C. Diniz

Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware 110
Yazhuo Dong, Yong Dou, and Jie Zhou

Adapting and Automating XILINX's Partial Reconfiguration Flow for Multiple Module Implementations 122
Rainer Scholz

A Linear Complexity Algorithm for the Automatic Generation of Convex Multiple Input Multiple Output Instructions 130
Carlo Galuzzi, Koen Bertels, and Stamatis Vassiliadis

Evaluating Variable-Grain Logic Cells Using Heterogeneous Technology Mapping 142
Kazunori Matsuyama, Motoki Amagasaki, Hideaki Nakayama, Ryoichi Yamaguchi, Masahiro Iida, and Toshinori Sueyoshi

The Implementation of a Coarse-Grained Reconfigurable Architecture with Loop Self-pipelining 155
Yong Dou, Jinhui Xu, and Guiming Wu

Hardware/Software Codesign for Embedded Implementation of Neural Networks 167
Cesar Torres-Huitzil, Bernard Girau, and Adrien Gauffriau

Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues 179
João Bispo, Ioannis Sourdis, João M.P. Cardoso, and Stamatis Vassiliadis

Mapping Techniques and Tools [Short Papers]

About the Importance of Operation Grouping Procedures for Multiple Word-Length Architecture Optimizations 191
Nicolas Hervé, Daniel Ménard, and Olivier Sentieys

Arithmetic [Regular Papers]

Switching Activity Models for Power Estimation in FPGA Multipliers 201
Ruzica Jevtic, Carlos Carreras, and Gabriel Caffarena

Multiplication over $\mathbb{F}_{p^m}$ on FPGA: A Survey 214
Jean-Luc Beuchat, Takanori Miyoshi, Yoshihito Oyama, and Eiji Okamoto

A Parallel Version of the Itoh-Tsujii Multiplicative Inversion Algorithm 226
Francisco Rodríguez-Henríquez, Guillermo Morales-Luna, Nazar A. Saqib, and Nareli Cruz-Cortés

A Fast Finite Field Multiplier 238
Edgar Ferrer, Dorothy Bollman, and Oscar Moreno

Applications [Regular Papers]

Combining Flash Memory and FPGAs to Efficiently Implement a Massively Parallel Algorithm for Content-Based Image Retrieval 247
Rayan Chikhi, Steven Derrien, Auguste Noumsi, and Patrice Quinton

Image Processing Architecture for Local Features Computation 259
Javier Díaz, Eduardo Ros, Sonia Mota, and Richard Carrillo

A Compact Shader for FPGA-Based Volume Rendering Accelerators 271
Guenter Knittel

Ubiquitous Evolvable Hardware System for Heart Disease Diagnosis Applications 283
Yong-Min Lee, Chang-Seok Choi, Seung-Gon Hwang, Hyun Dong Kim, Chul Hong Min, Jae-Hyun Park, Hanho Lee, Tae Seon Kim, and Chong-Ho Lee

FPGA-Accelerated Molecular Dynamics Simulations: An Overview 293
Xiaodong Yang, Shengmei Mou, and Yong Dou

Reconfigurable Hardware Acceleration of Canonical Graph Labelling 302
David B. Thomas, Wayne Luk, and Michael Stumpf

Reconfigurable Computing for Accelerating Protein Folding Simulations 314
Nilton B. Armstrong Jr., Heitor S. Lopes, and Carlos R. Erig Lima

Reconfigurable Parallel Architecture for Genetic Algorithms: Application to the Synthesis of Digital Circuits 326
Edson P. Ferlin, Heitor S. Lopes, Carlos R. Erig Lima, and Ederson Cichaczewski

Applications [Short Papers]

A Space Variant Mapping Architecture for Reliable Car Segmentation 337
Sonia Mota, Eduardo Ros, Javier Díaz, Rafael Rodriguez, and Richard Carrillo

A Hardware SAT Solver Using Non-chronological Backtracking and Clause Recording Without Overheads 343
Shinya Hiramoto, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima

Searching the Web with an FPGA Based Search Engine 350
Séamas McGettrick, Dermot Geraghty, and Ciarán McElroy

An Acceleration Method for Evolutionary Systems Based on Iterated Prisoner's Dilemma 358
Yoshiki Yamaguchi, Kenji Kanazawa, Yoshiharu Ohke, and Tsutomu Maruyama

Real Time Architectures for Moving-Objects Tracking 365
Matteo Tomasi, Javier Díaz, and Eduardo Ros

Reconfigurable Hardware Evolution Platform for a Spiking Neural Network Robotics Controller 373
Patrick Rocke, Brian McGinley, Fearghal Morgan, and John Maher

Multiple Sequence Alignment Using Reconfigurable Computing 379
Carlos R. Erig Lima, Heitor S. Lopes, Maiko R. Moroz, and Ramon M. Menezes

Simulation of the Dynamic Behavior of One-Dimensional Cellular Automata Using Reconfigurable Computing 385
Wagner R. Weinert, César Benitez, Heitor S. Lopes, and Carlos R. Erig Lima

Author Index 391

Author Index

Amagasaki, Motoki 142
Armstrong, Nilton B. (Jr.) 314

Benitez, César 385
Berekovic, Mladen 1, 26
Bertels, Koen 130
Beuchat, Jean-Luc 214
Bispo, João 179
Bollman, Dorothy 238
Bouwens, Frank 1

Caffarena, Gabriel 201
Cardoso, João M.P. 179
Carreras, Carlos 201
Carrillo, Richard 259, 337
Chikhi, Rayan 247
Cho, Kyoung-Rok 39
Choi, Chang-Seok 283
Cichaczewski, Ederson 326
Cruz-Cortés, Nareli 226

Derrien, Steven 247
Díaz, Javier 259, 337, 365
Diniz, Pedro C. 97
Dong, Yazhuo 110
Dou, Yong 110, 155, 293
Drimer, Saar 73

Ferlin, Edson P. 326
Ferrer, Edgar 238

Galuzzi, Carlo 130
Gauffriau, Adrien 167
Gaydadjiev, Georgi 1
Geraghty, Dermot 350
Girau, Bernard 167

Hervé, Nicolas 191
Hiramoto, Shinya 343
Hur, Jae Young 49, 61
Hwang, Dong-Guk 85
Hwang, Seung-Gon 283

Iida, Masahiro 142

Jevtic, Ruzica 201

Kanazawa, Kenji 358
Kanstein, Andreas 1, 26
Kim, Hyun Dong 283
Kim, Tae Seon 283
Knittel, Guenter 271

Lee, Chong-Ho 283
Lee, Hanho 283
Lee, Jae-Jin 85
Lee, Je-Hoon 39
Lee, Seung-Sook 39
Lee, Yong-Min 283
Lima, Carlos R. Erig 314, 326, 379, 385
Lopes, Heitor S. 314, 326, 379, 385
Luk, Wayne 302

Madsen, Jan 26
Maher, John 373
Mamagkakis, Stelios 91
Maruyama, Tsutomu 358
Matsuyama, Kazunori 142
McElroy, Ciarán 350
McGettrick, Séamas 350
McGinley, Brian 373
Ménard, Daniel 191
Menezes, Ramon M. 379
Min, Chul Hong 283
Miyoshi, Takanori 214
Morales-Luna, Guillermo 226
Moreno, Oscar 238
Morgan, Fearghal 373
Moroz, Maiko R. 379
Mota, Sonia 259, 337
Mou, Shengmei 293

Nakanishi, Masaki 343
Nakashima, Yasuhiko 343
Nakayama, Hideaki 142
Naous, Rawan 14
Noumsi, Auguste 247

Ohke, Yoshiharu 358
Okamoto, Eiji 214
Oyama, Yoshihito 214

Park, Jae-Hyun 283
Park, Joonseok 97

Quinton, Patrice 247

Rocke, Patrick 373
Rodriguez, Rafael 337
Rodríguez-Henríquez, Francisco 226
Ros, Eduardo 259, 337, 365

Saghir, Mazen A.R. 14
Saqib, Nazar A. 226
Scholz, Rainer 122
Sentieys, Olivier 191
Siozios, Kostas 91
Song, Gi-Yong 85
Soudris, Dimitrios 91
Sourdis, Ioannis 179
Stefanov, Todor 61
Stumpf, Michael 302
Sueyoshi, Toshinori 142

Thanailakis, Antonios 91
Thomas, David B. 302
Tomasi, Matteo 365
Torres-Huitzil, Cesar 167

Vassiliadis, Stamatis 49, 61, 130, 179

Weinert, Wagner R. 385
Wong, Stephan 49, 61
Wu, Guiming 155
Wu, Kehuai 26

Xu, Jinhui 155

Yamaguchi, Ryoichi 142
Yamaguchi, Yoshiki 358
Yamashita, Shigeru 343
Yang, Xiaodong 293

Zhou, Jie 110

Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array

Frank Bouwens[1,3], Mladen Berekovic[1,3], Andreas Kanstein[2], and Georgi Gaydadjiev[3]

[1] IMEC vzw, DESICS
Kapeldreef 75, B-3001 Leuven, Belgium
{frank.bouwens, mladen.berekovic}@imec.be
[2] Freescale Semiconducteurs SAS, Wireless and Mobile Systems Group
134 Av. du General Eisenhower, 31023 Toulouse Cedex 1, France
a.kanstein@freescale.com
[3] Delft University of Technology, Computer Engineering
Mekelweg 4, 2628 CD, Delft, The Netherlands
g.n.gaydadjiev@its.tudelft.nl

Abstract. Reconfigurable computational architectures are envisioned to deliver power efficient, high performance, flexible platforms for embedded systems design. The coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) and its compiler offer a tool flow to design sparsely interconnected 2D array processors with an arbitrary number of functional units, register files and interconnection topologies. This article presents an architectural exploration methodology and its results for the first implementation of the ADRES architecture on a 90nm standard-cell technology. We analyze performance, energy and power trade-offs for two typical kernels from the multimedia and wireless domains: IDCT and FFT. Architecture instances of different sizes and interconnect structures are evaluated with respect to their power versus performance trade-offs. An optimized architecture is derived. A detailed power breakdown for the individual components of the selected architecture is presented.

1 Introduction

The requirements for higher performance and lower power consumption are becoming more stringent for newest generation mobile devices. Novel chip architectures should be able to execute multiple performance demanding applications while maintaining low power consumption, small area, non-recurring engineering costs and short time-to-market. IMEC develops a coarse-grained reconfigurable array (CGRA) called *Architecture for Dynamically Reconfigurable Embedded Systems* (ADRES) [1]. ADRES is a power efficient, but still flexible platform targeting 40MOPS/mW with 90nm technology. ADRES provides the designer with the tools to design an application specific processor instance based on an architecture template. This enables the designer to combine an arbitrary number of functional units, interconnects and register files.

The main contributions of the paper are:

– A toolflow for energy, power and performance explorations of ADRES based CGRA architectures;

- A novel methodology for power analysis that replaces RTL simulation with instruction set simulation to obtain activity stimuli;
- Analysis of performance, power and energy tradeoffs for different array sizes and interconnect topologies;
- Derivation of an optimized architecture for key multimedia and wireless kernels and the power breakdown of its components.

This paper is organized as follows. Section 2 briefly introduces the ADRES architectural template. In Section 3 the toolflow and the developed power simulation methodology are presented. Section 4 contains the results of the architectural exploration experiments, the selection of the power/performance optimized instance and the detailed power breakdown of its components. Finally, the conclusions are presented in Section 5.

2 ADRES Template

The ADRES architecture is a tightly-coupled architecture between two views: VLIW and CGA. Figure 1 shows an example of a 4x4 ADRES instance.

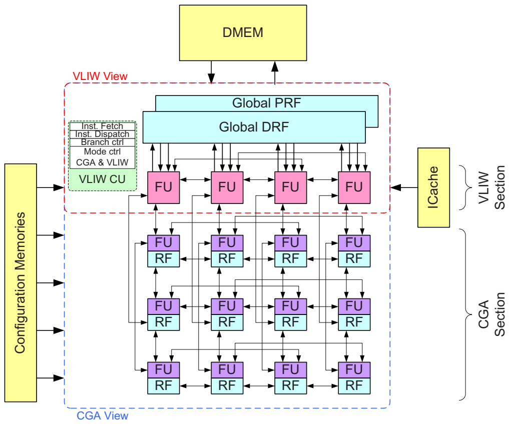


Fig. 1. ADRES Instance example

The VLIW control unit (CU) starts and stops the CGA loops when initiated by an CGA instruction in VLIW mode that works as a function call. It also accesses the instruction cache for the next instruction for the VLIW section and calculates the next Program Counter (PC).

The VLIW section in the example has an issue width of four instructions, while the CGA section has an issue width of 4 by 3 (12) instructions. The FUs in the VLIW view communicate through a multi-port global Data Register File (DRF), while those in the CGA use:

– global DRF;
– local register files (LRF);
– dedicated interconnects between the FUs.

Example of the data path in ADRES is depicted in Figure 2. All FUs have 1 destination and 3 source ports at most. There are local Data and Predicate Register Files (PRF) used for storage of variables. The ADRES template is so flexible that it allows all, operation without local RFs, single local RF per FU or shared register files among several FUs [8].

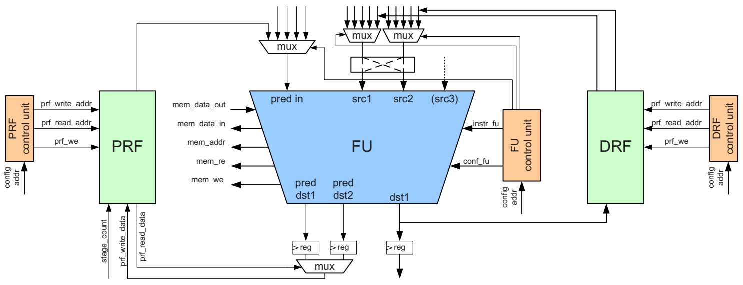


Fig. 2. ADRES Data path example

3 Tool Flow

Figure 3 depicts the proposed tool flow for architectural exploration. It consist of basically three parts: 1) Compilation and Assembly providing binary files and a compiled instruction level simulator, 2) Synthesis providing gate level netlist and physical characteristics needed for power calculation and 3) Simulation to obtain power and performance figures.

The *Compile and Assemble* part transforms the ANSI-C application code into an optimized binary file. The code is first processed by the IMPACT [3] frontend that performs various ILP optimization and transforms it into an intermediate representation called Lcode. The DRESC2.0 compiler in Figure 3 reads the Lcode, performs ILP scheduling, register allocation and modulo scheduling for CGA mode and generates the optimized DRE files. These files are used to create a high level simulator which provides basic performance parameters such as instructions per cycle (IPC). In addition, the Assembler tool creates the binary files needed for the cycle true Esterel and ModelSim simulations.

The ADRES instance XML description is transformed into VHDL files and is synthesized in the *Synthesize* part. Synopsys *Physical Compiler v2004.12-SP1* [5] is used to create the netlist for 90nm CMOS library, from which area, capacitance and resistor values are extracted to be used multiple times.

In the *Simulate* part three different simulators are used. The compiled ISA simulator (marked as A in Figure 3) provides the performance numbers. ModelSim RTL simulator is used to simulate the generated RTL VHDL files at highest level of hardware accuracy

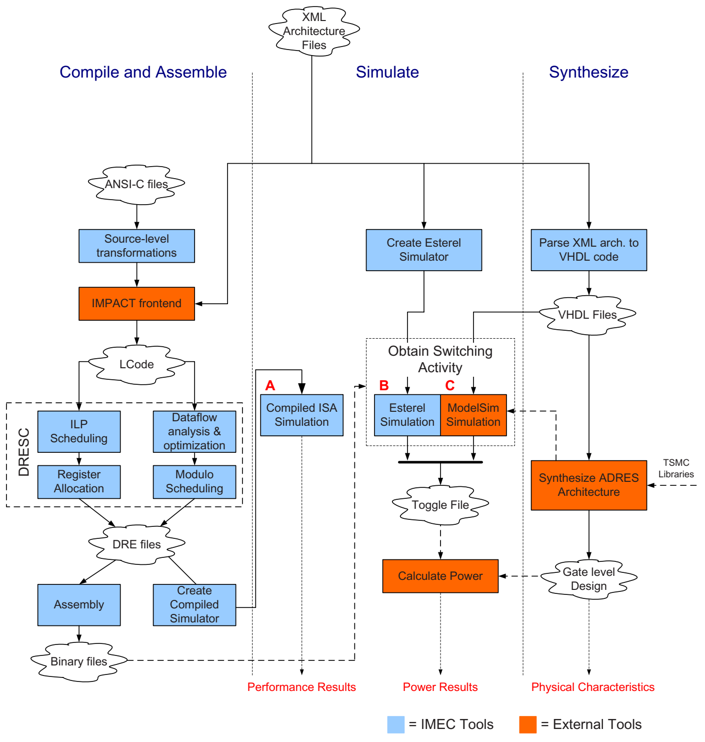


Fig. 3. Global Tool Flow overview

and to obtain the switching activity figures needed for RTL power calculations. The Esterel simulator [4] is based on the Esterel synchronous language dedicated to control-dominated reactive systems and models the entire ADRES instance as a state machine. The advantage of the Esterel simulator is the reduction in time to simulate a benchmark compared to ModelSim RTL simulations, as it is 8 - 12 times faster reducing overall time by a factor of 3 - 6.5 depending on the application. In addition, it is the behavior-level reference model for the core architecture available much earlier than the verified RTL.

Facilitated by its language, the Esterel simulator has the same structure as the actual implementation. Thus by enhancing it with bit toggle counting functions written in C, we are able of capturing the signal activity statistics of almost all relevant connections. This switching activity is what an RTL HDL simulator also generates for the power analysis tool. The match between the generated data is established by using the same signals as defined in the XML architecture description file.

The switching activities obtained after simulations are annotated on the gate level design created in the synthesize part. The toggling file and the gate level design are used by the *PrimePower v2004.12-SP1* of Synopsys [5] to estimate power.

For evaluation of the accuracy of the Esterel cycle accurate simulation versus the ModelSim VHDL simulation we used the IDCT kernel from MPEG. The IDCT benchmark is compiled for the VLIW only and for VLIW/CGA mode. The results are

Table 1. Differences between Esterel and ModelSim for ADRESv0

	IDCT	
Simulator	VLIW only (mW)	CGA (mW)
ModelSim	46.5	59.16
Esterel	57.75	65.65
Difference	24.2%	10.9%

depicted in the left column and in the right column of Table 1. The simulations are based on an non-pipelined ADRESv0 with a frequency of 100MHz.

From the results in Table 1 we can conclude that the proposed power simulation methodology with Esterel simulation works better in CGA mode that in VLIW mode. This is due to the fact that the CGA structure is captured by the simulator with more detail in the XML architecture file, while the VLIW part assumes a couple of implicit signals. Therefore the structure of the Esterel simulator and of the VHDL code match better for the CGA, resulting in more accurate signal switching statistics.

4 Architecture Explorations

In this section we propose a variety of architectural options to create an ADRES instance. ADRES explorations were performed in [8] producing scheduling and area results only as [9] provides energy results of a 32-tap Complex FIR filter based on abstract energy models. Different architectures are composed from the architectural options and

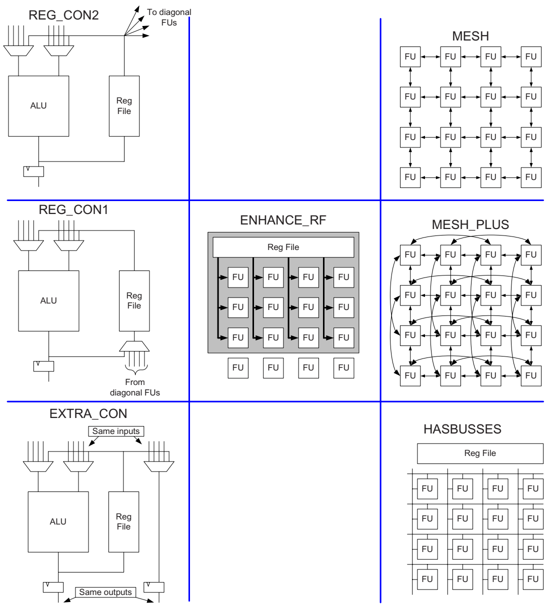


Fig. 4. Interconnection Options for Architectural Experiments

are evaluated for power, energy and performance. Finally, an optimized architecture is derived and a detailed power distribution for it is provided. This selection process utilizes two key benchmark kernels from multimedia and wireless application domains: IDCT and FFT. For the IDCT 396 8x8 block calculations are performed. For the FFT a single 1024 points FFT is performed. More benchmarks for evaluation were not available at the moment of writing.

4.1 Architecture Exploration Options

Fourteen different architectures are constructed from 7 different interconnection options as depicted in Figure 4. The architectures are described in the XML architecture file, which will be used in the tool flow described earlier. Exhaustive search of all configurations is impossible due to the large search space and required time.

The simplest interconnection option is *mesh*. It creates connections between destinations and sources of adjacent FUs in horizontal and vertical directions. The *mesh_plus* interconnection is an extension of mesh with additional connections that routes over the neighboring FUs. The *reg_con1* and *reg_con2* options create diagonal connections between neighboring FUs and RFs. This creates additional routing and the possibility of data sharing among FUs connecting directly to the RFs. The difference is that *reg_con1* receives data from its neighbors while *reg_con2* sends data to them. The *extra_con* option offers an extra FU bypass to enable parallel processing and routing. The *enhance_rf* option has shared read and write data ports from the global DRF to the vertically connected FUs as proposed by Kwok et al. [2]. This is beneficial for communication as the global DRF is used as communication medium in the array, however it increases power consumption due to the frequent accesses to the global DRF. Splitting up the power-hungry DRF into smaller, local DRFs was one of the main features of power reduction [6]. The option *has_busses* determines if predicate and data busses of 1 and 32 bits wide respectively are implemented in the design. By combining various interconnection options 14 different architectures are created as noted in Table 2.

Table 2. Architectural Exploration Selection Options

Architecture	mesh	mesh plus	reg con1	reg con2	extra con	enhance RF	has busses
mesh	X						
mesh_plus	X	X					
xtra_con	X	X			X		
reg_con1	X	X	X				
reg_con2	X	X		X			
reg_con_all	X	X	X	X			
enh_rf	X	X				X	
busses	X	X					X
arch_1	X	X		X	X	X	
arch_2	X	X	X		X	X	
arch_3	X	X		X		X	
arch_4	X	X	X			X	
all	X	X	X	X	X	X	X
ref	X	X	X		X	X	X

The physical properties of RFs and FUs are the same for all architectures. The local and global RFs have 16 and 64 words, respectively. The data bus width between DRFs, external memories and FUs is 32-bits. Each FU is capable of regular additions and multiplications of which only the FUs in the VLIW section are capable of load/store operations. The store capabilities of the FUs requires 12 read and 4 write ports for the global DRF. The global PRF has 4 read and write ports. The local DRFs have 2 read and 1 write ports and the local PRFs have 1 read and write port. The configuration memories (CM) are of fixed 128 words depth to store the benchmark configurations for the CGA. The CMs' bus sizes are variable and each FU, RF and multiplexor has its dedicated memory. We use standard 90nm general purpose libraries for both the standard cells and the memories.

The modeled non-pipelined architectures assume zero cache miss data memories and zero miss instruction cache. The external busses, the data memory and the I-cache are not considered in our power estimation study. However, the configuration memory, which serves as an instruction memory for the array mode, is considered in the CM module.

4.2 Instance Selection

The FFT and IDCT benchmarks are used to select the appropriate instance based on power, energy and performance trade-offs. As a comparison base a non-pipelined scalar RISC architecture is modeled as an array of size 1x1 (*np_1x1_reg*).

Figure 5 presents the leakage power results of the architectures. The *4x4_all* has high leakage due to the all interconnect options in Table 2. The Synopsys synthesis tool is not always consistent as the leakage power of *4x4_ref* is significantly reduced compared to *4x4_arch_1*. Figures 6 and 7 present the total power results for the two benchmarks. The order of components in the legends are the same in the charts of which PRF_CGA, PRF_VLIW and VLIW_CU are negligible. The following components are separately measured and divided in CGA and VLIW parts. There is a data register file DRF, predicated register file PRF, functional units FU and configuration memory CM.

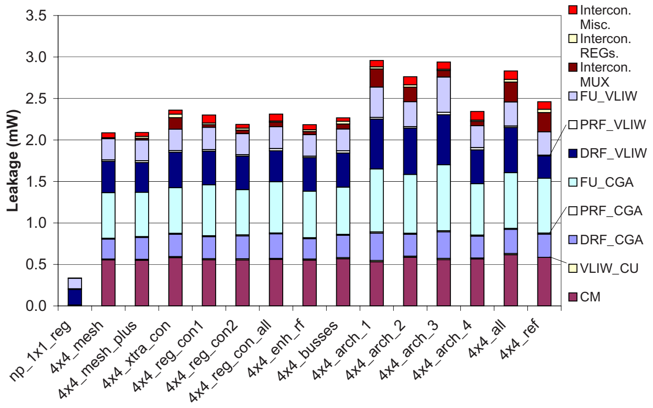


Fig. 5. Leakage Power

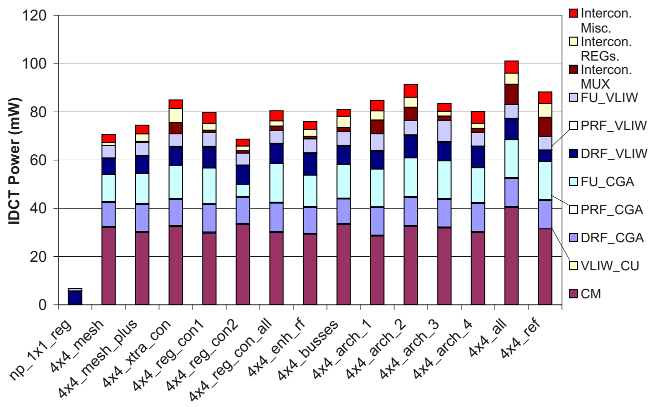


Fig. 6. IDCT Power @ 100MHz

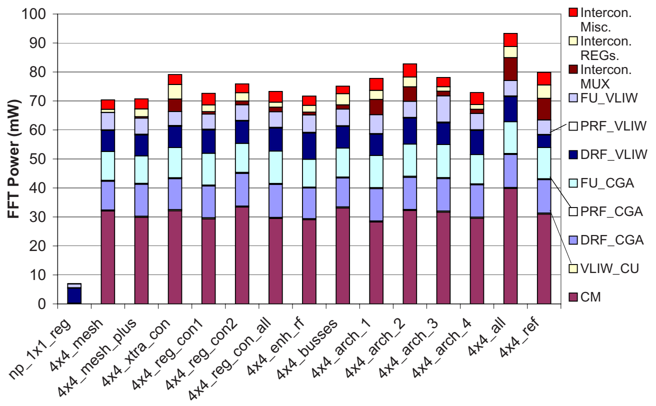


Fig. 7. FFT Power @ 100MHz

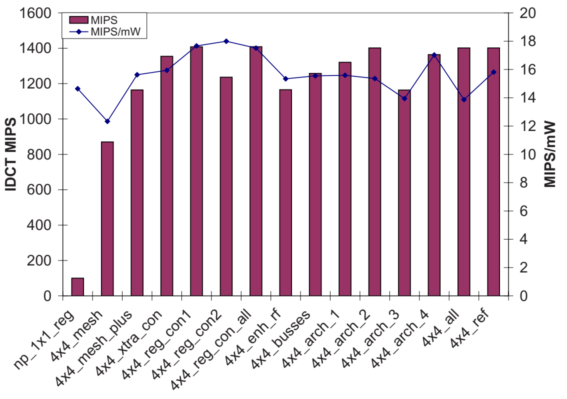


Fig. 8. IDCT Performance @ 100MHz

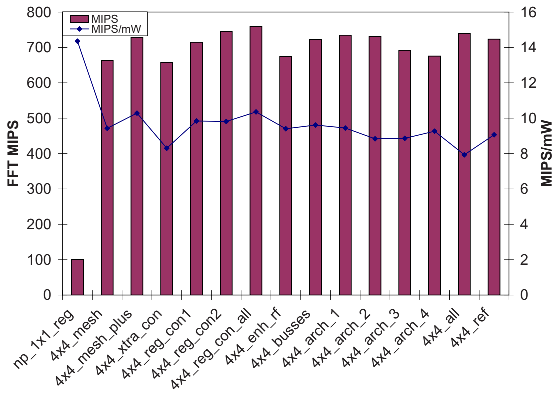


Fig. 9. FFT Performance @ 100MHz

The register files (PRF and DRF) in VLIW mode are global while in CGA mode the sum of all local register files is considered. The *4x4_all* option consumes highest total power compared to the other options mainly due to the large amount of interconnections requiring larger CMs.

We measured the performance in MIPS using the number of executed instructions obtained from the compiled instruction set simulator divided by the total number of execution cycles. The resulting performance charts are depicted in Figures 8 and 9. The *np_1x1_reg* shows a low MIPS result, but yet gives a high MIPS/mW. The performance charts also show that the fully interconnected architecture *4x4_all* is not an efficient option. The bypass feature in *4x4_xtra_con* has some advantage for IDCT only, but certainly not for FFT. This is an typical application that accesses DRFs quite often, which is shown in Figure 9 by *4x4_enh_rf* and *4x4_reg_con_all* having a higher MIPS/mW factor compared to *4x4_xtra_con*. The presence of busses in the design does not give much advantages, since it requires additional multiplexors, larger CMs and data networks, that all increase power.

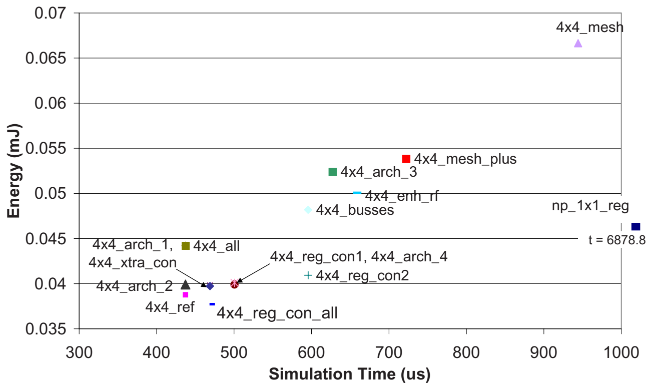


Fig. 10. IDCT Energy-Delay @ 100MHz

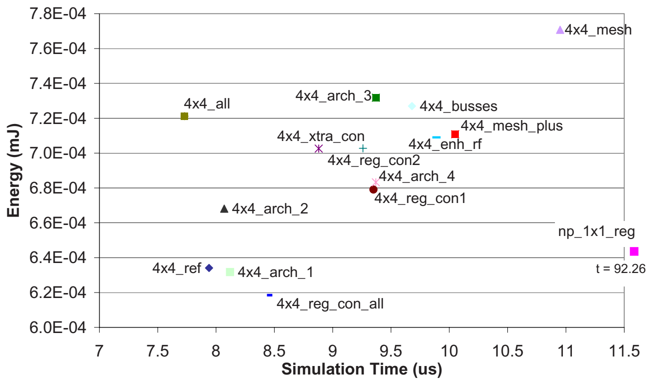


Fig. 11. FFT Energy-Delay @ 100MHz

The power and performance charts are a good indication for selection of the appropriate architecture, however the energy vs runtime shown in Figures 10 and 11 is of more significance.

The *np_1x1_reg* architecture does not have the highest energy consumption, however it is about 8 times slower than the *4x4_mesh* architecture. Of all considered 4x4 arrays the *4x4_mesh* has the highest energy consumption for both benchmarks. The *4x4_reg_con_all* has the lowest energy consumption for both benchmarks, but it is not the fastest option, which is *4x4_all*. The *4x4_ref* architecture is a good alternative between energy and performance results.

We select the architecture with the lowest energy consumption and reasonable performance, which can be seen from the energy-delay charts is *4x4_reg_con_all*. The *4x4_reg_con_all* interconnection architecture consists of a *mesh* and *mesh plus* interconnection topology and diagonal connections via multiplexors between FUs and local and global RFs as depicted in Figure 12.

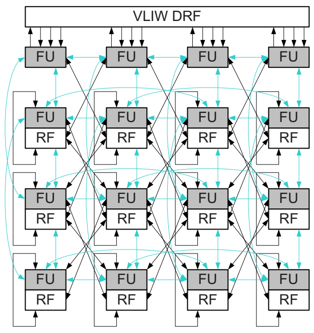


Fig. 12. Proposed ADRES Instance

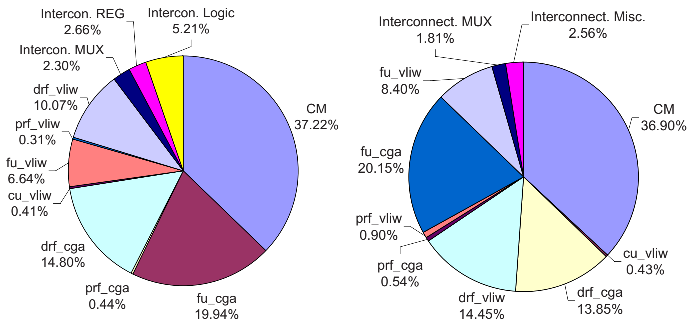
4.3 Power Breakdown

The power and performance figures of the proposed ADRES instance are summarized in Table 3. The detailed power and area breakdown for IDCT are depicted in Figure 13. The components with *_vliw* postfix are located in the VLIW section and are operational in both VLIW and CGA mode, while those with the *_cga* postfix are located in the CGA section.

Table 3. Characteristics of 4x4_reg_con_all

	Total		MIPS	MIPS/mW	mW/MHz
	Power (mW)	Energy (uJ)			
FFT	73.28	0.619	759	10.35	0.7328
IDCT	80.45	37.72	1409	17.51	0.8045

The power chart in Figure 13(a) shows that the CMs, FUs and DRFs consume most of the power. Interesting to note is the low power consumption of less than 10% of the interconnections between modules as opposed to 10 - 30% that where reported for regular DSP design [7]. The VLIW control unit (cu_vliw) and PRFs consume the least amount of power as these are the smallest components.



(a) IDCT Power: 80.45mW (b) Area: 1.59mm^2

Fig. 13. Power and Area distribution 4x4_reg_con_all @ 100MHz

The area chart in Figure 13(b) shows that the CMs, FUs and DRFs require most area. The CMs have a fixed depth of 128 words during the architectural explorations.

The power measurements of ADRES were based on the core architecture only. To get a power prediction of the complete ADRES processor including instruction cache and data memories we simulated the activities of these blocks. For the IDCT the data memories were activated for 100% of the cycles and the instruction cache for 4.3% respectively. According to the data sheets of SRAM memory models this results in a power consumption of 0.03mW/MHz for the data memories and 0.05 mW/MHz for the instruction cache. The power consumption of the not activated memory modules is negligible and assumed as zero. The distribution of all the components in the processor for the IDCT benchmark is shown in Figure 14. It shows that in addition to the FUs,

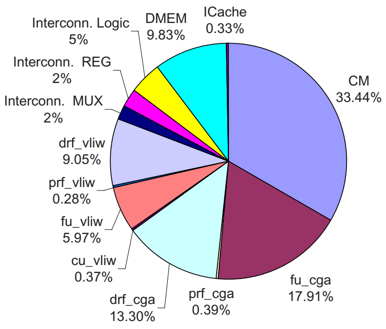


Fig. 14. Overview of ADRES Processor Power Consumption with IDCT

DRFs and CMs, the data memory (DMEM) consumes significant amount of power. The I-cache power consumption is negligible.

The experiments were obtained utilizing the non-pipelined version ADRESv0, however, a pipelined version ADRESv1 is under development at the moment. Preliminary results are encouraging reaching the performance target of 40MOPS/mW. The reconfigurable architecture Morphosys [10] is comparable to ADRES, but does not provide power results of IDCT and FFT. No other comparable architectures are know at the moment of writing.

5 Conclusions

In this paper we explored various architectures for the ADRES coarse-grained reconfigurable array and selected the appropriate one based on power, energy and performance trade-offs. An especially defined methodology and tool flow 1) maps the FFT and IDCT benchmarks on the array using the DRESC compiler for high performance, low energy execution, 2) synthesizes each architecture into a front-end design and 3) simulates with either the compiled ISA simulator, ModelSimv6.0a or the Esterel simulator for performance and power evaluations. Power is calculated by annotating the switching activity after Esterel or RTL simulation onto the gate-level design. Our three-fold simulation approach permits to obtain results quickly, which is important for architecture exploration.

Fourteen different ADRES instances were created based on 7 different architectural features and evaluated with the available tool flow. The obtained power, energy and performance charts show that a combination of mesh and mesh plus interconnection topologies with diagonal connections between functional units and local data register files results in good performance of 10.35 - 17.51 MIPS/mW and power of 73.28 - 80.45mW with the least amount of energy 0.619 - 37.72uJ for FFT and IDCT, respectively.

References

1. Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man and Rudy Lauwereins: *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, IMEC, 2003, Kapeldreef 75, B-3001, Leuven, Belgium, DATE 2004
2. Zion Kwok and Steven J. E. Wilton: *Register File Architecture Optimization in a Coarse-Grained Reconfigurable Architecture*, University of British Columbia, April 2005, 35–44, Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05) - Volume 00
3. The IMPACT Group. *http://www.crhc.uiuc.edu/Impact/*
4. *http://www-sop.inria.fr/esterel-org/*
5. *http://www.synopsys.com/*
6. Bingfeng Mei: *A Coarse-Grained Reconfigurable Architecture Template and its Compilation Techniques*, Katholieke Universiteit Leuven, Januari 2005, ISBN: 90-5682-578-X
7. Renu Mehra and Jan Rabaey: *Behavioral Level Power Estimation and Exploration*, April 1994, Proc. First International Workshop on Low Power Design, University of California at Berkeley
8. Bingfeng Mei, Andy Lambrechts, Jean-Yves Mignolet, Diederik Verkerst and Rudy Lauwereins: *Architecture Exploration for a Reconfigurable Architecture Template*, March 2005, IEEE Design & Test of Computers, IMEC and Katholieke Universiteit Leuven
9. Andy Lambrechts, Praveen Raghavan and Murali Jayapala: *Energy-Aware Interconnect-Exploration of Coarse Grained Reconfigurable Processors*, 4th Workshop on Application Specific Processors (WASP), September 2005
10. Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi and Nader Bagherzadeh: *MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications*, University of California (US) and Federal University of Rio de Janeiro (Brazil), May 2000, 465 – 481, IEEE Transactions on Computers

A Configurable Multi-ported Register File Architecture for Soft Processor Cores

Mazen A.R. Saghir and Rawan Naous

Department of Electrical and Computer Engineering
American University of Beirut
P.O. Box 11-0236 Riad El-Solh, Beirut 1107 2020, Lebanon
{mazen,rsn11}@aub.edu.lb

Abstract. This paper describes the architecture of a configurable, multi-ported register file for soft processor cores. The register file is designed using the low-latency block RAMs found in high-density FPGAs like the Xilinx Virtex-4. The latency of the register file and its utilization of FPGA resources are evaluated with respect to design parameters that include word length, register file size, and number of read and write ports. Experimental results demonstrate the flexibility, performance, and area efficiency of our proposed register file architecture.

1 Introduction

Growing interest in configurable and reconfigurable computing has increased the proportion of field programmable gate arrays (FPGAs) being used to implement computational platforms. High logic densities and rich on-chip features, such as hardware multipliers, memory blocks, DSP blocks, peripheral controllers, and microprocessor cores, make it possible to quickly implement highly tuned system designs that couple software programmable processor cores with custom hardware blocks that can be implemented inside the logic fabric of an FPGA.

The past few years have also witnessed the emergence of *soft* processor cores [1,2]. These processors are specified in a hardware description language, such as VHDL or Verilog, and implemented in the logic fabric of an FPGA. The main advantages of soft processors are that their datapaths can be easily reconfigured and their instruction set architectures extended to support user-defined machine instructions. To overcome the performance and area inefficiencies of implementing soft processors in FPGAs, the microarchitectures of these processors are typically optimized to make use of available on-chip resources [3]. For example, a common optimization is to use embedded memory blocks to implement register files [4]. Given the simplicity of current soft processor architectures, these register files typically feature two read ports and one write port. However, as the complexity of soft processor cores increases to support wider instruction issue [5,12], multithreading [6], and customized datapaths [7] flexible register file architectures capable of supporting multiple read and write ports are needed.

In this paper we describe the architecture of a configurable, multi-ported, register file for soft processor cores. We also evaluate the performance of our

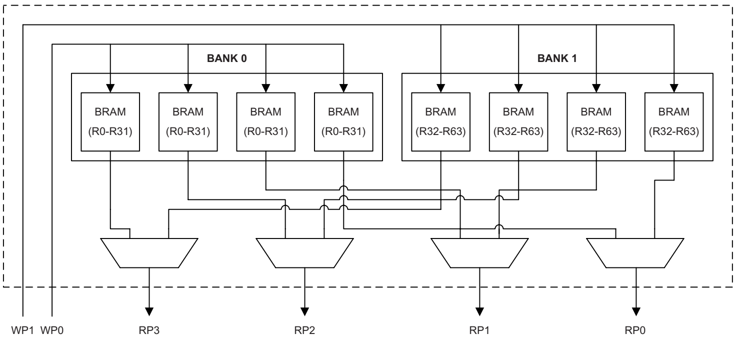


Fig. 1. A register file with four read and two write ports

design and measure its utilization of FPGA resources. In Sect. 2, we describe the architecture of our register file and highlight the characteristics and limitations of its design. Then, in Sect. 3 we describe how we evaluate the performance and area of the register file. Next, in Sect. 4, we analyze the effects of various register file design parameters such as data word length, register size, and number of read/write ports on both the latency and area of the register file. Then, in Sect. 5, we show how our register file can be used to customize the datapath of a soft VLIW processor and how this affects the overall performance and area of the processor. In Sect. 6, we describe related work and compare it to our own. Finally, in Sect. 7, we provide our conclusions and describe future work.

2 Multi-ported Register File Architecture

Our register file is designed around the embedded block RAMs (BRAMs) found in contemporary, high-density, FPGAs such as the Xilinx Virtex-4 [8]. The BRAMs on the Virtex-4 are dual-ported, synchronous, memory blocks capable of storing 16,384 data and 2,048 parity bits that can be organized in various aspect ratios. The two ports are independent and can each be configured as either a read or a write port. In our design, each BRAM is configured as a 512 $\times$ 36-bit memory block with one read port and one write port, and this configuration forms the basic building block in our register file. Figure 1 shows how eight such BRAMs can be used to implement a 64 $\times$ 32-bit register file with four read and two write ports.

The key to supporting multiple register ports is to organize the BRAMs into register banks and duplicate data across the various BRAMs within each bank. The register banks are each used to store a *subset* of the register file. In Fig. 1, the BRAMs are divided into two register banks used to store registers `R0` to `R31` and `R32` to `R63`, respectively. Since register banks hold mutually exclusive sets of registers, they can be updated independently. Associating each register bank with a separate write port provides support for multiple write ports.

Within each register bank, multiple BRAMs are used to store *duplicate* copies of the corresponding register subset. By duplicating register values across BRAMs, the values of different registers can be read from different BRAMs simultaneously, thus providing support for multiple read ports. Since the register file in Fig. 1 has four read ports, each register bank contains four BRAMs that store duplicate copies of the corresponding register subset. Since each register bank stores a different subset of the register file, multiplexers are used to provide access to the registers stored in each register bank. In Fig. 1, four multiplexers – one for each read port – are used to provide access to all the registers in both register banks. Finally, to maintain data consistency among the BRAMs in a register bank, each BRAM is connected to the corresponding register write port making it possible to update all BRAMs simultaneously.

2.1 Design Characteristics and Limitations

A major characteristic of our register file is that it is fully configurable and can be designed with an *arbitrary* word length, register size, and number of read or write ports. Here it is worth noting that the number of register ports does not have to be even or a power of two. This enables customizing the register file at a very fine level to the specific needs of the target processor microarchitecture. However, the size of the register subset associated with a register bank must be a power of two, and this causes some register file configurations to have register banks with different sized register subsets. For example, a register file of size 64 with five read ports and three write ports will consist of three register banks, two of which containing 16 registers and one containing 32 registers. However, such an asymmetric distribution of register subsets does not affect the performance, area, or functionality of the register file.

Another characteristic of our register file is that it influences the way registers and instructions get allocated and scheduled, respectively. Since registers are distributed across several banks associated with different write ports, registers must be allocated, and instructions scheduled, in a manner that avoids contention for the write ports. Thus, instructions cannot be scheduled to execute in parallel if they produce results in registers that belong to the same register bank. Although various software and hardware techniques, such as register renaming, can be applied to solving this problem [9,10], these techniques and their effects are beyond the scope of this paper.

Finally, our register file is limited by the number and size of the BRAMs available within the FPGA. If WL and DW are the word length and data width of the register file and BRAMs, respectively, implementing a register file with M read ports and N write ports requires $\lceil WL/DW \rceil \times M \times N$ BRAMs distributed across N banks. Since BRAMs typically have a fixed size and can only be configured in a limited number of aspect ratios, there is a limit on the number of registers that can be stored within each BRAM. In the Xilinx Virtex-4 FPGA, the BRAM aspect ratio that corresponds to the maximum word length is 512×36 bits. This limits the size of the register subset that can be stored in a BRAM to 512 registers. Although this limit can be easily extended by using more BRAMs in each register bank, we do not consider this case in this paper.

3 Performance and Area Evaluation

To evaluate the performance and area of our register file architecture, we developed a parameterizable register file generator called MPRFGen. MPRFGen reads design parameters such as word length, register size, and number of read and write ports and generates the corresponding VHDL code, which instantiates the appropriate number of BRAMs and organizes them as a multi-ported register file. Once the VHDL code is generated, we use the Xilinx ISE 8.2.03i tools, targeting a Xilinx Virtex-4 LX (XC4VLX160-12FF1148) FPGA, to synthesize and implement the register file. We chose the XC4VLX160 as our target FPGA because of its high logic density, which is representative of the types of FPGAs used for implementing computational platforms designed around soft processor cores, and because it contains 288 BRAMs, which is large enough to implement and evaluate a wide range of register file organizations. Finally, we used the Xilinx Timing Analyzer tool to measure the latency of the register file and the post-place-and-route synthesis reports to measure its utilization of FPGA resources. Although somewhat crude, the number of BRAMs and FPGA slices used to implement a register file is still an accurate measure of its area.

3.1 Register File Latency

The latency of the register file is determined by its critical path delay as reported by the timing analyzer tool. The critical path delay can be generally expressed by the following equation:

$$\mathit{Critical\ Path\ Delay} = T_{\mathrm{bram}} + T_{\mathrm{routing}} + T_{\mathrm{mux}} \ . \tag{1}$$

Here, T_{bram} is the latency of a BRAM block, T_{routing} is the routing delay along the critical path between the corresponding register bank and output multiplexer, and T_{mux} is the delay of the output multiplexer. Typically, T_{bram} is a constant that depends on the fabrication process technology and speed grade of the FPGA. For the Xilinx XC4VLX160-12FF1148, $T_{\mathrm{bram}} = 1.647$ ns. On the other hand, T_{routing} and T_{mux} depend on the organization of the register file and vary with word length and the number of read and write ports. Both delays also depend on the architecture of the FPGA routing network and the efficiency of the placement and routing tool. In Sect. 4, we examine how various design parameters affect the latency and FPGA resource utilization of our register file architecture.

4 Results and Analysis

In this section we present the results of three experiments we conducted to assess the impact of word length, register file size, and the number of read and

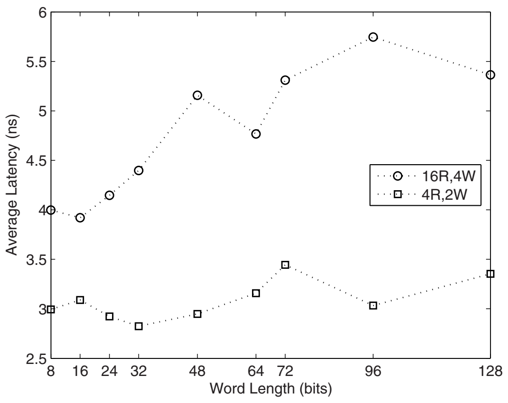


Fig. 2. Latency vs. word length

write ports on both the latency and FPGA resource utilization of our register file architecture. To minimize the effect of the place-and-route tool on latency measurements, we report the *average latency* for each register file organization, which we compute based on *nine* latency measurements corresponding to different starting placer cost tables [11]. We also report FPGA resource utilization in terms of the number of BRAMs and slices used to implement each register file.

4.1 Effect of Word Length

Our first experiment examined the impact of register word length on the latency and area of two register files configured with 16 read and 4 write ports and 4 read and 2 write ports, respectively. Figure 2 shows the latencies of both register files as the word length is increased from 8 to 128 bits. Our results show that latency tends to increase with word length since the amount of routing used to connect register banks to output multiplexers also increases with word length. The additional routing delays cause T_{routing} to increase, and this causes overall latency to increase. However, our results also suggest that the latency is sensitive to variations in placement and routing. In Fig. 2, the latencies associated with some word lengths are actually *lower* than those associated with smaller word lengths. Finally, our results show that the impact of word length becomes more pronounced as the number of register ports increases. This is due to the corresponding increase in routing resources needed to connect register banks to output multiplexers, which is also proportional to the product of the number of read and write ports.

Figure 3 shows the utilization of FPGA resource as a function of word length for both register files. In Sect. 3 we saw that the number of BRAMs used to implement a register file depends on the word length, the data width of the BRAM, and the number of read and write ports. Figure 3 shows that, for each register file, the number of BRAMs increases in steps that track the ratio of the register word length to BRAM data width. Figure 3 also shows that the number

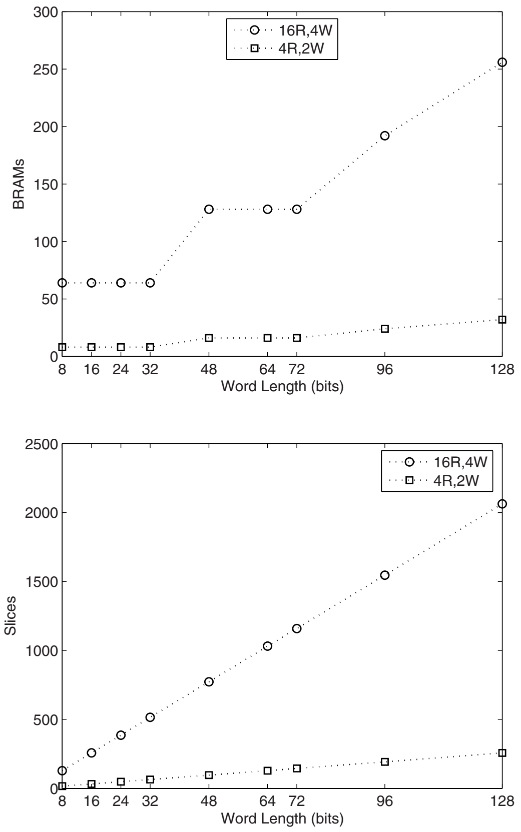


Fig. 3. Resource utilization vs. word length

of FPGA slices used to implement the register file is a linear function of its word length. This is due to the fact that slices are only used to implement output multiplexers and that the number of slices needed to implement a register file is proportional to the product of word length by the number of read ports.

4.2 Effect of Register File Size

Our second experiment examined the impact of register file size. For this experiment we used three register files that were each configured with 16 read ports and 4 write ports, and that had word lengths of 16-, 32-, and 64-bits, respectively. Figure 4 shows the average latencies of the three register files as we increased the size of the register file from 16 to 256.

Our results show that there is little correlation between the size of a register file and its latency. As the size of the register file increases, additional addressing lines are needed for each of its read and write ports, and the number of registers allocated to each bank increases. Routing the additional addressing lines can introduce some variation in the latency of different sized register files, but these

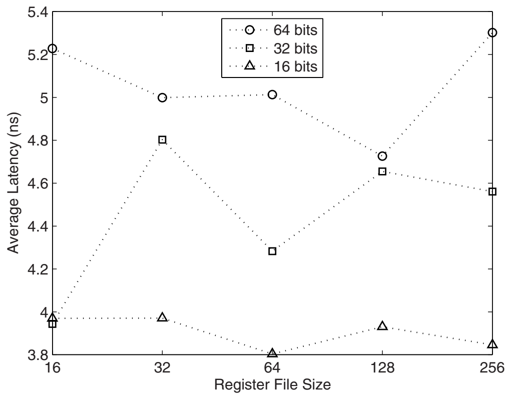


Fig. 4. Latency vs. register file size

Table 1. FPGA resource utilization for a register file with 16 read and 4 write ports

Word Length	BRAMs	Slices
16-bits	64	257
32-bits	64	515
64-bits	128	1,031

variations are mainly due to the effects of placement and routing. Moreover, the increase in the number of registers per bank is typically absorbed by the BRAMs, which can each store up to 512 registers. As a result, even if the size of the register file increases, its *organization* – the number of its BRAMs, register banks, output multiplexers, and the data paths used to connect them – does not change. This leads us to conclude that the latency of a register file is independent of its size, and that the variations in the latencies of different sized register files that otherwise have the *same* organization are mainly due to the effects of placement and routing. Of course, if more than 512 registers per bank are needed, additional BRAMs will be required, and this will effect both the organization and the latency of the register file. However, since we have assumed that this limit will not be exceeded, we do not consider this case any further. Finally, since the organization of a register file does not change with its size, it follows that the area occupied by the register file also does not change. Table 1 shows the utilization of FPGA resources for each of the register files used in this experiment. As we increase the size of each register file, its utilization of FPGA resources remains unchanged.

4.3 Effect of Read/Write Ports

Our third experiment examined the impact of the number of register read and write ports. For this experiment we used a 256 × 32-bit register file and varied the number of its read ports from 2 to 128 and the number of its write ports from

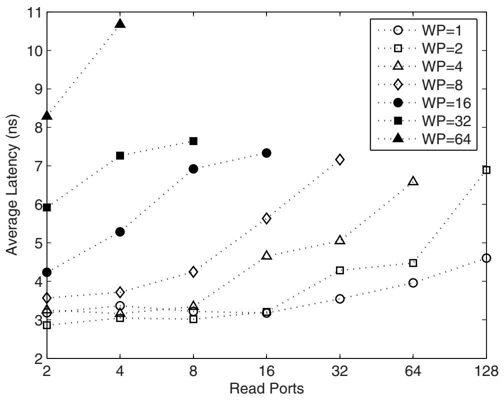


Fig. 5. Latency vs. number of read and write ports

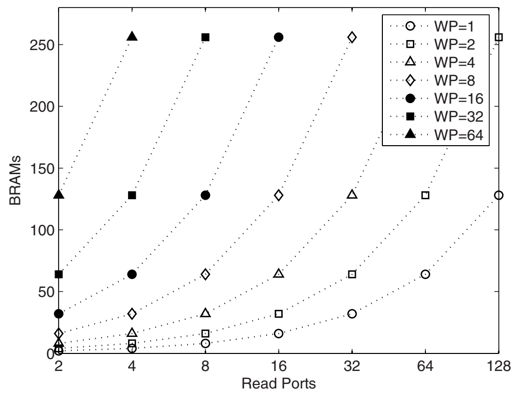
1 to 64. Since our target FPGA, the Xilinx XC4VLX160, contains 288 BRAMs, we only considered register file configurations whose BRAM requirements could be accommodated by the FPGA.

Figure 5 shows the average latency of the register file as a function of its read and write ports[1]. As expected, our results show that latency increases with the number of read and write ports. This reflects the architecture of our register file where the *width* of the output multiplexers – the number of inputs to select from – depends on the number of register banks, which, in turn, depends on the number of write ports. Since output multiplexers are implemented as a hierarchy of lookup tables and two-input multiplexers, the depth of the hierarchy and the corresponding delay, T_{mux}, increase as the width of the output multiplexers increases. Moreover, as the number of register banks increases, more routing resources are used to connect the register banks to the output multiplexers, causing T_{routing} to also increase. Similarly, increasing the number of read ports increases the *number* of output multiplexers and the amount of routing needed to connect the register banks to the output multiplexers, and this also increases T_{routing}. The latency of a register file therefore increases in proportion to the product of the number of read and write ports.

Figure 6 shows the utilization of FPGA resources as a function of read and write ports. In Sect. 3 we saw that BRAM utilization is proportional to the product of the number of read and write ports. Similarly, the utilization of FPGA slices depends on the width of the output multiplexer, which determines the number of slices needed to implement the multiplexer. The number of slices used to implement the register file is proportional to the product of the number of read and write ports.

Having gained some insight into the effects of various register file design parameters on latency and area, we next examine how our register file can be used

[1] Write ports are labeled WP on the graphs.



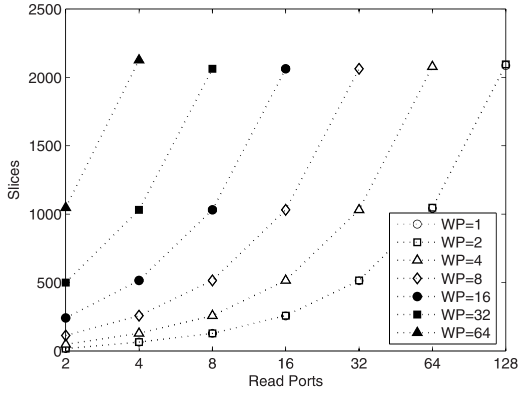


Fig. 6. Resource utilization vs. number of read and write ports

to customize the datapath of a soft VLIW processor, and we assess its impact on the overall performance and area of the processor.

5 Soft Processor Datapath Customization Using a Multi-ported Register File

To demonstrate the flexibility of our multi-ported register file and assess its impact on the overall performance and area of a high-performance soft processor core, we used two instances of our register file to customize the datapath of a configurable soft VLIW processor based on [12]. The datapath uses separate register files, DRF and ARF, for storing data and memory addresses, respectively. As the number of functional units in the datapath is increased, more register ports are generally needed to provide access to operands and results. For this experiment, we used our register file architecture to implement the DRF for two configurations of the processor datapath, DP1 and DP2. We configured the DRF of the first processor datapath (DRF1) as a 32 × 32-bit register file with 8 read ports

Table 2. Performance and FPGA resource utilization of DP1 and DP2

Datapath	IPC	Clock		Slices	BRAMs	MULT18×18
		Frequency	Period			
DP1	5	132.4 MHz	7.55 ns	2,347	23	6
DP2	8	119.3 MHz	8.38 ns	5,152	65	12

Table 3. Performance and FPGA resource utilization of DRF1 and DRF2

Register File	Word Length	Register Size	Read Ports	Write Ports	Latency	Slices	BRAMs
DRF1	32 bits	32	8	2	2.94 ns	128	16
DRF2	32 bits	64	14	4	3.61 ns	465	56

and 2 write ports. We then configured the DRF of the second processor datapath (DRF2) as a 64 × 32-bit register file with 14 read and 4 write ports.

Tables 2 and 3 show the performance and resource utilization of the two datapaths and register files, respectively. For each processor, we show the maximum number of instructions executable per clock cycle (IPC), the processor clock frequency and corresponding period, and the number of FPGA slices, BRAMs, and hardware multipliers used to implement the datapath. For each register file we show the corresponding latency and the number of FPGA slices and BRAMs used to implement the register file. Our results show that DRF1 uses 5.5% of the slices and 69.6% of the BRAMs used to implement DP1, and that its latency corresponds to 38.9% of the processor clock period. Our results also show that DRF2 uses 9.0% of the slices and 86.2% of the BRAMs used to implement DP2, and that its latency corresponds to 43.1% of the processor clock period. While the high BRAM utilization is expected – BRAMs are, after all, only used to implement register files and data memories – these results demonstrate the flexibility, performance, and area efficiency of our multi-ported register file.

6 Related Work

The use of embedded BRAMs to implement register files is a well known microarchitectural optimization for the efficient implementation of soft processor cores [4]. As soft processor architectures evolve to include wide instruction issue [5,12], multithreading [6], and customized datapaths [7], multi-ported register files become necessary. However, while most of the register files used in these processors provide multiple read ports, none, to the best of our knowledge, provides efficient support for multiple write ports.

Due to their central role in exploiting high levels of instruction parallelism, the design of efficient multi-ported register files for high-performance, general-purpose processors has been the subject of much research [13]. Most techniques focus on partitioning the register file into multiple banks with few ports each to improve access time and reduce area and power consumption. [14,15,16]. We also partition our register file into multiple banks, and benefit from BRAMs with latencies that are much smaller than the typical clock period of a soft processor. In [17], the register file is partitioned to reduce the number of ports in each partition, and data is duplicated to ensure data consistency. However, an additional clock cycle is needed to update the other partition when one of the partitions is updated. In our implementation we avoid this problem by connecting all BRAMs that contain duplicate information to the same write port, thus ensuring all BRAMs are updated simultaneously.

7 Conclusions and Future Work

In this paper we described and evaluated the architecture of a configurable, multi-ported register file for soft processor cores. Our register file is designed around the low-latency embedded memory blocks found in contemporary high-denisty FPGAs such as the Xilinx Virtex-4 FPGA. Our results show that both the latency of the register file and its utilization of FPGA resources are proportional to the product of its word length and the number of its read and write ports. The latency of the register file also depends on the latency of individual BRAM blocks. When integrated into the datapath of a configurable VLIW soft processor, our register file architecture provided the flexibility to support processor datapath customization while exhibiting low latencies and occupying a small proportion of the processor datapath.

To continue this work, we are developing register file latency and area models that we will use in a soft VLIW processor architecture exploration tool. We are also assessing the power consumption characteristics of our register file with the aim of also developing a power model for our register file architecture. Finally, we are developing new register allocation and instruction scheduling passes for our port of the GCC compiler to handle the proposed register file architecture.

References

1. *MicroBlaze Processor Reference Guide*. `http://www.xilinx.com`.
2. *NIOS-II Processor Reference Handbook*. `http://www.altera.com`.
3. P. Yiannacouras, J. G. Steffan, and J. Rose, “Application-Specific Customization of Soft Processor Microarchitecture”, *Proceedings of the 14th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2006)*, pp. 201–210, ACM, Feb. 22-24, 2006.
4. Xilinx Corportaion, “Using Block RAM in Spartan-3 Generation FPGAs”, *Xilinx Application Note XAPP463 (v2.0)*, March 1, 2005.
5. A. Jones et al., “An FPGA-based VLIW Processor with Custom Hardware Execution”, *Proceedings of the 13th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2005)*, pp. 107-117, ACM, Feb. 20-22, 2005.
6. R. Dimond, O. Mencer, and W. Luk, “CUSTARD - A Customizable Threaded FPGA Soft Processor and Tools”, *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL 2005)*, pp. 1–6, IEEE, Aug. 24-26, 2005.

7. D. Jain et al. "Automatically Customizing VLIW Architectures with Coarse-Grained Application-Specific Functional Units", *Proceedings of the Eighth International Workshop on Software and Compilers for Embedded Systems (SCOPES 2004)*, pp. 17-32, Springer Berlin/Heidelberg, Sept. 2-3, 2004.
8. *Virtex-4 User Guide*, UG070 (v1.6), `http://www.xilinx.com`, Oct. 6, 2006.
9. Steven Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers/Elsevier, 1997.
10. John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Third Edition, Morgan Kaufmann Publishers/Elsevier, 2003.
11. *Development System Reference Guide (8.2i)*, `http://www.xilinx.com`.
12. M. A. R. Saghir, M. El-Majzoub, and P. Akl, "Customizing the Datapath and ISA of Soft VLIW Processors", *Proceedings of the 2007 International Conference on High Performance Embedded Architectures and Compilers (HiPEAC 2007)*, pp. 276–290, Springer LNCS 4367, Jan. 28-30, 2007.
13. K. Farkas, N. Jouppi, and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors", *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, pp. 40–51, IEEE, 1996.
14. J. L. Cruz et al., "Multiple-Banked Register File Architectures", *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA 2000)*, pp. 316–325, ACM, 2000.
15. J. Zalema et al., "Two-Level Hierarchical Register File Organization for VLIW Processors", *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 2000)*, pp. 137–146, ACM/IEEE, 2000.
16. J. H. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors", *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA 2003)*, pp. 62–71, ACM, 2003.
17. R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture", *Proceedings of the International Conference on Computer Design (ICCD'98)*, pp. 90–95, IEEE, 1998.

MT-ADRES: Multithreading on Coarse-Grained Reconfigurable Architecture

Kehuai Wu[1], Andreas Kanstein[2], Jan Madsen[1], and Mladen Berekovic[3]

[1] Dept. of Informatics and Mathematic Modelling,
Technical Univ. of Denmark
{kw, jan}@imm.dtu.dk
[2] Freescale Semiconductor
a.kanstein@freescale.com
[3] IMEC, Netherland
Mladen.Berekovic@imec-nl.nl

Abstract. The coarse-grained reconfigurable architecture ADRES (Architecture for Dynamically Reconfigurable Embedded Systems) and its compiler offer high instruction-level parallelism (ILP) to applications by means of a sparsely interconnected array of functional units and register files. As high-ILP architectures achieve only low parallelism when executing partially sequential code segments, which is also known as Amdahl's law, this paper proposes to extend ADRES to MT-ADRES (Multi-Threaded ADRES) to also exploit thread-level parallelism. On MT-ADRES architectures, the array can be partitioned in multiple smaller arrays that can execute threads in parallel. Because the partition can be changed dynamically, this extension provides more flexibility than a multi-core approach. This article presents details of the enhanced architecture and results obtained from an MPEG-2 decoder implementation that exploits a mix of thread-level parallelism and instruction-level parallelism.

1 Introduction

The ADRES architecture template is a datapath-coupled coarse-grained reconfigurable matrix [3]. As shown in Figure 1, it resembles a very-long-instruction-word (VLIW) architecture coupled with a 2-D Coarse-Grained heterogeneous reconfigurable Array (CGA), which is extended from the VLIW's datapath. As ADRES is defined using a template, the VLIW width, the array size, the interconnect topology, etc. can vary depending on the use case. The processor operates either in VLIW mode or in CGA mode. The global data register file is used in both modes and serves as a data interface between both modes, enabling an integrated compilation flow. When compiling applications for ADRES with the DRESC compiler [2], loops are modulo-scheduled for the CGA and the remaining code is compiled for the VLIW by the DRESC compiler [2]. By seamlessly switching the architecture between the VLIW mode and the CGA mode at run-time, statically partitioned and scheduled applications can be run on the ADRES with a high number of instructions-per-clock (IPC).

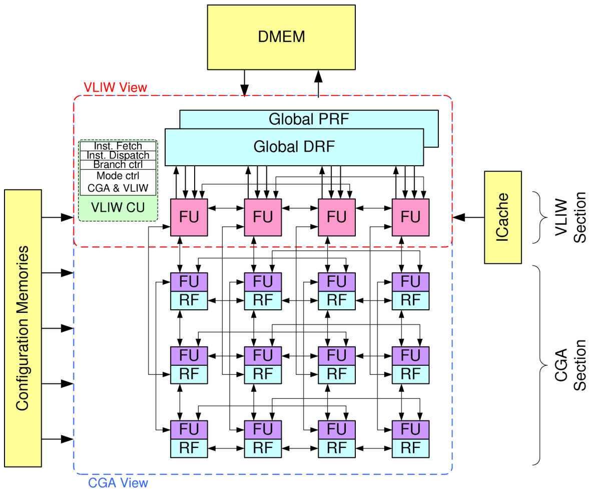


Fig. 1. ADRES architecture template and its two operation modes

We are using an MPEG2 decoder for our first demonstration of the multithreaded architecture. Based on the observation of our previous work [1], most MPEG2 decoder kernels can be scheduled on the CGA with the IPC ranging from 8 to 43. We have also observed that some modulo-scheduled kernels' IPC do not scale very well when the size of the CGA increases. Some of our most aggressive architectures have the potential to execute 64 instructions per clock cycle, but few applications can utilize this level of parallelism, resulting in a much lower average IPC. This is caused by two reasons: (1) The inherent ILP of the kernels is low and cannot be increased efficiently even with loop unrolling, or the code is too complex to be scheduled efficiently on so many units due to resource constraints, for example the number of memory ports. (2) The CGA is idle when executing sequential code in VLIW mode. The more sequential code is executed, the lower the achieved application's average IPC, and in turn, the lower the CGA utilization. In conclusion, even though the ADRES architecture is highly scalable, we are facing the challenge of getting more parallelism out of many applications, which fits better to be executed on smaller ADRES arrays. This is commonly known as Amdahl's law [4].

The ADRES architecture has a large amount of functional units (FU), thus the most appealing approach for increasing the application parallelism would be simultaneous multi-threading (SMT) [5]. An existing such architecture for the supercomputing domain has been developed at UT Austin [8]. However, because the ADRES implementation applies static scheduling due to low-power requirements, such dynamic/speculative [6,7] threading is inappropriate. Instead, our threading approach identifies an application's coarse-grained parallelism based on the static analysis results.

If properly reorganized and transformed at programming time, multiple kernels in the same application can be efficiently parallelized by the application designer. We can statically identify the low-LLP kernels through profiling, estimate the optimal choice of ADRES array size for each kernel, and partition a large ADRES array into several small-scaled ADRES sub-arrays that fits each kernel, which is parallelized into threads if possible. When an application is executed, a large ADRES array can be split into several smaller sub-arrays for executing several low-LLP kernels in parallel. Similarly, when a high-LLP kernel is executed, sub-arrays can be unified into a large ADRES array. Such a multi-threaded ADRES (MT-ADRES) is highly flexible, and can increase the over utilization of large-scaled ADRES arrays when the LLP of application is hard to explore .

This paper presents a demonstrative dual-threading experiment on the MPEG2 decoder implemented on top of the current single-threaded architecture and its matching compilation tools. Through this experiment, we have proven that the multithreading is feasible for the ADRES architecture.

Previously, a superscalar processor loosely coupled with a reconfigurable unit has been presented in [12]. Their approach enables the superscalar processor and the reconfigurable unit to execute different threads simultaneously. The CUSTARD architecture presented in [13] has a tighter coupling between the RISC and the reconfigurable unit. Their architecture and tools support the block/interleaved multithreading. To our knowledge, the parallelism supported by the reconfigurable unit has not been exploited by threading so far, and the MT-ADRES is the first attempt to support SMT both on the processor and the reconfigurable unit.

The rest of the paper is organized as follows. Section 2 discusses the threading solution we will achieve in the long run. Section 3 describes the proof-of-concept experiment we carried out, and what design issues must be addressed in terms of architecture and compilation methodology. Section 4 concludes our experiment and discuss the future work.

2 ADRES Multithreading

We propose a scalable partitioning-based threading approach for ADRES. The rich resource on the ADRES architecture allows us to partition a large ADRES

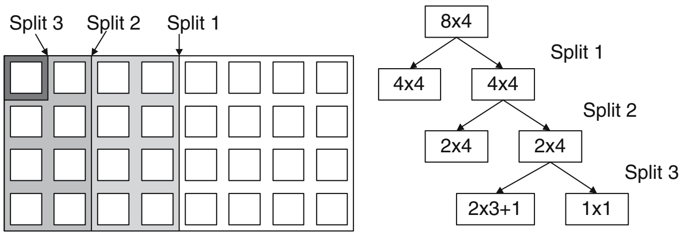


Fig. 2. Scalable partitioning-based threading

into two or more sub-arrays, each of which can be viewed as a down-scaled ADRES architecture and be partitioned further down hierarchically, as shown in Figure 2. With the proposed partitioning technique it is possible to dynamically share HW resources between threads without the cost of the control logic of dynamic out-of-order execution, as used in general-purpose processors. This technique can serve as a template for future FPGA-like platforms.

Each thread has its own resource requirement. A thread that has high ILP requires more computation resources, thus executing it on a larger partition results in the optimal use of the ADRES array and vise versa. A globally optimal application design demands that the programmer knows the IPC of each part of the application, so that he can find an efficient array partition for each thread.

The easiest way to find out how many resources are required by each part of a certain application is to profile the code. A programmer starts from a single-threaded application and profiles it on a large single-threaded ADRES. From the profiling results, kernels with low IPC and are less dependent to the other kernels are identified as the high-priority candidates for threading. Depending on the resource demand and dependency of the threads, the programmer statically plans on how and when the ADRES should split into partitions during application execution. When the threads are well-organized, the full ADRES array can be optimally utilized.

2.1 Architecture Design Aspects

The FU array on the ADRES is heterogeneous. There exists dedicated memory units, special arithmetic units and control/branch units on the array that constrain the partitioning. When partitioning the array, we have to guarantee that the program being executed on certain partition can be scheduled. This requires that any instruction invoked in a thread to be supported by at least one of the functional unit in the array partition. The well-formed partitions usually have at least one VLIW FU that can perform branch operation, one FU that can perform memory operation, several arithmetic units if needed, and several FUs that can handle general operations.

On the ADRES architecture, the VLIW register file (RF) is a critical resource that can not be partitioned easily. The most recent ADRES architecture employs a clustered register file that has previously been adapted in VLIW architectures [9,10]. If we prohibit the RF bank to be shared among several threads, the RF cluster can be partitioned with the VLIW/CGA, and the thread compilation can be greatly simplified. In case a single RF is used, the register allocation scheme must be revised to support the constrained register allocation, as suggested in our proof-of-concept experiments.

The ADRES architecture requires ultra-wide memory bandwidth. Multi-bank memory has been adapted to our recent architecture [14], and proven to cope nicely with our static data-allocation scheme. On ADRES, the memory and the algorithm core are interfaced with a crossbar with queues. Such a memory interface offers a scratchpad style of memory presentation to all the load/store units, thus the multi-bank memory can be used as a shared synchronization memory.

Besides the shared memory, other dedicated synchronization primitives like register-based semaphores or pipes can also be adapted to the ADRES template. These primitives can be connected between pairs of functional units that belongs to different thread partitions. Synchronization instruction can be added to certain functional units as intrinsics, which is supported by the current DRESC tools.

In the single-threading ADRES architecture, the program counter and the dynamic reconfiguration counter is controlled by a finite-state-machine (FSM) type control unit. When implementing the multithreading ADRES, we need an extendable control mechanism to match our hierarchically partitioned array. As shown in Figure 3, we duplicate the FSM type controller and organized the controllers in a hierarchical manner. In this multithreading controller, each possible partition is still controlled by an FSM controller, but the control path is extended with two units called merger and bypasser. The merger and bypasser form a hierarchical master-slave control that is easy to manage during program execution. The merger path is used to communicate change-of-flow information to the master controller of a partition, while the bypasser propagates the current PC or configuration memory address from the master to all slaves within a partition.

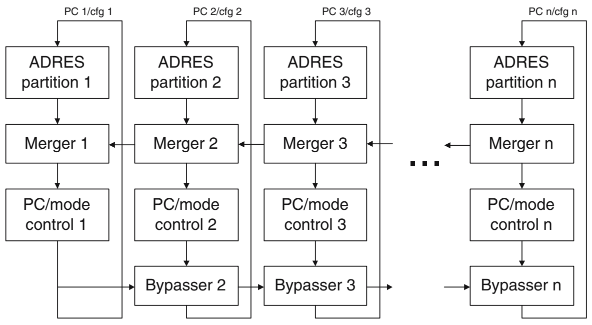


Fig. 3. Hierarchical multithreading controller

The principle of having such a control mechanism is as follows. Suppose we have an ADRES architecture that can be split into two halves for dual threading, while each half has its own controller. In order to reuse the controllers as much as possible, we need each controller to control a partition of the ADRES when the program is running in dual threaded mode, but also prefer one of the controller to take full control of the whole ADRES when the program is running in the single-threaded mode. By assigning one of the controller to control the whole ADRES, we created the master. When the ADRES is running in the single-thread mode, the master controller also receives the signal from the slave partition and merge them with the master partition's signal for creating global control signal. At the same time, the slave partition should bypass any signal generated from the local controller and follow the global control signal generated from the master partition. When the ADRES is running in the dual-threaded mode, the master and slave controller completely ignores the control signals coming from the other partition and only responds to the local signals. This strategy can be easily extended to cope with further partitioning.

2.2 Multithreading Methodology

The multithreading compilation tool chain is extended from our previous DRESC compiler[2]. With several modifications and rearrangements, most parts of the original DRESC tool chain, e.g. the IMPACT frontend, DRESC modulo scheduler, assembler and linker can be reused in our multithreading tool chain. The most significant modifications are made on the application programming, the architecture description and the simulator.

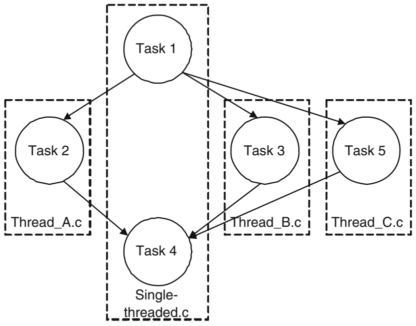


Fig. 4. Source code reorganization

Before the threaded application can be compiled, the application needs to be reorganized. As shown in Figure 4, the application is split into several C-files, each of which describes a thread that is executed on a specific partition, assuming the application is programmed in C. The data shared among threads are defined in a global H file that is included in all the thread C-files, and protected with synchronization mechanism. Such reorganization takes modest effort, but makes it easier for the programmer to experiment on different thread/partition combinations to find the optimal resource budget.

The multithreading ADRES architecture description is extended with the partition descriptions, as shown in Figure 5. Similar to the area-constrained placement and routing on the commercial FPGA, when a thread is scheduled on an ADRES partition, the instruction placement and routing is constrained by the partition description. The generated assembly code of each thread goes though the assembling process separately, and gets linked in the final compilation step.

As in the original DRESC tool chain, the simulator reads the architecture description and generates an architecture simulation model before the application simulation starts. As shown in Figure 3, each partition has its own controller, thus the generation of the controller's simulation model depends on the partition

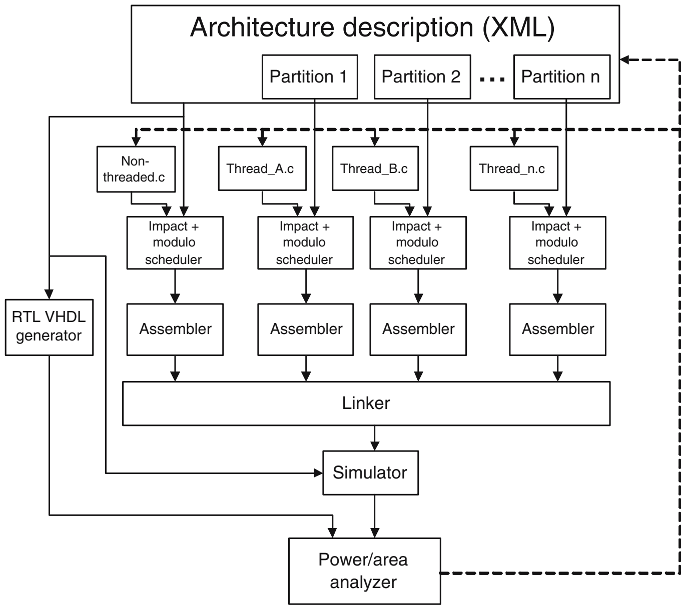


Fig. 5. Multithreading compilation tool chain

description as well. Furthermore, the control signal distribution is also partition-dependent, thus requires the partition description to be consulted during simulation model generation.

Some other minor practical issues needs to be addressed in our multithreading methodology. The most costly problem is that different partitions of the ADRES are conceptually different ADRES instances, thus a function compiled for a specific partition cannot be executed on any other partitions. When a function is called by more than one thread, multiple partition-specific binaries of this function has to be stored in the instruction memory for different caller. Secondly, multiple stacks need to be allocated in the data memory. Each time the ADRES splits into smaller partitions due to the threading, a new stack need to be created to store the temporary data. Currently, the best solution to decide where the new stack should be created is based on the profiling, and the thread stacks are allocated at compile time. And finally, each time the new thread is created, a new set of special purpose registers needs to be initialized. Several clock cycles are needed to properly initial the stack points, the return register, etc. immediately after the thread starts running.

3 Experiment

In order to understand what feature is needed in future DRESC tool chain for supporting the multi-threaded methodology described in section 2 and prove its feasibility, we carried out an experiment based on the MPEG2 decoder, a well-understood benchmark. Our objective is to go through the whole process of generating the threaded application executable, partitioning the instruction/data memory for threads, upgrading the cycle-true architecture simulation model and successfully simulating the execution of MPEG2 decoder with our simulator. By going through the whole process, we can acquire ample knowledge on how to automate the compilation for threads and simulation/RTL model generation of MT-ADRES.

3.1 Application and Methodology

Our proof-of-concept experiment achieves dual-threading on the MPEG2 decoder. The MPEG2 decoder can be parallelized on several granularities [11],

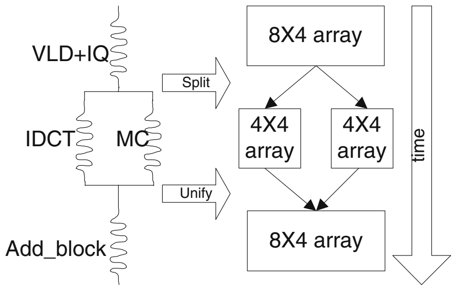


Fig. 6. Threading scenario on MPEG2 decoder

thus it is a suitable application to experiment on. We choose the Inverse Discrete Cosine Transform (IDCT) and Motion Compensation (MC) as two parallel threads, and reorganized the MPEG2 decoder as shown in Figure 6. The decoder starts its execution on an 8x4 ADRES, executes the Variable Length Decoding (VLD) and Inverse Quantization (IQ), and switches to the threading mode. When the thread execution starts, the 8x4 ADRES splits into two 4x4 ADRES arrays and continues on executing the threads. When both threads are finished, the two 4x4 arrays unify and continue on executing the add block function. We reorganized the MPEG2 program as described in Figure 4, and added “split” and “unify” instructions as intrinsics. These instructions currently do nothing by themselves, and are only used to mark where the thread mode should change in the MPEG2’s binary code. These marks are used by the split-control unit at run time for enabling/disabling the thread-mode program execution.

The current dual-threading compilation flow is shown in Figure 7. The lack of partition-based scheduling forces us to use two architectures as the input to the scheduling. The 8x4 architecture is carefully designed so that the left and the right halves are exactly the same. This architecture is the execution platform of

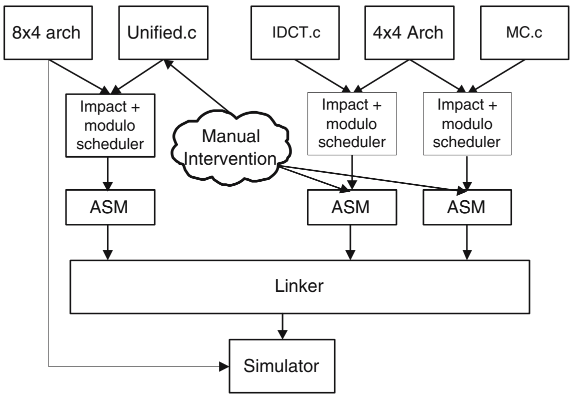


Fig. 7. Experimental Dual-threading compilation flow

the whole MPEG2 binary. We also need a 4x4 architecture, which is a helping architecture that is compatible to either half of the 8x4 array. This architecture is used as a half-array partition description of the 8x4 architecture. With these two architectures in place, we compile the single-threaded C-file and the threads on the 8x4 architecture and the 4x4 architecture, respectively. The later linking stitches the binaries from different parts of the program seamlessly.

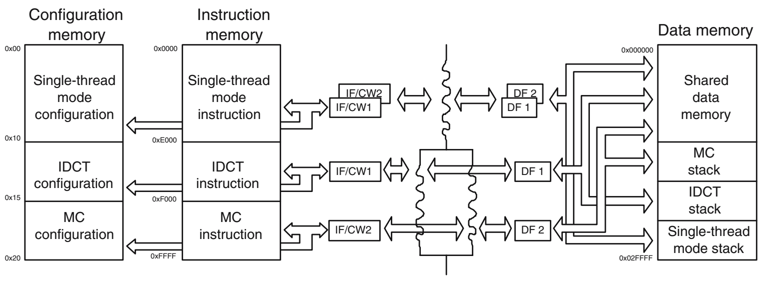


Fig. 8. Dual-threading memory management

3.2 Memory and Register File Design

The memory partitioning of the threaded MPEG2 is shown in Figure 8. The instruction fetching (IF), data fetching (DF) and the configuration-word fetching (CW) has been duplicated for dual-threading. The fetching unit pairs are step-locked during single-threaded program execution. When the architecture goes into the dual-threading mode, the fetching unit pairs split up into two sets, each of which is controlled by the controller in a thread partition.

During the linking, the instruction memory and data memory are divided into partitions. Both the instruction and configuration memory are divided into three partitions. These three partition pairs store the instructions and configurations of single-threaded binaries, IDCT binaries and MC binaries, as shown on Figure 8. The data memory is divided into four partitions. The largest data memory partition is the shared global static data memory. Both single-threaded and dual-threaded program store their data into the same memory partition. The rest of the data memory is divided into three stacks. The IDCT thread's stack grows directly above the single-threaded program's stack, since they uses the same physical controller and stack pointer. The base stack address of the MC thread is offset to a free memory location at linking time. When the program execution goes into dual-threading mode, the MC stack pointer is properly initialized at the cost of several clock cycles.

In the future, we aim at partitioning the clustered register file among the array partitions so that each thread has it's own register file(s). However, due to the lack of a partitioning-based register allocation algorithm at the current stage, the partitioning approach is not very feasible. We experiment on the ADRES architecture with a single global register file and go for the duplication based approach to temporary accommodate the register file issue. As shown in Figure 9, a shadow register file has been added into the architecture. When the single-threaded program is being executed, the shadow register file is step-locked with the primary register file. When the program initiate the dual-thread execution, the MC thread gets access to the shadow register file and continues the execution on the array partition and shadow register file. When the program resume to the single threaded execution, the shadow register file become hidden again. The MPEG2 program is slightly modified so that all the data being shared between threads and all the live-in and live-out variables are passed through the global data memory.

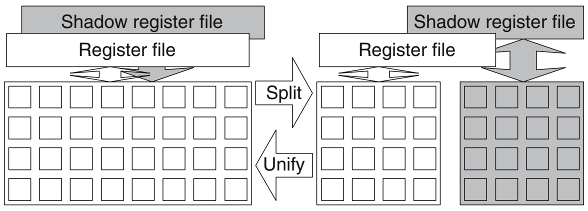


Fig. 9. Shadow Register file setup

3.3 Control Mechanism Design

The scalable control concept in Figure 3 has been verified in our simulation model. By having our control scheme tested on the dual-threading, we are positive that this scheme can be extended to a certain scale, and the control unit simulation model generation can be automated.

During the program linking, we identify where the "split" and "unify" instructions are stored in the instruction memory. These instructions' physical addresses mark the beginning and the ending point of the dual-threading mode. During the simulation model generation, these instructions' addresses are stored in a set of special-purpose registers in a split-control unit. After the program starts executing, the program counter's (PC) values are checked by the the split-control unit in each clock cycle. When the program counter reach the split point, the split-control unit sends control signals to the merger and bypasser to enable the threading mode. After the program goes into the threaded mode, the split-controller waits for both threads to join in by reaching the PC value where the "unify" instructions are stored. The first thread that joins in will be halt till the other thread finish. When the second thread eventually joins in, the split-control unit switch the ADRES array back to single-threaded mode, and the architecture resumes to the 8x4 array mode. The overhead of performing split and unify operation mainly comes from executing several bookkeeping instructions on some special-purpose registers, and such overhead is negligible.

When an application gets more complicated and has multiple splitting/unifying point, the current approach will become more difficult to manage, thus the future architecture will only rely on the instruction decoding to detect the "split" and "unify" instructions. The split-control unit will be removed in the future, and part of its function will be moved into each partition's local controller.

3.4 Simulation Result

The simulation result shows that the threaded MPEG2 produces the correct image frame at a slightly faster rate. Table 1 shows the clock count of the first 5 image frames decoded on the same 8x4 ADRES instance with and without threading. The **cc count** column shows the clock count of the overall execution time when an image frame is decoded, while the **decoding time** column shows the clock count between two frames are decoded. The dual-threaded MPEG2 is about 12-15% faster than the single-thread MPEG2 for the following reasons.

Both IDCT and MC algorithm have high loop-level parallelism, thus can optimally utilize the single-threaded 8X4 architecture. When scheduled on the 4x4 architecture as threads, the IPCs of both algorithms are reduced by half due to the halved array size, thus the overall IPCs of the non-threaded and the threaded MPEG2 are nearly the same. As mentioned earlier, when the ADRES' size is increased to certain extend, the scheduling algorithm has difficulty exploring parallelism in the applications and using the ADRES array optimally. It is clear that doubling/quadrupling the size of the ADRES array or choosing low-parallelism algorithm for threading will result in more speed-up. This will be proved with our future work on AVC decoder, which often has less than 50% of utilization rate of the full 8X4 processor.

As we have observed, the marginal performance gain is mostly achieved from the ease of modulo-scheduling on the smaller architecture. When an application is scheduled on a larger CGA, many redundant instructions are added into the kernel for routing purpose. Now the IDCT and MC kernels are scheduled on a half-CGA partition instead of the whole ADRES, even if the overall IPC of the application is not improved much, the amount of redundant instructions added during scheduling for placement and routing purpose has been greatly reduced.

Table 1. Clock cycle count of single and dual threaded MPEG2 on the same architecture

frame number	single-thread cc count	dual-thread cc count	single-thread decoding time	dual-thread decoding time	speed-up
1	1874009	1802277			
2	2453279	2293927	579270	491650	15.1%
3	3113150	2874078	659871	580151	12.1%
4	3702269	3374421	589119	500343	15.1%
5	4278995	3861978	576726	487557	15.5%

4 Conclusions and Future Work

By carrying out the dual-threading experiment on MPEG2 decoding algorithm, we have gained ample knowledge on the MT-ADRES architecture. The simulation results shows that the MPEG2 has gain 12-15% of speed up. We are convinced that more speed can be gained in future benchmarking, when full tool support permits the optimization of the template and the software architecture. The results so far demonstrate that our threading approach is adequate for the ADRES architecture, is practically feasible, and can be scaled to a certain extend. So far, the only extra hardware cost added onto ADRES is a second control unit, the size of which can be neglected for an ADRES larger than 3X3. Based on the success of our proof-of-concept experiments, we have very positive view on the future of MT-ADRES.

In the near future, we will implement the partition-based scheduling as the first priority. This will make the process of creating partitions more flexible. The resource-constrained register allocation algorithm, multiple stack and multiple special-purpose register supports are another group of urgently needed compiler support. With these supports, many application modification will no longer be necessary, and the nonrecurrent work will be reduced to minimum. And finally, better synchronization mechanism will be supported by ADRES architecture. We will evaluate the next generation MT-ADRES on more multimedia applications and study the impact on their performance.

References

1. B. Mei *A Coarse-grained Reconfigurable Architecture Template and its Compilation Techniques*, Ph.D. thesis, IMEC, Belgium.
2. B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, International Conference on Field Programmable Technology, Hong Kong, December 2002, pages 166-173.

3. B. Mei, S. Vernalde, D. Verkest, H. D. Man and R. Lauwereins *DRESC: A Retargetable Compiler for Coarse-Grained Reconfigurable Architectures*, FPL 2003
4. G.M. Amdahl *Validity of the single processor approach to achieve large-scale computing capabilities*, Proc. AFIPS Spring Joint Computer Conf. 30, 1967 Page(s):483-485
5. Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, Dean M. Tullsen *SIMULTANEOUS MULTITHREADING: A Platform for Next-Generation Processors*, Micro, IEEE Volume 17, Issue 5, Sept.-Oct. 1997 Page(s): 12 - 19
6. H. Akkary, M.A. Driscoll *A dynamic multithreading processor* ,31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998. MICRO-31. Proceedings. 30 Nov.-2 Dec. 1998 Page(s):226 - 236
7. E. Ozer, T.M. Conte *High-performance and low-cost dual-thread VLIW processor using Weld architecture paradigm* , IEEE Transactions on Parallel and Distributed Systems, Volume 16, Issue 12, Dec. 2005 Page(s):1132 - 1142
8. D. Burger, S. W. Keckler, K. S. McKinley, et al. *Scaling to the end of Silicon with EDGE architectures* IEEE Computer (37), 7 2004, Page(s)44 - 55
9. J. Zalamea, J. Llosa, E. Ayguade, M. Valero *Hierarchical clustered register file organization for VLIW processors*, International Parallel and Distributed Processing Symposium, 2003. Proceedings. 22-26 April 2003 Page(s):10 pp.
10. A. Capitanio, N. Dutt, and A. Nicolau. *Partitioned register files for VLIWs: A preliminary analysis of tradeoffs.*, The 25th Annual International Symposium on Microarchitecture, pages 103–114, Dec. 1992.
11. E. Iwata, K Olukotun *Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm*, Stanford University Computer Systems Lab Technical Report CSL-TR-98-771, September 1998.
12. S. Uhrig, S. Maier, G. Kuzmanov, T. Ungerer *Coupling of a Reconfigurable Architecture and a Multithreaded Processor Core with Integrated Real-Time Scheduling*, International Parallel and Distributed Processing Symposium. IPDPS 2006. 25-29 April 2006 Page(s):4 pp
13. R. Dimond, O. Mencer and W. Luk *CUSTARD - a customisable threaded FPGA soft processor and tools* , International Conference on Field Programmable Logic and Applications, 24-26 Aug. 2005 Page(s):1 - 6
14. B. Mei , S. Kim, R. Pasko, *A new multi-bank memory organization to reduce bank conflicts in coarse-grained reconfigurable architectures*, IMEC, Technical report, 2006

Asynchronous ARM Processor Employing an Adaptive Pipeline Architecture

Je-Hoon Lee, Seung-Sook Lee, and Kyoung-Rok Cho

CCNS Lab., San12, Gaeshin-dong, Cheongju, Chungbuk, Korea
{leejh, sslee}@hbt.cbnu.ac.kr, krcho@cbu.ac.kr

Abstract. This paper presented an asynchronous ARM processor employing adaptive pipeline and enhanced control schemes. This adaptive pipeline employed stage-skipping and stage-combining. The stage-skipping removed the redundant stage operations, bubbles. The stage-combining was used to unify the neighboring stage when the next stage is idle. Each stage of our implementation had several different datapaths according to the kind of instruction. The instructions in the same pipeline stage could be executed in parallel when they need different datapaths. The outputs obtained from the different datapaths were merged before the WB stage, by the asynchronous reorder buffer. We designed an ARM processor using a 0.35-μm CMOS standard cell library. In the simulation results, the processor showed approximately 2.8 times speed improvement than its asynchronous counterpart, AMULET3.

Keywords: Asynchronous design, adaptive pipeline, processor.

1 Introduction

Rapid advances in VLSI design technologies leads the trend of chip design to the platform-based SoC design, which reduces the design cost. Most SoC platforms include more than one embedded processor to perform many complex operations at high speed; thereby it results in higher power consumption. It is a design objective to reduce the power consumed by the processor. Also, most processors pursuing high performance employ two fundamental parallel concepts at the instruction level: pipelining and replication. Both of them can be applied simultaneously. Pipelining boosts performance by reducing the input latency of each stage. The replication of functional units is a more natural way to perform the allocated workload in parallel, even though it needs multiple execution units. The pipelining is tuned in a way that each stage performs its workload in the same period of time. However, it is difficult to keep the balance for all stages. In addition, the replication of functional units inevitably increases the H/W size in proportion to the number of replication.

The main advantage of asynchronous pipeline is their elasticity, where any pipeline stage can vary its processing time without any timing restriction [1]. In a pipeline system, each stage of the pipeline may take a different time to implement its workload. In conventional synchronous pipeline architecture, all stages are completed within a time slot given by the global clock, whose length is set to the slowest time to complete its operation; thereby it yields the worst performance. In contrast, the asynchronous pipeline uses the handshaking protocol for evaluation of the next stage. It operates in a flexible manner. This operation style yields an average-case performance. Another advantage of an asynchronous pipeline is that it only activates the stage requiring work. This feature is very similar to what one obtains quite naturally with asynchronous designs, in terms of not activating parts of the pipeline that are not requested. Some functional blocks which have no work to accomplish are idle in an asynchronous system and will not waste energy. Efthymiou presented a new class of latch controllers [2]. This system can be dynamically configured as collapsed, on a per-request basis, thus changing the structure of an asynchronous pipeline.

Many research results indicate that asynchronous design becomes one of the promising design features yielding lower power consumption. At present, the many implementations have been evaluated, such as the asynchronous ARM processor, AMULET3 and 32-bit processor TITAC-2 at the Tokyo Institute of Technology [3–6]. Because RISC architecture has easily adapted pipelining, most asynchronous processors utilize RISC architecture. However, various asynchronous CISC processors exist, such as Lutonium and A8051 [7–8].

This paper gives a high-level view of the architectural innovations, in terms of asynchronous pipeline organization. The asynchronous design can result in adaptive pipeline, owing to the data-dependent control mechanism based on a handshaking protocol. The proposed pipeline architecture allows stage-skipping and stage-combining. The stage-skipping skips the pipeline stages that are not requested. The stage-combining is to unify the neighboring stages when the next stage is empty. This paper presents the evaluation of the asynchronous multiprocessing scheme by splitting the datapath. It divides a whole datapath of the processor to several independent datapaths according to the kind of instructions. If the datapaths are different, some instructions can be executed in parallel without increasing hardware size. We evaluate the ARM9 processor to prove the efficiency of the proposed execution scheme.

This paper is organized as follows. Section 2 describes the proposed adaptive pipeline architecture. Section 3 introduces the proposed asynchronous multiprocessing scheme. Section 4 shows the evaluation of an asynchronous ARM processor employing the proposed control mechanism. Section 5 shows the simulation results. Finally, concluding remarks are presented in section 6.

2 Proposed Adaptive Pipeline Architecture

The proposed adaptive pipeline architecture employs stage-skipping and stage-combining. These schemes support the flexibility of asynchronous pipelining, so that the pipeline stages can be merged and skipped according to the instruction. This enhances processor performance and reduces power dissipation.

The functional block is connected by a pipeline latch as shown in Fig. 1. It is difficult to move these latches after the architecture is fixed, even though

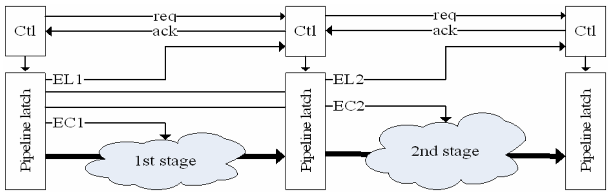


Fig. 1. Proposed pipeline architecture

the bubble stages occur. Note that the bubble stages are caused by preserving the instruction cycle as a multiple of a machine cycle. This paper proposes the adaptive pipeline structure in an asynchronous processor employing stage-skipping and stage-combining. This avoids performance scratching from these bubble operations. We add extra inputs, EL_N and EC_N, to the latch controller and pipeline stage, respectively. These signals are bundled with the latched data, and effect the pipeline stages when the request is sent through to the next stage. EL_N signals determine whether the N_{th} latch becomes transparent. EC_N signals determine whether the operation of the N_{th} pipeline stage is skipped.

The processor can skip the bubbles using the stage-skipping. The ID stage propagates the micro-instruction and EC_N signals to the next stages. The decoder sets the EC_N signals because the decoder knows which stage is a bubble after instruction decoding. The stage-skipping eliminates this redundant stage and yields shorter machine cycles. It skips the operation of a bubble without executing. For example, the EC_1 signal determines that the 1st stage operation will be skipped in Fig. 2a. The combinational logic of the 1st pipeline stage is executed when the signal is '0'. On the contrary, this logic is not performed and the input passes the 1st stage when the EC_1 signal is '1'. Stage-combining joins two consecutive stages into one stage when the next stage is empty. The processor knows which stage can be merged with the neighboring stage after instruction decoding. For example, the latch between 1st and 2nd stages becomes transparent when the EL_1 signal is high, as shown in Fig. 2b. The latches become transparent when the downstream stage has finished its allocated job and is ready for the next stage and the upstream stage is empty. However, the pipeline works in normal mode when the EL_1 signal is low. Each EL_N signal is issued by the ID stage and then passed on from each stage to the next until they reach their destination latch controller, as shown in Fig. 1.

Figure 3a shows a timing diagram of the stage-skipping operation. There is a MUX in front of each stage. The EC_N signal determines whether the corresponding N_{th} stage is operated. The upstream stage of pipeline is considered as an invisible stage when EC_N is asserted high. This means the processing time of the pipeline is reduced as much as the time passing since the 1st stage. The time slots in Fig. 3a, a and b, present the delay times in normal mode and stage-skipping mode, respectively. It shows the reduction in processing time caused by the stage-skipping.

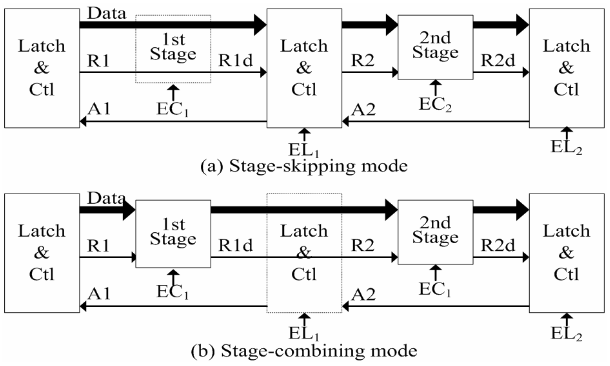


Fig. 2. Example of the pipeline operation: (a) stage-skipping, (b) stage-combining

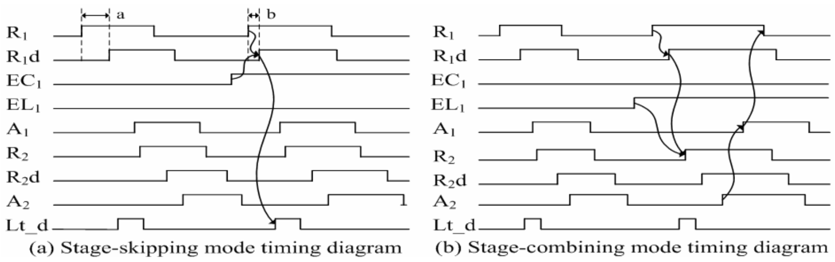

R1
R1d
EC1
EL1
A1
R2
R2d
A2
Lt_d
a
b
(a) Stage-skipping mode timing diagram
(b) Stage-combining mode timing diagram

Fig. 3. Timing of the pipeline operation: (a) stage-skipping , (b) stage-combining

The stage-combining mode makes the latch between the two stages transparent. A latch becomes transparent when the EL_N signal is high. Two consecutive pipeline stages are joined into one stage as shown in Fig. 3b. This shows that the signal wave form above the two stage pipeline with the latch can be transparent. The operation result of the first stage is passed to the next stage without saving data in the latch. A request signal, R1d is sent to 2nd stage directly. When the two stages joined by the EL_1 signal are high, the first stage is not acknowledged even though it has finished its operation.

3 Proposed Parallel Executing Architecture

Most processors employ pipelining and replication. The pipelining performs the required operation in parallel as shown in Fig 4a that maximize operation of the functional unit. On the other hand, the processors replicate of the functional units to exploit parallelism such as a superscalar and VLIW processor. Replicated functional units execute the same operation simultaneously as many replicated functional units as the processor has. This architecture inevitably increases their hardware size.

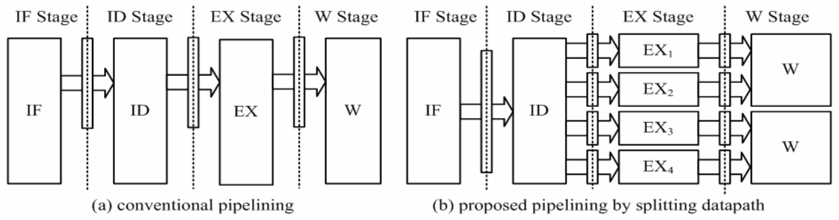


Fig. 4. The comparison of conventional and proposed pipeline

Every instruction does not pass through the same datapath. For example, *ADD R1 R2 R3* and *MOV M1 R2* have a different datapath. *ADD* instruction needs operand fetch from the register files, ALU execution, and saving the results to the register files. *MOV* instruction passes the data obtained from the register file to a memory unit. These instructions requiring the independent datapath and resources can execute in parallel without the replication of the functional units. The synchronous processor needs complex control to split the datapath and to execute instructions in parallel. It is more natural for the asynchronous processor to split the datapath and execute instructions simultaneously.

The proposed datapath splitting method makes instructions to be executed in parallel without the replication of functional units when they have no dependency as shown in Fig. 4b. This method is based on the data-dependent control provided by the asynchronous design. This elastic control scheme is to split the entire system to several datapaths according to the kind of instruction. The Petri-net diagram for the proposed pipelining in Fig. 4b is presented as shown in Fig. 5. ID stage is responsible for decoding instruction and removing the data dependency using an instruction windowing and the scoreboard method [9]. Both methods can be implemented similarly with those of the synchronous design. Instruction windowing is responsible for removing data dependency among the fetched instructions. Scoreboard scheme eliminates the data dependency between the fetched instructions and executed instructions in the upstream pipeline. ID stage issues the multiple instructions after the dependency is removed. Datapath splitting requires multiple joins and multiple forks. It can be resolved by the asynchronous handshaking protocol. The operation can be represented by Eq. 1. EX_1 and EX_2 are executed in parallel when they have not any dependency between them. The results obtained from EX_1 and EX_2 preserve the sequential consistency using the ROB (reorder-buffer) that is almost same with the synchronous one. The case for EX_3 and EX_4 is same with the above case and the results is saved the register files preserving the sequential consistency in W_2 stage. The result of each EX unit is saved into the register files after it assures sequential consistency of execution in the case of multiple EXs operating in parallel using ROB.

$$ParallelExecution = IF * ID * [(EX_1|EX_2) * W_1]|[(EX_3|EX_4) * W_2] \quad (1)$$

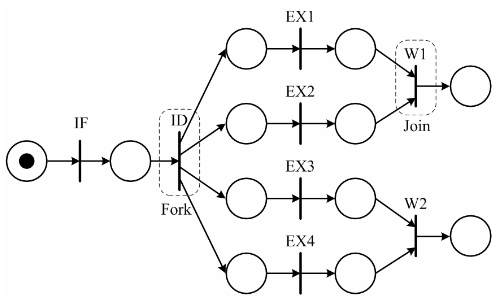


Fig. 5. Petri-net diagram for the proposed pipelining

4 Design of an ARM Compatible Asynchronous Processor

The proposed pipeline structure and multi-processing is adapted to an ARM9 instruction compatible processor to confirm the performance. The processor architecture is presented in Fig. 6. The processor consists of 8 pipeline stages; instruction fetch (IF), thumb decoder, instruction decode1 (ID1), instruction decode2 (ID2), operand fetch (OF), execute (EX), reorder buffer (RB) and write back (WB). The handshaking control blocks are between each stage.

There are two additional stages, ID2 and RB, in the proposed processor compared with ARM9. At ID2 stage, instructions which can be operated concurrently in the next stage are chosen and issued. ID2 includes instruction window and scoreboard to resolve the dependency problem. This processor allows the multi-processing feeding instructions into the pipeline when these instructions have different datapath after ID2 stage. The instruction window is responsible for identifying the available instructions executed in parallel among the fetched instructions. Dependency between instructions in instruction window and instructions which are being executed in upsteream pipeline are checked and removed by scoreboard block. If dependency is detected, the scoreboard schedules the instructions which have dependency to be operated in next sequence. In the RB stage, results of instructions which are operated out of order are rearranged to be saved to register file.

To adopt the proposed adaptive pipeline to processor, EL and EC signal in Fig. 6 is added to control the pipeline. In IF stage, pre-decoding is performed that classifies the fetched instructions into ARM instruction or thumb instruction. If an instruction is a thumb instruction and ID is empty, EL signal is asserted to high then it makes thumb decoder and ID1 to be combined. If an instruction is ARM instruction, EC signal is asserted high and thumb decoder operation is skipped. In micro-pipeline, like above cases, datapath is reduced using stage-skipping and stage-combining. For instance, barrel shifter is connected to ALU. If shifter is used and ALU is empty stage, EL signal is asserted high and stage is combined. If shifter is not used, EC signal is asserted high and stage-skipping is

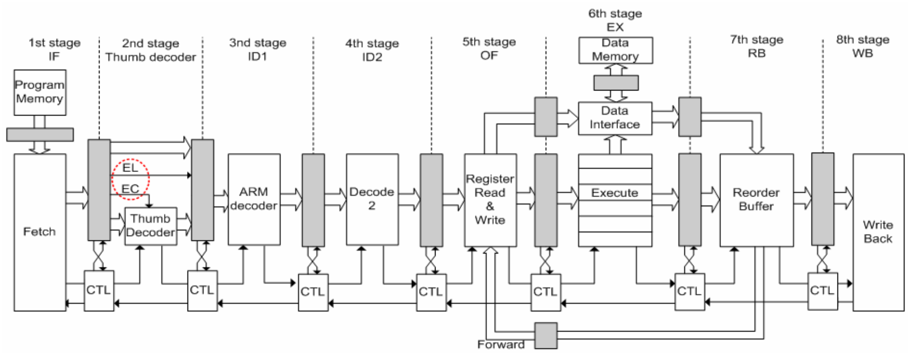


Fig. 6. The architecture of asynchronous ARM processor

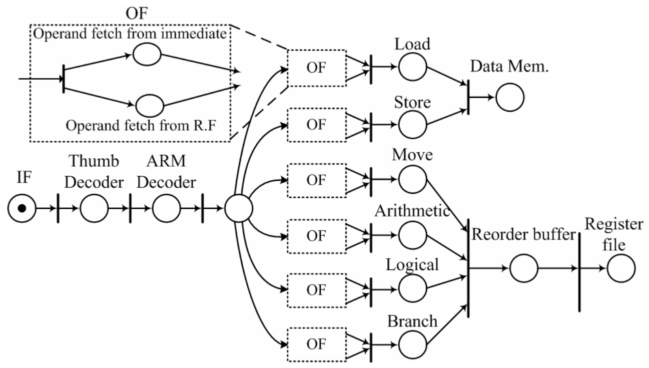


Fig. 7. Petri-net for the asynchronous ARM processor

operated. The proposed processor can be represented by Petri-nets as shown in Fig. 7. Datapath is divided by kind of instructions. Each instruction in different datapath can be operated by multi-processing. Also, OF stage is divided into two paths which are for operand from immediate value and that from register file. For example, in the case of instruction *MOV A, immediate*, there is not an operand from the register file, operation of operand fetch stage is unnecessary. This instruction is operated in order of *IF-Arm Decoder-MOV-RB-WB*. It skips the operation of Thumb decoder because it does not need that operation as shown in Fig. 7. Even though all instructions finish in an out-of-order fashion, overall instruction execution should mimic sequential execution using a reorder buffer that is operated as similar way of the synchronous one.

5 Simulation Results

We designed an ARM processor based on the proposed adaptive pipeline architecture using 0.35-μm CMOS technology. Fig. 8 shows the simulation results of

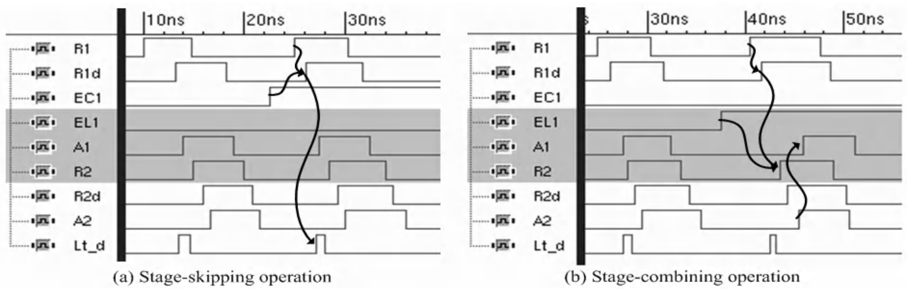


Fig. 8. Simulation results of adaptive pipeline architecture

Table 1. Average execution time of each stage

Stage	IF	ID		OF	EX	RB	RW	WB
		ID1	ID2					
Execution time(ns)	4.93	6.1	5.1	3.8	6.3	3.8	4.1	0.5

the stage skipping and stage combining between two consecutive pipeline stages in the Fig. 6. The stage-skipping occurs when the EC_1 signal transits to high as shown in Fig. 8a. The latch becomes transparent and two consecutive stages are combined when the signal EL_1 is *high*. These pipeline modes shorten the required datapath and enhance the processor performance.

Table 1 shows the maximum execution time of each pipeline stage of the asynchronous ARM processor. For efficient pipelining, each stage has to be almost same execution time. As shown in Table 1, the proposed architecture shows some execution time difference among the pipeline stages. The unbalancing of each stage makes complicate control. It also decreases the system performance.

To get the average execution time of each stage, we used 8 workloads from the SPEC2000 benchmark suite to measure the performance of a processor. Figure 9 shows the comparison results when the processor is operated in the non-pipeline mode, conventional pipeline, and the proposed pipeline mode including the datapath splitting. The non-pipelined mode shows 50 MIPS on average. The conventional pipeline mode shows performance of 161 MIPS. The proposed pipeline mode shows 365 MIPS on average. Thus, the proposed design is about 2.3 times faster than a conventional pipeline mode.

Figure 10 shows the parallel processing ratio of the total instructions for each benchmark program. The proposed processor shows 48% parallel processing ratio that means the processor performs 1.48 instructions simultaneously. It performs 64% instructions per machine cycle compared to that of the superscalar processor that executes 2.3 instructions in parallel. The throughput of AMULET3 is around 110 to 140 MIPS [4]. The performance of our implementation with a conventional pipeline is similar with that of the AMULET3. However, the

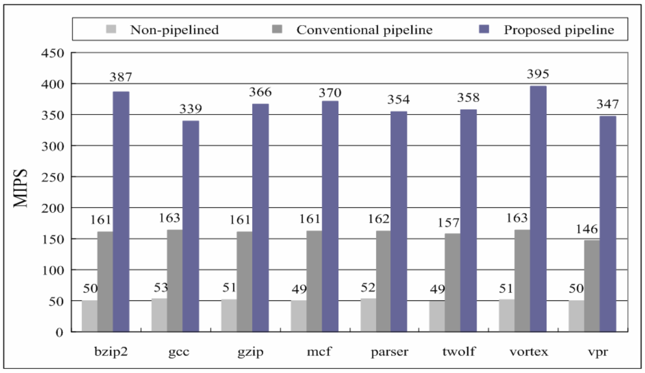


Fig. 9. Petri-net for the asynchronous ARM processor

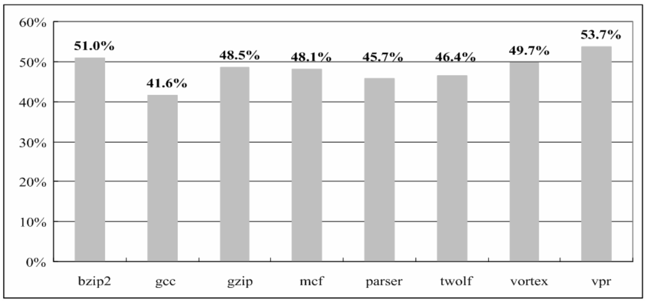


Fig. 10. Parallel processed instruction ratio

datapath splitting architecture accelerates instruction throughput extremely. It is 2.3 times higher comparing to the asynchronous counterpart.

6 Conclusion

This paper presents adaptive pipeline architecture for a low-power embedded asynchronous processor. The architecture employs stage-skipping and stage-combining scheme to make shorten length of the datapath. The proposed architecture also supports parallel processing during execution stage based on the datapath splitting. It enhances processor performance by multi-processing of the instructions without duplication of the functional unit. We synthesized an asynchronous ARM processor to prove the efficiency of the proposed pipeline architecture using a 0.35-μm CMOS standard cell library. As the results, the designed processor shows 48% parallel processing at EXE stage that is performing 1.48 instructions simultaneously. The instruction throughput is around 364 MIPS that is about 2.8 times higher speed than the asynchronous counterpart, AMULET3.

Acknowledgments. This work was supported by the Regional Research Centers Program of the Ministry of Education & Human Resources Development in Korea and Dr. J. H. Lee participated in this work is supported by the Second Phase of the Brain Korea 21 Project at Chungbuk National University.

References

1. J. M. Colmenar, O.Garnica, S. Lopez, J. I. Hidalgo, J. Lanchares, and R. Hermida, "Empiri-cal characterization of the latency of long asynchronous pipelines with data-dependent module delays," *Proc. 12th EUROMICRO-PDP'04*, pp. 311–321, Feb. 2004.
2. A. Efthymiou and J. D. Garside, "Adaptive pipeline structures for speculation control," *Proc. ASYNC'03*, pp. 46–55, May 2003.
3. M. Ozawa, M. Imai, Y. Ueno, H. Nakamura, and T. Nanya, "A cascade ALU architecture for asynchronous super-scalar processor," *IEICE Trans. Electron*, vol. E84–C, no. 2, pp. 229–237, Feb. 2001.
4. S. B. Furber, D. A. Edward, and J. D. Garside, "AMULET3: A 100 MIPS asynchronous embedded processor,"*Proc. Computer Design*, pp. 329–334, 2000.
5. A. Takamura, M, Kuwakao, M. Imai, T. Fujii, M. Ozawa, I. Fukasaku, Y. Ueno, and T. Nanya, "TITAC–2 A 32–bit scalable–delay insensitive microprocessor," *Proc. ICCD*, pp. 288–294, Oct. 1997.
6. S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver, "AMULET2e An asynchronous embedded controller," *Proc ASYNC'1997*, pp. 290–299, Apr. 1997.
7. A. J. Martin, M. Nystrom, K. Papadantonakis, P. I. Penzes, P. Prakash, C. G. Wong, J. Chang, K. S. Ko, B. Lee, E. Ou, J. Pugh, E. Talvala, J. T. Tong, and A. Tura, "The lutonium : sub–nanojoule asynchronous 8051 microcontroller," *Proc ASYNC'03*, pp. 14–23, 2003.
8. Je-Hoon Lee, YoungHwan Kim, and Kyoung-Rok Cho, "Design of a fast asynchronous embedded CISC microprocessor, A8051," *IEICE trans. on Electron*, vol. E87–C, no. 4, pp. 527–534. Apr. 2004.
9. D. Sima, T. Fountain, and P. Kacsuk, *Advanced Computer Architectures: a design space approach*, Addition–Wesley, 1997.

Partially Reconfigurable Point-to-Point Interconnects in Virtex-II Pro FPGAs

Jae Young Hur, Stephan Wong, and Stamatis Vassiliadis

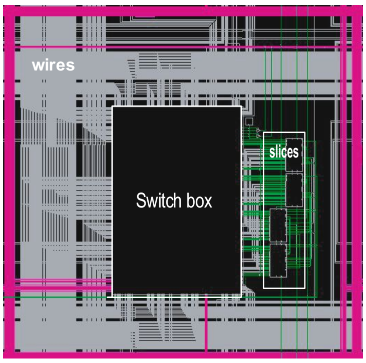
Computer Engineering Lab., TU Delft, The Netherlands
http://ce.et.tudelft.nl

Abstract. Conventional rigid router-based networks on chip incur certain overheads due to huge occupied logic resources and topology embedding, i.e., the mapping of a logical network topology to a physical one. In this paper, we present an implementation of partially reconfigurable point-to-point (ρ-P2P) interconnects in FPGA to overcome the mentioned overheads. In the presented implementation, arbitrary topologies are realized by changing the ρ-P2P interconnects. In our experiments, we considered parallel merge sort and Cannon's matrix multiplication to generate network traffic to evaluate our implementation. Furthermore, we have implemented a 2D-mesh packet switched network to serve as a reference to compare our results with. Our experiment shows that the utilization of on-demand ρ-P2P interconnects performs $2\times$ better and occupies 70% less area compared to the reference mesh network. Furthermore, the reconfiguration latency is significantly reduced using the Xilinx module-based partial reconfiguration technique. Finally, our experiments suggest that higher performance gains can be achieved as the problem size increases.

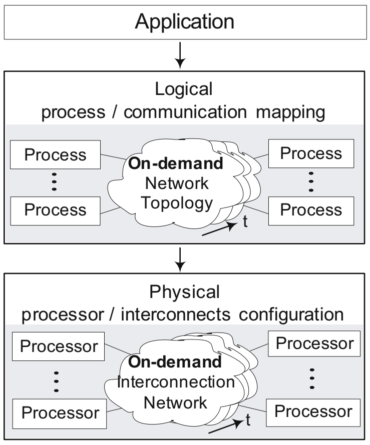
1 Introduction

In modern on-chip multi-core systems, the communication latency of the network interconnects is increasingly becoming a significant factor hampering performance. Consequently, network-on-chips (NoCs) as a design paradigm has been introduced to deal with such latencies and related issues. At the same time, NoCs provide improved scalability and an increased modularity[1][2][3][4][5]. However, these multi-core systems still incorporate rigid interconnection networks, i.e., mostly utilizing a 2D-mesh as the underlying physical network topology combined with packet routers. More specifically, the interconnection network will be fixed in the design stage leading to the modification of algorithms to suit the underlying topology or the embedding of the logical network (intended by the algorithm) onto the physical interconnection network. In both cases, reduced performance is the result. The topology embedding techniques are well-studied [6] and usually require the introduction of a router module to handle network dilations and congestions. Furthermore, worm-hole flow control for the packet switched network (PSN) is widely used due to its insensitivity to multi-hop delays. As a result, these systems that still utilize rigid network interconnects have the following limitations. First, the programmer must have intricate knowledge of the underlying physical network in order to fully exploit it. Second, performance is lost due to topology embedding that is likely to increase communication delays and/or create traffic congestions. Third, a router (with virtual channels) occupies significant on-chip resources.

Our work is motivated by several key observations. First, different applications (or algorithms) generate different traffic patterns that require different topologies to handle them in the best possible manner. Therefore, the ability to 'construct' topologies on-demand (at application start time or even during run-time) is likely to improve performance. Second, direct point-to-point connections eliminate the communications overheads of utilizing packet switched networks. Traditionally, P2P networks were not popular due to complex wiring and scalability issues. However, modern reconfigurable hardware fabrics contain rich intra-chip wiring resources and additionally provide a capability to change the interconnections (possibly) in run-time [7]. Figure 1(1) shows a Configurable Logic Block (CLB) cell in the Virtex-II Pro device indicating that the wiring resources occupy more than 70% of the cell. The Virtex-II Pro xc2vp30 device contains 3680 CLBs and abundant wires such as long, hex, double, and direct lines. Moreover, the interconnect wires do not require logic slices, which are valuable computational resources. Therefore, we aim to design and implement a (dynamically) adaptive and scalable interconnection network suited to meet demands of specific applications. Figure 1(2) depicts our general approach. Our system adaptively provides demanding interconnection networks that realize the traffic patterns. The reconfigurable interconnects consist of set of on-chip wiring resources. The presented ρ-P2P implementation combines the advantages of high performance of traditional P2P networks and the flexibility of reconfigurable interconnects.



(1) CLB cell in Virtex-II Pro FPGA
(snapshot from Xilinx FPGA Editor tool)



(2) Our design flow

Fig. 1. Our approach

In this work, we implement 2D-mesh PSNs and our ρ-P2P interconnects. We perform a comparative study to re-examine the performance gain and the area reduction of the ρ-P2P network over the PSN in modern reconfigurable platforms. We propose to utilize partial reconfiguration techniques to realize the ρ-P2P networks. The main contributions of this work are:

- An arbitrary topology can be rapidly configured by updating small-sized partial bitstreams, which is verified by an actual prototyping. The proof-of-concept experiment in Virtex-II Pro device using the Xilinx module-based partial reconfiguration technique shows that the actual system performance gain increases as the problem size increases.
- The experiments show that our ρ-P2P network performs $2\times$ better and occupies 70% less area compared to a mesh-based PSN for the considered applications.

The organization of this paper is as follows. In Section 2, related work is described. System designs and their hardware implementations are described in Sections 3 and 4. In Section 5, conclusions are drawn.

2 Related Work

The general concept of on-demand reconfigurable networks was introduced in [8]. In this paper, we present ρ-P2P networks to realize on-demand interconnects as an implementation methodology using modern FPGA technology. Recently, a number of NoCs have been proposed [1][2][3][4][5]. These networks employ rigid networks fixed at design time. Additionally, a packet router with virtual channels occupy significant on-chip logic resources. As an example, 16 routers with 4 virtual channels in [4] occupy 25481 slices, while a common device such as the xc2vp30 Virtex-II Pro contains in total only 13696 slices. Those networks route packets over multi-hop routers and consequently we consider a wormhole flow control based PSN as a reference. Our approach is different from the afore-mentioned networks in that the interconnection network consists of directly connected native on-chip wires. Our ρ-P2P network is different from the traditional point-to-point network, such as a systolic array, in that the physical interconnects of our work are reconfigurable. Our work is similar to [9] in that an on-chip packet processing overhead is discussed. While [9] considers a butterfly fat tree topology and does not discuss topology embedding, our work presents a network design to avoid packet processing overheads together with topology embedding. A partial reconfiguration technique for the NoC design is presented in [10]. While [10] presents dynamic insertion and removal of switches, our work presents configurations of physical interconnects. Reconfigurations of buses are presented in [11] and our work is similar to [11] in that the topology is dynamically changed. However, the buses in [11] are static as they use pre-established lines, we use only the required wiring resources.

3 Design and Implementation

3.1 2D-Mesh PSN

We consider a 2D-mesh PSN as a reference network as depicted in Figure 2(1). The PSN system consists of processing nodes which are interconnected in a rigid 2D-mesh topology. Each node is composed of a processing element (PE), a dual-ported memory, a network interface, and a router. Each node communicates utilizing a handshaking protocol. The packet is composed of 3 control flits and a variable amount of data flits as depicted in Figure 2(2). The first flit is a header containing the target node address. The second flit is the physical memory address of the remote node to store the transmitted data. The trailer indicates the last flit of a packet. Figure 2(3) depicts how a packet is processed. A flit-based worm-hole flow control is adopted for more efficient buffer utilization. When a packet arrives, the switch controller checks the packet header in the buffer. If the current node address and a target node address are different, the switch controller forwards the packet to the appropriate port. If current node address and a target node address are identical, the switch controller checks how many local memory ports are idle. The switch controller permits up to two packets to simultaneously access a local dual-ported memory. This feature is useful, provided that two-operand computations are common in many applications. Figure 2(4) depicts how a 2D-mesh node is organized. The network interface deals with packet generation, assembly, and memory management. The memory is private to the PE and the packet size is determined by the PE using trailer signal. In this way, direct memory access (DMA) of burst data transfers is supported. Each port entails full-duplex bidirectional links. A buffer is contained in each input port and the XY routing algorithm is utilized for its simplicity. A buffer accommodates 8 flits, while the individual flit width is 16 bit. The packets are arbitrated based on the round robin scheduling policy. The processors are wrapped inside a router in order for the dual-ported memory to simultaneously store the 2 incoming packet payloads. Consequently, the system conforms to the non-uniform memory access (NUMA) model. Our implementation is similar to [3][5] in that the same flow control and routing scheme are used. However, our PSN system in this work differs in the following ways. First, variable length burst packets are directly communicated between distributed memories. Second, dual-ported embedded memories are utilized to simultaneously accommodate two incoming packets. Third, a topology embedding (e.g., binary tree traffic embedding using [12]) has been implemented. A single packet containing N_l elements requires the following amount of cycles to conduct a source-to-destination communication:

$$L_l = N_l + \#hop \cdot L_h + P_{overhead}, \tag{1}$$

where L_l refers to the communication latency in number of cycles to transfer N_l elements. L_h denotes the header latency per router. The $P_{overhead}$ refers to the packetization overhead. $\#hop$ refers to the number of intermediate routers.

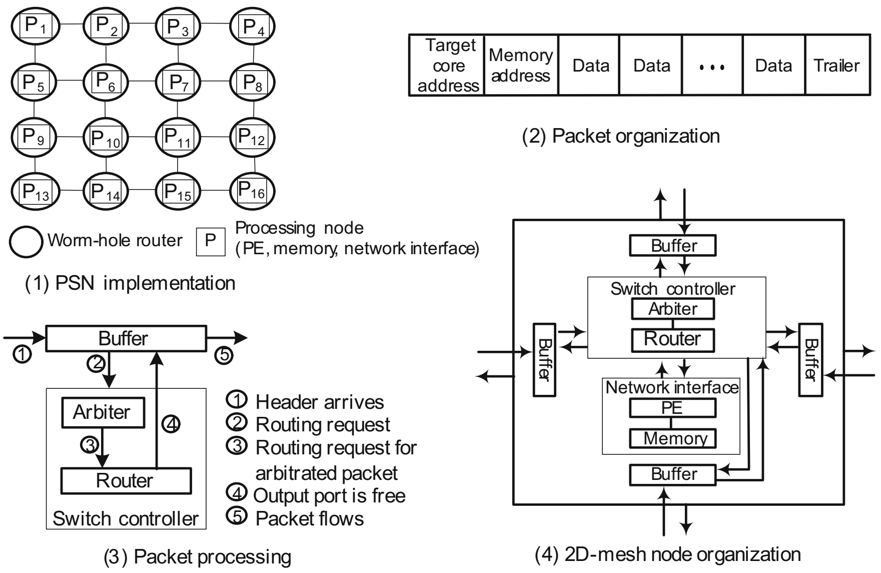


Fig. 2. The 2D-mesh PSN system organization

3.2 ρ-P2P Network

The proposed ρ-P2P system consists of directly interconnected PEs as depicted in Figure 3(1). Figure 3(2) shows the device floorplan. PEs are located in a static region and interconnects in the reconfigurable region are adaptively (re)configured on demand. The interconnects in the reconfigurable region correspond to the required wiring resources. The bus macro region separates the reconfigurable region from the static region and consists of symmetrical tri-state buffer arrays. The left-hand side belongs to the reconfigurable region and the right-hand side belongs to the static region. The interconnects are enabled or disabled by controlling the tri-state buffers. Two tri-state buffers constitute a single-bit wire line. The reconfigurable region and bus macros constitute the topology component. The topology component is modular and can be replaced by other topologies. In our experiments, the computational region remains fixed over time and only the communication region is reconfigurable. Those regions span whole vertical columns since the Virtex-II Pro device is configured in full column by column. When the required communication patterns change, the physical interconnects can be quickly changed. We exploit dynamic and partial reconfiguration techniques [13] of Xilinx FPGAs to adaptively configure ρ-P2P interconnects. Figure 3(3) depicts the exemplified reconfiguration steps, where the processors are initially interconnected in a 2D-mesh. After that, the on-demand topology is (re)configured by updating partial bitstreams only for the reconfigurable topology component during the respective time $t_2 - t_1, t_3 - t_2$. The network topology is implemented as a partially and dynamically reconfigurable component of the system. N_l elements requires N_l cycles to conduct a source-to-destination communication. This approach can be generally applied to the recent reconfigurable hardware, such as partially reconfigurable Xilinx Virtex series devices, though we target Virtex-II Pro in this work.

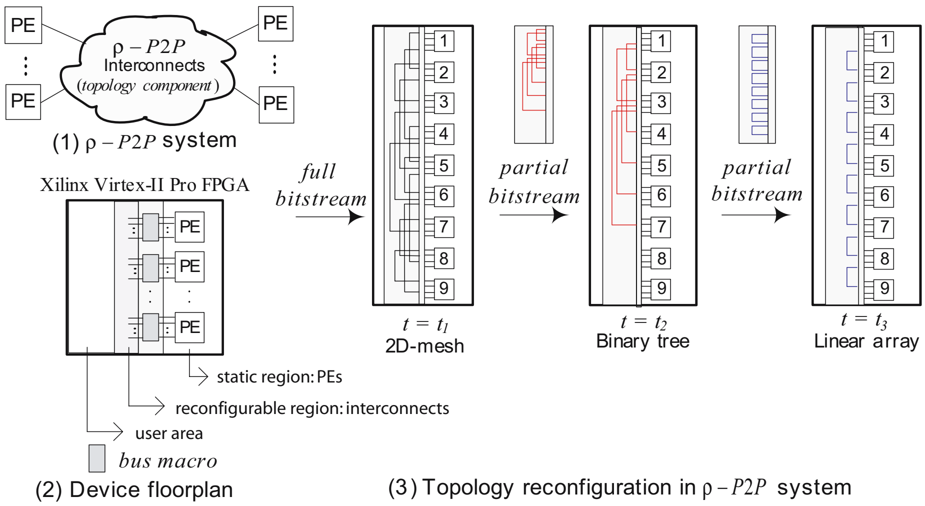


Fig. 3. The ρ-P2P system

The layout of static region is identical for each system configuration and remains unchanged during the interconnects reconfiguration. The small sized topology components can be reconfigured while processors are in operation within the static region. The reconfiguration latencies are directly proportional to corresponding partial bitstream sizes. The bitstream size is determined by required on-chip resources. In Virtex-II Pro, a bitstream is loaded in a column basis, with the smallest load unit being a configuration frame of the bitstream. Different network configurations are implemented in VHDL using the bus macros. Figure 4 depicts the design of a topology component using bus macro region as an example. Virtex-II Pro xc2vp30 contains abundant (6848) tri-state buffers which can be used for bus macro designs. As can be observed from Figure 4, the network topology can be implemented using tri-state buffer configurations. The interconnects do not require any logic resources such as slices, while the look-up tables (LUTs) in the bus macro regions are only required for power and ground signals of tri-state buffers. Since the data is communicated in a point-to-point fashion without packetization and multi-hop routing overheads, faster communication can be achieved compared to PSN. The area is also significantly reduced, provided that the interconnection network can be realized by configuring hard-wired bus-macros and using on-chip wires. Additionally, the partial bitstream is automatically generated using the Xilinx modular design flow.

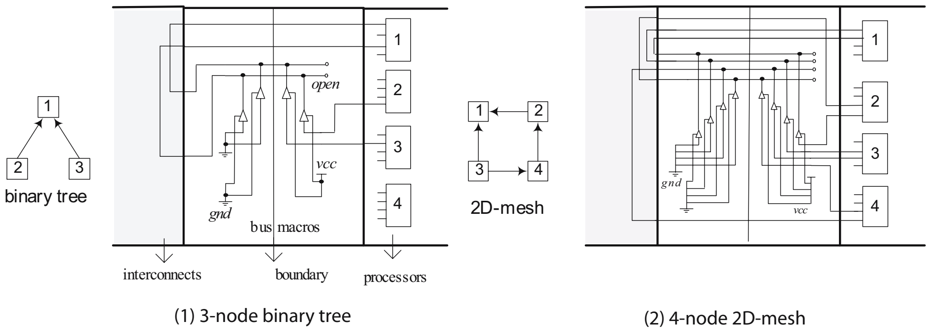


Fig. 4. The topology implementation using bus macros

3.3 Configuration Examples

In order to evaluate the presented network, we consider two types of workloads. The parallel merge sort (binary tree, burst traffic) and the Cannon's matrix multiplication (2D-torus, element-wise traffic) have been chosen as network traffics. Those algorithms are common and the implemented PEs are small in area such that larger networks can be realized in a single chip.

Parallel Merge Sort: The logical traffic pattern of the parallel merge sort is as follows. Firstly, sequential sort is performed at the bottom-most nodes. Secondly, the parent nodes sort elements taken from child nodes until the root node finishes the task. Sequential and parallel computation require $O(n\,\log n)$, $O(n)$ steps, respectively, and communication requires $O(n)$ steps, for the problem size n. Sequential PEs are identical for ρ-P2P and PSN systems. Consider p processors performing the parallel merge sort of N integers. System cycles are derived as shown in Equations (2a),(2b), where MS_PSN, MS_P2P refer to the number of cycles when the merge sort is operated in the PSN, ρ-P2P, respectively. In PSN system, the system cycles are calculated by (computation cycles) + (communication cycles). Consider a complete binary tree with p-processors, $\alpha_q \frac{2N}{p+1} \log \frac{2N}{p+1}$ cycles are required for a sequential sort. In PSN, when the sequential PE points to a remote memory address and commands a message transaction, the network interface generates a packet containing $\frac{2N}{p+1}$ elements. After that, each packet requires $(\beta_s \frac{2N}{p+1} + \#hop \cdot L_h + P_{overhead})$ cycles to move up to upper nodes. When each packet arrives, the network interface assembles the packets and stores the elements in local memory. Upper nodes perform a parallel sort, in which each element requires α_s cycle(s) for the parallel computation. $(\log_2(p+1) - 1)$ corresponds to the level of binary tree. In ρ-P2P system, $\alpha_p N$ cycles are required for the parallel computation for N total elements, in which α_p cycles are required to perform load, compare, forward operations for each element. $\beta_p N$ are required for communications. In case α_p and β_p are same, all the communication cycles $\beta_p N$ are completely hidden by the computation, since communication and computation can be overlapped.

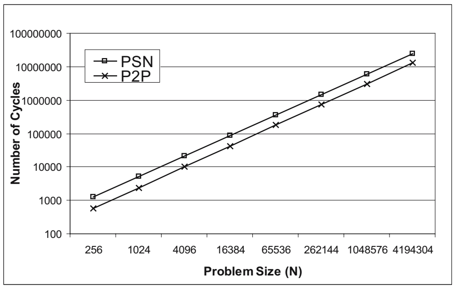
$$MS\_PSN = \alpha_q \frac{2N}{p+1}\log\frac{2N}{p+1} + 2\alpha_s N(1-\frac{2}{p+1}) + \beta_s N(1-\frac{2}{p+1})$$
$$+ (\log_2(p+1)-1)(\#hop \cdot L_h + P_{overhead}) \tag{2a}$$
$$MS\_P2P = \alpha_q \frac{2N}{p+1}\log\frac{2N}{p+1} + \alpha_p N \tag{2b}$$

Cannon's Matrix Multiplication: The logical 2D-torus traffic pattern of the Cannon's matrix multiplication is as follows. Firstly, a scalar multiplication is performed at each node and secondly, the intermediate result is transferred to left/upper node. These two steps are repeated until the task is finished. A sequential computation requires $O(m^3)$ steps for the matrix of size $m \times m$. Each PE is assumed to contain a single multiplier. Consider $\sqrt{p} \times \sqrt{p}$ processors and 2 symmetric matrices with size $M \times M$. System cycles are derived as shown in Equations (3a), (3b), where MM_PSN, MM_P2P refer to the number of cycles when the matrix multiplication is operated in the PSN, ρ-P2P, respectively. In ρ-P2P system, there are $\sqrt{p}$ phases of computations and $\frac{M^3}{p\sqrt{p}}$ computation cycles are required for each phase. PEs require α_p cycles to perform multiply, add, transfer operations for each element. The communication is performed between directly connected neighbor PEs, or #hop is 1. Totally $\alpha_p \frac{M^3}{p}$ cycles are required for the system cycles. In 2D-mesh PSN system, L_h, $P_{overhead}$ and worst case $\sqrt{p}$ hops for each packet are required. Additionally, the communication is directly performed between distributed memories.

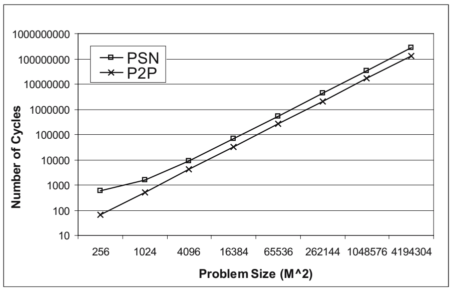
$$MM\_PSN = \alpha_q \frac{M^3}{p} + \sqrt{p}\,(\beta_s \frac{M^2}{p} + \#hop \cdot L_h + P_{overhead}) \tag{3a}$$
$$MM\_P2P = \alpha_p \frac{M^3}{p} \tag{3b}$$

4 Experimental Results

In this work, 3 experiments have been conducted. First, the system cycles are analytically derived from Equation (2) and (3) to measure the performance gain of $\rho-$P2P over PSN in the reconfigurable hardware. The coefficients are obtained from the highly optimized hardware implementations. For the parallel merge sort, $\alpha_q = 2.6$, $\alpha_s = 2.3$, $\beta_s = 1$, $L_h = 6$, $\alpha_p = 2$, $\beta_p = 2$ have been obtained. For the matrix multiplication, $\alpha_q = 2$, $\alpha_p = 1$, $\beta_s = 1$ are obtained. α_q is 2, since the PE requires 2 cycles to access a local memory, while α_p is 1, since the data is communicated directly between PEs. We can fairly compare $\rho-$P2P and PSN, since the computational latency is same for both systems. Figure 5(1) depicts the system cycles for different problem sizes. As can be observed, the $\rho-$P2P network performs on average 2$\times$ better (Note: graphs have a log scale). $\rho-$P2P is better in performance, since PSN suffers from a multi-hop communication latency, a packetization overhead while $\rho-$P2P provides a single-hop communication. Figure 5(2) shows the performance gain of $\rho-$P2P over PSN in terms of the execution time. As the problem size increases, the performance gain in the actual amount of cycles also increases. Assuming the systems operate at 100MHz, the performance gain is obtained by (PSN system cycles - P2P system cycles)$\times$10ns. Figure 5 depicts the performance gain that can grow up to 120 ms, 1347 ms for the merge sort, matrix multiplication, respectively. It can be noted that the Virtex-II Pro xc2vp30 device requires 26ms as a whole chip reconfiguration time [7]. The experiment suggests that partially reconfigured on-demand interconnects can be more beneficial for large problem sizes, since the reconfiguration latency is relatively smaller.

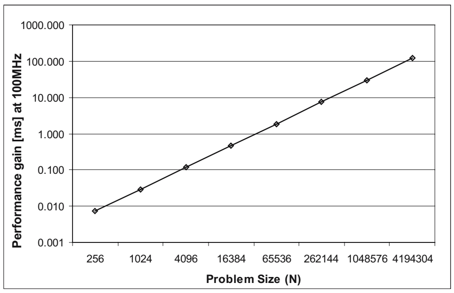


(a) Parallel merge sort

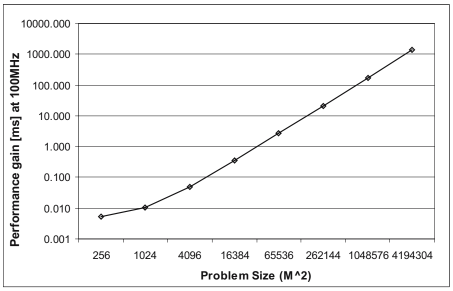


(b) Cannon's matrix multiplication

(1) System cycles



(a) Parallel merge sort



(b) Cannon's matrix multiplication

(2) Performance gain

Fig. 5. System cycles and performance gains of ρ-P2P and PSN

Second, PSN and $\rho-$P2P systems have been implemented in VHDL, placed and routed using the Xilinx ISE tool, in order to evaluate the system cycle models in Equations (2),(3). The systems have been implemented for N=512 (merge sort) and M=16 (matrix multiplication). Each PE has been implemented in an application-specific finite state machine, occupying 1% of area. Virtex-II Pro xc2vp100-6 has been used as a target device in order to experiment on larger networks. The implementation results for the merge sort are presented in Figure 6(1), in which network size, type of network, topology, number of nodes, system cycles, system area, clock frequency and system execution time are shown. The sequential merge sort requires 11981($\approx 2.6 \times 512 \times \log_2 512$) cycles in the implementation. To implement a PSN, the binary tree is embedded in 2D-mesh using the algorithm of [12]. The binary tree P2P system reduces on average 67% area and 37% of execution time compared to PSN. In Figure 6(1), PSN and $\rho-$P2P for the same network size can be fairly compared, since an actual number of PEs which participate in the computation is the same. Figure 6(2) shows the comparison of PSN and $\rho-$P2P for the matrix multiplication. The sequential matrix multiplication requires 12546($\approx 3 \times 16^3$) cycles. Embedded hardwired 18×18 bit multiplier, an adder and a simple control unit are implemented for each PE, which is identical for ρ-P2P and 2D-mesh PSN systems. 2D-torus P2P performs Cannon's matrix multiplication with a single-hop communication. Each PE performs an integer multiplication in a single cycle. In $\rho-$P2P, 94.6% of an execution time is reduced, since in PSN with 16×16 PEs, in the worst case 16 hops are required to transfer packets, while single cycle per each hop is required. Additionally, 82% of area is reduced, since complex router modules are eliminated.

Size	Network	Topology	#nodes	#cycles	Area		Max.	Execution time	
					#slices	reduction(%)	Freq. (MHz)	[us]	reduction(%)
	Sequential	-	1	11981	491	-	125.5	95.4	-
1	PSN	2D-mesh	4	6429	2821	60.9	117.3	54.8	9.0
	P2P	Binary tree	3	6154	1102		123.5	49.8	
2	PSN	2D-mesh	9	4276	6828	66.1	116.4	36.7	25.6
	P2P	Binary tree	7	3338	2316		122.1	27.3	
3	PSN	2D-mesh	16	3415	15533	68.8	113.3	30.1	40.9
	P2P	Binary tree	15	2063	4851		115.8	17.8	
4	PSN	2D-mesh	36	3094	33915	71.0	111.1	27.8	48.7
	P2P	Binary tree	31	1623	9852		113.7	14.3	
5	PSN	2D-mesh	64	2873	64057	69.1	105.5	27.2	58.9
	P2P	Binary tree	63	1247	19791		111.4	11.2	

(1) Merge sort (N=512)

Network	Topology	#nodes	#cycles	Area		Max.	Execution time	
				#slices	reduction(%)	Freq. (MHz)	[us]	reduction(%)
Sequential	-	1	12546	124	-	125.9	99.7	-
PSN	2D-mesh	256	300	44094	82.2	100.2	3.0	94.6
P2P	2D-torus	256	16	7837		99.4	0.2	

(2) Matrix multiplication (M=16, p=256)

Fig. 6. Implementation results

Third, intended as a proof-of-concept, the run-time reconfiguration of the interconnects has been realized on the Virtex-II Pro xc2vp30 in the Digilent XUP-V2P prototyping board[14]. Figure 7 demonstrates the procedure of the partial run-time reconfiguration. The 2D-mesh and the binary tree interconnects have been reconfigured using Xilinx module-based partial reconfiguration technique [13] in a boundary scan mode. Actual network interconnects in Figure 7(2) and (4) correspond to the topology components depicted in Figure 3. The small sized topology components can be reconfigured while processors are in operation. As an example, we reconfigure the binary tree interconnects by updating partial bitstream (Figure 7(4)). The layout of static region (Figure 7(1)) is identical for each system configuration and remains unchanged during the interconnects reconfiguration. In this experiment, the partial bitstream size in number of frames is 79 out of 1757, indicating that 4.4% of the reconfiguration latency is required compared to the full reconfiguration. It can be observed that the reconfiguration latency is significantly reduced by utilizing the partial bitstream. Furthermore, 162 LUTs (out of 27396) of logic resources and 240 TBUFs (out of 6848) primitives were used for bus macros. Regarding wiring resources, 231 switch boxes (out of 4592) are used for the binary tree partial bitstream. The experiment clearly shows that the occupied logic resources are significantly reduced by utilizing ρ-P2P interconnects.

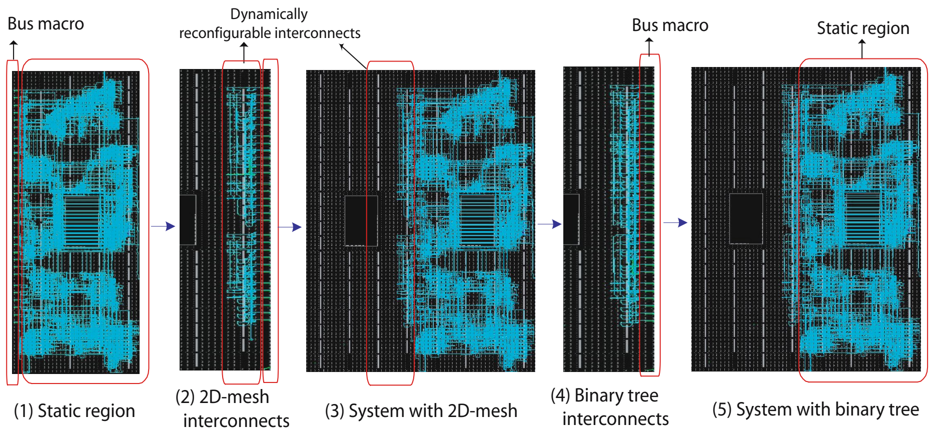

Bus macro
Dynamically reconfigurable interconnects
Bus macro
Static region
(1) Static region
(2) 2D-mesh interconnects
(3) System with 2D-mesh
(4) Binary tree interconnects
(5) System with binary tree

Fig. 7. Partial run-time reconfiguration

5 Conclusions

In this work, we proposed the partially reconfigurable interconnects on demand. Our ρ-P2P networks have been implemented and evaluated by comparing them with the 2D-mesh PSNs. We showed that the ρ-P2P network provides a better performance and reduced area by avoiding the use of complex routers. We also showed that the bitstream size is significantly reduced using partial reconfiguration of interconnects. Finally, our experiment projects adequate performance benefits to offset the reconfiguration latency as the problem size increases. Therefore, systems facilitating processors directly interconnected with the proposed reconfigurable interconnects can be suitable for our general approach.

Acknowledgement. This work was supported by the Dutch Science Foundation (STW) in the context of the Architecture, Programming, and Exploration of Networks-on-Chip based Embedded System Platforms (ARTEMISIA), project number LES.6389.

References

1. W. J. Dally and T. Brian, "Route Packets, Not Wires: On-Chip Interconnection Networks," Proceedings of 38th International Conference on Design Automation Conference (DAC'01), pp. 684–689, Jun 2001.
2. T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," ACM Computing Surveys, vol. 38, no. 1, pp. 1–51, Mar 2006.
3. F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an Infrastructure for Low Area Overhead Packet-switching Netwoks on Chip," Integration, the VLSI Journal, vol. 38, no. 1, pp. 69–93, Oct 2004.
4. A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC," Proceedings of 18th Symposium on Integrated Circuits and Systems Design (SBCCI'05), pp. 178–183, Sep 2005.
5. A. Mello, L. Möller, N. Calazans, and F. Moraes, "MultiNoC: A Multiprocessing System Enabled by a Network on Chip," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'05), pp. 234–239, Mar 2005.
6. F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes. Morgan Kaufmann Publishers, Inc., 1992.
7. Virtex-II Pro Handbook, http://www.xilinx.com.
8. S. Vassiliadis and I. Sourdis, "FLUX Networks: Interconnects on Demand," Proceedings of International Conference on Computer Systems Architectures Modelling and Simulation (IC-SAMOS'06), pp. 160–167, Jul 2006.
9. N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet Switched vs Time Multiplexed FPGA Overlay Networks," Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), pp. 205–216, Apr 2006.
10. T. Pionteck, R. Koch, and C. Albrecht, "Applying Partial Reconfiguration to Networks-on-Chips," Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL'06), pp. 155–160, Aug 2006.
11. M. Huebner, M. Ullmann, L. Braun, A. Klausmann, and J. Becker, "Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems," Proceedings of 14th International Conference on Field Programmable Logic and Applications (FPL'04), pp. 1037–1041, Aug 2004.
12. S.-K. Lee and H.-A. Choi, "Embedding of Complete Binary Tree into Meshes with Row-Column Routing," IEEE Transactions on Parallel and Distributed Systems, vol.7, no.5, pp. 493–497, May 1996.
13. "Two Flows for Partial Reconfiguration: Module Based or Difference Based," Xilinx Application Note, xapp 290, Sep 2004.
14. Digilent, Inc., http://www.digilentinc.com/XUPV2P.

Systematic Customization of On-Chip Crossbar Interconnects

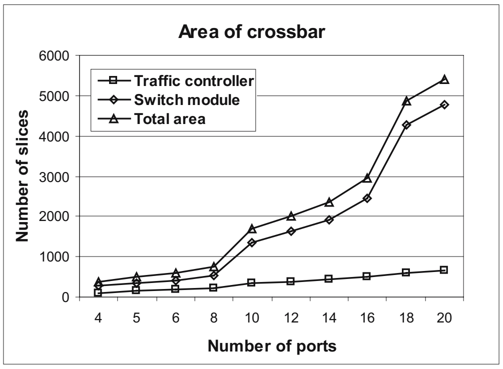
Jae Young Hur[1], Todor Stefanov[2], Stephan Wong[1], and Stamatis Vassiliadis[1]

[1] Computer Engineering Lab., TU Delft, The Netherlands
http://ce.et.tudelft.nl
[2] Leiden Embedded Research Center, Leiden University, The Netherlands
http://www.liacs.nl

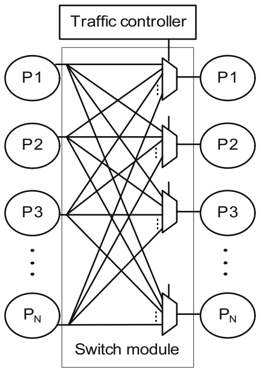
Abstract. In this paper, we present a systematic design and implementation of reconfigurable interconnects on demand. The proposed on-chip interconnection network provides identical physical topologies to logical topologies for given applications. The network has been implemented with parameterized switches, dynamically multiplexed by a traffic controller. Considering practical media applications, a multiprocessor system combined with the presented network has been integrated and prototyped in Virtex-II Pro FPGA using the ESPAM design environment. The experiment shows that the network realizes on-demand traffic patterns, occupies on average 59% less area, and maintains performance comparable with a conventional crossbar.

1 Introduction

A crossbar is widely used as an internet switch due to its non-blocking dedicated nature of communication and its simplicity, providing minimum network latency and minimum network congestion. It accommodates all possible connections, since traffic patterns are in most cases unknown. It is also widely used for networks-on-chip (NoC) as a basic building block [1]. Nevertheless, a major bottleneck of a conventional crossbar is the increasing amount of wires, as the number of ports grows. Figure 1 depicts the area of a conventional crossbar and a typical implementation. A crossbar consists of a traffic controller and a switch module. As the number of ports increases, the area of the switch module increases in a more unscalable way than the traffic controller. This work alleviates the scalability problem of a crossbar in a NoC-based reconfigurable platform. In general, communication patterns of different applications represent different logical topologies. The application performs best when the underlying physical interconnects are identical to the communication behavior of the parallel application. In modern NoC platforms, the logical topology information is available from the parallel application specification and the applications in most cases require only a small portion of all-to-all communications. Figure 2 depicts realistic applications [2], indicating that the required topologies are application-specific, much simpler than all-to-all topologies. Moreover, Figure 2 depicts that a single application can be specified differently (see MJPEG specifications).



Virtex-II Pro (xc2vp20-7-896)



Full crossbar interconnects

Fig. 1. A motivational example

Modern reconfigurable hardware can realize on-demand reconfigurability. Therefore, we designed a partial crossbar network, which dynamically adapts itself to traffic patterns, while using only the necessary wiring resources. In this work, we present a systematic design of fully customized crossbar interconnects. The presented network combines the high performance of a conventional crossbar and the reduced area of fully customized point-to-point interconnects for raw data communications. The main contributions of this work are:

- The developed network provides identical physical topologies to arbitrary logical topologies for an application.
- Experiments on realistic media applications show that on average 59% of area reduction is obtained compared to a full crossbar.
- The network component has been integrated in the ESPAM design environment [3][4].

The organization of this paper is as follows. In Section 2, related work is described. System designs including our network and their hardware implementation results are described in Sections 3 and 4. In Section 5, conclusions are drawn.

2 Related Work

The concept and our general approach of on-demand reconfigurable networks are described in [5]. In this paper, partial crossbar interconnects in an FPGA are proposed to implement an on-demand network. Our design proposed in this paper is integrated in and verified using the ESPAM tool chain [3][4]. In [3][4], a design methodology from sequential application specifications to multiprocessor system implementations prototyped on an FPGA board is presented and considers a full crossbar interconnection network. Since a full crossbar is not scalable in terms of area, we present a customized partial interconnection network.

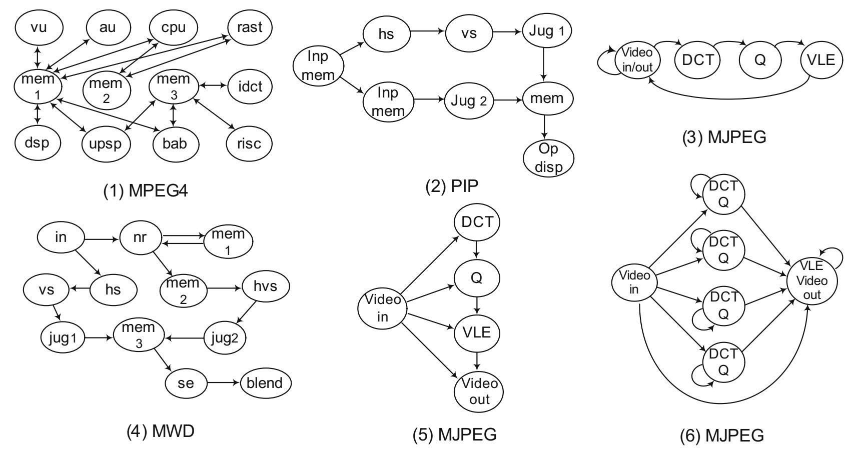


Fig. 2. Parallel specifications of practical applications

Numerous NoCs targeting ASICs (surveyed in [1]) employ rigid underlying networks and typically packet routers constitute tiled NoC architectures. In a typical tiled organization, one router is connected to a single processor (or IP core). Those packet routers accommodate a buffer at each port and internal full crossbar fabrics. NoCs targeting FPGAs [6]∼[11] employ fixed topologies defined at design-time. In these related work, the topology is defined by the interconnections between routers or switches, while the topology of our work is defined by the direct interconnection between processors. Figure 3 summarizes related work in terms of system sizes, clock frequencies, target devices, data widths, buffer depths, number of I/O ports, occupied network areas per switch (or router) and design characteristics of [6]∼[11]. Each of them entails specific design goals and different characteristics. As an example, [7] is designed for a partially reconfigurable module supported by the OS. [6]∼[8] adopt the packet switching and each router contains a full crossbar fabric. In [10], 10 switches are connected with 8 processors. In [11], 4 different topologies are presented and each circuit router is connected with multiple nodes. [9] presents a topology adaptive parameterized network component, which is a close approach to our work. While the physical topology in [9] is constructed between multi-ported packet routers, our adaptive topology is constructed between processors using partial crossbar interconnects. Our centralized partial crossbar constitutes a system network.

Recently, a couple of application-specific NoC designs are proposed. [12] presents a multi-hop router based network customization, while our network is single-hop based. [13][14] present an internal STbus crossbar customization and verified it through simulation. Our work is similar to [13][14] in that a crossbar is customized. In [13][14], each arbiter of the bus-based partial crossbar is connected to all processors, while in our work, a point-to-point direct link is established and different sized multiplexors are utilized. Additionally, our work is verified by actual prototyping. Finally, [15][16] present dynamic configurations of bus segments, while our work presents a full customization of NoC topologies.

NoC	Year	System #cores	Freq. MHz	Chip	Switch				
					DataWidth	BufferSize	I/O	Area (#slices, #BRAMs)	Characeristic
[6]	2004	2X2	25	V2-1000-4	8-bit	8 flits	5	316	worm-hole
[7]	2004	3X3	33	V2-6000	16-bit	BRAM	5	446, 5 BRAMs	virtual cut-through
[8]	2005	3X3	33	V2Pro30	8-bit	BRAM	5	352, 10 BRAMs	parallel routing
[9]	2005	3X3	50	V2Pro40	16-bit	BRAM	5	552, 5 BRAMs	flexible topology
[10]	2006	8	166	V2-6000-4	32-bit	no buffer	4	732 or 1464	time-multiplexed
[11]	2006	8	134	V2Pro30-7	16-bit	no buffer	8	1223, 1 BRAM	circuit switching

Fig. 3. Summary of related work

3 Design and Implementation

As mentioned earlier, our goal is to design reconfigurable interconnection networks, in which the physical topology is identical to the logical topology specified by the application partitioning. The physical interconnects are also required to be instantly switched to adaptively meet the dynamic traffic patterns. The logical topology is represented by a point-to-point graph, in which each node has possibly a different number of input and output links. In this work, a parameterized multiplexor array has been implemented for switch modules as a design technique. Topology-specific and different sized multiplexors ensure that demanding interconnects are established. Additionally, routing paths are selected by a crossbar traffic controller. The switch module is generic in terms of data widths, number of processors, and custom topologies. In this way, the network interconnects can be adapted to an arbitrarily specified logical topology and arbitrary number of processors.

3.1 Design Environment

The parameterized switch module has been integrated as a modular communication component in the ESPAM tool chain as depicted in Figure 4, in which the MJPEG data flow specification in Figure 2(3) is considered as an example. Details of the ESPAM design methodology can be found in [3][4]. In ESPAM, 3 input specifications are required, namely application / mapping / platform specification in XML. An application is specified as a Kahn Process Network (KPN). A KPN is a network of concurrent processes that communicate over unbounded FIFO channels and synchronize by a blocking read on an empty FIFO. The KPN is a suitable model of computation on top of the presented crossbar-based interconnection network, due to its point-to-point nature of communication and its simple synchronization scheme. However, the design technique of the presented network can be used in other systems. A KPN specification is automatically generated from a sequential Matlab program using the COMPAAN tool [17]. From the KPN specification, a task dependency graph is manually extracted.

Each process is assigned to a specific processor in the mapping specification. The number of processors, type of network and a port mapping information are specified in the platform specification. Figure 4 depicts how the customized interconnects can be implemented from the specified application. In the platform specification, four processors are port-mapped on a crossbar. From the mapping and platform specifications, port-mapped logical network topology is extracted as a static parameter and passed to ESPAM. Subsequently, ESPAM refines the abstract platform model to an elaborate parameterized RTL (hardware) and C/C++ (software) models, which are inputs of the commercial Xilinx Platform Studio (XPS) tool. Finally, the XPS tool generates the netlist with regard to the parameters passed from the input specifications and generates a bitstream for the FPGA prototype board to check the functionality and performance.

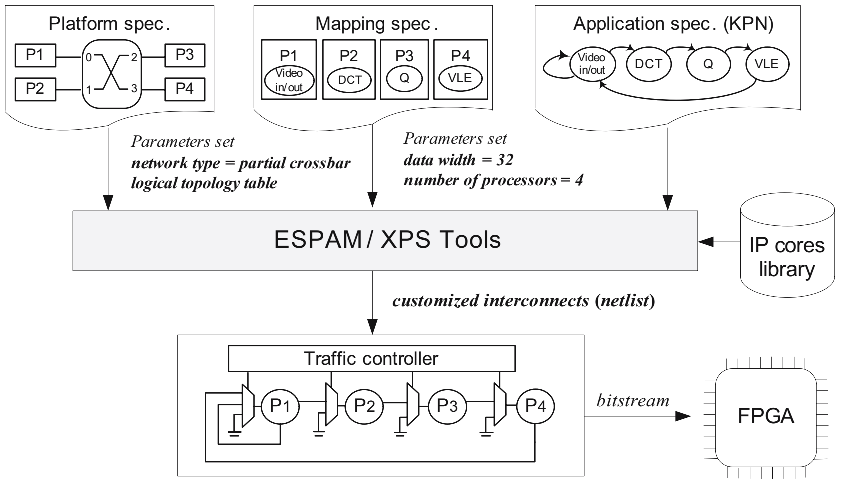


Fig. 4. An integration in the ESPAM design framework

3.2 Implementation of the Partial Crossbar

The switch module is customized with parameterized multiplexor arrays using a generic VHDL function. Our network has been implemented with the following steps:

1. Port mapping: Given an application and a mapping specification, topologically sorted processors are associated with crossbar ports in the platform specification.
2. Topology extracting: The topology information is extracted from the three input specifications.
3. Parameter passing: The extracted topology tables are passed as static parameters to the switch module. Multiplexor select signals are generated in the traffic controller and also passed as a dynamic parameter to the switch module.

Figure 5 depicts how actual interconnects are customized for the MJPEG application specification in Figure 2 (3). First, priorities are given to task-assigned processors based on the topological sort in the task graph. Figure 5(1) depicts the topological sort and linear ordering of processors. The dotted lines represent the levels of sorting. Considering *Video_in* node P1 as a root in the first level, a next priority is given to the directly connected processor in the second level. Following the precedence graph, these steps are repeated until the priority is given to all candidates. Consequently, the processors are linearly ordered in the following sequence: P1,P2,P3,P4. After that, all of these processors are associated with the crossbar ports. The round-robin scheduler in the traffic controller performs an arbitration of the requests during run time using the topology sort. Note that the processor without incoming links in the task graph does not request for data and the scheduler excludes those processors in the scheduling candidate lists. As an example, $Video\_in$ nodes in Figure 2(5) and (6) are not considered for the scheduling in the traffic controller.

Second, the graph topology is extracted from the three input specifications. Each processor port has a set of incoming and outgoing links, as depicted in Figure 5(3). Table 1 shows the number of incoming links and a list of ports from which the links are originated (for the task graph in Figure 2(3)). As an example, port P1 has two incoming links from ports P1 and P4, indicating that processor P1 reads the data located in the FIFOs connected to either P1 or P4. Table 2 shows the number of outgoing links and a list of ports to which the links are directed. As an example, P1 port has two outgoing links to ports P1 and P2, indicating that the data in FIFOs connected to P1 is transferred to either processor P1 or processor P2. The topology of the physical interconnection network can be efficiently constructed using customized multiplexor arrays. Table 1 and 2 are used to systematically implement customized multiplexor arrays instead of full multiplexors. Note that an N-port full crossbar contains N-way multiplexors per port, while our network contains variable-way multiplexors, depending on the graph topology. There are two types of multiplexors, namely processor-side multiplexors and FIFO-side multiplexors, as depicted in Figure 5(3). Table 1 is used to implement processor-side multiplexors controlled by $CTRL\_FIFO$ signals and Table 2 is used to implement FIFO-side multiplexors controlled by $CTRL\_PROC$ signals. A state diagram of the traffic controller which controls signals $CTRL\_FIFO$ and $CTRL\_PROC$ is depicted in Figure 5(2). The traffic controller deals with a handshaking protocol (i.e., data request, acknowledgement, data transfer) and a round-robin arbitration. The arrows indicate state transition conditions and the bold words indicate the actions in each state. The traffic controller checks whether there is a request using a circular round robin policy. The request signal contains a target port and a target FIFO index. In case there is a request, the request is registered and the traffic controller checks whether the target port is busy or idle. If the target port is idle and the designated FIFO contains data, $CTRL\_FIFO$ and $CTRL\_PROC$ signals are generated and multiplexor input links are selected by those two signals.

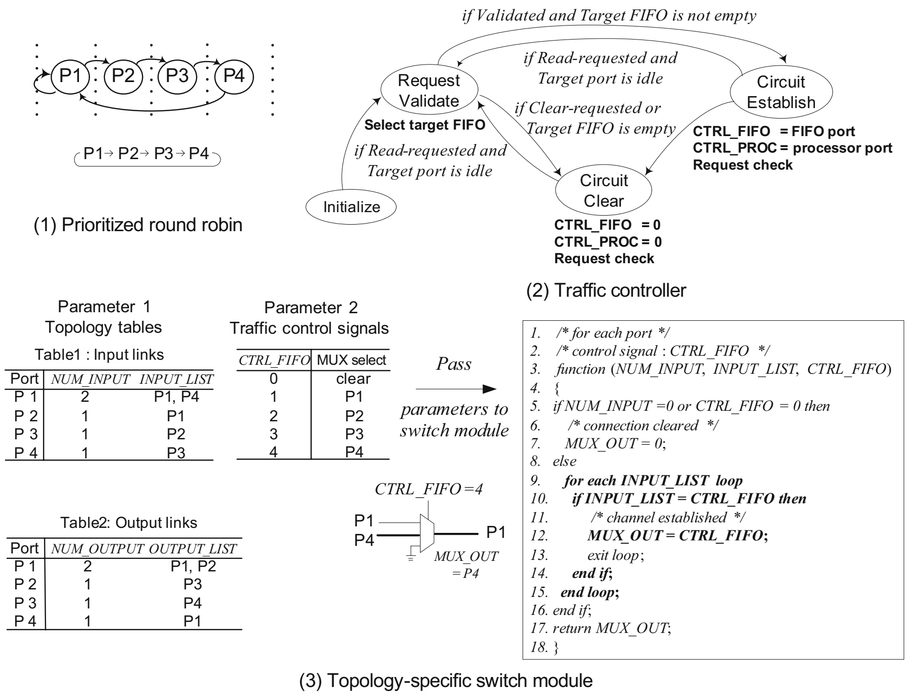


Table1 : Input links

Port	*NUM_INPUT*	*INPUT_LIST*
P 1	2	P1, P4
P 2	1	P1
P 3	1	P2
P 4	1	P3

CTRL_FIFO	MUX select
0	clear
1	P1
2	P2
3	P3
4	P4

Table2: Output links

Port	*NUM_OUTPUT*	*OUTPUT_LIST*
P 1	2	P1, P2
P 2	1	P3
P 3	1	P4
P 4	1	P1

```
1.  /* for each port */
2.  /* control signal : CTRL_FIFO */
3.  function (NUM_INPUT, INPUT_LIST, CTRL_FIFO)
4.  {
5.  if NUM_INPUT =0 or CTRL_FIFO = 0 then
6.     /* connection cleared */
7.     MUX_OUT = 0;
8.  else
9.     for each INPUT_LIST loop
10.      if INPUT_LIST = CTRL_FIFO then
11.         /* channel established */
12.         MUX_OUT = CTRL_FIFO;
13.         exit loop;
14.      end if;
15.    end loop;
16. end if;
17. return MUX_OUT;
18. }
```

Fig. 5. Parameterized switch module and traffic controller

Third, a VHDL function has been implemented to generate multiplexor arrays. The two parameters described above are passed to a VHDL function to actually establish a circuit link. As an example, processor P1 reads a data located in a FIFO connected to P4, with $CTRL\_FIFO = 4$ generated by the traffic controller, as depicted in Figure 5(3). The function to generate processor-side multiplexors has been implemented with a simple priority encoder as described in lines 9∼15 in Figure 5(3). The function for FIFO-side multiplexors can be implemented in the same way. Once the request is given the priority, 2 cycles are required to establish a circuit link to the designated target port. Once a link is established, a remote memory behaves as a local memory until the link is cleared.

The communication controller in [3][4] is also used in this work as a common network interface in order to integrate the presented network component in ESPAM. Figure 6(1) depicts that P1 connected to the crossbar port 0 reads from a remote memory connected to crossbar port 3 (represented by the bold line). The switch module requires three multiplexors per crossbar port as depicted in Figure 6(2). Multiplexors for other ports are not depicted for the sake of clarity. P1 sends a read-request to the traffic controller and the traffic controller sends the designated FIFO index to the communication controller connected

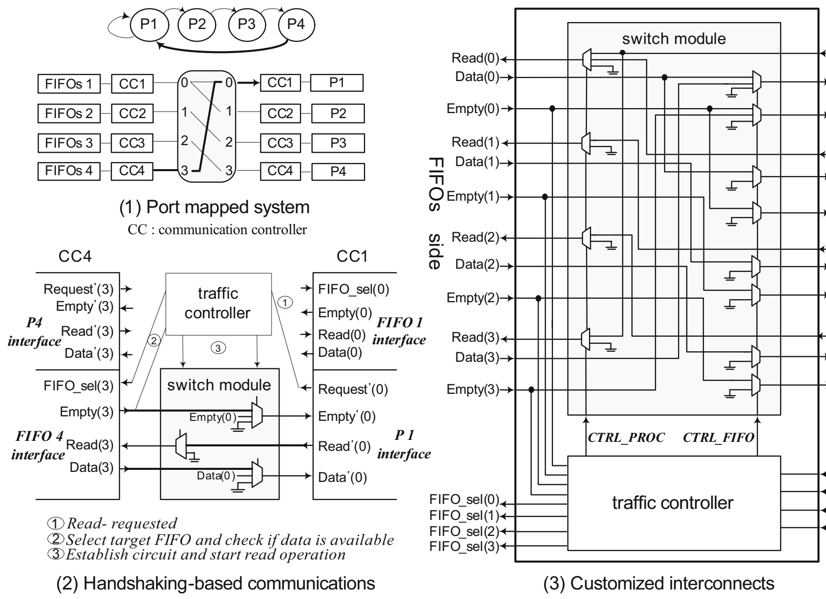


Fig. 6. Customized interconnection network

to P4. P4 responds to the traffic controller whether the FIFO is empty or not. If the FIFO is not empty, the traffic controller generates *CTRL_PROC* and *CTRL_FIFO* signals to establish a communication line. Figure 6(3) depicts the finally customized 4-port switch module, in which 12 multiplexors are instantiated. In general, a N-port switch module in our network contains $3\times N$ variable-way multiplexors. In our network, the data is communicated in a point-to-point manner without a packetization, without multi-hop header processing overheads, and without intermediate buffering overheads. Therefore, a low latency and a high throughput communication is achieved. The occupied network area depends on the logical topology of an application. Only in case the application requires an all-to-all topology, the network is identical to a full crossbar. Additionally, our network efficiently utilizes the available bandwidth, since only required links are established and the links are fully utilized.

4 Experimental Results

In this work, two experiments have been conducted for realistic applications. First, a full crossbar and the presented partial crossbar have been compared in terms of area utilization in order to measure the area reduction. The application task graphs of MPEG4, PIP, MWD are taken from [2]. The task graphs of H.263 encoding, MP3 encoding, and MMS are taken from [18]. The task graphs of 802.11 MAC, TCP checksum, VOPD are taken from [15],[19],[20], respectively.

Assuming each node is associated with a single crossbar port, the full and partial crossbar networks are synthesized, placed and routed using the Xilinx ISE tool on Virtex-II Pro (xc2vp20-7-896) FPGA and the areas have been obtained. Figure 7 depicts topologies, number of nodes, number of required links, area of the traffic controller, area of the switch module, area of the crossbar, chip occupation and area reduction in percentage. The crossbar area is the summation of the area of the traffic controller and switch module. As Figure 7 shows, the area of the network is highly dependent on the number of nodes and links. The partial crossbar network requires on average 61% less area, compared to the full crossbar. The same traffic controller is used for both cases. The area of our network is not only dependent on the number of nodes that determine its size but also on the network topology. It is observed that the higher area reduction is obtained as the network size increases. This is due to the fact that the average number of incoming and outgoing links per node does not increase as the number of nodes increases.

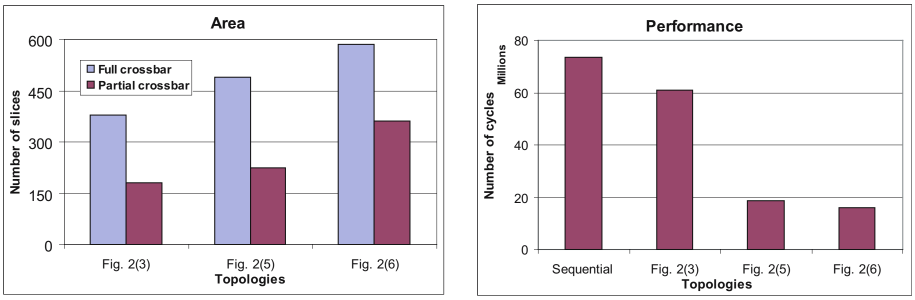
Topologies	#nodes	#links	Type	Area (#slices)			Chip area(%)	Reduction(%)
				Controller	Switch	Combined		
TCP Checksum	5	14	Full	148	340	488	5.3	40.4
			Partial	148	143	291	3.1	
MP3 enc	5	10	Full	148	340	488	5.3	47.7
			Partial	148	107	255	2.7	
H.263 enc	7	14	Full	196	476	672	7.2	49.7
			Partial	196	142	338	3.6	
PIP	8	8	Full	211	544	755	8.1	55.6
			Partial	211	124	335	3.6	
802.11 MAC	9	20	Full	315	1224	1539	16.6	64.5
			Partial	315	231	546	5.9	
MPEG4	12	26	Full	387	1632	2019	21.8	64.6
			Partial	387	328	715	7.7	
MWD	12	13	Full	387	1632	2019	21.8	70.9
			Partial	387	200	587	6.3	
VOPD	16	20	Full	493	2448	2941	31.7	73.3
			Partial	493	291	784	8.4	
MMS	25	47	Full	798	6800	7598	81.9	81.8
			Partial	798	587	1385	14.9	

Fig. 7. Experiments on topologies of practical applications

Second, actual systems have been implemented in order to measure the performance and the area of the presented network on a prototype board. We considered an MJPEG encoder application that operates on a single image with size 128 $\times$ 128 pixels. We experimented with different MJPEG task graph topologies. We used the ESPAM tool chain and prototyped a multiprocessor MJPEG system onto the ADM-XPL FPGA board[21]. Figure 8 depicts the experimental results. We have experimented with the three alternative task graphs in Figure 2(3),(5),(6), where our network provided the on-demand topologies. The implemented system is homogeneous in that each node contains a MicroBlaze processor. It can be observed that the topology plays an important role for the system performance. The partial crossbar network requires on average 48% less area, compared to the full crossbar. The system cycles decrease as the number of processors increase and the performance of the partial crossbar is comparable to the full crossbar. The operating clock frequency varies with the network configurations. As an example, the network component for Figure 2(3) is operated at 119MHz and the data width is 32-bit. Therefore, 3.8Gbps of bandwidth per link is available in the standalone network. The network component is not a bottleneck for the system performance, since the embedded block RAMs (used for FIFOs) operate at 100MHz.

Topology	from Fig. 2 (3)				from Fig. 2 (5)				from Fig 2 (6)			
#nodes	4				5				6			
#links	5				7				14			
Type	Performance	Area (#slices)			Performance	Area (#slices)			Performance	Area (#slices)		
	System cycles	Controller	Switch	Combined	System cycles	Controller	Switch	Combined	System cycles	Controller	Switch	Combined
Full	61104796	107	272	379	18580930	148	340	488	15940149	176	408	584
Partial	60862613	107	73	180	18580768	148	76	224	15940023	176	186	362

(1) MJPEG case study



(2) Area and performance

Fig. 8. Experiments on the prototype for MJPEG applications

5 Conclusions

In this paper, we presented an actual design and implementation of novel partial crossbar interconnects designed for reconfigurable hardware. We showed that on-demand interconnects can be implemented using parameterized multiplexor arrays. The network was integrated in the ESPAM tool and multiprocessors interconnected with our networks were implemented on a prototype board. We showed that the performance of our partial crossbar-based network is comparable to the performance of a conventional crossbar. The presented network efficiently utilizes the available bandwidth. Moreover, our network provides a single hop communication latency and the network area is significantly reduced compared to a full crossbar.

Acknowledgement. This work was supported by the Dutch Science Foundation (STW) in the context of the Architecture, Programming, and Exploration of Networks-on-Chip based Embedded System Platforms (ARTEMISIA), project number LES.6389.

References

1. T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," ACM Computing Surveys, vol. 38, no. 1, pp. 1–51, Mar 2006.
2. D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G.D. Micheli, "NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip," IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 2, pp. 113–129, Feb 2005.
3. H. Nikolov, T. Stefanov, and E. Deprettere, "Efficient Automated Synthesis, Programming, and Implementation of Multi-processor Platforms on FPGA Chips," Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL'06), pp. 323–328, Aug 2006.
4. H. Nikolov, T. Stefanov, and E. Deprettere, "Multi-processor System Design with ESPAM," Proceedings of 4th IEEE/ACM/IFIP International Conference on HW/SW Codesign and System Synthesis (CODES-ISSS'06), pp. 211-216, Oct 2006.
5. S. Vassiliadis and I. Sourdis, "FLUX Networks: Interconnects on Demand," Proceedings of International Conference on Computer Systems Architectures Modelling and Simulation (IC-SAMOS'06), pp. 160–167, Jul 2006.
6. F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, "HERMES: an Infrastructure for Low Area Overhead Packet-switching Netwoks on Chip," Integration, the VLSI Journal, vol. 38, no. 1, pp. 69–93, Oct 2004.
7. T. Marescaux, V. Nollet, J.Y. Mignolet, A.B.W. Moffat, P. Avasare, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Run-time Support for Heterogeneous Multitasking on Reconfigurable SoCs," Integration, the VLSI Journal, vol. 38, no. 1, pp. 107–130, Oct 2004.
8. B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri, "LiPaR: A Lightweight Parallel Router for FPGA-based Networks-on-Chip," Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'05), pp. 452–457, Apr 2005.
9. T.A. Bartic, J.-Y. Mignolet, V. Nollet, T. Marescaux, D. Verkest, S. Vernalde, and R. Lauwereins, "Topology adaptive network-on-chip design and implementation," IEE Proceedings of Computers & Digital Techniques, vol. 152, no. 4, pp. 467–472, Jul 2005.
10. N. Kapre, N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, "Packet Switched vs Time Multiplexed FPGA Overlay Networks," Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06), pp. 205–216, Apr 2006.
11. C. Hilton and B. Nelson, "PNoC: A flexible circuit-switched NoC for FPGA-based systems," IEE Proceedings of Computers & Digital Techniques, vol. 153, no. 3, pp. 181–188, May 2006.
12. K. Srinivasan and K.S. Chatha, "A Low Complexity Heuristic for Design of Custom Network-on-chip Architectures," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'06), pp. 130–135, Mar 2005.

13. S. Murali and G.D. Micheli, "An Application-specific Design Methodology for STbus Crossbar Generation," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'05), pp. 1176–1181, Feb 2005.
14. M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing On-Chip Communication in a MPSoC Environment," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'04), pp. 752 – 757, Feb 2004.
15. K. Sekar, K. Lahiri, A. Raghunathan, and S. Dey, "FLEXBUS: A High-Performance System-on-Chip Communication Architecture with a Dynamically Configurable Topology," Proceedings of 42th International Conference on Design Automation Conference (DAC'05), pp. 571–574, Jun 2005.
16. M. Huebner, M. Ullmann, L. Braun, A. Klausmann, and J. Becker, "Scalable Application-Dependent Network on Chip Adaptivity for Dynamical Reconfigurable Real-Time Systems," Proceedings of 14th International Conference on Field Programmable Logic and Applications (FPL'04), pp. 1037–1041, Aug 2004.
17. B. Kienhuis, E. Rijpkema, and E. Deprettere, "Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures," Proceedings of 8th International Workshop on Hardware/Software Codesign (CODES'2000), pp. 13–17, May 2000.
18. J. Hu and R. Marculescu, "Energy-Aware Mapping for Tile-based NoC Architectures Under Performance Constraints," Proceedings of the 8th Asia and South Pacific Design Automation Conference (ASP-DAC'03), pp. 233–239, Jan 2003.
19. K. Lahiri, A. Raghunathan, G. Lakshminarayana and S. Dey, "Design of High-Performance System-On-Chips Using Communication Architecture Tuners," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 23, no. 5, pp. 620–636, May 2004.
20. S. Murali and G.D. Micheli, "Bandwidth-Constrained Mapping of Cores onto NoC Architectures," Proceedings of International Conference on Design, Automation and Test in Europe (DATE'04), pp. 896–901, Feb 2004.
21. Alpha Data Parallel Systems, Ltd., http://www.alpha-data.com/adm-xpl.html.

Authentication of FPGA Bitstreams: Why and How

Saar Drimer[⋆]

Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, UK
saar.drimer@cl.cam.ac.uk

Abstract. Encryption of volatile FPGA bitstreams provides confidentiality to the design but does not ensure its authenticity. This paper motivates the need for adding authentication to the configuration process by providing application examples where this functionality is useful. An examination of possible solutions is followed by suggesting a practical one in consideration of the FPGA's configuration environment constraints. The solution presented here involves two symmetric-key encryption cores running in parallel to provide both authentication and confidentiality while sharing resources for efficient implementation.

1 Introduction

The most significant security shortcoming of volatile *Field Programmable Gate Arrays* (FPGA) is the transmission, on every power-up, of their functional definition, the bitstream, from an off-chip source. Field update is what FPGAs are designed for and is one of their distinct advantages, but also what exposes them to various attacks. Some FPGAs now have the ability to process encrypted bitstreams in order to maintain the design's confidentiality. Authenticity, however, is not provided and results in the FPGA accepting bitstreams of *any* form and from *anyone* with the consequence of, at best, denial of service, or at worst, the execution of unauthorized code.

This paper describes the benefits of adding bitstream authentication to the configuration process, a topic that has only been superficially covered to date. While previously suggested, a thorough discussion about the particulars has not been made. The goal of the paper is to motivate the topic and to propose a solution that considers all constraints in order for it to be appealing enough for integration into commercial FPGAs. The paper starts with a short introduction to relevant security goals, followed by examples of real world applications that would benefit from authentication. After considering available and previously suggested schemes, the proposed solution is described.

[⋆] This work has been supported by a grant from Xilinx Inc., 2100 Logic Dr., San Jose, CA 95124, USA.

2 The Case for Authentication

Confidentiality is typically preserved by encryption—an invertible non-linear function that relies on a secret: a *key*. The most common form of encryption uses *symmetric ciphers* where the two parties communicating share a pre-arranged key. This allows the data to be transmitted on an insecure channel as either party encrypts or decrypts it at each end. The *Advanced Encryption Standard* (AES) became the most commonly used symmetric cipher, after NIST chose to replace the *Data Encryption Standard* (DES) in 2001 [1]. For our purposes, it is sufficient to know that AES processes data in 10 rounds, each needing a sub-key that is derived from the main key. The element that contributes the most to the circuit size is the *substitution box* (s-box), which simply converts an n bit input value to a different n bit value. These s-boxes are part of both the main round-function and the key-scheduling function and are typically implemented as look-up tables.

Public-Key Cryptography (PKC) is a different, much more computationally demanding protocol, where the parties are not required to agree on mutual keys prior to communicating. The process is slower than symmetric ciphers and requires longer keys for equivalent security. PKC is most often used for producing digital signatures and establishing symmetric session keys during the initial stages of secure communication.

Authentication provides cryptographic-level assurance of data integrity and its source. Hash functions process arbitrary-length data and produce a fixed-length string called a *message digest*, *hash*, or *tag*. When the process incorporates a secret key, the outcome is called a *Message Authentication Code* (MAC) and allows the receiver to verify that the message is authentic: it has not been tampered with and the sender knows the key. This family of algorithms is designed such that it is computationally infeasible to find two different inputs that result in identical MACs, find an input that yields a given MAC, or find a MAC for a given input without knowing the key.

"In general, authentication is more important than encryption" conclude Schneier and Ferguson in [2, pp. 116]. Being able to impersonate someone to provide falsified information can be more devastating than just being able to eavesdrop on communications. This applies to secure communications in general and emphasizes the importance of authentication in many applications.

For bitstreams, authentication allows the FPGA to verify that the bitstream's content it is about execute was created by someone with whom it shares a secret and that it has not been altered in any way. Doesn't encryption provide the same assurance? As we shall see, the answer is no. Each function should be considered on its own, as each serves a different purpose that sometimes does not require the other. In essence, encryption protects the bitstream's content independently of the device (from cloning, reverse engineering, etc.) while authentication ensures the *correct* and *intended* operation of the FPGA.

The rest of this section describes applications where authentication plays a crucial role in the overall correct and secure operation of the system. To illustrate these, a fictitious company is formed, "Honest Voting Machines, Inc." (HVM) which is set on producing verifiably honest voting machines. The field upgradability and agility of FPGAs appeal to them, and they cannot justify the design of an ASIC. They are looking at ways in which they can achieve their security goals while still being able to use an FPGA as their main processor.

2.1 Data Integrity

HVM is considering using encryption for their bitstreams. They are aware, though, that encryption alone does not provide them with data integrity since data can be manipulated without knowledge of the key. Symmetric ciphers have several *modes of operation* that define how data is processed [3]. The various modes allow the core function of a symmetric cipher to operate as a block cipher, stream cipher or even produce a MAC. Each of these modes have respective security properties that must be evaluated per the application. A popular mode is *Cipher Block Chaining* (CBC) that is used to encrypt arbitrary-length messages such that ciphertext blocks are not the same for identical plaintext blocks. CBC is self recovering in that it can recover from ciphertext bit errors within two decryption operations. If m bits are manipulated in a block of size n, $n/2$ incorrect bits, on average, will be incorrect in the resulting plaintext block. An additional m bits will be flipped in the subsequent plaintext block at the exact location of the original manipulated bits. When using CBC for bitstream encryption, an adversary can toggle m bits in a single block to change the design without knowing the key. The previous block will be completely corrupt but it may not contain data that is relevant to the attacked portion of the chip. This attack can be fruitful, for example, if the goal is to find a collection of bits that, when toggled, disable a defence mechanism. Other modes have similar, or worse, vulnerabilities with regards to manipulation of ciphertexts.

Hadžić et al. were the first to tackle the implication of malicious tampering of bitstreams [4]. The paper describes how an FPGA is permanently damaged due to shorts caused by manipulating configuration bits that control pass gates. The high currents caused by these shorts may cause the device to exceed its thermal tolerance, permanently damaging it. A bitstream of random content will create considerable contention and is relatively simple to construct; if one knows the bitstream construction, an even more devious one can be made. This attack, however, is not at all likely to succeed for most fielded systems. To damage the FPGA, the power supplies need be able to provide the high current needed for sufficiently long periods while the supporting circuitry be able to handle that current. This is not the case in most systems where the voltage regulators collapse upon high current demand, or some other failures occur, long before the device can be damaged. In addition, FPGAs have circuitry that detects voltage drops, usually 15% below nominal, in which case, they reset themselves. Although not suggested in [4], authentication solves all issues presented there.

Error correction (and detection) codes, such as *Cyclic Redundancy Codes* (CRC) do not satisfy HVM either. These codes are currently used to prevent a bitstream that was accidentally corrupted in transmission from being loaded onto the device and potentially causing damage due to circuit shorts. These linear functions can be forged with relative ease and do not detect all types of errors, especially ones put there maliciously [5]. Linear checks are therefore not adequate for cryptographic-level data integrity. Authentication protocols provide this level of assurance by being designed such that even a single wrong bit will result in an incorrect MAC.

2.2 Code Audit

HVM has been contracted by a large municipality to provide it with voting machines. The municipality would like to assure itself, and its constituents, that the code running on the FPGA-based system does not have back-doors or election-altering functions. HVM, in turn, insists that only their proprietary code is run on their voting machines. Those interests are met as follows.

HVM provides the source code to the municipality, or posts it on its web site, along with a MAC for the compiled design. That MAC was computed using a key that HVM embeds into the FPGA that will be part of the system sold to the municipality. HVM also provides the software version and settings such that the code would compile to exactly the same bitstream every time, provided that there are no design changes. Given that information, the municipality compiles the verified code to produce a bitstream to which they append the MAC that was provided by HVM. The resultant bitstream is then loaded onto the FPGA by the municipality. Only HVM knows the key to compute a valid MAC and therefore, no-one else can create a bitstream that will run on that particular FPGA. Since the design was not encrypted and the code verified, the municipality is assured of its correctness.

2.3 Versioning of Identical Hardware

It makes economic sense for HVM to design hardware that can function, depending on the bitstream, in “standard” or “enhanced” modes, the latter having more functionality and is more expensive. HVM embeds the MAC key to either version in each system’s FPGA. The code audit scheme allows for the code for both variants to be publicly available, yet would not let the customers replace the standard bitstream with the enhanced one.

This and the previous scheme require that the MAC key be permanently embedded within the FPGA, otherwise it could be erased and the system used with the publicly available, or other, code. Some FPGAs already allow permanent key storage for encryption; storage for the authentication key will need to be provided as well. It also requires strict inventory control by both seller and customer in order to prevent counterfeit products from being manufactured or purchased with the public design modified for malicious intent.

2.4 Authenticated Heart Beat

Consider a critical system that is operating in an insecure and remote environment. For example, HVM’s voting machine during an election. The goal is to provide to both the code running on the FPGA and a remote monitoring center (MC) a guarantee that they are continuously connected to one another. If either the code or the MC detects a lapse in connectivity, a breach is assumed. An internal authentication core with a unique key supplemented by a volatile counter can accomplish this.

On election day, officials load the voting machine with the previously audited code along with a predetermined initial state of the counter, C_i. Once configuration is complete, an initial authentication process is performed as follows.

$$A \rightarrow B : N_A \in \{0,1\}^{128}, \mathrm{H}_K(N_A||C_{i+1})$$
$$B \rightarrow A : \mathrm{H}_K(N_A||C_{i+2})$$

A, the monitoring center, sends B, the FPGA, a nonce N_A and a MAC, H_K, of the nonce concatenated with the current value of C under the key, K. The FPGA computes the MAC on its end and stores the result in a status bit that is constantly monitored by the design. The FPGA then replies with a MAC computed with C_{i+2} and N_A. The authentication process repeats in the background as often as needed, each time with a unique nonce from the MC an incremented counter value, C_i. The counter and nonce provide freshness; the nonce should never be reused to assure that previously recorded challenges can not be used in the future by an attacker. The authentication process does not interfere with the normal operation of the device.

3 Constraints

Now that some of the benefits of authentication have been described, the constraints that bound a solution will be considered for arriving at an optimal solution.

3.1 Configuration Environment

Systems deal with remote and local configuration of an FPGA in a variety of ways. Some have non-volatile memory placed close to the FPGA, some configure via a microcontroller, while in others, the configuration is fetched, on every power-up, via a network or a local PC. Authentication, or other bitstream pre-processing, can be done using devices external to the FPGA, such as microcontrollers or dedicated processors, but only if the trust is set around the system itself. If this is the case, elaborate tamper proofing is necessary, or that the system be placed in a secure environment. This is not feasible for many, or most, applications. Thus, in this paper, the boundary of trust is set around the FPGA itself without delegating the bitstream pre-processing to other devices. Whether the adversary has physical access to the FPGA or not, malicious code must be prevented from being executed. Incorporating the authentication process into the FPGA accounts for bitstreams arriving by any means, be they from a local non-volatile device, USB, ethernet, or the various other sources. This requirement means that implementing the authentication protocol in the FPGA's user logic will not work since an initial, trusted, bitstream must be loaded first. Since this is what we are trying to achieve to begin with, we must rely on a core that is part of the configuration process. Further, the requirement excludes verification mechanisms that rely on external devices and the continued secrecy of the bitstream's proprietary encoding, as suggested in [6,7] and the recently announced "Device DNA" from Xilinx [8].

Current FPGAs can only hold a single configuration data at a time. Once a new configuration process has begun, the previous one can not be reverted back to without full reconfiguration. No history is kept of previous failed attempts. Since we put the trust boundary around the FPGA, there is no way to guarantee that the *same* bitstream will be provided twice from an external source. Therefore, authentication and encryption operations must be done within the same configuration cycle.

Denial-of-service attacks can not be prevented with today's configuration process. If the attacker has physical access to the device he can destroy or unplug it. Short of effective tamper-proofing of the whole system, or placing it in a secure environment, there is not much to be done against a destructive attacker. If the attacker is remote, and has the ability to reconfigure the device, he can initiate a failed configuration attempt to leave the device un-configured. Authentication will prevent unauthorized configuration from running, but not denial-of-service. In addition, there is no way to prevent *replay attacks* where the adversary records a valid bitstream and sends it at a later time, perhaps to revert back to a previous bitstream with known vulnerabilities. All these issues stem from the fact that the FPGA does not preserve state between configurations.

Table 1. Configuration throughput and time of largest FPGA family members

FPGA	device	config bits	mode	frequency	throughput	time
Virtex-5 [9]	LX330T	82,696,192	8 bit[a]	100 MHz	800 Mbits/s	0.1 s
Stratix-III [10]	L340	125,829,120	8 bit	25 MHz[b]	200 Mbits/s	0.63 s
Spartan-3 [11]	5000	13,271,936	8 bit	50 MHz	400 Mbits/s	0.033 s

[a] Longer bus width, x16 and x32, are not supported when encryption is used.
[b] Actual is 100 MHz; data must be x4 slower than clock if decryption is used.

The adopted solution must meet the current configuration throughput and not adversely affect total configuration times. Table 1 shows the maximum throughput and configuration times of the largest family members of recent FPGAs.

The throughput of the authentication core needs to meet the maximum throughput of the decryption core, since they most likely operate together. According to Table 1, this is set around 800 Mbit/s, which is a modest requirement. Typically, as the throughput of a circuit increases, so does its size. It is important that the core be designed such that it does not have greater throughput than required, so not to "waste" resources.

The effect of longer configuration times depends on the application and the mode. Some applications configure in serial mode at low frequencies, and therefore, doubling the configuration times may be undesirable. Applications that require frequent scrubbing to mitigate *Single Event Upsets*, for example, would suffer from longer configuration times. In general, therefore, it is best to find a solution that does not adversely affect the total time of configuration. That said, users may need to accept extra latency for the added functionality.

3.2 Manufacturers

Economical considerations should be mentioned as they play a crucial role when a new feature or technology is evaluated or considered for adoption. If the goal is to suggest something that would be considered to be viable, these constraints must be accounted for. From the perspective of the FPGA manufacturers, incorporating new functionality into their devices must yield a return on their investment. This means that either the functionality is required or demanded by sufficient amount customers, or, it is small and does not adversely affect performance, yield, support and development costs. Standardization is important as well as manufacturers prefer the use of "approved" algorithms that will be accepted by the majority of their customers. In the case of security, this is doubly important, as the manufacturers and users may not be experts in the field and must rely on certification by other authorities.

Although authentication is important, as this paper demonstrates, it seems as though manufacturers are not yet willing to sacrifice large portions of silicon to implement it. The most significant constraint, then, is to strike a balance between functionality and size.

4 Bitstream Authentication

Many protocols exist that provide authenticity. This section evaluates the most often used protocols for their suitability under our constraints. To help with the analysis, Table 2 summarizes Parelkar's results [12] that provides an estimates of the size and performance of the circuits in 90 nm technology.

Table 2. Parelkar's results targeting 90 nm ASIC architecture shown as the ratio of AES-128 in combination with SHA and AE schemes to stand-alone AES-128 [12]

criteria	AES-128(ECB)	+SHA1	+SHA512	+EAX	+CCM	+OCB
area	1 (41,250 μm^2)	2.8	4.4	1.5	1.4	1.5
throughput	1 (1097 Mbit/s)	1	1.2	0.4	0.4	0.8

4.1 Public-Key Cryptography and Keyed-Hashes

PKC is theoretically suitable for bitstream encryption but is costly in hardware. The large keys involved and the many exponentiations and additions on wide bus length [13] make it unattractive for our application as it does not meet our constraint of small area. As mentioned, PKC is also slow compared to symmetric ciphers, and is mostly used for signatures and establishing session keys. As we shall see, there are faster and more nimble algorithms available for achieving our goal.

Keyed-hash message authentication codes (HMAC) produce a fixed length MAC from an arbitrary length message. The addition of the key into the input of a hash function provides authenticity [14]. A commonly used HMAC function is NIST's *Secure Hash Algorithm* (SHA) with various output lengths. Recently, flaws have been found in SHA-1 [15] and while these flaws are only of theoretical threat, it would be prudent to consider the stronger successor, SHA-2 [16] or its widely used variants, SHA-256 and SHA-512. If we use an AES-128 encryption core as the base for comparison, Parelkar's results as shown in Table 2, indicate that SHA-512 in combination with AES is 4.4 times as large as stand-alone AES, making this option unattractive. Other solutions may prove more efficient.

4.2 Authenticated Encryption

Since the late 1990s, there has been increased interest in protocols that securely provide encryption and authentication using a single key; there are collectively called *authenticated encryption* (AE) protocols. Black provides a concise introduction to AE algorithms in [17] and a summary of their properties is shown in Table 3. "Single- or dual-pass" AE, as the name suggests, refers to the times the message needs to be processed. Dual-pass algorithm perform each operation in a separate pass. "Provably secure" means that the scheme has been shown to be at least as secure as the underlying symmetric-key cipher it is based on. "Associated data" is a portion of the message that needs to be authenticated but not encrypted. An example is the routing header of a network packet. Since only the source and destination know the key, this header can not be encrypted and the destination to be sure that the packet has indeed arrived from the right source, it must be authenticated. Complete operational description of each of the AE schemes is outside the scope of this paper and only the relevant attributes will be discussed.

Parelkar was the first to suggest the use of AE for FPGA bitstreams and concluded in [18,12] that the dual-pass *Counter with CBC-MAC* (CCM) would be best suited for bitstream authentication with the added benefit of it being a NIST recommended mode of operation [19]. Parelkar and Gaj also suggested EAX for bitstream processing in [20].

Dual-pass AE are not well suited for bitstream processing as they would require significant changes, circuit additions, and increased time to the configuration process. The reason is the lack of accessible temporary storage for the bitstream within the FPGA. After the first pass, the data has been loaded onto the configuration cells and if a second pass is needed, this data would need to be read-back, passed through the internal configuration processor again, and re-written. In addition, not all of the bitstream's content, such as "no-ops", header and footer commands are stored in the configuration memory, making a second

Table 3. Properties of authenticated encryption algorithms

protocol	passes	provably secure	associated data	free	standard
IAPM,XECB,OCB	1	yes	no	no	no
EAX, CWC	2	yes	yes	yes	no
CCM	2	yes	yes	yes	NIST

pass even more problematic. This complication can not be solved by sending the bitstream twice since, as stated, the security boundary is set at the FPGA, so there is no way to guarantee that the same bitstream will be received on both passes, allowing the attacker sending modified bitstreams for each pass, compromising the security.

Single pass algorithms are better suited, but currently, they are all proprietary. This is not a major hurdle for the FPGA manufacturers, but is still a hindrance for adoption given the other, royalty-free, options. Manufacturers would rather stay out of constrictive licenses for their hardware because in case of dispute, they can not simply "patch" the device. The other issue of lack of confidence in non-"approved" protocols has been previously mentioned.

The main disadvantage of AE for bitstream processing, however, is in what it was originally trying so solve: the separation of authentication and encryption. As we have seen in the examples of Section 2, there are applications that actually benefit from authentication-only schemes.

AE schemes do not seem to be ideal. Dual-pass schemes will require significant overhead while single-pass ones require licensing. Some modes, such as *Offset Codebook* (OCB), require that both the encrypt and decrypt functions be implemented at the receiving end, the FPGA. In addition to that, designers are averse to using new schemes that are not standardized or widely used; many AE protocols are still being evaluated. Having said that, in the future, AE schemes may become more appealing and their attributes better suited for our application.

4.3 Proposed Scheme

When discrete operations are combined to perform encryption and authentication the result is called *generic composition* and can be applied in the following ways: *MAC-then-encrypt*, *encrypt-then-MAC*, or *MAC-and-encrypt*. The benefits and disadvantages of each are discussed in [17] and [2, pp. 115–117] and needs to be chosen according to the application. The following proposed scheme is a generic composition where two symmetric key ciphers operate in parallel on the same bitstream, one for decryption and the other for producing a MAC.

The CBC mode was briefly described in Section 2.1 when it is used for encryption, but it can also be used to produce a MAC. The last encrypted block of CBC is a MAC of the message since it depends on all previous blocks. CBC, however, has security shortcomings for variable length messages which may be an issue with partial reconfiguration and the variation of the bitstream's length, such as adding no-ops or commands after the device has been released. These deficiencies are corrected in *Cipher-based MAC* (CMAC) mode, which was recommended by NIST in 2005 [21]. CMAC is a wrapper around the symmetric block cipher and only uses the encryption/forward function.

The straight forward generic composition would be to have the two ciphers operate in parallel, one in CBC mode for decryption, and the other in CMAC mode, for authentication. This would mean both a decryptor and encryptor need to be implemented. If the amount of resources is to be minimized, this would be a disadvantage. As mentioned, AES's most resource demanding element is the s-box. For AES, these s-boxes are inverse of each other for the encrypt and decrypt functions, and therefore, must be implemented, or generated, separately. More effectively, in terms of hardware utilization, would be to use another mode for decryption that only uses the forward function.

Counter (CTR), *output feedback* (OFB), and *cipher feedback* (CFB) are all modes that use the forward function for both encryption and decryption. Using these modes allows the generic composition of two encryption cores to share resources for a more efficient implementation. Since the s-boxes are the same, their look-up tables can be shared. This may be done by multiplexing them within a clock cycle, as they can be made to be fast compared to other operations.

Readback, available in some FPGAs, allow the user to retrieve the current state of the FPGA while it is still in operation. Currently, when bitstream encryption is used, readback is disabled since the configuration data would come out in plaintext, defeating the purpose. The use of an forward-only mode will enable the data to be encrypted as it is read out of the FPGA.

The use of the same key for both operations may seem appealing since it will require only a single key-scheduling circuit. In general, this is bad practice that often leads to vulnerabilities and should be avoided. Aside from not having a proof of security, using the same keys would not enable the complete separation of operations. In some applications there may be a need for different distribution of keys where, for example, many devices share the same authentication key while having different encryption keys.

The proposed scheme is depicted in Figure 1 and operates as follows. The construction of the bitstream is very similar to the one used today with the exception of replacing the linear checksum (such as CRC) with a MAC. In the cases where authentication is not used, the CMAC can replace the checksum for all configurations by using a key of value 0. As an option, the implementation may allow the user to choose whether to MAC-then-encrypt or encrypt-then-MAC according to their preference, with the former resulting in a latency of a single block. The associated data, namely, the header and footer that provide the FPGA with initialization information and instructions is authenticated but not encrypted, preventing an attacker from enabling or disabling certain commands. The only portion of the bitstream that will not be authenticated is the startup sequence that could be removed if the commands are hard coded into the FPGA. The encrypted portion is

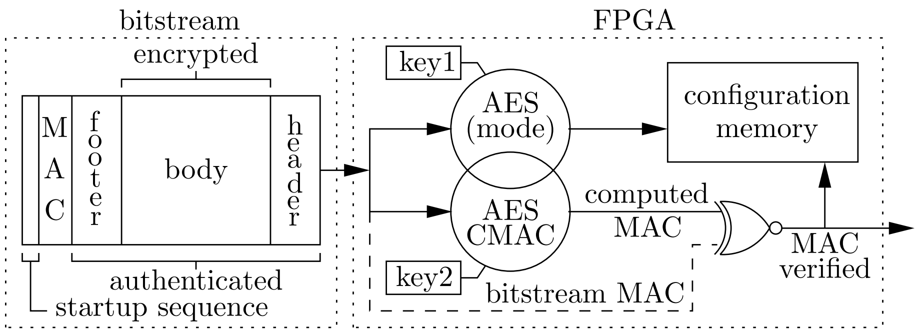


Fig. 1. Two parallel AES cores provide decryption and authenticity. The amount of shared resources depends on the modes.

the design itself that needs to be kept secret. After the footer has been processed, the computed and bitstream's MACs are compared. If they are equal, the FPGA will continue to the startup sequence. Otherwise, configuration will abort and the cells be cleared. After configuration has succeeded, the CMAC core can continue to operate without interfering with the configuration content or the operation of the device. This can be used for the authenticated heart beat described in section 2.4 or for authenticating partial reconfigurations.

5 Conclusion and Future Work

Encryption protects the design but does not prevent malicious code from running on the FPGA. Authentication allows the FPGA to operate as intended and be configured only from a bitstream generated by someone with whom it shares a secret. The examples given show that many applications can benefit from authentication. With the purpose of arriving at an appealing scheme that may be adopted by manufacturers, the constraints of the configuration environment were considered to narrow down the various protocols that provide authentication. The solution proposed allows for the separation to the two processes and does not demand significant increase to resources beyond what is currently used for bitstream encryption. The solution also has minimal impact on the current configuration process.

Future work includes ASIC implementations to quantify resource-sharing using the various modes of operation and optimization techniques.

Acknowledgement

Thanks to Steven J. Murdoch, Markus Kuhn, Ross Anderson, Milind Parelkar, and anonymous reviewers for their valuable comments and suggestions.

References

1. NIST, U.S. Dept. of Commerce: FIPS 197: Advanced encryption standard. (2001)
2. Ferguson, N., Schneier, B.: Practical Cryptography. John Wiley & Sons, Inc., New York, NY, USA (2003)
3. Dworkin, M.: Special Publication 800-38A: Recommendation for block cipher modes of operation. NIST, U.S. Dept. of Commerce. (2001)
4. Hadžić, I., Udani, S., Smith, J.M.: FPGA viruses. In: FPL '99: Proceedings of the 9th International Workshop on Field-Programmable Logic and Applications. Volume 1673 of LNCS., London, UK, Springer-Verlag (1999) 291–300
5. Stigge, M., Plötz, H., Müller, W., Redlich, J.P.: Reversing CRC – theory and practice. Technical Report SAR-PR-2006-05, Humboldt University Berlin (2006)
6. Baetoniu, C., Sheth, S.: XAPP780: FPGA IFF copy protection using Dallas Semiconductor/Maxim DS2432 Secure EEPROM. Xilinx Inc. (2005)
7. Altera Corp.: FPGA design security solution using MAX II devices. (2004)
8. Xilinx Inc.: UG332: Spartan-3 generation configuration user guide. (2006)
9. Xilinx Inc.: DS202: Virtex-5 data sheet: DC and switching characteristics. (2006)
10. Altera Corp.: Stratix III design handbook. (2006)
11. Xilinx Inc.: DS099: Spartan-3 FPGA family: Complete data sheet. (2006)
12. Parelkar, M.M.: Authenticated encryption in hardware. Master's thesis, George Mason University, Fairfax, VA, USA (2005)
13. Batina, L., Örs, S.B., Preneel, B., Vandewalle, J.: Hardware architectures for public key cryptography. VLSI Journal, Integration **34**(1-2) (2003) 1–64
14. NIST, U.S. Dept. of Commerce: FIPS 198: The keyed-hash message authentication code (HMAC). (2002)
15. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: CRYPTO. (2005) 17–36
16. NIST, U.S. Department of Commerce: FIPS 180-2: Secure hash standard. (2002)
17. Black, J.: A. Authenticated encryption. In: Encyclopedia of Cryptography and Security. Springer (2005) 10–21
18. Parelkar, M.M.: FPGA security – bitstream authentication. Technical report, George Mason University (2004) http : //mason.gmu.edu/ mparelka/reports/ bitstream$_a$uth.pdf
19. Dworkin, M.: Special Publication 800-38C: Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality. NIST, U.S. Dept. of Commerce. (2005)
20. Parelkar, M.M., Gaj, K.: Implementation of EAX mode of operation for FPGA bitstream encryption and authentication. In: Field Programmable Technology. (2005) 335–336
21. Dworkin, M.: Special Publication 800-38B: Recommendation for block cipher modes of operation: The CMAC mode for authentication. NIST, U.S. Dept. of Commerce. (2005)

Design of a Reversible PLD Architecture

Jae-Jin Lee[1], Dong-Guk Hwang[2], and Gi-Yong Song[2]

[1] Electronics and Telecommunications Research Institute, Korea
[2] School of Electrical and Computer Engineering, Chungbuk National University, Cheongju, 361-763, Korea
gysong@chungbuk.ac.kr

Abstract. Reversible gate is a circuit that has the same number of inputs and outputs satisfying one-to-one mapping between the vectors of input and output. So far several logic synthesis methods for reversible logic have been proposed, however, they are not able to synthesize a reversible function with input and output of arbitrary width in a constructive manner based on building blocks and interconnect. This paper proposes a new reversible PLD(programmable logic device) architecture that enables any reversible function to be implemented by cascading the building blocks, or logic units through interconnect, and fits well on arithmetic circuits in particular. Also, a new reversible gate, T2F gate, is suggested and adopted in the proposed reversible PLD architecture. Both reversible PLD and T2F gate offer significant alternative view on reversible logic synthesis.

1 Introduction

As proved by Landauer [1], using traditional irreversible logic gates such as AND or multiplexer type inevitably lead to energy dissipation regardless of the realization technology. Bennett [2] showed that for power not to be dissipated in a circuit, it is necessary that the circuit be built from reversible gates. A circuit is said to be reversible if it has the same number of inputs and outputs and there is a one-to-one and onto mapping between the vectors of input and output, thus the vector of input can always be reconstructed from the vector of output [3][4][5]. Several reversible gates have been proposed over the past decades. Among them are the controlled-not(CNOT) gate [6] proposed by Feynman, Toffoli gate [7] and Fredkin gate [8].

Little has been published on logic synthesis and optimization methods for reversible and quantum logic. Perkowski et al. [3] suggested a RPGA(reversible programmable gate array). Recently, Maslov et al. [5] proposed a reversible design method that uses the minimum number of garbage outputs. Our work has been motivated by the fact that reversible logic synthesis methods mentioned above are not able to synthesize a binary function with a large number of inputs and outputs in a constructive manner based on logic units and interconnect. The RPGA in [3] is based on a regular structure to realize a binary function in reversible logic. If the number of inputs and outputs of a binary function becomes large, the RPGA itself should be extended as much as needed according to the inputs and outputs to implement the binary function, namely, we need an infeasibly big RPGA.

In this paper, as an alternative that can be put into practice, a RPLD(reversible programmable logic device) architecture with logic units and interconnect inside is proposed which enables any reversible function to be synthesized by cascading the building blocks in a constructive manner. The relation between RPGA in [3] and RPLD in this paper is much like the relation between PLA/PAL and CPLD/FPGA. Also, a reversible gate with less quantum cost than Kerntopf gate adopted in RPGA [3] is suggested and named T2F gate, and adopted in the RPLD architecture proposed in this paper. The RPLD and T2F gate in this paper offer significant alternative view on reversible logic synthesis.

2 Reversible Logic Gates

Let us briefly introduce some representative reversible gates, and then explain about T2F gate suggested in this paper. Reversible circuits which are hierarchically composed of reversible primitives have two types of outputs in general; functionality outputs, and outputs that are needed only to achieve reversibility, being called "garbage" [8].

2.1 Fundamental Reversible Gates

Feyman gate: A Feynman gate with input vector (A, B) and output vector (P, Q) implements the logic function : $P = A$ and $Q = A \oplus B$ [6]. The Feynman gate is a reversible $(2, 2)$ gate. One of the input, B, serves as a control signal: If $B = 0$, then the output Q simply duplicates the input A; if $B = 1$, then the output $Q = A \oplus 1 = A'$, an inverse of the input. For this reason, the Feynman gate is also called the controlled-not(CNOT) gate, or the quantum XOR gate due to the popularity in the field of quantum computing.

Toffoli gate: A Toffoli gate [7] is one of the most important 3×3 gate among all 3×3 gates. This gate has input-output relation : $P = A, Q = B$ and $R = AB\oplus C$ (AB stands for A AND B). The Toffoli gate is a 2-CNOT gate since it passes the two inputs (A and B) unchanged and it inverts the input C when $A = 1$ and $B = 1$.

Fredkin gate: This gate has been exploited by many reversible logic circuits for its usefulness. It realizes $P = C'A+CB, Q = C'B+CA, R = C$ where (A, B, C) is input vector and (P, Q, R) is output vector.The Fredkin gate [8] uses C as a control input : if $C = 0$, then the outputs are simply copies of inputs; otherwise, the input A and B are interchanged.

Kerntopf gate: The 3×3 Kerntopf gate [3] is described by equations: $P = 1 \oplus A \oplus B \oplus C \oplus AB, Q = 1 \oplus AB \oplus B \oplus C \oplus BC, R = 1 \oplus A \oplus B \oplus AC$. When $C = 1$, then $P = A + B, Q = AB, R = B'$, so OR and AND operations are realized on outputs P and Q, respectively, with C being the control input.

When $C = 0$, then $P = A'B', Q = A + B', R = A \oplus B$, therefore, for control input '0', the gate realizes NOR, IMPLICATION, and EXOR on its outputs P, Q and R, respectively. Despite the theoretical advantages of Kerntopf gate over classical Fredkin and Toffoli gates, there are no published results on optical or CMOS realization of this gate so far. RPGA proposed in [3] adopted Kerntopf gate to create AND and OR configurations.

2.2 T2F Gate

This paper proposes a new reversible gate consisting of one Toffoli gate and two Feynman gates as shown in Fig. 1, and named it 3×3 T2F gate. This gate described by equation: $P = A, Q = AB \oplus C, R = AB \oplus C \oplus A \oplus B$. The AND and OR operation can be realized on Q and R, respectively, when $C = 0$. The T2F gate is used to realize both AND and OR operations in a logic unit of the proposed RPLD. The T2F gate is estimated to have a smaller quantum cost than Kerntopf gate since quantum cost of a gate G is the number of basic operations (one-bit and controlled-V type required to realize the function defined by G [5]). Moreover, the number of garbage outputs entailed in T2F is the lower bound for the circuit realizing both AND and OR operations.

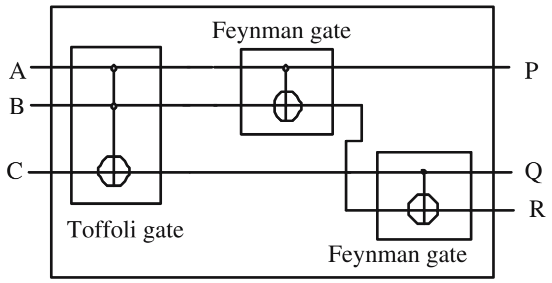


Fig. 1. T2F gate

Theorem 1. *A circuit realizing both AND and OR operations simultaneously has at least one garbage output.*

Proof. In a circuit realizing both AND and OR operations simultaneously, only output pattern (0 1) occurs two times in a truth table. In order to make the circuit reversible, at least one more bit which enables each output pattern to correspond to unique input pattern is required since n identical output patterns should be distinguished from each other by adding at least $\lceil log_2^n \rceil$ garbage bits, here $n = 2$.

3 Reversible PLD Architecture

The architecture, or fabric of the proposed RPLD(reversible programmable logic device) which is a combination of Logic Units(LUs) and interconnect. Each LU is connected to the immediate neighboring LUs through shortlines. Other routing network is switch matrix allowing a signal to be routed from one switch matrix to another, eventually connecting LUs that are relatively far away from each other. The third type of routing network is a longline, which synthesizer can use to connect LUs that are physically far away from each other on critical path without introducing long delay

The LU is designed to implement a 1-bit reversible full adder as a building block with a implementation of various arithmetic processors by cascading LUs on the RPLD fabric according to EXOR-based Davio expansion in mind. To design an LU, first, an RPSFG capable of implementing a 1-bit reversible full adder is set up, and then upgraded by adding Fredkin gate to accommodate the EXOR-based Davio expansion. Let us call this upgraded RPSFG with Fredkin gate inside RPSFG-F. The structure of a RPSFG-F is shown in Fig. 2(a). The LU in the proposed RPLD consists of one RPSFG-F and three Toffoli gates as shown in Fig. 2(b), and synthesizes arbitrary symmetric functions.

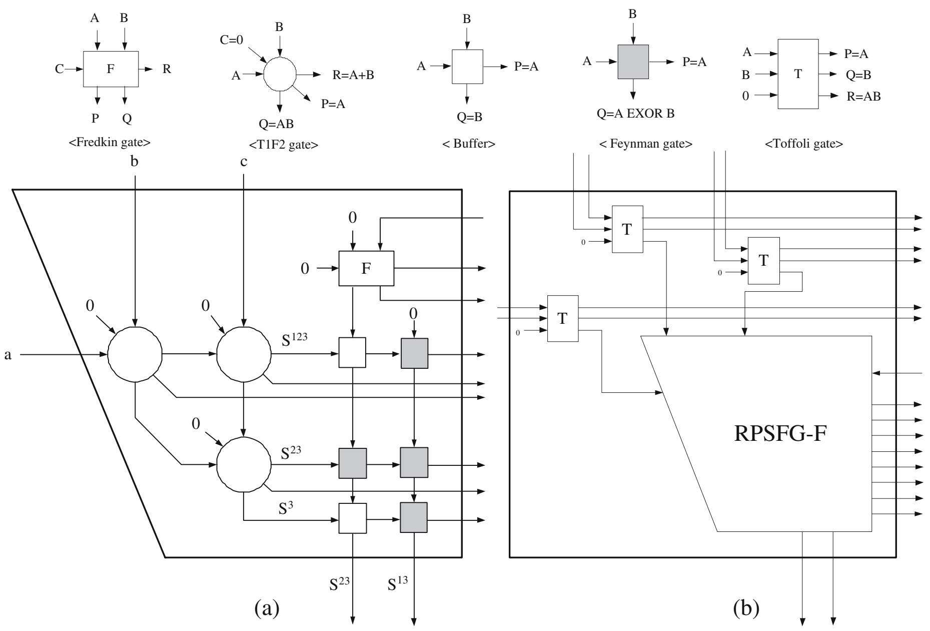


Fig. 2. (a)Structure of RPSFG-F (b)Structure of Logic Unit

Three Toffoli gates control the inputs of RPSFG-F. Two outputs of each Toffoli gate should be forwarded to primary outputs to guarantee reversibility. Each LU requires 14 garbage bits as shown in Fig. 2(b). One RPSFG-F slice implementing both sum and carry of a 1-bit reversible full adder with 8 garbage outputs can implement arbitrary 2- or 3-input reversible symmetric function. The structure of an RPSFG-F which looks similar to that of an RPGA in [3] is distinguished by two facts. First, T2F gate with a smaller quantum cost than Kerntopf gate is used for realizing AND and OR operation simultaneously. Next, let us consider the synthesis of a symmetric function with more than three inputs. In the case of an RPGA in [3], an extension of RPGA itself as necessary is required to accommodate the synthesis of such a function. Although an extension of an RPGA to any size is theoretically feasible, not only is the availability of such a device more unlikely, but also it is difficult to synthesize a non-symmetric function with a large number of input and outputs like n-bit multiplier due to its symmetrization cost. However, a symmetric function with more than three inputs can be synthesized as a cascade of RPSFG-F's through EXOR-based Davio expansion on the proposed RPLD architecture. In order to implement arbitrary n-input symmetric function($n \geq 4$) on the proposed RPLD architecture, we use EXOR-based Davio expansion. The readers should refer to [9] for more details on Davio expansion.

4 Experimental Results

Previous works have mainly concentrated on minimizing the number of gates used and the number of garbage outputs, and there have been no published reports so far on a structure that enables any reversible function with input and output of arbitrary width to be implemented by programming the building blocks and interconnect without extending the structure itself as necessary. To show the versatility of the proposed RPLD architecture, arithmetic or signal processing circuits such as adders, multipliers and FIR filter are synthesized on it. Table 1 shows the synthesis results that are an estimation through strict inference, so these values would not be justified as physical data for the proposed architecture. Even though we verify both reversibility and functionality of each design using HDL simulation instead of checking them on a real RPLD chip, it is interesting to see that the proposed RPLD is able to synthesize such designs that are highly regular like arithmetic circuits in a constructive manner.

Table 1. Synthesis results

Design	T2F	Toffoli	Fredkin	Feyman	Garbage outputs	LUs
8-bit adder	24	24	8	32	112	8
32-bit adder	96	96	32	128	448	32
4×4 multiplier	39	39	13	52	182	13
8×8 multiplier	171	171	57	228	798	57
4-TAP FIR filter	1104	1104	368	1472	5152	368

5 Conclusions

In this paper, we propose a new RPLD architecture that implements any reversible function by cascading the logic units through interconnect, and fits well on arithmetic circuits in particular. Reversible arithmetic circuits such as adder, multiplier and FIR filter are synthesized in a constructive manner on the proposed RPLD. Also, a new reversible gate, T2F gate, is suggested and adopted in the proposed RPLD architecture. The RPLD and T2F gate offer significant alternative view on reversible logic synthesis. Synthesis tools for LU configuration and EXOR-based Davio expansion as well as automatic routing need to be developed to complete the process of reversible logic synthesis. In addition, as a post-processing, the compaction of garbage outputs through a top-down approach deserves an attention.

Acknowledgement

This work was supported by the research grant of the Chungbuk National University in 2005.

References

1. R.Landauer, "Irreversibility and Heat Generation in the Computing Process," IBM, J. Research and Development, vol. 3, pp. 183-191, 1961.
2. C.H.Bennett, "Notes on the History of Reversible Computation," IBM, J. Research and Development, vol. 32, pp. 16-23, 1988.
3. M.Perkowski, A.Al-Rabadi, P.Kerntopf, A.Mishchenko and M.Chrzanowska-Jeske, A.Buller, A.Coppola, and L.Jozwiak, "Regularity and Symmetry as a Base for Efficient Realization of Reversible Logic Circuits," IWLS 2001, pp. 90-95, 2001.
4. M.Nielsen and I.Chuang, Quantum Computation and Quantum Information, New York: Cambridge Univ. Press, 2000.
5. D.Maslov and G.W.Dueck, "Reversible Cascades with Minimal Garbage," IEEE Trans. CAD, vol. 23, no.11, 2004.
6. R.Feynman, "Quantum Mechanical Computers," Opt. News, vol. 11, pp. 11-20, 1985.
7. T.Toffoli, "Reversible Computing," MIT Lab for Comp. Sci., Tech. Memo MIT/LCS/TM-151, 1980.
8. E.Fredkin and T.Toffoti, "Conservative Logic," Int. J. Theor. Phys., vol. 21, pp. 219-253, 1982.
9. J.J.Lee and G.Y.Song, "A New Application-Specific PLD Architecture," IEICE Trans. Fundamentals., vol. E88-A, no. 6, pp. 1425-1433, 2005.

Designing Heterogeneous FPGAs with Multiple SBs[*]

K. Siozios, S. Mamagkakis, D. Soudris, and A. Thanailakis

VLSI Design and Testing Center,
Department of Electrical and Computer Engineering,
Democritus University of Thrace, 67100, Xanthi, Greece
{ksiop, smamagka, dsoudris, thanail}@ee.duth.gr

Abstract. The novel design of high-speed and low-energy FPGA routing architecture consisting of appropriate wire segments and multiple Switch Boxes is introduced. For that purpose, we develop a new methodology consisting of two steps: (i) Exploration and determination of the optimal wire length and (ii) Exploration and determination of the optimal combination of multiple switch-boxes, considering the optimal choice of the former step. The proposed methodology for designing the high performance interconnection architecture is fully-supported by the software tool called EX-VPR. For both steps, the selection criterion for a minimal Energy×Delay Product is chosen. Depending on the localized performance and energy consumption requirements of each specific region of FPGA architecture, we derive a set of corresponding spatial routing information of the applications mapped onto FPGA. We achieved Energy×Delay Product reduction by 55%, performance increase by 52%, reduction in total energy consumption by 8%, at the expense of increase of channel width by 20%.

1 Introduction

The FPGA architecture characteristic changed and improved significantly the last two decades, from a simple homogeneous architecture with logic modules, and horizontal and vertical interconnections to FPGA platforms (e.g. Virtex-4 family [7]), which include except logic and routing, microprocessors, block RAMs etc. Furthermore, the FPGA architecture changed gradually from homogeneous and regular architecture to a heterogeneous (or piece-wise homogeneous) and irregular (or piece-wise regular). The platform-based design allows to designer to build a customized FPGA architecture, depending on the application domain requirements. The platform-based strategy changed the FPGAs role from a “general-purpose” machine to an “application-domain” machine, closing the gap with ASIC solutions. Having in mind the current trend about the design FPGA architecture, we proposed a new software-supported methodology for selecting appropriate interconnection architecture.

Due to the fact that about 60% of an FPGA power is occupied by routing resources [4], many researchers have spent much effort on minimizing power leading to smaller

[*] This work was partially supported by the project IST-34793-AMDREL, the PYTHAGORAS-II and the project PENED ’03, which are funded by the European Commission and the GSRT of Ministry of Development.

devices, achieving higher frequencies and consuming less energy. A typical interconnection network of FPGA consists of: (a) the wire segments and (b) the Switch Boxes (SBs). Moreover, the components of the total power consumption are: (a) the dynamic power and (b) the leakage power. More specifically, the dynamic power dissipation is proportional to the wire interconnection capacitance. Also, due to the fact that the wires have more capacitance compared to SBs, the proposed methodology targets first to minimize the impact of segments to the total power and secondly to minimize the capacitance associated with SBs.

In this paper, we propose a novel methodology for designing a high-performance and low-energy interconnection structure of an island style-based FPGA platform. The main goal of the new methodology is to find out the appropriate segment length, as well as the associated optimal combination of multiple SBs, taking into account the considered application-domain characteristics. The efficiency of a wire segment and SB is characterized by analyzing parameters such as energy dissipation, performance, and the minimum number of required routing tracks. We made an exhaustive exploration with all the kinds of MCNC benchmarks [3] (i.e. combinatorial, sequential and FSM), to find out both the optimal segment length for minimizing the Energy×Delay Product (EDP) of a conventional FPGA, as well as the optimal combination among three existing SBs, i.e. Wilton [1], Universal [1] and Subset [1], assuming the selected segment. Also, the optimal SB combination is found under the EDP criterion considering the heterogeneous (or proposed) FPGA architecture. The methodology provides the optimal ratio among the different chosen SBs. Having EDP as a selection criterion, we proved that the optimal segment length is the L4 for all SBs, and the SB combination “Subset-Universal” is the optimal one for the chosen segment.

The paper is organized as follows. In Section 2, the proposed FPGA interconnection architecture composed by longer segment wires and multiple SBs, as well as the exploration procedure for specifying them is described. Section 3 presents the comparison results, while conclusions are summarized in Section 4.

2 Proposed FPGA Interconnection Architecture

In this section, we discuss the spatial information of Switch Box (SB) connections as well as the usage of longer segments and their impact on the FPGA interconnection architecture. For that purpose, we introduce a new method for deriving special maps, each of which describes the number, as well as the location (spatial) of used transistors within a SB. In order to build these maps, MCNC benchmarks, the EX-VPR tool [2] and a Virtex-like FPGA architecture [8] were used.

The first step of the methodology is to find out the connectivity, the performance, the energy and the area requirements of MCNC benchmarks. For that purpose, a specific map (or 3-D curve) can be created for each design parameter which shows the parameter variation across the whole FPGA device. In particular, Fig. 1 shows the overall connectivity of the whole FPGA. It can be seen that the connectivity varies from point to point of FPGA. If we define a certain threshold of the connectivity value, and project the diagram to (X,Y) plane of FPGA, we create maps for connectivity requirements.

Considering connectivity threshold equal to 2, Fig. 1 shows the connectivity requirements of MCNC applications mapped into conventional FPGAs. The connectivity is defined as the total number of connections (i.e., “ON” pass-transistors) that take place into the SB. The number of distinct regions is based only to the designer requirements. By increasing the number of regions, the FPGA becomes more heterogeneous, as it is consisted by more regions. On the other hand, this increase leads to performance improvement for the device, due to the better routing resources utilization. As we can see from the exploration results, the number of the connections is gradually decreases from the centre of the map to the borders. The connectivity requirement for more tracks in the center of the device than the I/O boundary elements depends on the chosen placement and routing algorithm [2].

The introduction of connectivity map is very useful instrument to FPGA device designers to specify the interconnection requirements over each (*x*,*y*) point of FPGA device. Determining the “hot spots” locations of FPGA device, the designer can concentrate his/her efforts for efficient device optimization on certain regions only, but not on the whole device.

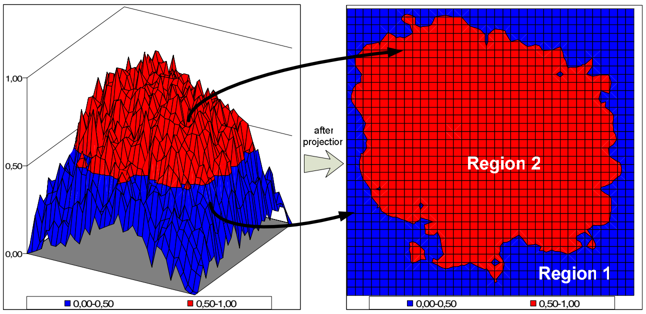


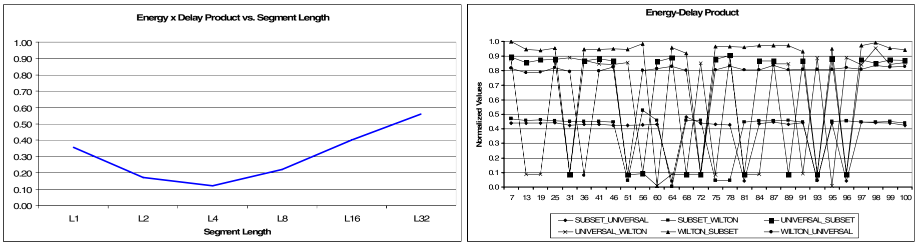
Fig. 1. Overall connectivity across the whole FPGA

The energy dissipation is critical issue of an FPGA design process. Since the power consumed in routing is more than 60% of total power of the FPGA device [9], the proposed technique aims at the minimization of this factor. For that purpose, we take into account the SB pass-transistors utilization in the various regions of FPGA map, shown in Fig. 1. Thus, in regions with smaller connectivity (i.e. fewer transistors) we can use appropriate type of SB with low-energy features. The connectivity degree of any (*x*,*y*) point of FPGA array is directly related with the energy consumption of (*x*,*y*) SB location, since less number of active SB connections means less energy consumption.

Furthermore, as we increase the number of distinct SB regions, the designer can identify in more detailed manner the spatial distribution of energy consumption and therefore, he/she can choose the most appropriate SB for each region at the expense of the increased heterogeneity of FPGA features. On the other hand, increase of the SB regions has a penalty at the fabrication cost of the device. For this work we choose to use an FPGA array with two distinct SB areas.

The second step of the proposed methodology is to determine the most appropriate wire length of a homogeneous FPGA, (i.e. considering the Subset, Wilton, and

Universal SBs). As it is mentioned, the selection of the optimal segment length is based on EDP criterion. Fig.2(a) gives the average variation of EDP curve for various segment lengths and the available SBs, where it can be seen that the segment L4 provides the minimal EDP. All the values at Fig. 2(a) and (b) are normalized to largest EDP value and they are the average values from all MCNC benchmarks. The horizontal axis represents the length of the routing wire segments, while the vertical one is the normalized value for a design parameter. It should be noted that the three curves (i.e. three SBs) are almost identical.



(a) (b)

Fig. 2. (a) EDP for different segments, (b) EDP for different SBs

Employing the spatial information regarding with a SB location, the rest of the paragraph provides detailed data about the selection procedure of the optimal combination of SBs, considering the EDP criterion. This is the 3rd step of the proposed methodology. Assuming two FPGA regions each of which uses a different type of SB, we performed exploration for all possible values, of *SB_ratio*'s:

$$SB\_ratio = \frac{SB\_Type\_1\,(\%)}{SB\_Type\_2\,(\%)} \tag{1}$$

where *SB_Type_Region*_1 and *SB_Type_Region*_2 denote the percentage of chosen SBs of *Region*_1 and *Region*_2. Fig. 2(b) shows the exploration results for EDP, assuming placement and routing to the smallest square FPGA with different ratios between the two distinct SBs. The values of the horizontal axis show the percentage of the first SB compared to the second one into the array. Moreover, having a combination {*SB_Type*_1} and {*SB_Type*_2}, the latter SB type is placed is an orthogonal located in the centre of FPGA, while the *SB_Type*_1 placed around the orthogonal up to I/O pads of FPGA. The exploration procedure was done by the EX-VPR tool, which can handle both the above three SBs and user-specified SBs [2].

We can deduct that the ratio 80%/20% of "Subset-Universal" combination, minimizes the EDP value. The Subset SB is assigned to *Region*_1, while Universal SB to *Region*_2 (center of FPGA). The aforementioned exploration procedure for the EDP can be also applied for the performance, the energy dissipation and the area requirements. Due to lack of space, we cannot provide the corresponding curves for these design parameters. However, we found that the proposed interconnection architecture provides the optimal results. It should be stressed that the primary goal of the proposed methodology is to prove that the usage of proper segment and combination of different properly-chosen SBs results into performance and energy consumption optimization.

3 Experimental Results

The proposed interconnection architecture was implemented and tested by a number of MCNC benchmarks. The chosen MCNC benchmarks are the twenty largest ones, and they were placed and routed in an island-style FPGA array [6, 8], using the EX-VPR tool. All the benchmarks were mapped to the smallest FPGA array. Table 1 shows the results for delay and energy both for homogeneous architectures and the proposed one with multiple SBs and segment L4. In the homogeneous FPGAs, the whole device is composed by only one of the available SBs (Subset, Wilton or Universal). Since our primary goal is the design of *both* high performance and low energy FPGA architecture, we choose the optimal EDP value from the exploration results (Fig. 2(a) and (b)) for our exploration results.

Table 1. Comparison results between the proposed FPGA architecture (with multiple-SBs & L4 segment) and single SB FPGA architectures in terms of delay and energy

Bench-mark	Subset		Wilton		Universal		Multiple SBs + L4 Architecture	
	Delay $x10^{-8}$	Energy $x10^{-9}$	Delay $x10^{-8}$	Energy $x10^{-9}$	Delay $x10^{-8}$	Energy $x10^{-9}$	Delay $x10^{-8}$	Energy $x10^{-9}$
alu4	9.77	5.83	9.74	5.77	10.9	5.84	4.29	5.36
apex2	9.37	6.75	9.35	6.66	13.1	6.94	5.24	6.66
apex4	8.86	4.17	8.86	4.17	10.3	4.24	4.51	4.10
bigkey	7.71	8.05	6.97	8.04	6.26	7.87	3.10	7.34
clma	15.1	11.4	14.8	10.3	14.9	10.5	9.30	7.65
des	8.29	10.4	9.06	10.2	8.65	10.2	4.56	9.83
diffeq	6.15	3.51	6.51	3.45	6.03	3.43	5.46	3.55
dsip	7.99	7.82	7.93	7.63	7.93	7.61	3.66	6.75
elliptic	10.9	12.4	10.9	12.0	12.3	12.4	7.15	12.8
ex5p	9.26	4.32	10.6	4.32	9.55	4.27	4.31	4.52
ex1010	18.3	16.2	17.4	15.4	26.4	17.1	8.16	15.6
frisc	16.1	12.1	15.7	11.0	15.8	11.2	9.89	8.14
misex3	11.7	5.55	11.8	5.49	10.1	5.28	4.08	4.66
pdc	20.4	22.3	23.9	21.6	15.2	18.7	7.58	18.2
s298	13.4	6.88	13.8	6.78	13.4	6.92	8.24	6.95
s38417	10.3	22.6	10.3	22.7	9.91	22.5	5.80	23.4
s38584	9.59	19.5	9.59	18.9	9.59	19.0	4.70	19.1
seq	12.4	6.56	18.4	7.18	8.87	6.10	4.43	6.28
spla	15.6	13.1	17.5	12.7	19.0	13.1	6.54	11.4
tseg	5.55	3.18	6.72	3.19	6.03	3.15	4.82	2.96

It can be seen that the proposed method achieved significant reduction in EDP of average about 55%, reasonable gain in performance up to 52%, energy savings up to 8%, at the expense of increase channel width by 20%. The reported gains that reported results from the average value of partial gains of the proposed architecture to each single-SB architecture. We have to point out that during the exploration procedure we used the optimal channel width for all the benchmarks and interconnection devices. It should be stressed that we achieved to design a high performance FPGA, without any negative impact on energy, although high performance circuit means high switching activity and eventually increased energy.

4 Conclusions

A novel FPGA interconnection methodology for high speed and energy efficient island-style FPGA architectures was presented. Using appropriately, the spatial information of various FPGA parameters, a new routing architecture with multiple-SBs and segment length L4 was designed. Using the minimal EDP value, the comparison results proved that heterogeneous FPGA platform outperforms with a conventional FPGA. More specifically, delay reduction up to 52% and energy savings up to 8% were achieved. Furthermore, the design of the new FPGA architecture is fully software-supported approach.

References

1. G. Varghese, J.M. Rabaey, "Low-Energy FPGAs- Architecture and Design", Kluwer Academic Publishers, 2001.
2. K. Siozios, et al., "An Integrated Framework for Architecture Level Exploration of Reconfigurable Platform", 15th Int. Conf. FPL 2005, pp 658-661, 26-28 Aug. 2005
3. S.Yang, "Logic Synthesis and Optimization Benchmarks, Version 3.0", Tech.Report, Microelectronics Centre of North Carolina, 1991
4. K. Leijten-Nowak and Jef. L. van Meerbergen, "An FPGA Architecture with Enhanced Datapath Functionality", FPGA'03, California, USA, pp. 195-204, Feb. 2003
5. V. Betz, J. Rose and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs", Kluwer Academic Publishers, 1999
6. http://vlsi.ee.duth.gr/amdrel
7. http://www.xilinx.com/products/silicon-solutions/fpgas/virtex/virtex4/overview
8. Deliverable Report D9: "Survey of existing fine-grain reconfigurable hardware platforms," AMDREL project, available at http://vlsi.ee.duth.gr/amdrel/pdf/d9_final.pdf
9. Guy Lemieux and David Lewis, "Design of Interconnection Networks for Programmable Logic", Kluwer Academic Publishers, 2004

Partial Data Reuse for Windowing Computations: Performance Modeling for FPGA Implementations[*]

Joonseok Park and Pedro C. Diniz

In-ha University, Dept of Computer Science and Engineering
Nam gu Yonghyun 4 dong, Hitech center 1012,
Inchon, Republic of Korea, 402-751
joonseok@inha.ac.kr
Department of Information Systems and Computer Engineering
Instituto Superior Técnico, Technical University of Lisbon
Tagus Park, Porto Salvo 2780-990, Portugal
ped@dei.rnl.ist.utl.pt

Abstract. The mapping of applications to FPGAs involves the exploration of a potentially large space of possible design choices with long and error-prone design cycles. Automated compiler analysis and transformation techniques aim at improving the design productivity of this mapping process by reducing the design cycles while still leading to good desigs. Scalar replacement, also known as, register promotion, leads to designs that reduce the number of external memory accesses, and thus reduce the execution time, by the use of storage resource. In this paper we present the combination of loop transformation techniques, namely loop unrolling, loop splitting and loop interchange with scalar replacement to enable partial data reuse on computations expressed by tightly nested loops pervasive in image processing algorithms. We describe an accurate performance modeling in the presence of partial data reuse. Our experimental results reveal that our model accurately captures the non-trivial execution effects of pipelined implementations in the presence of partial data reuse due to the need to fill-up data buffers. The model thus allows a compiler to explore a large design space with high accuracy, ultimately allowing compiler tools to find better design than using brute-force approaches.

Keywords: Field Programmable Gate Arrays (FPGA), Reconfigurable Computing, data reuse, scalar replacement, loop splitting, loop interchange.

1 Introduction

Scalar replacement, also known as register promotion, allows for performance improvement by the elimination of external memory accesses. The elimination of these accesses comes at the cost of storing in hardware (typically in registers) the data from memory locations that are frequently accessed. In the context of configurable or custom computing machines, this storage is implemented using internal registers or

[*] This research is supported by the Inha University Research Grant 34394.

internal RAM blocks. The fundamental distinction between the use of scalar replacement for configurable machines and cache-based architectures is that in the former the location and the data replacement policy has to be done explicitly, *i.e.* the designer must know at all times where each data item is store for it to be reused in the future, and when it is no longer need so that it can be discarded and the corresponding storage to be reclaimed.

Loop transformations, most notably loop interchange and loop unrolling, are important to expose vast amounts of fine-grain parallelism and to promote data reuse. By unrolling and interchanging the loops in a loop nest the amount of data that can be reused and the corresponding storage requirements needed to capture that data reuse varies drastically. The variety and interaction between these loop transformations, thus creates a large design space with radically distinct area and time trade-offs that are far from trivial to the average designer. To make matters worse, when targeting FPGA devices the amount of hardware resources, and thus of storage space is limited.

In this paper, we describe a notion of partial data reuse to mitigate the adverse effects of the potentially large number of required storage in the presence of scalar replacement. We borrow the notions of loop iteration space developed in the context of parallelizing compilation techniques, and develop a execution time model that captures important aspects of the execution time of the hardware design resulting from the application of various loop transformations. We can thus estimate the performance of designs in the aspects of both of resource utilization and of performance. Our preliminary results reveal that our approach, scalar replacement and loop transformations for partial reuse, combined with accurate performance modeling, explores more design space with high accuracy and ultimately able to find better designs than contemporary approaches.

This paper makes the following specific contributions:

- It explores the impact of *partial scalar replacement* in FPGA design space exploration in the context of limited register and Block RAMs.
- It describes the application of the loop transformations, such as loop splitting and loop interchange, that changes the data reuse vector, and thereby changes the impact of scalar replacement both in design area and performance.
- It presents a performance modeling approach that combines analytical and estimation techniques for complete FPGA designs.
- It validates the proposed modeling for a sample case study application on a real Xilinx® Virtex4™ FPGA device. This experience reveals our model to be very accurate even when dealing with the vagaries of synthesis tools.

Overall this paper argues that performance modeling for complete designs, either for FPGA or not, is an instrumental technique in handling the complexity of finding effective solutions in the current reconfigurable as well as future architectures. This paper is organized as follows. In section 2 we describe the motivating example to illustrate our approach, and we describe related work in section3. In section 4, and 5 compile time analysis and analytical performance modeling approach are described in detail, respectively. In section 6 we present experimental results for a sample image processing kernel. In section 7 we discuss about future work and conclude in section 8.

2 Motivating Example

We now present an application example that highlights our approach. Fig. 1 depicts an implementation of the Sobel edge detection algorithm written in C. The computation traverses a 2-dimensional array using a 3-by-3 “window”.

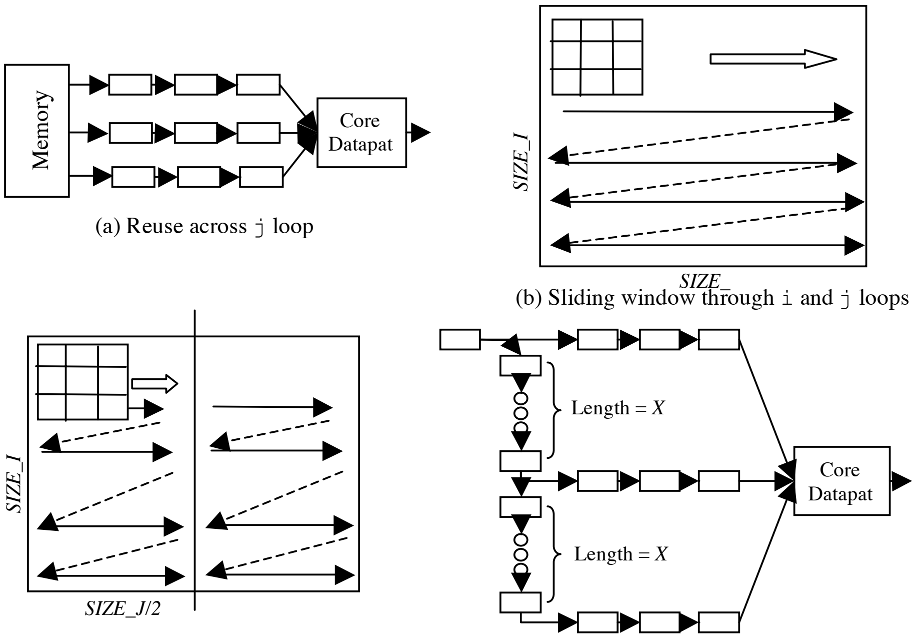
```
for (i= 0; i < SIZE_I - 2; i++) {

  for (j= 0; j < SIZE_J - 2; j++) {
     uh1 = (((- img[i][j]) + (- (2 * img[i+1][j]))+(- img[i+2][j]))
              + ((img[i][j+2]) +
              (2 * img[i+1][j+2])+(img[i+2][j+2])));
     uh2 = (((-img[i][j]) + (img[i+2][j])) +
            (-(2 * img[i][i+1])) +(2*img[i+2][j+1]) +
            (- img[i][j+2]) + (img[i+2][j+2])));
     if ((abs(uh1) + abs(uh2)) < threshold)
       edge[i][j]="0xff";
     else
       edge[i][j]="0x00";
  }

}
```

Fig. 1. Sobel Edge Detection core using a two-level nested loop

The inner loop of this algorithm reads 8 `img` array elements. However, if `j` is not zero, then 5 out of 8 array elements have been accessed in the previous iteration. Using a sophisticated data reuse analysis, a compiler tool can identify the reuse vector for scalar replacement, which is (0,1), (0,2) for the `j` loop in this example. It is possible to compute at compile time analysis the amount of register required to exploit a specific reuse vector, 6 registers in this case [2]. The design implementation after scalar replacement can be achieved by using 3 sets of shift registers; organized as tap-delay-line structures, whose 8 elements can be accessed concurrently. Data in a tap-delay-line is shifted along at specific iterations, as shown in fig 2. (a).

The compiler analysis can also identify data reuse across the `i` loop, by uncovering reuse vectors (1,*), (2,*). If we apply data reuse across `i` loop by saving whole row of image, the required registers for tap-delay-lines would be proportional to the size of image column, as shown in figure 2. (d). We can use registers to save all the reused data or take advantage of Block RAM if available. By exploiting data reuse, inside the loop body, only one data access is required, instead of three in (a). However, for large sized inputs, there would always be storage limitation unit in the FPGA device. An alternative solution to this capacity issue is to split the `j` loop, say in half and interchange the `i` and `j` loops. These transformations would reduce the required register (or BlockRAM) for scalar replacement and the reuse distances associated with reuse distance vectors would be halved. As a consequence the area usage in the resulting FPGA design is dramatically reduced.

Loop splitting and interchange and the resulting control of the flow of the data in the implementation leads to two non-trivial complications in the implementation. First, loop splitting introduces a discontinuity in the loop iteration space. By dividing `j` loop into 2 loops, the number of discontinuities will be increased by a factor of 2,



(a) Reuse across `j` loop

(b) Sliding window through `i` and `j` loops

(c) Splitting the `j` loop and interchanging with `i`

(d) Reuse in both `i`, `j` loop, X is *size_J* before in (b) implementation and *size_J*/2 in (c) implementation

Fig. 2. FPGA implementations and possible iterations in input images

which will lead to some performance loss. Second, and in the context of a pipelined implementation with tapped-delay lines, the implementation must incur a long cycle overhead to fill-up the many registers in a tapped-delay line. This overhead is increasingly large for longer tapped-delay lines which are tied to the maximum data reuse distance. These issues can be mitigated by reducing the reuse distance or by shortening the loop counts. This will, however, lead to a lower storage utilization efficiency as relative to the time it takes to reuse the data, the implementation spends more time accessing memory to load the tapped-delay lines.

In this paper we describe an approach that uses area and performance modeling to assess the benefits and cost of the use of splitting and interchange in the presence of scalar replacement for computations that take advantage of these tapped-delay line storage structures. The approach presented here targets windowing computations where the data reuse patterns can be uncovered by existing data dependence analysis techniques.

3 Related Work

The main focus of existing data reuse analysis has been on the detection and translation of data reuse into data locality, as exploited by contemporary cache-based architectures

[5]. Most of the work has focused on high-end computing and has thus ignored the issue of precise data accesses. So and Hall have extended the work by Carr and Kennedy [1] for scalar replacement targeting FPGAs. They focus on identifying the number of accesses during a given computation by analysis of the arrays index subscript functions [2]. This work exploits full data reuse and has never addressed the issue of limited capacity from the perspective of reducing the iteration spaces by application of tiling and loop splitting. In our previous research, we have exploited the opportunities of scalar replacement by using both BlockRAM and registers in FPGAs [3]. In this work, however, we never exploited the space/performance trade-off associated with partial reuse using loop tiling and splitting.

Weinhardt and Luk emphasized the importance of loop transformations in FPGA synthesis. They proposed a partial unrolling scheme using loop tiling and loop unrolling based on initial resource utilization of the statements in the loop [14]. We used same combination of loop transformation for different purpose, which is control of data reuse factor. Their other paper presented the optimization scheme to manage the hierarchy of storage unit based on compile time data reuse analysis focusing on the implementation of shift register and mapping between arrays and on-chip/off-chip memory [15]. The inclusion of data dependence vectors and thereby choice of BlockRAM mapping and analysis on overall performance and area utilization distinguish their researches from our work.

Other researchers have also recognized the potential of window-based computations as its data reuse properties. Guo, *et. al.* propose a simplified HDL code generations by capturing data reuse pattern at compile time [7]. They call "Window Operation" if there exist window set, managed set, and killed set to map hardware buffer in the application, automate code generations for it. Their research focuses on automating hardware generations rather than in the transformation and optimization of applications. Yu and Lesser described a performance estimation and optimization technique in special "sliding window operation" [8]. They focus on effective utilization of FPGA-on-chip memory under given constraints, such as area or bandwidth.

Regarding the application of loop transformations in the mapping of computations to FPGAs most of the work has naturally focused on loop unrolling to expose additional instruction-level parallelism (ILP). In addition to the replication of operators, loop unrolling also exposes many data references that can be reused either within a single iteration of the unrolled loop body or even across iterations. Researchers have also exploited loop fission [10] and loop dissevering [11] to temporally partition the execution of loop whose bodies require an excessive amount of FPGA resources.

Other authors have addressed the issues of memory bandwidth and data partition as FPGAs offer the opportunity of custom memory layouts tailored to a specific computation. Kandemir and Chouldhary [8] describe a memory design algorithm that uses reuse vectors to determine the required capacity of local memory for the embedded system. Bairagi *et. al.* developed an array-memory allocation algorithm based on the footprint of each array reference within a loop nest [12].

4 Compiler Analysis and Loop Transformation

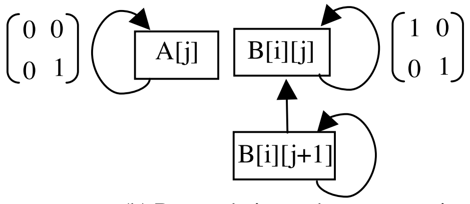
We now present a compiler analysis, whose goal is to discover the maximized internal data reuse. We now highlight loop transformations that help a compiler to adjust the amount of possible data reuse a kernel exhibits. We assume a perfectly nested loop with compile-time known loop bounds.

4.1 Data Reuse Analysis and Reuse Vectors

In order to capture the data reuse in loop nests we take advantage of compile time data dependence analysis. There exists a data dependence between two array references if they read/write the same array elements in the different loop iteration. It is called a self (-temporal) reuse if reuse is induced by the same array reference in two distinct iterations. It is called a group (-temporal) reuse if it is induced by two distinct array references in two loop iterations. In the example in Figure 3(a) there exist self reuses for reference `A[j]` and group reuse between references `B[i][j]` and `B[i][j+1]`.

```
for (i=0;i<M;i++){
  for (j=0;j<N;j++){
     ......... A[j]
     ......... B[i][j]......
     .........
```

(a) Self (A) and group (B) data reuse example code


$\begin{pmatrix}0 & 0\\0 & 1\end{pmatrix}$
A[j]
B[i][j]
$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$
B[i][j+1]

(b) Reuse chains and access matrix

Fig. 3. Sample code and reuse chain

Given a reference A, the compiler identifies its access matrix, in this case H = [(1,0),(0,1)]. Its self-reuse occurs for distinct iterations that satisfy the reuse equation, given by the ker(H). The solution to this equation can be characterized by a set of reuse vectors, the basis of the reuse space. In the case of group-reuse, the equation must satisfy I2 – I1 = ker(H) + C, where C is a constant matrix. Through this reuse analysis, we can identify reuse vectors as well as distance vectors for self- and group-reuse. The distance between dependent accesses, found by traditional data dependence analyses, in an n-level loop nest is captured by an n-dimensional dependence vector $D=<d_1, d_2, \ldots, d_n>$, where d_i is constant, a '+' (all positive) or *(unknown). The distance vector usually represented as a form of directed graph, called a reuse chain [2]. Details of the analysis to determine reuse vectors and distance vectors for single-induction-variable (SIV) and Multiple Induction Variables (MIV) is described for instance in [3].

4.2 Scalar Replacement

For a reuse chain that exclusively exploits data reuse in array references with single induction variables, the expressions below determine the number of distinct array elements accessed between reuses. Here i_k denotes the iteration count of the kth loop and d_i denotes the ith element of the distance vector d. These expressions described in detail in [2] and [3] provide an analytical model to calculate the number of required registers, for the self-reuse and group-reuse cases. The key aspect is that the number of required registers depends on the reuse distance and hence of the reuse vectors in addition to the loop bounds.

$$\begin{gathered}\alpha_s\ D, i, n) = \prod \quad (d_l, x) \\ \text{where} \quad (d_1, x) = \ x, \text{ if } d_i \text{ is a constant, 1 otherwise}\end{gathered} \tag{1}$$

$$\begin{gathered}\alpha_g(D, i, n) = \sum\{(\prod i_k)*\Gamma(d_1)\} + \Gamma(d_n)\} \\ \text{where} \quad (d_x) = \ c, \text{ if } d_i \text{ is a constant, 1 if positive unkown,0 otherwise}\end{gathered} \tag{2}$$

4.3 Strip-Mining and Loop Interchange

Strip-mining and loop interchange or loop tiling, decomposes single loop into two loops; the outer loop steps over blocks of consecutive iterations – the control looo, whereas the inner loop steps between single iterations within a block. Loop tiling thus increases the number of loops in a nest and changes the iteration space, the data dependence vector, and consequently the reuse distance vector associated with data reuse of an array reference. Compared to the original loop iterations, whose bound is say N, if we tile a loop by a factor of K, the reuse distance will be reduced by a factor of K, as it will be the number of required registers to capture the reuse of an array reference. This reduction of reuse distance and thus of required registers will therefore therefore increase the possibility of the resulting hardware design to meet FPGA area constraints.

5 Performance Modeling

We now present our target hardware structure that serves as the basis of our FPGA designs. We then provide an analytical performance model for this structure. The model handles the performance estimation for scalar replaced design using the loop transformation presented in the previous section.

5.1 Hardware Structure

The hardware structure underlying our FPGA design implementation consists of two primary components as shown in figure 4. The first part is datapath, which implements the core computation of application. The datapath consists of pure computational discrete logic and FSM which generates control signals for internal/external data accesses. The second part is the configurable memory interface. This interface generates physical memory address and protocol signals for both input and output data. The details of the proposed architecture are described in [4]. Our model exploits the pipelined execution of memory accesses as well as the core datapath.

The memory units inside datapath design are either registers or BlockRAM, which will store scalar replaced value of array data. We can select register or BlockRAM for a given data reuses based on the resource constraints [4]. Other than the combinatorial logics to implement the computation defined in the application, the FSM controls the execution of the datapath as well as the I/O port activity. The other part, the memory interface should fetch the data (from external memory) and provide it to datapath input port (or vice versa for an output port). To adequately support this model, our memory interface design includes the concepts of streamed data channels. Associated with each channel the memory interface includes resources to generate the corresponding sequence of memory addresses – address generation unit (AGU).

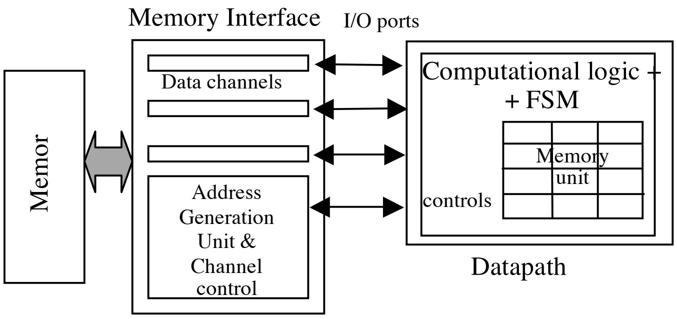


Fig. 4. FPGA Hardware structure model for a generic application

5.2 Parameters for the Performance Model

Using our model, a compiler can estimate the number of execution cycles of a given hardware design in the presence of pipelined execution of datapath and memory interface. To estimate the overall performance we separate the estimates for the datapath and memory activity cycles. The important parameters for the performance modeling of the datapath are its initiation interval, iteration count, and the number of pipeline stages. The initiation interval and pipeline stages are implementation dependent; hence, we can use high-level synthesis results after synthesis stage. If the pipeline stage is long or there exists a long reuse chain to exploit scalar replacement, the input data is loaded *cold* as the data in the tapped-delay line from a previous iteration is not valid. As such we need to fill up all of

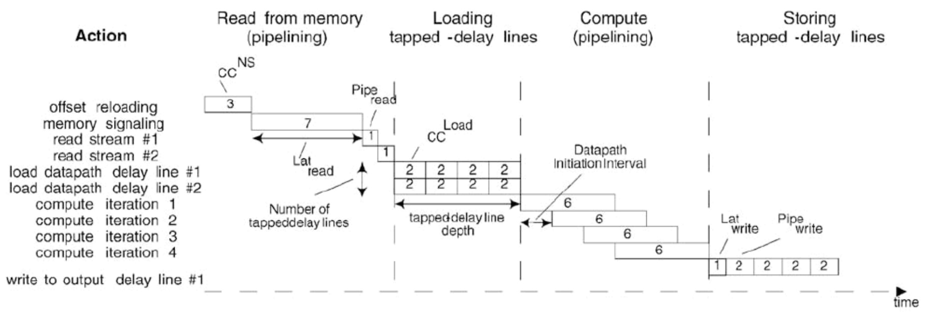


Fig. 5. Execution modeling parameters illustration

pipeline stages before we get any valid data. Since we implement tapped-delay-line in the datapath, the depth of tap-delay-line should be added to this latency cycles. This may occur whenever the data reuse chain is reset, which only occurs at the iteration space boundaries.

The identification of reuse chain reset at the boundary of iterations is very important in the estimation of memory activity. In a streamimg data accesses the AGU only needs to access memory addresses in a sequential incremental mode. However, at the loop boundary the AGU must change its address generation patterns. Setting up and resetting memory interface components whenever the datapath requires data from channels incurs an overhead. As to the memory interface, its important parameters are the *reloading overhead*, the *read* and *write latencies* and the *setup* for reading and writing in pipelined mode. Hence the estimate of the total number of execution cycles includes the number of continuous and discontinuous memory accesses as well as datapath execution cycles. We can summarize the performance estimation as follows:

$$\begin{aligned}\textit{(Total Cycles)} &= \textstyle\sum(\textit{external memory cycles(Mem)} + \textit{computation cycles(CC)} \\ &\qquad + \textit{non\_continuous overhead(NC)}) \\ &= \textstyle\sum(\textit{total\_memory\_access} + \textit{memory\_reload\_overhead} \\ &\qquad + \textit{iteration\_count} * \textit{initiation\_interval} \\ &\qquad + \textit{pipline\_stages} + \textit{pipeline\_reload\_overhead})\end{aligned} \tag{3}$$

Most of the parameters of equation (3) are known at compile time using traditional compiler analysis or high-level synthesis. The key issue is how to identify iteration space discontinuity to calculate pipeline stage stalls (reload_overhead) and memory access discontinuity by analyzing the shape of loop nests and array indexes. At this stage of research, we assume there is control discontinuity across loop boundary and memory access discontinuity as well. This is conservative assumption for memory access overhead, since some of array might be continuous across the different depth of loop nests because of its layout and index. However, we can provide an upper bound of performance estimation with this scheme. The application of scalar replacement shown in sections 4.1 and 4.2 will reduce total memory accesses while increasing slightly the number of pipeline stages.

A key aspect in the performance estimation is the impact of strip-mining. As described in section 4.3 strip-mining to the loop increases the loop nests, as a consequence, the reuse (distance) vector also changes. Since the loop bound and iteration space is transformed, pipeline/memory reload overhead from the formula will be increased because of the newly introduced control loops in the loop nest. However, the total iteration count is unchanged in strip mining, and so is the number of pipeline stages. Memory access count will not be changed by strip-mining; however, combined with loop interchange it will be dramatically changes since it changes the data reuse amount. Loop interchange itself will not change total any of parameters for performance estimation.

6 Experimental Results

We now evaluate the performance model outlined about in the context of scalar replacement using loop interchange and loop strip-mining for a small set of kernel codes – the Sobel edge detection (SOBEL), and the Binary Image Correlation (BIC). The source code of SOBEL is depicted in figure1 whereas figure 2 depicts a possible implementation. The source code for BIC is shown in figure 6. For its hardware implementation we fully unroll the `x` and `y` loops. The entire `mask` array is reused across iterations of the the `i` and `j` loops. For the `img` array, `size_x` times`(size_y-1)` elements will be reused across iterations of the `j` loop using `size_x` number of shift registers.

```
for (i= 0; i < SIZE_I; i++)
  for (j= 0; j < SIZE_J; j++){
      res = 0;
      for (x = 0; x < size_X; x++) // fully unroll
          for (y = 0; y < size_Y; y++) {  // fully unroll
            if (mask[x][y] > 0) and (mask[x][y] < img[i+x][j+y]))
               res += abs(img [i+x][j+y]);
          }
     result_img[i][j]=res;
  }
```

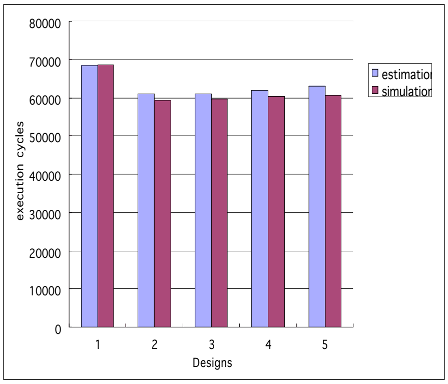
Fig. 6. Souce code for the kernel of BIC

We have implemented five versions of scalar replaced datapath designs for SOBEL and BIC corresponding to different choices of which loops to split and interchange. For BIC we apply full loop unroll to the inner most two loops. Both computational kernels scan through 2 dimensional iteration space. The base versions simply apply data reuse in the inner-loop. We designate the five versions as D1 through D5 and apply scalar replacement for outer loops, fully or partially as follows.

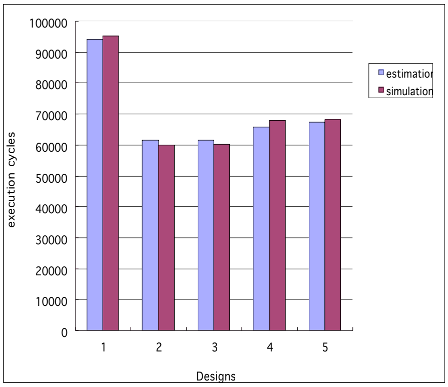
- D1 : scalar replaced with only j loop.
- D2 : scalar replaced with both i and j loop using register only (*fully reuse across i loop*)
- D3 : scalar replaced with both i and j loop using blockRAM
- D4 : loop split j loop, interchange j_outer with i loop, scalar replaced with i and j_inner loop (*partially reuse across i loop*)
- D5 : data reuse using blockRam to D4.

We implement the datapath of these codes directly in standard behavioral VHDL description and take advantage of the structural memory interface units described in [12]. We synthesized the resulting designs using the Xilinx® ISE 8.1 TM targeting a Virtex4TM device. We simulated the resulting design using Mentor Graphics TM ModelSim TM to measure the execution cycles and compare to the estimated performance. Estimated cycle count is calculated through equation **(3)**. We used 128-by-128 input images for both kernels and 8-by-8 mask input for BIC.

As expected, all scalar-replaced versions for `i` and `j` loops, show improved performance over design D1, which is scalar-replaced only for `j`-loop. Design D3 (and D5) usess BlockRAMs instead as a reuse buffer. The iteration spaces are identical as in D2 (and D4). Designs D2 and D4 differ in iteration space. We obtain D4 from D2 by splitting the `j` loop, followed by `i` and `j` control loop interchanging, as shown in fig 2. (b) and (c). Our estimation works very well and shows approximately a 5% gap relative



(a) Sobel Edge Detection



(b) Binary Image Correlation

Fig. 7. Comparison of performance estimation and simulation results

to the simulation results for all designs. Designs D2, and D3 are better than designs D4 and D5 performance-wise, as they fully reuse data across the `i` and `j` loops and exhibit the least amount of discontinuity in their execution. Designs D3 and D4 reuse data across the `i` and `j` loops, however, they incur a discontinuity of memory access twice as oftern as designs D2 and D3. The reductions in execution cycles mainly result from the reduced external memory accesses.

Table 1. Synthesis results targeting for Xilinx Virtex4 FPGA device

Designs		Datapath only					Complete design			
		LUTs	LUT (S-reg)	BlkRam (16Mbit)	Equiv. Gates	Clk (MHz)	LUTs	Equiv. gates	Clk (MHz)	W.Time (μsec)
SOBEL	D1	229	0	0	3,035	421	570	8,704	253	271.4
	D2	499	200	0	17,066	337	801	22,496	273	217.0
	D3	311	0	1	4,180	367	701	10,140	264	225.9
	D4	407	100	0	10,713	291	704	16,114	259	232.9
	D5	312	0	1	4,175	378	707	10,168	279	217.1
BIC	D1	828	0	0	16,826	169	1786	37,374	169	563.5
	D2	921	128	0	25,237	166	1916	45,706	166	361.1
	D3	679	0	2	17,237	174	1778	37,454	174	346.7
	D4	740	64	0	21,411	167	1854	41,622	167	406.0
	D5	679	0	2	17,229	174	1794	37,446	174	391.1

Table 1 presents the synthesis results for the 5 designs. Design D2 which uses full registers for data reuse requires more area than any other design, as it needs storage units for 2 full-lines of the input image. Using BlockRAMs instead, we can reduce the register utilization dramatically. Design D1 is the most compact design in area usage, while it is the worst performing design of all. The results reveal that we can find the D4 and D5 design as a compromising point as they exhibit better performance than D1 and require a smaller slice count than D2. All simulation and synthesis results are predicted by our compile time analysis using reuse vector comparison and loop transformations.

Table 1 summarizes the synthesis results using Xilinx™ ISE 8.1 and targeting a Virtex4™ device(XC4V1x15 sf363). As expected D2 is the largest design as it uses a large amount of shift registers, exhibiting a very comparable performance to the other designs. Designs D3 and D5 require less LUTs than D2 and D4, as registers are counted as LUTs (and total gates), and BlockRAMs are not.

Comparing designs D2 and D4 we can identify the impact of partial data reuse (loop splitting followed by loop interchange) in actual design implementation. D4 requires only half of registers used of D2 with respect to the loop transformation. Since the computational resource and control logic of both designs are almost equivalent, the difference on area usage is the impact of allocated registers. As a BlockRAM macro cannot be partitioned in Virtex 4™, 16K bit DPRAM block are assigned to both D3 and D5 registers, while D5 use only 128 entries of BlockRAM and D3 uses 256 entries. If we use larger input images, the amount of registers (or DPRAM entryies) would increase. Hence, the reduction of number of registers by partial-reuse will provide more choices in design space explorations in the performance, area trade-off curve.

7 Discussion and Future Work

The results reveal there is only a slight performance improvement in `i-j` loop scalar replaced versions (D2) over `j`-loop versions (D1). This is due to a good balance between memory accesses and computation in the loop body. While for SOBEL, D1 accesses three 8-bit elements per iteration, it is still less than 1 cycle pipelined 32-bit external memory accesses. Applying scalar replacement for the designs that stress the external memory, the impact on the performance gap will be bigger.

One limiting aspect of our apporach is the simplified performance estimation model. We assumed there always exists a memory access discontinuity at the loop boundaries. For the estimation accuracy, we should consider data layout with loop boundary. The parameters, such as memory latency, reload overhead may vary on diverse system. The memory access latency or other parameters to estimate overall performance is system dependant, hence the impact of loop transformation will vary the actual results on the specific system. One other hand, overlapping of memory access time and datapath execution is partially taken into account in our design and estimation. We need to enhance our memory interface design to enable data prefetching for full overlapping. BIC has 4 level nested loops and we only explored tiling of outer loop. Partially unrolling the inner most loops, would substantially increase the size of the design space to explore, as explored in our own previous research [4].

8 Conclusion

The efficient mapping procedure of an application to an FPGA requires designs to explore a wide range of mapping choices and program transformations. This is a long and error-prone process making the application of automatic analysis and tools extremely desirable. In this paper we have presented a technique that combines loop- and data-oriented transformations for the mapping of computations to hardware designs on FPGAs. We have presented a performance model that captures the execution of pipelined implementations of computations that exploit data reuse in internal storage structures such as tapped-delay lines and BlockRAMs. This model is sufficiently accurate to allow a compiler to explore a wide range of design choices when mapping computations onto FPGAs.

References

1. Carr, S., Kennedy.K.: Scalar Replacement in the Presence of Conditional Flow, Software-Practice and Experience, Vol. 24(1), (1994) 51–77
2. So, B, Hall, M.W.: Increasing the Applicability of Scalar Replacement, In the Proc. of the 2004 ACM Symp. On Compiler Construction (CC'04). ACM Press. (2004)
3. Baradaran, N, Diniz, P. C, Park, J.: Extending the Applicability of Scalar Replacement to Multiple Induction Variables, Lecture Notes in Computer Science, Vol. 3602 Springer-Verlag Berlin Heidelberg New York (2005) 455–469
4. Park, J, N, Diniz P.C, Shayee, S. K.R.: Performance and Area Modeling of Complete FPGA Designs in the Presence of Loop Transformations, IEEE Trans. on Computers, Vol. 53, No. 11, IEEE Computer Society Press (2004) 1420 – 1435
5. Wolf, M, and Lam, M.: "A Data Locality Optimization Algorithm", In Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI), ACM Press (1991)
6. Guo, Z, Buyukkurt, B, Najjar,W.:Input Data Reuse in Compiling Window Operations onto Reconfigurable Hardware, in Proc. of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM Press (2004)
7. Yu, H and Lesser, M.:Automatic Sliding Window Operation Optimization for FPGA-Based Computing Boards, in Proc. of the 14th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'06), IEEE CS Press, (2006) 76-88
8. Kandemir, M, Choudhari, A.: Compiler-Directed Scratch Pad Memory Hierarchy Design and Management. In Proc. of 2002 ACM/IEEE Design Automation Conference (DAC'02), IEEE Computer Society Press (2002)
9. Xilinx® Virtex-4 FPGAs User Guide Xilinx® (2006)
10. Kaul, M. Vemuri, R, Givindarajan, S and Ouaiss, I.E.: An Automated Temporal Partitioning and Loop Fission Approach to FPGA Based Reconfigurable Synthesis of DSP Applications, In Proc. IEEE Design Automation Conf. (DAC '99), (1999)
11. Cardoso, J.: Loop Dissevering: A Technique for Temporally Partitioning Loops in Dynamically Reconfigurable Computing Platforms, Proc. 10th Reconfigurable Architectures Workshop (RAW 2003) (2002)
12. Diniz, P, Hall, M, Park, J, So, B, and Ziegler, H.: Bridging the Gap between Compilation and Synthesis in the DEFACTO System, in the Proc. of the 14th Workshop on Languages and Compilers for Parallel Computing (LCPC'2001) (2001)..
13. Bairagi, D, Pande, S, and Agrawal, D.:Framework for Containing Code Size in Limited Register Set Embedded Processor, in Proc. of ACM workshop on Languages. Compilers and Tools for Embedded Systems (LCTES 00), ACM Press (2000)
14. Weinhardt, M., Luk, W.: Pipeline Vectorization, IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems. Vol. 20. No. 2, IEEE (2001) 234 – 248
15. Weinhardt, M., Luk, W : Memory Access Optimisaztion for reconfigurable Systems, In IEE Proceedings of Computer and Digital Tech., Vol. 148, No. 3, (2001)

Optimized Generation of Memory Structure in Compiling Window Operations onto Reconfigurable Hardware

Yazhuo Dong, Yong Dou, and Jie Zhou

School of Computer Science, National University of Defense Technology, Changsha, Hunan, China, 410073
{dongyazhuo,yongdou,zhoujie}@nudt.edu.cn

Abstract. Window operations which are computationally intensive and data intensive are frequently used in image compression, pattern recognition and digital signal processing. The efficiency of memory accessing often dominates the overall computation performance, and the problem becomes increasingly crucial in reconfigurable systems. The challenge is to intelligently exploit data reuse on the reconfigurable fabric (FPGA) to minimize the required memory or memory bandwidth while maximizing parallelism. In this paper, we present a universal memory structure for high level synthesis to automatically generate the hardware frames for all window processing applications. Comparing with related works, our approach can enhance the frequency from 69MHZ to 238.7MHZ.

1 Introduction

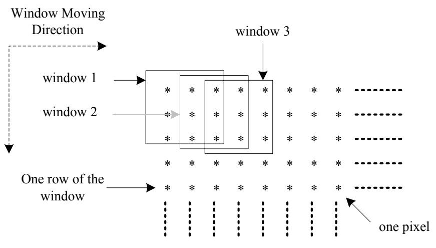
FPGA has become the medium of choice for fast hardware prototyping and a popular vehicle for the realization of custom computing machines that target multi-media applications. But developing programs that execute on FPGAs are extremely cumbersome [1]. To deal with the problem, high level synthesis (HLS) tools are developed to implement the hardware system using behavioral level languages, as opposed to register transfer level languages.

HLS tools can be classified into two approaches: the annotation and constraint-driven approach and the source-directed compilation approach. The first approach preserves the source programs in C or C++ as much as possible and makes use of annotation and constraint files to drive the compilation process, such as SPARK [2], Sea Cucumber [3], SPC [4], Streams-C [5], Catapult C [6] and DEFACTO [7]. The second approach modifies the source language to let the designer to specify, for instance, the amount of parallelism or the size of variables, such as ASC [8], C2Verilog [9], Handel-C [10], Handy-C [11], Bach-C [12] and SpecC [13] etc. All of these design automation tools aim to raise the level of design.

This paper concentrates on one class of applications called window operations. This kind of applications are widely used in signal, image and video processing and require much computation and data manipulation. Fig.1(a) shows the Sobel edge detection example code in C and the window operations are depicted in fig.1(b). All these operations have similar calculation patterns: a loop or a loop nest operates with array variables. There are multiple references to an array element in the same or a subsequent iteration. Thus, the memory structure can be designed to exploit data reuse.

```
char img[SIZE][SIZE], edge[SIZE][SIZE];
int uh1, uh2, threshold;
for (i=0; i < SIZE - 4; i++) {
    for (j=0; j < SIZE - 4; j++) {
        uh1 = img[i-1][j+1]+2*img[i][j+1]+img[i+1][j+1]-(img[i-1][j-1]
                +2*img[i][j-1]+img[i+1][j-1];
        uh2= img[i+1][j-1]+2*img[i+1][j]+img[i+1][j+1])-(img[i-1][j-1]
                +2*img[i-1][j]+img[i-1][j+1];
    if ((abs(uh1)+abs(uh2))<threshold)
            edge[i][j]=”0xff”;
        else
            edge[i][j]=”0x00;
    }
}
```

(a)



(b)

Fig. 1. Sobel edge detection algorithm in C code and the window operations

One of general-purposed approaches to exploit data reuse is to identify multiple memory accesses to the same memory location as reused data, and keep these data in a group of registers called smart buffer. It is an important issue how to organize the smart buffer and data layout effectively.

A number of efforts have been carried out to deal with the problem [14][15] [16][17][18]. Some traditional methods use a set of either vertical or exclusively horizontal queues to realize data reuse. They demand a good deal of registers which are the critical resources in FPGA. Since most of them ignored system level scheduling when dealing with external memories, there is a long memory latency to initialize array elements into the internal RAM blocks before starting the processing.

In order to overcome these problems, ROCCC [20][21][22] which is a reconfigurable computing compiler system, presents a new approach to the reuse of data when compiling window operations [19]. It use less number of registers, but it did not realize data reuse completely. Since ROCCC has to access the same location of off-chip memory multiple times, there exist a space for optimization in clock frequency.

In this paper we put forward a new approach to deal with these problems. We execute data accessing to off-chip and on-chip memory overlapping with calculation. We exploit data reuse fully, and at the same time keep the RAM blocks and smart buffer as small as possible.

This paper makes the following contributions:

– It presents a universal parameterized memory structure and a novel data scheduling scheme for window operations.

– It gives the algorithms about how to generate VHDL automatically according to some parameters which are obtained from the compiler.
– It illuminates experimental simulation results for the automatic translation of a set of window processing applications onto FPGA.

The rest of the paper is structured as follows. We compare different mapping approaches via an example in section 2. Next we describe the universal memory structure of our approach. Section 4 describes the VHDL code generation and the compiler support. Section 5 presents simulation experimental results for a set of window applications. In section 6 we give a conclusion.

2 Background

We now illustrate the use of different storage and control structures in the automatic mapping of an example computation onto a FPGA-based computing engine. The computation is Sobel edge detection written in C as depicted in fig.1(a).

A traditional strategy to reduce the number of required memory accesses is shown in fig.2. Smart buffer holds the data input queues to exploit the fact that consecutive iterations of the inner loop use data that previous iterations have fetched. This strategy layout all reused data in smart buffer, so it uses a large number of registers.

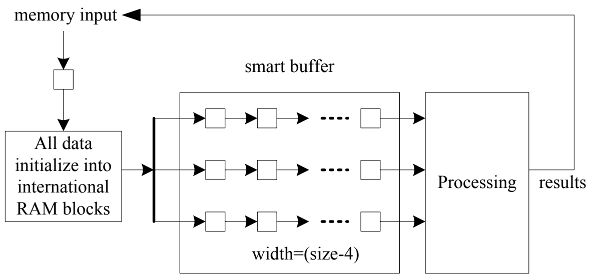


Fig. 2. Traditional strategy to exploit data reuse

The memory structure generated in ROCCC is illustrated in fig.3(a). The smart buffer receives data from the input memory directly, which use less number of registers. The input data used by each outer-loop iteration are loaded only once, but there are overlaps between adjacent outer iterations since they can not afford a smart buffer to hold whole rows of data. It has to access the same location of input memory multiple times which makes the processing speed slow. Fig.3(b) shows the FSM of smart buffer. We notice that ROCCC arranging data in smart buffer in different order at each cycle incurs the complex connection between smart buffer and processing elements. There are twenty-one states needed for Sobel program.

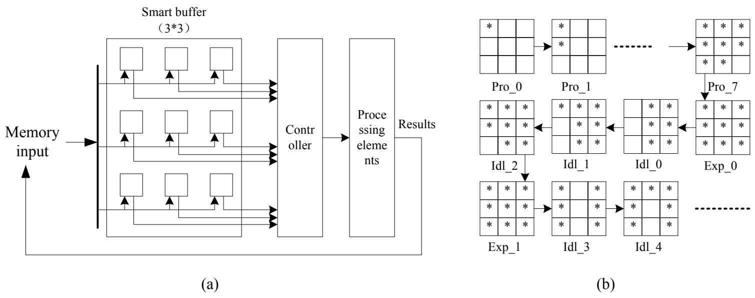


Fig. 3. ROCCC's approach to exploit data reuse and the FSM of smart buffer

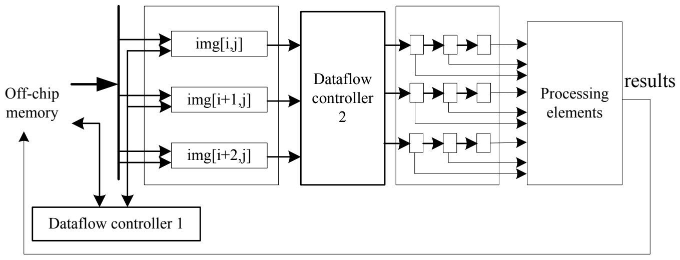


Fig. 4. Our approach to exploit data reuse

In this paper, we propose a novel memory structure, which uses less number of registers than traditional ways and obtains higher speed than ROCCC. Fig.4 shows our approach to resolve Sobel edge detection problem.

There are three RAM blocks with the depth of the inner loop dimension (SIZE-4) designed in the target architecture. One of them holds 1-dimensional img array data. Fig.5 illustrates FSM of the smart buffer. When the data of row i, i+1 and the first three data of row i+2 are ready, the whole processing can be started. In case the row i is done, the new input data of row i+3 will take the place of row i. We use shift registers during the whole processing. It can be noticed that we do not initialize all array elements into the on-chip memory before starting the processing. The structure execute in a data driven mode which means starting current operations as soon as possible. There are still pipelined off-chip memory accessing during the whole processing.

Comparing with traditional ways, we use less number of RAM blocks and registers to exploit data reuse. Comparing with ROCCC, we put all reused data in internal RAM blocks. When the data is used later, we can get it from the on-chip RAM blocks but not have to access the off-chip memory. Thus, the processing speed of our approach is faster than ROCCC.

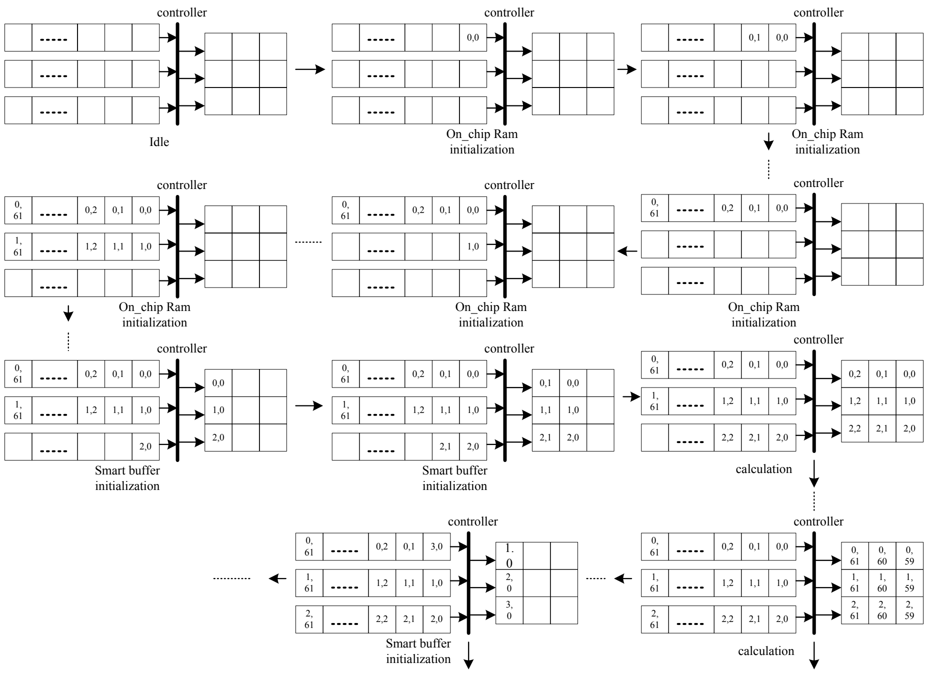


Fig. 5. FSM of on-chip RAM and smart buffer in our approach

3 Memory Architecture

In window operations, the input and output arrays are separate and therefore there is no loop-carried dependency on a single array. Fig.6 presents the universal layout of the target memory structure with which compiler generates VHDL codes for window operations.

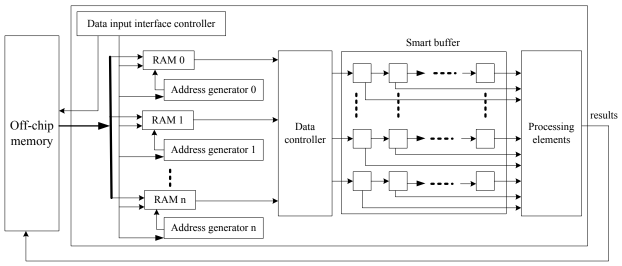


Fig. 6. Overall window operations execution architecture

Data that will be used in the following interations be kept in the RAM blocks until it will never be used again. Data to be used in the current iteration shift in the registers of smart buffer. Data input interface controller and data scheduling controller keep track of which iterations of the loop are currently in execution, and generate the appropriate control signal to realize the pipelined memory accesses. The address generation unit is a programmable one with auto-increment and auto-decrement capabilities.

4 VHDL Code Generation

In this section, we present our approach to generate efficient VHDL code for the controller and related components. The goal is to minimize run-time control calculation and maximize input data reuse.

4.1 Compiler Support

Window operations have one or more windows sliding over one or more arrays. Both addresses of the read and the write are calculated at compiler time. According to the memory load reference and the loop unrolling parameters, the following parameters are known at compiler time:

1. Starting and ending addresses;
2. The size of array, *width* and *height*;
3. The size of unrolled window, *buffer_width* and *buffer_height*;
4. The unrolled window's strides in each dimension, *buffer_span_width* and *buffer_span_height*;
5. The number of RAM blocks and the depth, *Ram_num*, *Ram_depth*;
6. The number of results generated for once interation, *Result_num*;
7. The number of unrolled times of outer loop, *Control_num*;

Fig.7 gives the unrolled C code of 2D_lowpass_filter. We unroll the loop twice in both horizontal and vertical directions. Therefore, each iteration computes four of these 3*3 windows, and produces a 2*2 output window.

```
for(i=1;i<62;i=i+2) {
    for(j=1;j<62;j=j+2) {
        C[i-1][j-1]=(A[i-1][j-1]+A[i-1][j]+A[i-1][j+1]+A[i][j+1]+A[i+1][j-1]+A[i+1][j]+
A[i+1][j+1])>>3+(A[i][j]>>1)-B[i-1][j-1];
        C[i-1][j]=(A[i-1][j]+A[i-1][j+1]+A[i-1][j+2]+A[i][j+2]+A[i+1][j]+A[i+1][j+1]+
A[i+1][j+2])>>3+(A[i][j+1]>>1)-B[i-1][j];
        C[i][j-1]=(A[i][j-1]+A[i][j]+A[i][j+1]+A[i+1][j+1]+A[i+2][j-1]+A[i+2][j]+
A[i+2][j+1])>>3+(A[i+1][j]>>1)-B[i][j-1];
        C[i][j]=(A[i][j]+A[i][j+1]+A[i][j+2]+A[i+1][j+2]+A[i+2][j]+A[i+2][j+1]+A[i+2][j+2])
>>3+(A[i+1][j+1]>>1)-B[i][j];
        }
}
```

Fig. 7. Motion detection C code, 2*2 unrolled loop

For the C code in fig.7, *width*=62, *height*=62. And for data array A, there are (4*4) registers in smart buffer, *buffer_width*=4, and *buffer_height*=4. The program is 2*2 unrolled, *buffer_span_width*=2, *buffer_span_height*=2. There are 4 RAM blocks needed, *Ram_num*=4. The depth of RAM is as same as the width of data array A, *Ram_width*=*width*=62. For once interation, there would generate four results, *Result_num*=4. We assume the memory bus in twice the width of the pixel bit-size, then each memory load reads in two pixels. In this example, Ram0 and Ram1 receive data at the same clock period, and Ram2 and Ram3 receive data at the same clock period. The outer loop is unrolled two times, *Control_num*=2. Fig.8 shows the status of array A's smart buffer at different clock cycles.

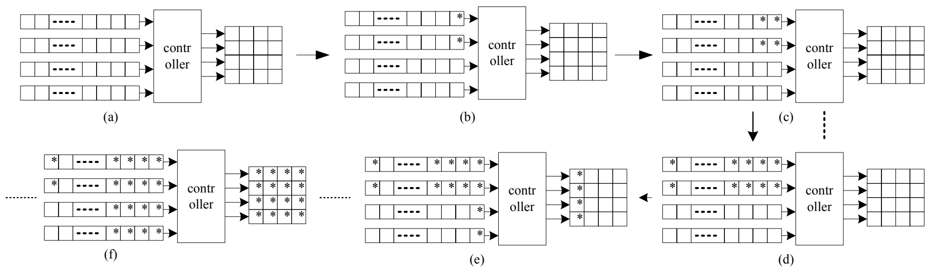


Fig. 8. FSM Status of 2D_Lowpass_filter smart buffer

Fig.9 shows the 5-tap FIR in C which is 1D window operations. In this case, data is only reused in the same loop iteration. We did not design RAM blocks for 1D applications. Smart buffer is enough to keep the reused data. *width*=62, *height*=1, *buffer_width*=5, *buffer_height*=1, *buffer_span_width*=1, *buffer_span_height*=0, *Ram_num*=0, *Ram_depth*=0, *Result_num*=1, *Control_num*=0.

```
For(i=0;i<62;i=i+1){
            B[i]=C0*A[i]+C1*A[i+1]+C2*A[i+2]+C3*A[i+3]+C4*A[i+4];
}
```

Fig. 9. A 5-tap FIR in C

4.2 FSM Generation

The FSM is in charge of the whole processing, determines when the data initialization is finished, traces which register is expired and would be overwritten by new data, determines when a window of data is ready to processing elements, and manages the counterpart relationship between the on-chip RAM and registers in smart buffer. The FSM is assigned one of the four states as shown in fig.5.

- Idle: Waiting for the start signal. When a procedure is started, go to on-chip RAM initialization;
- On-chip RAM initialization: On-chip RAM blocks are in a warm-up state;
- Smart buffer initialization: The smart buffer is collecting data to form the first window. Once all the new data have arrived, the smart buffer goes to processing;
- Processing: In this state, when a window of data is ready, send data in smart buffer to processing elements to calculate. In the next clock cycle, the window data is updated again. The processing keeps going until current row is finished, then the FSM changes to the smart buffer initialization state again to collect new data of the next row.

In the four states, processing state needs some complex controls, because we want to do data transferring and calculation concurrently to speed up the processing. Fig.10 illustrates the parameterized FSM of data transmission between off-chip and on-chip memory. The parameters can be obtained from the compiler as we have discussed in subsection 4.1. Fig.11 gives the parameterized FSM of sending data from on-chip RAM blocks to smart buffer, then to processing elements, and at the same time receiving the results. The FSM also ensures that the whole calculation processing carry through accurately.

5 Experiments

This section presents experimental results that characterize the impact of different methods. We use four window operations as benchmarks: 5-tap FIR, image sharpening, Sobel edge detection (SOBEL), and 2D_Lowpass_filter. These benchmarks are selected for the diversity of size of smart buffer and control structure. 5-tap FIR is a constant-coefficient finite-impulse (FIR) filter. Its source code is given in Fig.9. The size of smart buffer is (1*5). Image sharpening program deals with 2D data array. There are two RAM blocks designed, and the window size is (2*2). Sobel edge detection is shown in fig.1, (3*3) registers are used in smart buffer. The code of 2D_Lowpass_filter is given in fig.7, and the size of smart buffer is (4*4). The input data set of all 1D examples is 256 and the input data set size of all 2D examples is 64*64.

Table 1 shows the number of registers used in smart buffer to exploit data reuse in three methods.

Table 1. Number of registers in smart buffer

Benchmarks	5-tap FIR	Image sharpening	Sobel	2D_Lowpass filter
Traditional approaches	5	128	192	256
ROCCC	5	4	9	16
Ours	5	4	9	16

```
case (sending states)
        1: Primal state: set the Token-Ring to the RAM block  of number
Ring_counter, if the current RAM is the number (height-Control_num), set the
Token-Ring to the first RAM block again;
        2: Send data request signal to the off-chip memory. Ready to receive a new
data;
        3: Receive a data and keep it in the current on-chip RAM block who has the
Token-Ring. The counter increase to note how many data has been received for the
current RAM: counter_num<=counter_num+1;
        4:  Check.
                if (counter_num< width) goto2;
                else the current RAM block is full,
                Ring_counter<=Ring_counter+Control_num, goto 1;
endcase
```

Fig. 10. The FSM of sending data from off-chip memory to on-chip memory

```
case (control states)
        1:Idle state.
                Wait for (height-Control_num) rows of RAM blocks being initialized;
        2: Send data.
                Send data of smart buffer to the processing elements;
        3: Wait for the results;
        4: Send results.
                Increase the result counter,
                result_counter<=result_counter+Result_num;
                        If (result_counter== width*buffer_span_height)
                                result_counter<=0;
                        Increase the row counter which registers how many rows have
been done;
                                row_done<=row_done+buffer_span_height;
        5: Increase counter and shift the registers in smart buffer.
                RAM_current_do<=RAM_current_do+1;
                // RAM-current-do records which RAM blocks data is currently being
        dealt with.
                                If (RAM_current_do==Ram_num/buffer_span_height)
                                     RAM_current_done=1;
        6. Receive the next data.
                                If(row_done==height) finish,
                                goto 1;
                                else goto 2;
endcase
```

Fig. 11. The FSM of data transmission from on-chip RAM blocks to smart buffer and then to processing elements

Traditional appraches hold all reused data in smart buffer. Thus, they need a big smart buffer with a good many registers. ROCCC and our approach only keep calculational data in smart buffer, so the smart buffer is small.

We use the Xilinx ISE 7.1i tool chain to do synthesis and place-and-route reports. The generated VHDL codes are simulated using ModelSim 5.8. The target architecture of all synthesis is Xilinx XC2V8000-5. ROCCC also take 5-tap FIR and 2D_lowpass_filter as benchmarks. Table 2 below gives the simulation results of ROCCC and ours.

Table 2. The synthsis and simulation results of 5-tap FIR and 2D_Lowpass filter

	5-tap FIR		2D_Lowpass_filter		Image sharping	Sobel
	ROCCC	Ours	ROCCC	Ours	Ours	Ours
Smart buffer's statues	14	5	18	4	2	3
Control area (slices)	210	262	542	514	131	210
Clock rate(MHz)	94	238.664	69	238.664	238.664	238.664
Execution time(cycles)	262	263	5980	2057	4042	4108
Throughput(results number/cycles)	0.96	0.96	0.64	1.87	0.98	0.94

Area is the number of slices obtained from place-and-route reports. State is the number of states in the smart buffer's FSM. Clock rate is the clock rate of the whole placed-and routed circuit. Execution time is the number of cycles obtained from the simulation waveforms.

The shift registers in smart buffer of our approach make the connection between smart buffer and processing elements simpler. Thus, the smart buffer's status of ours is less than ROCCC. The whole control area of our approach and ROCCC are almost equal. Our approach can exploit data reuse completely, so the processing speed of ours is much faster than ROCCC, and the clock frequency is much higher than ROCCC. The clock frequency of our approach for the four benchmarks are almost same. It is because we bring forward a universal parameterized memory structure for the window operations. There are only some parameters need to be changed for a different design, and the control components are almost equivalent for all applications.

6 Summary and Conclusions

In this paper, we present a optimization generation memory structure for window operations. We exploit data reuse to reduce the number of accesses to the off-chip memory. We design special control unit to dominate the dataflow which makes it possible to store a small part of the data in internal RAM blocks and smart buffer while still providing sufficient memory bandwidth for the custom data path. We have applied our technique to a set of window processing tasks, and do some comparisons with related works. The results show that the generated memory structure can speed up the processing using less number of memory resources.

Acknowledgement

This work is sponsored by the National Science Foundation of China under the grant NO. 60633050.

References

1. Byoungro So, Mary W. Hall, Pedro C. Diniz: 'A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems'. PLDI 2002, 165-176.
2. Gupta, S., Dutt, N.D., Gupta, R.K., and Nicolau, A.: 'SPARK: a high-level synthesis framework for applying parallelizing compiler transformations'. Proc. Int. Conf. on VLSI Design, January 2003.
3. Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings: 'Sea Cucumber: A Synthesizing Compiler for FPGAs'. M. Glesner, P.Zipf, and M. Renovell(Eds.), FPL 2002, LNCS 2438, pp. 875-885, 2002. Springer-Verlag Berlin Herdelberg 2002
4. Weinhardt, M., and Luk, W.: 'Pipeline vectorization', IEEE Trans. Comput.-Aided Des., 2001, 20, (2), pp. 234-248.
5. Jan Frigo, Maya Gokhale, Dominique Lavenier: 'Evaluation of the StreamsC C to FPGA Compiler: An Applications Perspective'. FPGA 2001, February 11-13, 2001, Monterey, CA.
6. http://www.mentor.com/products/c-based_design/catapult_c_synthesis/index.cfm.
7. Heidi Ziegler and Mary Hall: 'Evaluating Heuristics in Automatically Mapping Multi-Loop Applications to FPGAs'. FPGA'05, February 20-22, 2005, Monterey, California, USA.
8. Mencer, O., Pearce, D.J., Howes, L.W., and Luk, W.: 'Design space exploration with a stream compiler'. Proc. IEEE Int. Conf. on Field Programmable Technology, 2003.
9. Donald Soderman and Yuri Panchul: 'Implementing C algorithms in reconfigurable hardware using C2Verilog'. In Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM), pages 339-342, Los Alamitos, CA, April 1998.
10. Celoxica, 'Handel-C Language Reference Manual for DK2.0', Document RM-1003-4.0, 2003.
11. De Figueiredo Coutinho, J.G., and Luk, W.: 'Source-directed transformations for hardware compilation'. Proc. IEEE Int. Conf. on Field-Programmable Technology, 2003.
12. Takashi Kambe, Akihisa Yamada, Koichi Nishida, Kazuhisa Okada, Mitsuhisa Ohnishi, Andrew Kay, Paul Boca, Vince Zammit, Toshio Nomura,: 'A C-based Synthesis System, Bach, and its Application'.
13. Daniel D. Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. 'SpecC: Specification Language and Methodology'. Kluwer, Boston, Massachusetts, 2000.
14. Byoungro So, HMary W. HallH, HHeidi E. ZieglerH: 'Custom Data Layout for Memory Parallelism'. CGO 2004, 291-302.

15. Gwenole Corre, Eric Senn, Pierre Bomel, Nathalie Julien, Eric Martin:'Memory Accesses Management During High Level Synthesis' CODES+ISSS 2004: 42-47.
16. Pedro C. Diniz, Joonseok Park: 'Automatic Synthesis of Data Storage and Control Structures for FPGA-Based Computing Engines'. FCCM 2000: 91-100.
17. Nastaran Baradaran, Pedro C. Diniz, Joonseok Park: 'Extending the Applicability of Scalar Replacement to Multiple Induction Variables'. LCPC 2004: 455-469.
18. Andersson P.H and Kuchcinski K.H 'Automatic Local Memory Architecture Generation for Data Reuse in Custom Data Paths', in Proc. of Engineering of Reconfigurable Systems and Algorithms, 2004.
19. Z. Guo, B. Buyukkurt and W. Najjar. "Input Data Reuse In Compiling Window Operations Onto Reconfigurable Hardware", Proc. ACM Symp. On Languages, Compilers and Tools for Embedded Systems (LCTES 2004), Washington, DC, June 2004.
20. Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers. "Optimized Generation of Datapath from C Codes for FPGAs", Int. ACM/IEEE Design, Automation and Test in Europe Conference (DATE 2005), Munich, Germany, March, 2005.
21. A. Mitra, Z. Guo and W. Najjar. ""Dynamic Co-Processor Architecture for Software Acceleration on CSoCs", Int. Conference on Computer Design (ICCD 2006), San Jose, California, 2006.
22. Z. Guo, W. Najjar and B. Buyukkurt. "Efficient Hardware Code Generation for FPGAs", ACM Transaction on Architecture and Code Optimizations (TACO), (Accepted 2006).

Adapting and Automating XILINX's Partial Reconfiguration Flow for Multiple Module Implementations

Rainer Scholz

Universität der Bundeswehr München, Germany
rainer.scholz@unibw.de

Abstract. In this paper, we present a modification of XILINX's Partial Reconfiguration Design Flow. Starting with either HDL-Design files or synthesised netlists, the presented flow generates all partial as well as the complete configuration bitstreams. In contrast to the established XILINX design flows, our flow is completely automated by a generator. By checking partial reconfiguration constraints it assists the user to avoid typical errors in module and bus macro placement. Compared with the PlanAhead partial reconfiguration flow, it is a single flow for generating multiple implementation for each reconfigurable area.

1 Introduction

Dynamic Partial Reconfiguration (DPR) has been widely proposed as revolutionary computing paradigm over at least a decade. But until now, it was mainly an academic research topic and didn't find it's way to many real-life products. Some quite promising prototypes for productive systems have been created, but none of these approaches has lead to a commercial break-through.

Some of the reasons for this, is the complicated building process for systems using DPR and the restrictive requisitions for designing them.

The following section describes the solutions XILINX makes with a special focus on the partial reconfiguration flow generated by the tool PlanAhead. In Section 3 we introduce our adaptation and extension of the baseline XILINX flow. Section 4 presents a tool for automating the modified flow. The paper concludes with a summary including plans for future work and code release.

2 XILINX and Partial Reconfiguration

In [5] XILINX presents the flow for building reconfigurable designs based on their Modular Design [3, Chapter 4]. This flow allows the reconfiguration of entire columns of Configurable Logic Blocks (CLB) and does not support static routes through reconfigurable areas.

In a new version of the baseline flow target [6] reconfigurable modules may span any rectangular area of an FPGA and routing static logic through reconfigurable modules is possible. However users are still limited, because tests on

Virtex II devices showed, that the generated partial configuration bitstreams are not operating correctly when modules are located in the same CLB columns, even if they are not intersecting.

2.1 The Partial Reconfiguration Flow by PlanAhead

A partial automation of the XILINX design flow can be achieved with the hierarchical floorplanning and design tool PlanAhead. Its flow for generating partial bitstreams is depicted in Fig. 1 and described in [6,7]. Dashed interconnections in the illustration denote steps automatically performed by the framework, solid lines denote required user interactions. Besides the high price-tag for the frame, users report technical drawbacks.

First, it lacks support of using multiple implementations for one reconfigurable area. So users still have to create a project for each implementation and to spend time in manually distributing the specification of the static logic (standardized in the file `static.used`) routed through the reconfigurable areas.

Second, for generating partial reconfigurable designs using PlanAhead, all input HDL-files have to be synthesized manually. So for each file, a synthesis-project has to be created.

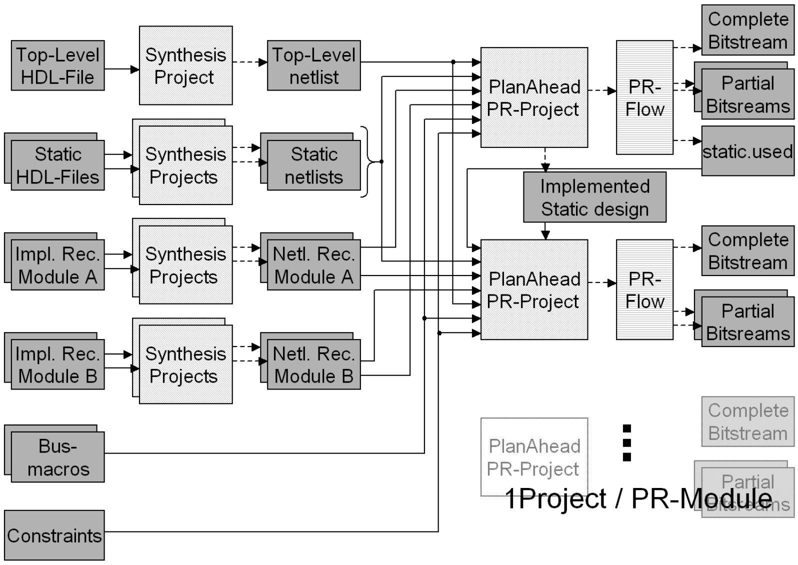


Fig. 1. The flow for generating partial bitstreams using PlanAhead

The Plan Ahead flow needs the static netlist files as well as the bus macros and the constraints as input.

The flow executes in three phases: Initial Budgeting, Active Module Implementation and Assembly.

In the Initial Budgeting phase, it performs the steps translate, map and place & route only for the static components, producing a design with areas not containing any logic (sometimes called "holes") for the partial reconfigurable (PR) modules and information about wires routed through the PR-areas (in the file `static.used`). In the Active Module Implementation phase translate, map and place & route are carried out for one implementation of each PR-area. In a final step, the Assembly phase, the PR-areas are merged and the complete as well as the partial bitstreams generated.

To produce more than one bitstream for a PR-area, another PlanAhead flow containing one set of not yet implemented PR-modules has to be generated. The implemented static design with the holes for the PR-modules as well as the `static.used` file has to be manually injected into the flow. This flow is now able to translate, map, place and route the PR-modules and to generate the complete and partial bitstreams for those modules. This has to be repeated for every set of reconfigurable module implementations.

3 Adaptation of the PlanAhead Reconfiguration Flow

Based on the developments of XILINX and discussions with users, we decided to implement an enhanced flow for our research work, depicted in Fig. 2. Dashed lines mean that these steps are performed automatically and drawn through lines stand for the need of user interaction. Since the development cycle requires frequent redesigns, it was the major goal to reduce user interaction with the flow framework. Another intent was to reduce the needed knowledge about partial reconfiguration to a minimum.

Our flow starts with the synthesis phase of the design. In this phase, HDL files and the information whether they implement the top-level design, a part of the static design or a partial reconfigurable module has to be defined. From this information all the needed setting can be retrieved and so the synthesis can be run.

The netlists resulting from the synthesis phase can be automatically taken by the PR-flow and so all modules are translated, mapped, placed and routed without any further user interaction.

One of the major benefitsis, that in the last step not only one set of reconfigurable modules is merged into the top-level design, but all modules are consecutively merged into it and the corresponding bitstreams are generated.

4 The ReconfGenerator

The ReconfGenerator synthesizes HDL designs with all options needed for partial reconfiguration and builds the partial and complete bitstreams of the design.

It is typically operated by a Graphical User Interface (GUI), but all settings for the tool may as well be specified in a file, that can directly be imported into the GUI for editing and exporting. These saved settings can be run completely

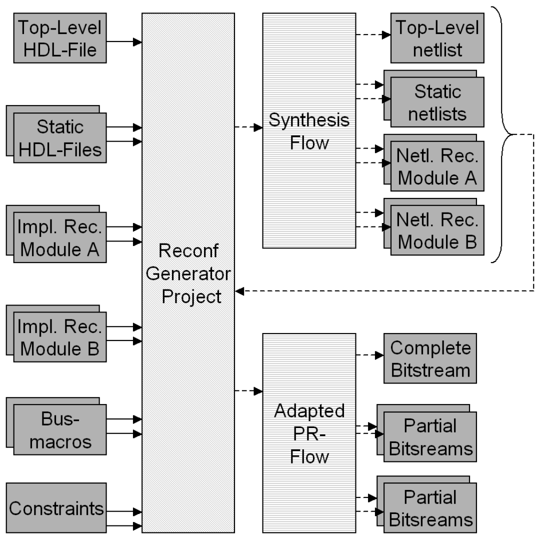


Fig. 2. The novell flow for generating partial bitstreams using the ReconfGenerator

autonomous without the GUI. This offers the possibility to apply this flow in bigger scripted or generated build systems, and ensures, that Partial Reconfigurable Systems remain archivable and reproducible.

Since most problems with designing PR-systems arise from erroneous settings for the tools, it is remarkable, that the ReconfGenerator sets all necessary options automatically.

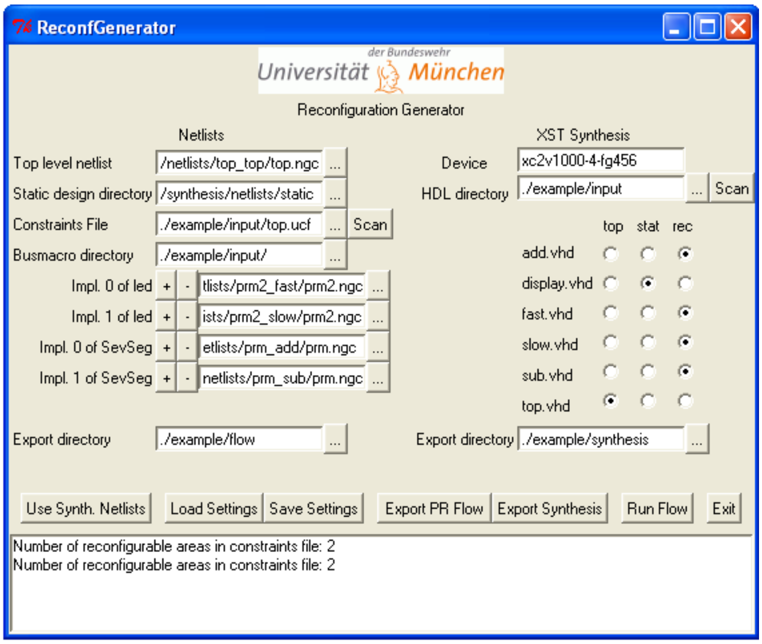
4.1 Synthesis

To use the generator for the synthesis of partial reconfigurable designs it is required to specify the design in an HDL (either VHDL or Verilog). At the moment the generator expects designs composed of a top-level HDL-file and several HDL-files, which define the modules (according to [3, Chapter 4]).

Here, the top-level file only instantiates and connects all the modules and bus macros. Its description should not contain any further logic.

The module files may only implement one module each. Any additional hierarchy is to be avoided. Since this flow only utilizes one top-level module, it is recommended to name all entities defining the same reconfigurable area identically and only give different names to their files.

The synthesis part of the generator creates a flow for synthesising all the input HDL-files, generating a netlist-file each. All options needed by the synthesis tool are set automatically. The resulting netlist files are sorted to folders in a way that eases building of the reconfiguring bitstreams. At the moment the only supported synthesis-tool is XILINX's XST.


ReconfGenerator

Reconfiguration Generator

Netlists

Top level netlist	/netlists/top_top/top.ngc
Static design directory	/synthesis/netlists/static
Constraints File	./example/input/top.ucf
Busmacro directory	./example/input/
Impl. 0 of led	tlists/prm2_fast/prm2.ngc
Impl. 1 of led	ists/prm2_slow/prm2.ngc
Impl. 0 of SevSeg	etlists/prm_add/prm.ngc
Impl. 1 of SevSeg	netlists/prm_sub/prm.ngc
Export directory	./example/flow

XST Synthesis

Device	xc2v1000-4-fg456
HDL directory	./example/input

	top	stat	rec
add.vhd	○	○	●
display.vhd	○	●	○
fast.vhd	○	○	●
slow.vhd	○	○	●
sub.vhd	○	○	●
top.vhd	●	○	○

Export directory	./example/synthesis

```
Number of reconfigurable areas in constraints file: 2
Number of reconfigurable areas in constraints file: 2
```

Fig. 3. The GUI of the Generator

4.2 Building from Netlists

For using other synthesis-tools for generating reconfiguring bitstreams, the ReconfGenerator supports building bitstreams out of already synthesised netlists.

To build reconfigurable designs starting with netlists, the generator needs either XILINX NGC or EDIF netlists as a starting point. The synthesized top-level netlist must contain IO-buffers, keep the hierarchy and use the same bus-delimiters as the bus macros. Settings for the modules are identical but IO-buffers are not required [1,2].

Besides the netlists, a constraints file is necessary. In this file, the areas for the reconfigurable modules, as well as the placement of the bus macros, are to be defined as described in [6]. The most common errors in module and bus macro placement will be detected by the generator.

At the moment the following placement errors are detected:

– Overlapping modules
– Modules located in the same CLB-columns
– Modules spanning their region in any other way than from the upper left to the lower right corner
– Bus macros that are not connected to a module
– Bus macros that are not located on the intersection between a PR-module and its surrounding

If the synthesis flow as described in section 4.1 is used, its outputs are directly taken as input files for this process.

As soon as the needed input is specified, an adapted PR-flow as described in section 3 for building the bitstreams is exported. Batch files for each step of this flow as well as a batch file for executing those steps consecutively are generated as well.

4.3 Environment

To run the proposed tool chain, the following software is required on the host-machine:

- MS Windows
- XILINX ISE 8.1.01i (the only ISE Version supporting the PR-Tools)
- XILINX PR-Tools (Modified map, par and library files for partial reconfiguration. Tested with Version 11.)
- a Perl interpreter (tested with ActivePerl v5.8.6)
- Perl/TK (a Perl library for GUIs, included in ActivePerl)

If users don't need the GUI, it is as well possible to run the ReconfGenerator using XILINX's Perl interpreter Xilperl, as delivered with the ISE.

4.4 Graphical User Interface

The Generator is comfortably operated by means of a GUI, shown in Fig. 3. In this GUI, all the files needed to execute the flow are to be specified. The GUI adapts automatically to the number of reconfigurable areas and the number of their implementations.

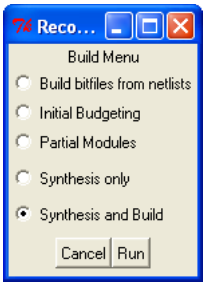
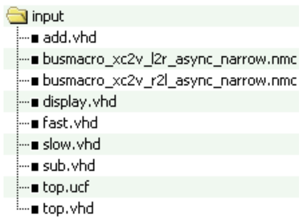


Fig. 4. The Build Menu

To export of as well the synthesis flow as the bitstream generation flow a button is available within the GUI, that activates the Build Menu as shown in Fig. 4. In this Menu, the user may select which steps of the build process are to be run.

4.5 Example

The exemplar design embodies two reconfigurable areas with two implementations each on a XILINX Virtex II FPGA [4] (xc2v1000).



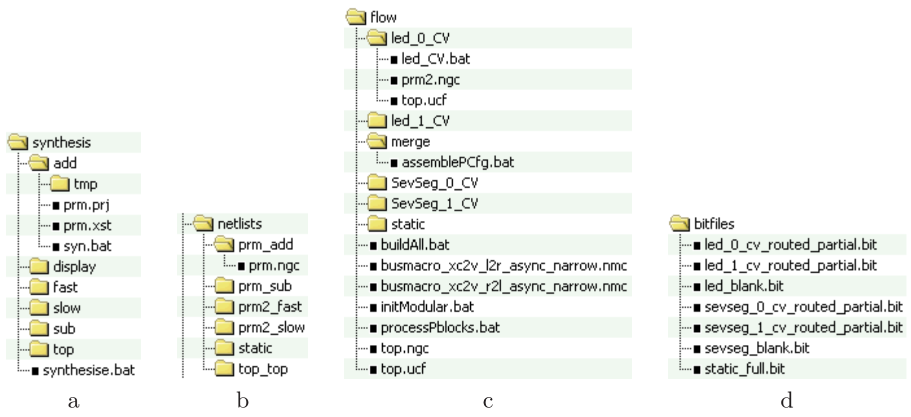
```
input
  add.vhd
  busmacro_xc2v_l2r_async_narrow.nmc
  busmacro_xc2v_r2l_async_narrow.nmc
  display.vhd
  fast.vhd
  slow.vhd
  sub.vhd
  top.ucf
  top.vhd
```

Fig. 5. Input files for the example

The first reconfigurable area (called PRM) contains a clock divider. The implementations of this module consist of different dividers for different output clock rates (implemented in `fast.vhd` and `slow.vhd`). The second reconfigurable area (called PRM2) increases an eight bit number by one (`add.vhd`) or decreases it (`sub.vhd`).

The static design consists of a top-level HDL-file (`top.vhd`), which routes the output of PRM to an LED and a static module, which shows the output of PRM2 on two seven segment displays (`display.vhd`).

Additionally, a constraints file specifying the I/O pins and the reconfigurable areas (top.ucf) and the applied bus macros are needed. Please refer to Fig. 5 for an overview of the required input files.



a
```
synthesis
  add
    tmp
    prm.prj
    prm.xst
    syn.bat
  display
  fast
  slow
  sub
  top
  synthesise.bat
```

b
```
netlists
  prm_add
    prm.ngc
  prm_sub
  prm2_fast
  prm2_slow
  static
  top_top
```

c
```
flow
  led_0_CV
    led_CV.bat
    prm2.ngc
    top.ucf
  led_1_CV
  merge
    assemblePCfg.bat
  SevSeg_0_CV
  SevSeg_1_CV
  static
  buildAll.bat
  busmacro_xc2v_l2r_async_narrow.nmc
  busmacro_xc2v_r2l_async_narrow.nmc
  initModular.bat
  processPblocks.bat
  top.ngc
  top.ucf
```

d
```
bitfiles
  led_0_cv_routed_partial.bit
  led_1_cv_routed_partial.bit
  led_blank.bit
  sevseg_0_cv_routed_partial.bit
  sevseg_1_cv_routed_partial.bit
  sevseg_blank.bit
  static_full.bit
```

Fig. 6. Example: a: Synthesis Flow, b: Synthesis Output, c: Bitstream Generation Flow, d: Bitstreams

After filling out the fields in the GUI, the Synthesis Flow (Fig. 6a) can be generated. The Synthesis Flow produces the netlists (Fig. 6b), that are used as an input for the export of the Bitstream Generation (Fig. 6c). This flow generates the full and the partial bitstreams (Fig. 6d), ready to configure the device.

5 Summary, Conclusions and Future Work

In this contribution, we present an adapted flow for partial reconfiguration. The ReconfGenerator tool automates this flow and enables users to build dynamically reconfigurable designs automatically. This gives users the chance to concentrate primarily on the design of the functionality, not needing any knowledge of the building process.

In future work, we will expand the ReconfGenerator by additional support in placing bus macros and in specifying reconfigurable areas. We also intend to build a wizard, that generates the bodies of the input HDL files for the flow, readily prepared for partial reconfiguration.

Furthermore, we plan to make the ReconfGenerator, including its source-code, publicly available.

References

1. Braeckman, Geert et al.: Module Based Partial Reconfiguration: a quick tutorial, July 2004. `http://iwt5.ehb.be/typo3/fileadmin/files/ Quick_tutorial/Module_Based_Partial_Reconfiguration_-_A_quick_tutorial.zip`
2. Van den Branden, Gerd et al.: Module Based Partial and Dynamic Reconfiguration, September 2006. `http://iwt5.ehb.be/typo3/fileadmin/files/Downloads/DynamicReconfiguableModuleBased.pdf`.
3. Xilinx Inc.: Development System Reference Guide, 2005. `http://toolbox.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf`.
4. Xilinx Inc.: Virtex-II Platform FPGAs: Complete Data Sheet, March 2005. `http://direct.xilinx.com/bvdocs/publications/ds031.pdf`.
5. Xilinx Inc.: XAPP290: Two Flows for Partial Reconfiguration: Module Based or Difference Based, September 2004.
6. Xilinx Inc.: Early Access Partial Reconfiguration User Guide, March 2006.
7. Xilinx Inc.: Partial Reconfiguration Software Users Guide: Partial Reconfiguration of Virtex 4 using PlanAhead 8.1.
8. Xilinx Inc.: XAPP 255: Using Partial Reconfiguration to Time-Share Device Resources in Virtex-II and Virtex-II Pro, May 2005.

A Linear Complexity Algorithm for the Automatic Generation of Convex Multiple Input Multiple Output Instructions

Carlo Galuzzi, Koen Bertels, and Stamatis Vassiliadis

Computer Engineering, EEMCS
Delft University of Technology, The Netherlands
{C.Galuzzi, K.L.M.Bertels, S.Vassiliadis}@ewi.tudelft.nl

Abstract. The Instruction-Set Extensions problem has been one of the major topic in the last years and it consists of the addition of a set of new complex instructions to a given Instruction-Set. This problem in its general formulation requires an exhaustive search of the design space to identify the candidate instructions. This search turns into an exponential complexity of the solution. In this paper we propose an efficient linear complexity algorithm for the automatic generation of convex Multiple Input Multiple Output (MIMO) instructions, whose convexity is theoretically guaranteed. The proposed approach is not restricted to basic-block level and does not impose limitations either on the number of input and/or output, or on the number of new instructions generated. Our results show a significant overall application speedup (up to x2.9 for ADPCM decoder) considering the linear complexity of the proposed solution and which therefore compares well with other state-of-art algorithms for automatic instruction set extensions.

1 Introduction

The last years have shown an increasing popularity of reconfigurable architectures thanks to the capability to provide high overall performances of execution of an application, by tuning the architecture towards the specific requirements of the application. More and more often this is achieved via the automatic extension of a given Instruction-Set (IS) with new customized instructions for the specific application.

An example of reconfigurable architecture can be realized by combining a GPP and a reconfigurable hardware as an FPGA. The execution of an application on an architecture with a given Instruction-Set usually involves instructions belonging to the Instruction-Set, and implemented in hardware, and many instructions executed in software, and typically more costly in terms of execution time than the ones executed in hardware. Roughly speaking the idea is to group clusters of instructions executed in software in new complex instructions to implement on the reconfigurable hardware and whose execution time is faster in hardware than in software. Once these new complex instructions are hardwired, they represent an extension of the given Instruction-Set. Additionally, the reconfigurability of the architecture allows the creation of new complex instructions ad hoc, that is Application-Specific Instruction-Set Extensions.

In this paper we propose a linear complexity algorithm for the automatic generation of new Application-Specific Instructions under hardware resources constraints. The proposed approach targets the Molen organization [1] which allows for a virtually unlimited number of new instructions without limiting the number of input/output values of the function to be executed on the reconfigurable hardware.

A set of convex Multiple Input Multiple Output (MIMO) operations is identified by a two steps approach. Firstly, the operations are clustered in Maximal Multiple Input Single Output (MAXMISO) operations, the elementary building blocks of our approach, and subsequently these MAXMISOs are clustered per levels as new application-specific instructions. The result is a cluster of operations with Multiple Input Multiple Output, called MIMO, which is executed on the reconfigurable hardware and which provides the maximum performance improvement under reconfigurable hardware resource constraints. More specifically, the main contributions of this paper are:

- an overall linear complexity of the proposed solution. The generation of complex instructions is a well known NP problem and its solution requires, in the worst case, an exhaustive search of the design space which turns into an exponential complexity of the solution. Our approach heuristically extends a given Instruction-Set with convex MIMO instructions based on MAXMISOs clustering in order to exploit the MAXMISO level parallelism. Single MAXMISOs usually do not provide significant improvement. Thus we propose MAXMISOs combination (Section 3.4) in order to take advantage of the parallelism inherent to the hardware execution and the Theorem 2 that guarantees the MIMO convexity by construction. The proposed solution addresses the problem with two linear complexity steps, MAXMISO clustering and MAXMISO combination, that are linear in the number of processed elements.
- elimination of the restrictions of the types and number of new instructions (in contrast with most of the existing approaches). There is no limitation on the number of input/output values or on the number of new instructions.
- the proposed approach is not restricted to basic-block level analysis and can be applied directly to large kernels.

The paper is organized as follows. In Section 2, background information and related works are provided. In Section 3, the context is further formalized and the theoretical contribution is presented. Section 4 presents the experimental setup and results. Concluding remarks and an outline of research conducted are given in Section 5.

2 Background and Related Works

The algorithms for Instruction Set Extensions usually select clusters of operations which can be implemented in hardware as single instructions while providing maximal performance improvement. Basically, there are two types of clusters that can be selected, based on the number of output values: MISO or MIMO. Accordingly, there are two types of algorithms for Instruction Set Extensions that are briefly presented in this section.

Concerning the first category, a representative example is introduced in [2] which addresses the generation of MISO instructions of maximal size, called MAXMISO. The proposed algorithm exhaustively enumerates all MAXMISOs. Its complexity is linear with the number of nodes. The reported performance improvement is of few processor cycles per newly added instruction. The approach presented in [3] targets the generation of general MISO instructions. The exponential number of candidate instructions turns into an exponential complexity of the solution in the general case. As a consequence, heuristic and additional area constraints are introduced to allow an efficient generation. The difference between the complexity of the two approaches is due to the properties of MISOs and MAXMISOs: while the enumeration of the first is similar to the subgraph enumeration problem (which is exponential) the intersection of MAXMISOs is empty and then once a MAXMISO is identified, its nodes are removed from the set of nodes that have to be successively analyzed. In this way the MAXMISOs are enumerated with linear complexity in the number of nodes.

The algorithms included in the second category are more general and provide more significant performance improvement. However, they have exponential complexity. For example, in [4] the identification algorithm detects optimal convex MIMO subgraphs but the computational complexity is exponential. A similar approach described in [5] proposes the enumeration of all the instructions based on the number of inputs, outputs, area and convexity. The selection problem is not addressed. In [6] the authors target the identification of convex clusters of operations under given input and output constraints. The clusters are identified with a ILP based methodology similar to the one proposed in [7]. The main difference is that in [6] the authors iteratively solve ILP problems for each basic block, while in [7] the authors have one global ILP problem for the entire procedure. Additionally, the convexity is addressed differently: in [6], the convexity is verified at each iteration, while in [7] it is guaranteed by construction. Other approaches cluster operations by considering the frequency of execution or the occurrence of specific nodes [8,9] or regularity [10]. Still others impose limitation on the number of operands [11,12] and use heuristics to generate sets of custom instructions which therefore can not be globally optimal.

The algorithm we introduce in this paper combines concepts of both categories: at first, MAXMISOs are identified as proposed in the first category, afterward they are combined in convex MIMOs. Our algorithm requires linear complexity for the MAXMISO enumeration and the MAXMISO combination. Additionally, the proposed algorithm does not impose any limitations on the number of input/output values (as in [13,14,11]) or on the number of newly added instructions.

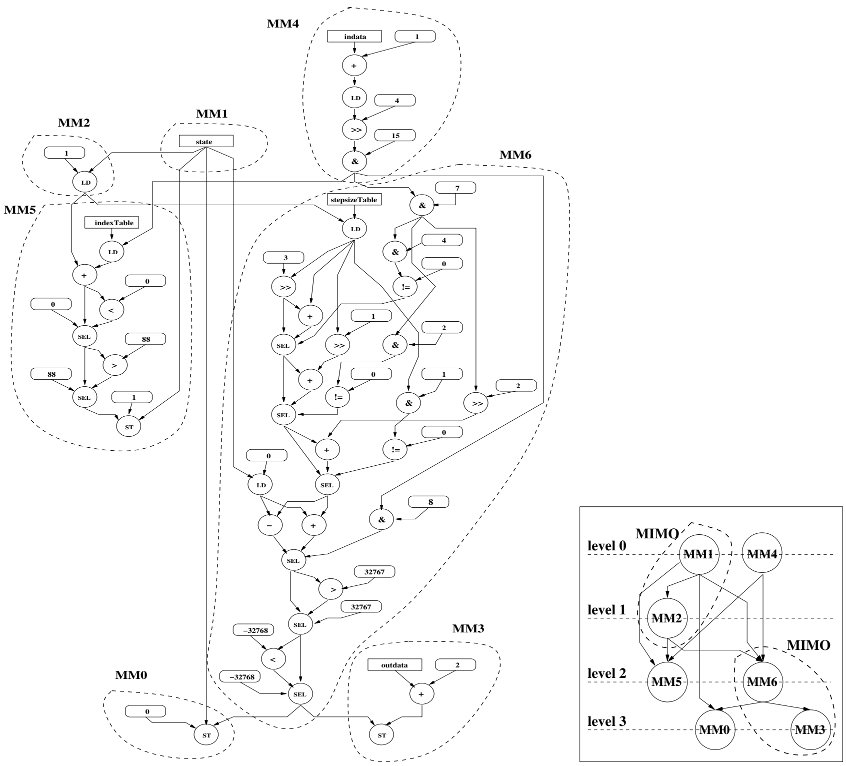


Fig. 1. Motivational example: Dataflow subgraph from *ADPCM* Decoder. **a)** the MAXMISO identification and **b)** the collapsed graph with the convex MIMOs identified by the algorithm.

3 Theoretical Background

In this section, we begin by introducing a motivational example to informally outline the main concept of the proposed algorithm and then we present the theoretical foundation of our approach. Last but not least, we present in detail the steps of the MIMO clustering algorithm.

3.1 Motivational Example

In Figure 1, we present the dataflow subgraph of the ADPCM application as implemented in the MediaBench benchmark suite [15]. Our algorithm identifies the convex MIMO in two steps. Firstly, the graph is partitioned in MAXMISOs (see Figure 1(**a**)). Each MAXMISO is then collapsed as a single node in the reduced graph presented in Figure 1(**b**). Since single MAXMISO execution in hardware doesn't provide significant performance improvement, the main idea of our algorithm is to combine, per levels, MAXMISOs available at the same level in the reduced graph, into a convex MIMO that is executed as a single instruction in hardware. The combination per level and successively per levels is the key difference between the solution presented in this paper and the one we proposed in [7]. Let assume the hardware latency for $MAXMISO_i$ to be l_i. When k MAXMISOs at the same level are clustered, the execution time of the cluster in hardware is $\max_{i=1..k} l_i$. The performance gain in this case is $\sum_{i=1..k}(l_i) - \max_{i=1..k}(l_i)$. If successively we cluster per levels, the overall performance gain increases. Let assume that $\alpha_1, ..., \alpha_h$ are the levels of the nodes belonging to a cluster generated combining per levels. The overall performance gain in this case is:

$$\sum_{j=\alpha_1}^{\alpha_h} (\sum_{i_j} l_{i_j} - \max_{i_j}(l_{i_j})) \tag{1}$$

Although the clustering algorithm is detailed in the following sections, we can roughly underline the main clustering steps. Starting from a node v belonging to level m, the algorithm grows a convex cluster of nodes C taking v as a seed. At first, an initial cluster C' composed by v and its predecessors at level $m-1$ is grown. C' is further extended with the successors at level m of the nodes of $C' \backslash \{v\}$. By Theorem 2, this extension C'' is convex. Once C'' is generated, the algorithm iteratively extends at each iterations C'' with nodes having all inputs coming from nodes belonging to the cluster generated in the previous iteration. When the cluster is no more extendable, its nodes are removed from the nodes to analyze and the algorithm restarts a new cluster from a node v'. The final clusters $C_1, .., C_l$ built in this way are convex MIMOs (see Section 3.4). For the graph presented in Figure 1(**a**) the algorithm identifies the convex clusters $C_1 = \{MM_1, MM_2\}$ and $C_2 = \{MM_3, MM_6\}$.

MAXMISO clustering is limited by the size of the reconfigurable hardware. This means that the algorithm stops the generation of convex MIMO as soon as there is no more available area in the FPGA.

3.2 MISO Properties and MAXMISO-Clustering

In order to formally express the problem previously presented, we first introduce the necessary definitions and the theoretical foundation of our solution. We assume that the input dataflow graph is a DAG $G = (V, E)$, where V is the set of nodes and E is the set of edges. The nodes represent primitive operations, more specifically assembler-like operations, and the edges represent the data dependencies. The nodes can have two inputs at most and their single output can be input to multiple nodes.

Let $\wp(G)$ be the power set of G[1] and let n be the order of V. The order of $\wp(G)$ is 2^n. Basically, there are two types of subgraphs that can be identified: MISOs and MIMOs. Let G_{MISO} and G_{MIMO} be the subsets of $\wp(G)$ containing all MISOs and MIMOs respectively. The following chain of inclusions is valid:

$$G_{MISO} \subset G_{MIMO} \subset \wp(G). \tag{2}$$

[1] $\wp(G)$ is the set of all subgraphs of G, including the empty graph $\emptyset$ and G.

Definition 1. *Let $G^* \subseteq G$ be a subgraph of G with $V^* \subseteq V$ set of nodes and $E^* \subseteq E$ set of edges. G^* is a MISO of root $r \in V^*$ provided that $\forall\ v_i \in V^*$ there exists a path*[2] *$[v_i \rightarrow r]$, and every path $[v_i \rightarrow r]$ is entirely contained in G^*.*

By Definition 1, A MISO is a connected graph. A MIMO, defined as the union of $m \geq 1$ MISOs can be either connected or disconnected.

Definition 2. *A subgraph $G^* \subsetneq G$ is **convex** if there exists no path between two nodes of G^* which involves a node of $G \backslash G^*$*[3]*.*

Convexity guarantees a proper and feasible scheduling of the new instructions which respects the dependencies. Definitions 1 and 2 imply that every MISO is a connected and convex graph. MIMOs can be convex or not. For example the subgraph $G^* = \{MM_1, MM_6\}$ in Figure 1(**b**) is not a convex graph.

Nevertheless the following property holds.

Theorem 1. *Let $G^* \subsetneq G$ be a convex subgraph of G. Then there exists $k \in \mathbb{N}$ such that $G^* = \bigcup_{i=1}^{k} MISO_i$.*

Proof. Let G^* be a MISO. Then $k = 1$ and $MISO_1 = G^*$. Let G^* be a MIMO. Every node has single output and therefore each node is trivially a MISO. This concludes the proof since every (sub)graph can be decomposed as the union of its nodes. □

The previous theorem implies that an exhaustive enumeration of the MISOs contained in G gives all the necessary building blocks to generate all possible convex MIMOs. This faces with the exponential order of G_{MISO} and, by (2), of G_{MIMO}. A reduction of the number of the building blocks reduces the total number of convex MIMOs which it is possible to generate. Anyhow, it reduces the overall complexity of the generation process as well. A trade-off between complexity and quality of the solution can be achieved considering MISO graphs of maximal size.

Let $\subset$ be the usual subset inclusion and $\wp(G)$ the power set of G. The couple $(\wp(G), \subset)$ is an ordered set and an element $G_{\overline{\iota}} \in \wp(G)$ is said to be **maximal** if, for all $G_i \in \wp(G)$, $G_{\overline{\iota}} \not\subset G_i$. A **maximal MISO** (MAXMISO) is a maximal element of $(G_{MISO}, \subset)$. A MAXMISO can formally be defined as follows.

Definition 3. *A MISO $G^*(V^*, E^*) \subset G(V, E)$ is a MAXMISO if $\forall v_i \in V \backslash V^*$, $G^+(V^* \cup \{v_i\}, E^+)$ is not a MISO.*

We know from the set-theory that each element of G_{MISO} is either maximal (a MAXMISO) or there exists a maximal element containing it. [2] observed that if $A, B \in G_{MISO}$ are two MAXMISOs, then $A \cap B = \emptyset$. Since every node is trivially a MISO, the following equality holds:

$$G = \bigcup_{i \in I} MAXMISO_i,\ I \subset \mathbb{N}. \tag{3}$$

[2] A path is a sequence of nodes and edges, where the vertices are all distinct.

[3] G^* has to be a *proper subgraph* of G. A graph itself is always convex.

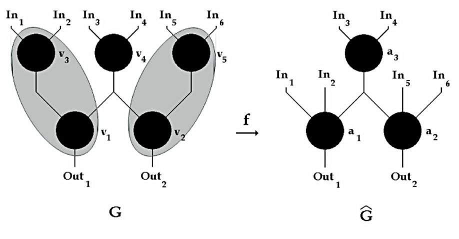


Fig. 2. The collapsing function $f : G \to \hat{G}$. $G_{MM} = \{MM_1, MM_2, MM_3\}$ where $MM_1 = \{v_1, v_3\}, MM_2 = \{v_4\}, MM_3 = \{v_2, v_5\}$. Then $MM_1 \mapsto a_1$, $MM_2 \mapsto a_3$ and $MM_3 \mapsto a_2$. In this case we have $G_{MISO} = \{v_1, v_2, v_3, v_4, v_5, MM_1, MM_3\}$ and clearly $G_{MM} \subsetneq G_{MISO}$.

The empty intersection of two MAXMISOs implies that the MAXMISOs of a graph can be enumerated with linear complexity in the number of its nodes.

3.3 Convex MIMOs Generation

Let $G_{MM} \subset G_{MISO}$ be the set of all MAXMISOs of G. Let $f : G_{MM} \subset G \to \hat{G}$ be the function that collapses the MAXMISOs of G in nodes of the graph $\hat{G}$, (Fig. 2): $MM_i \in G_{MM} \subset G \mapsto a_i \in \hat{G}$.

By definition f is a surjective function. Two MAXMISOs cannot overlap and then f is also injective and therefore bijective. Let f^{-1} be the inverse function of f. $f^{-1} : \hat{G} \to G_{MM}$ is the function such that maps $a_i \in \hat{G}$ into $MM_i \in G_{MM} \subset G$. f and f^{-1} are called the **collapsing** and **un-collapsing function** and $\hat{G}$ is called the $MAXMISO$**-collapsed graph**.

Let $v \in V$ be a node of G and let $\textsc{Lev} : V \to \mathbb{N}$ be the integer function defined as follows:

- $\textsc{Lev}(v) = 0$, if v is an input node of G;
- $\textsc{Lev}(v) = \alpha > 0$, if there are α nodes on the longest path from v and the level 0 of the input nodes.

Clearly $\textsc{Lev}(\cdot) \in [0, +\infty)$ and the maximum level $d \in \mathbb{N}$ of its nodes is called the **depth** of the graph.

Definition 4. *The level of $MAXMISO_i \in G$ is defined as follows:*
$$\textsc{Lev}(MAXMISO_i) = \textsc{Lev}(f(MAXMISO_i)).$$

Theorem 2. *Let G be a DAG and $A_1, A_2 \subset G$ two MAXMISOs*[4]*. Let $\textsc{Lev}(A_1) \geq \textsc{Lev}(A_2)$ be the levels of A_1 and A_2 respectively. Let $C = A_1 \cup A_2$. If*

$$\textsc{Lev}(A_1) - \textsc{Lev}(A_2) \in \{0, 1\} \tag{4}$$

then C is a convex MIMO. Moreover C is disconnected if the difference is 0.[5]

[4] Clearly $A_1 \cap A_2 = \emptyset$.

[5] $\textsc{Lev}(A_1) - \textsc{Lev}(A_2) = 0$ is a particular case studied in [7].

Proof. Let G be decomposed as union of MAXMISOs and let f be the collapsing function. Let a_1, a_2 and c be the images through f of A_1, A_2 and C respectively. f transforms equation (4) in $\textsc{Lev}(a_1) - \textsc{Lev}(a_2) \in \{0, 1\}$. In both cases, by contradiction, if $c = a_1 \cup a_2$ is not convex, there exists at least one path from a_1 to a_2 involving a node a_k different from a_1 and a_2. Then

$$\textsc{Lev}(a_2) < \textsc{Lev}(a_k) < \textsc{Lev}(a_1). \tag{5}$$

Since by hypothesis $\textsc{Lev}(a_1) - \textsc{Lev}(a_2) \in \{0, 1\}$, it follows that $\textsc{Lev}(a_k) \notin \mathbb{N}$ which contradicts the hypothesis $\textsc{Lev}(\cdot) \in \mathbb{N}$. As a result c is a convex MIMO and then considering the un-collapsing function f^{-1}, $f^{-1}(c) = C = A_1 \cup A_2$ is a convex MIMO graph. In particular, if the difference is 0, c is disconnected. By contradiction if c is connected there exists a path from a_1 to a_2. Then $\textsc{Lev}(a_1) \neq \textsc{Lev}(a_2)$ which contradicts the assumption $\textsc{Lev}(a_1) - \textsc{Lev}(a_2) = 0$. As a result c is disconnected and therefore $C = f^{-1}(c)$ is disconnected. □

Corollary 1. *Any combination of MAXMISOs at the same level or at two consecutive levels is a convex MIMO.*

Proof. Let $C = A_1 \cup ... \cup A_k \subset G$ be the union of $k \geq 3$ MAXMISOs of G. Two scenarios are possible: $\textsc{Lev}(A_i) = \overline{\imath}\ \ \forall i$, or $\textsc{Lev}(A_i) \in \{\overline{\imath}, \overline{\imath}+1\}$. In both cases, let $c = f(C) = a_1 \cup ... \cup a_k$[6] be the image through f of the MAXMISOs-union. By contradiction if c is not convex, there exists al least a path between two nodes that is not included in c. This contradicts the previous Theorem 2. Then c is a convex MIMO as well as $C = f^{-1}(c)$. □

3.4 The Algorithm

Each convex MIMO is identified in two steps: first of all a cluster of nodes is grown within $\hat{G}$, the $MAXMISO$-collapsed graph, and afterward the cluster is further extended with additional nodes.

1st **step**. Let $a_{\overline{\imath}}$ be a node of $\hat{G} = (\hat{V}, \hat{E})$ with $\textsc{lev}(a_{\overline{\imath}}) = \alpha \in [0, d]$ and let $C = \{a_{\overline{\imath}}\}$. Let us define the following sets:

$$\begin{aligned}
\textsc{pred}'(a_{\overline{\imath}}) &= \begin{cases} \{m \in \hat{V} \mid \textsc{lev}(m) = \alpha - 1 \ \wedge\ \exists\ (m, a_{\overline{\imath}}) \in \hat{E}\} & \text{if } \alpha \geq 1 \\ \emptyset & \text{if } \alpha = 0 \end{cases} \\
\textsc{succ}'(a_{\overline{\imath}}) &= \begin{cases} \{m \in \hat{V} \mid \textsc{lev}(m) = \alpha + 1 \ \wedge\ \exists\ (a_{\overline{\imath}}, m) \in \hat{E}\} & \text{if } \alpha \leq d - 1 \\ \emptyset & \text{if } \alpha = d \end{cases} \\
\textsc{succ}(a_{\overline{\imath}}) &= \begin{cases} \{m \in \hat{V} \mid \exists\ [a_{\overline{\imath}} \rightarrow m] \wedge\ \textsc{lev}(m) > \textsc{lev}(a_{\overline{\imath}})\} & \text{if } \alpha \leq d - 1 \\ \emptyset & \text{if } \alpha = d. \end{cases}
\end{aligned} \tag{6}$$

$C' = C\ \cup\ \textsc{pred}'(a_{\overline{\imath}})$ is a convex MIMO. This holds for $\alpha \geq 1$ as a consequence of Theorem 2 and for $\alpha = 0$ since a node is trivially a convex graph.

Let us consider $\textsc{succ}'(\textsc{pred}'(a_{\overline{\imath}}))$ and let $N_{In}(n)$ be the number of inputs of a node n and let $N_{In_C}(n)$ be the number of inputs coming from a set C of a node n. For each node n and each set C the following inequality can be satisfied:

$$2 * N_{In_C}(n) \geq N_{In}(n). \tag{7}$$

[6] $f(A \cup B) = f(A) \cup f(B)$.

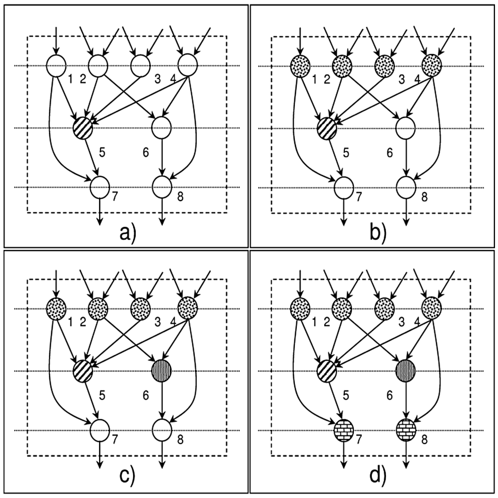


Fig. 3. Example of an application of the algorithm. **a**) $C = \{5\}$, **b**) $C' = \{5\} \cup \{1, 2, 3, 4\}$, **c**) $C'' = \{1, .., 6\}$, **d**) $C''' = \{1, .., 8\}$.

This can be reformulated by saying that the number of inputs of n coming from the set C has to be greater than or equal to at least half the total inputs of n. We define the following set:

$$C'' = \begin{cases} C' \;\cup\; \{n \in \textsc{succ}'(\textsc{pred}'(a_{\overline{C}})) \mid (7) \text{ holds}\} & \text{if } n \text{ exists} \\ C' & \text{otherwise.} \end{cases} \tag{8}$$

If there exists n such that (7) holds, by Theorem 2, $C'' = C' \;\cup\; \{n\}$ is a convex MIMO.

2nd **step**. Let us define the following set:

$$\textsc{succ}(C'') = \bigcup_{m \in C''} \textsc{succ}(m). \tag{9}$$

For each $m \in \textsc{succ}(C'')$ such that

$$N_{In}(m) = N_{In_{C''}}(m), \tag{10}$$

$C'' \;\cup\; \{m\}$ is a convex MIMO. This follows from (10). If the total number of inputs of m is equal to the number of inputs coming from C'', it follows that it doesn't subsist the possibility of having a path between a node of C'' and m which includes a node not belonging to $C'' \;\cup\; \{m\}$. As a consequence $C''' = C'' \;\cup\; \{m \in \textsc{succ}(C'') \mid N_{In_C}(m) = N_{In}(m)\}$ is a convex MIMO. In Figure 3, we present an example that shows the way by which the algorithm generates the convex MIMO C'''.

In summary, the steps required to generate the set of convex MIMOs are the following:

– Step a: MAXMISO identification: using an algorithm similar to the one presented in [2]
– Step b: Construction of the reduced graph: each MAXMISO is collapsed on one node
– Step c: HW/SW estimation: evaluate the HW/SW execution latency for each MAXMISO
– Step d: Until the area constraint is violated, choose a node of $\hat{G}$, generate C''' and remove the nodes of C''' from the node to further analyze.

Remark 1. By Step d, it follows that the convex MIMOs are generated linearly with the number of node of $\hat{G}$. Additionally, the inequality (7) is introduced to limit the total number of inputs of C''. The node which is selected to be the seed to grow C''' is chosen as the one with smallest latency in hardware. In case many nodes have the same latency, at present we select randomly the seed starting from the lowest levels.

4 Results

To evaluate the speedup achieved by the proposed approach, a dedicated tool chain is built and the algorithm is applied on a set of four well-known kernels and benchmarks applications. The above described algorithm is part of a larger automatic tool chain that aims to support the hardware designer in the design process. The tool chain for the experiments has been already described in our previous paper [7].

The software execution time for each MAXMISO is computed as the sum of the latencies of its operations. The hardware execution time is estimated through behavioral synthesis of the MAXMISO's VHDL models and then converting the reported delay into PowerPC cycles. We consider implementation of our approach on the Molen prototype that is built on Xilinx's Virtex-II Pro Platform FPGA. Since the PowerPC processor does not provide floating-point instructions, the floating-point operations in the benchmark suite kernels are converted into the proper integer arithmetic. The DCT, IDCT, and ADPCM decoder kernels have been unrolled by a factor of 8/16 in order to increase the selection space of our algorithm. In the current MOLEN prototype, the access to the Exchange Registers (used for GPP-FPGA communication) (XRs) for the input/output values is significantly slower compared to the GPP register. As this is a limitation only in the current prototype and taking into account that other approaches on Instruction-Set Extension do not consider register accesses, for a fair comparison we report two set of results: with and without XR accesses. For our experiments, we consider a set of three well-known MediaBench [15] benchmarks and MJPEG encoder application. In Figure 4, we present the overall application speedup for FPGAs of various sizes compared to the pure software execution.[7] For ADPCM and SAD we only presented a small set of FPGAs since a further increase of the FPGA size does not provide additional improvement. The speedup estimation is

[7] The occupied area is not shown, since it is almost equal to the available area on the FPGAs.

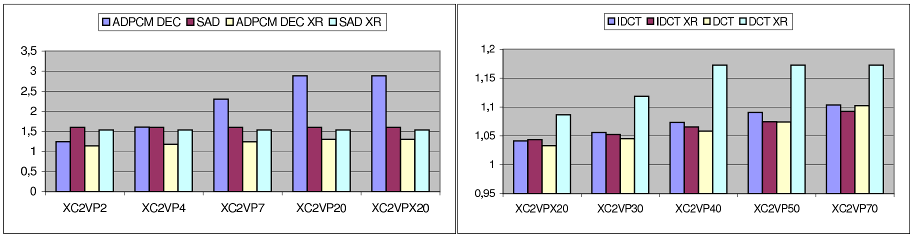


Fig. 4. Overall speedup for different FPGA sizes of **a**) ADPCM Decoder/MPEG2 Encoder, and **b**) MJPEG Encoder/MPEG2 Decoder

based on Amdahl's law, using the profiling results and the computed speedup for the kernels. The achieved speedup varies from x1.2 up to x2.9 for the ADPCM Decoder and different FPGA sizes. For the other benchmarks, the speedup is limited due to the shape of the dataflow graphs of the application.

An expected observation is that the impact on performance of the MM-Level algorithm increases with the size of the available FPGA area. This is explained by the fact that more MAXMISOs can be combined for hardware execution on the FPGAs. Additionally, we notice that for some applications/kernels (MPEG2 Encoder/SAD) the estimated speedup does not depend on the FPGA size. This is due to the fact that SAD kernel contains only one MAXMISO which fits on all FPGAs. A second observation is that the impact of the XR accesses on the overall speedup is higher for the ADPCM decoder compared to the SAD/DCT/IDCT kernels. This is due to the larger number of input/output values of the selected MAXMISOs and high XR access latency. A last observation is about the shape of the graph. The algorithm we propose in this paper is designed to work properly with graph having a large depth. The benchmarks we proposed to test our algorithm are usually represented by wide graphs with a small depth which limits the extension of C'' in C'''. Promising benchmarks to test our algorithm are, for example, the cryptographic benchmarks which usually are applications represented by graph with a large depth but at present they are not ready to test the algorithm.

It is important to note at this point that even though the overall speedup is limited, the overall complexity of our solution is linear in the number of processed elements[8]. Moreover, we emphasize that our approach does not impose limitations on the number of operands and on the number of new instructions.

5 Conclusions

In this paper, we have introduced an algorithm which combines clusters of MAXMISO for execution as new application-specific instructions on the reconfigurable

[8] More specifically, the number of processed elements is at most $n+m$, where n is the order of G and m is the number of MAXMISOs in G.

hardware. One of the main features of our approach is the linear overall complexity. Additionally, the proposed algorithm is general: new instructions have no limitation on their types and numbers and the algorithm is applied beyond basic block level. The used SW model in the experiments is simplified and does not reflect the available processor optimizations (pipelines, cache hits, etc). Besides we do not consider also the possible penalties like branch miss-predictions, cache misses, etc. Even though the generated VHDL is not optimized, we consider our results reliable and promising for future studies. A more general MAXMISO-clustering and operation-clustering concern a task for our future research plan.

References

1. Vassiliadis, S., Wong, S., Gaydadjiev, G., Bertels, K., Kuzmanov, G., Panainte, E.M.: The molen polymorphic processor. IEEE Trans. Comput. **53**(11) (2004) 1363–1375
2. Alippi, C., Fornaciari, W., Pozzi, L., Sami, M.: A dag-based design approach for reconfigurable vliw processors. (In: Proceedings of DATE '99)
3. Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-specific instruction generation for configurable processor architectures. (In: Proceedings of FPGA '04)
4. Atasu, K., Pozzi, L., Ienne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. (In: Proceedings of DAC '03)
5. Yu, P., Mitra, T.: Scalable custom instructions identification for instruction-set extensible processors. (In: Proceedings of CASES '04)
6. Atasu, K., Dündar, G., Özturan, C.: An integer linear programming approach for identifying instruction-set extensions. (In: Proceedings of CODES+ISSS '05)
7. Galuzzi, C., Panainte, E.M., Yankova, Y., Bertels, K., Vassiliadis, S.: Automatic selection of application-specific instruction-set extensions. (In: Proceedings of CODES+ISSS '06)
8. Kastner, R., Kaplan, A., Memik, S.O., Bozorgzadeh, E.: Instruction generation for hybrid reconfigurable systems. ACM Trans. Des. Autom. Electron. Syst. **7**(4) (2002) 605–627
9. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: Synthesis of custom processors based on extensible platforms. (In: Proceedings of ICCAD '02)
10. Brisk, P., Kaplan, A., Kastner, R., Sarrafzadeh, M.: Instruction generation and regularity extraction for reconfigurable processors. (In: Proceedings of CASES '02)
11. Baleani, M., Gennari, F., Jiang, Y., Patel, Y., Brayton, R.K., Sangiovanni-Vincentelli, A.: Hw/sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. (In: Proceedings of CODES '02)
12. Clark, N., Zhong, H., Mahlke, S.: Processor acceleration through automated instruction set customization. In: Proceedings of MICRO 36. (2003) 129
13. Goodwin, D., Petkov, D.: Automatic generation of application specific processors. (In: Proceedings of CASES '03)
14. Choi, H., Hwang, S.H., Kyung, C.M., Park, I.C.: Synthesis of application specific instructions for embedded dsp software. (In: Proceedings of ICCAD '98)
15. Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In: Proceedings of MICRO 30. (1997) 330–335

Evaluating Variable-Grain Logic Cells Using Heterogeneous Technology Mapping

Kazunori Matsuyama, Motoki Amagasaki,
Hideaki Nakayama, Ryoichi Yamaguchi,
Masahiro Iida, and Toshinori Sueyoshi

Department of Mathematics and Computer Science, Graduate School of Science and Technology, Kumamoto University,
2-39-1 Kurokami Kumamoto-shi, 860-8555, Japan

Abstract. Generally, reconfigurable logic devices have been classified as fine-grained or coarse-grained devices depending on the input size of their logic cells. These architectures have conflicting characteristics, which limits their application domain for an efficient implementation. In order to solve this constraint, we propose a variable grain logic cell (VGLC) architecture that exhibits the characteristics of both fine-grained and coarse-grained cells. In this study, we investigate a VGLC structure and its mapping technique. We evaluate the capability of VGLC with respect to its critical path delay, implementation area, and configuration data bits; we propose a maximum improvement of 49.7%, 54.6%, and 48.5% in these parameters, respectively.

1 Introduction

Reconfigurable Logic Devices (RLDs), which are typified by Field Programmable Gate Arrays (FPGAs), are becoming one of the main methods for implementing digital circuits. Due to their programmability, applications can be easily implemented; further, their Non-Recurring Engineering (NRE) costs are lower than those of Application-Specific Integrated Circuits (ASICs). However, [1] reported the logic density gap between ASICs and FPGAs to be ten times more implementation area and power consumption, and the operation speed gap to be at least three times lesser. These disadvantages can be attributed to their flexibility; in order to make them flexible, they require a larger number of wires, switches, and configuration memories as compared to ASICs. Many of applications impose critical requirements on performance, power consumption, and thus require innovative hardware architectures to meet these applications. Therefore, the development of high-performance RLDs is required. We propose a variable-grain logic cell (VGLC) architecture[2] to implement circuits efficiently. In this paper, we describe the VGLC structure and its mapping technique. We evaluate the capability of VGLC with respect to its critical path delay, configuration data bits, and implementation area.

The remaining part of this paper is organized as follows. Section 2 discusses the issues regarding conventional RLDs and elaborates on the results of the applications analysis. Section 3 describes our VGLC architectures. Section 4 discusses the VGLC mapping technique. Section 5 describes the evaluation methodology, and the results are shown in Section 6. Finally, the conclusion and an outline for future works are provided in Section 7.

2 Issues Regarding Conventional RLD

2.1 Operation Granularity of Logic Cells

Generally, RLDs are classified as fine-grained or coarse-grained devices depending on the operation granularity of their logic cells. A fine-grained logic cell mostly consists of Look-up Tables (LUTs) or multiplexers (MUXs), which can be observed in the logic blocks of the FPGA architecture. These logic cells can efficiently implement any circuit for typical random logic. In contrast, a coarse-grained cell consists of computational units such as an Arithmetic Logic Unit (ALU). Since ALUs provide higher circuit density and greater speed than LUTs, they are suitable for implementing arithmetic operations. However, such RLDs can be efficiently implemented only within their own application domains. When arithmetic applications are implemented using fine-grained devices, large resources are required and the operation speed is low. On the other hand, when typical random logic applications are implemented using coarse-grained devices, the area overheads are large. Therefore, conventional RLDs can be equipped with certain hard macros such as multipliers to reduce this disadvantage; however, if these macros are not utilized in an application, they are wasting space on the chip. oth random logic operations and arithmetic functions.

2.2 Analysis of Application Characterization

In this subsection, we present a type of application domain analysis. For this purpose, we use six applications from OPENCORES[3] as benchmark circuits; these circuits were synthesized by using the Synopsys Design Compiler with a 0.35-um library. We classified all the design components into three groups: Controllers, Ciphers and DSPs. We selected two applications for each class. The list of applications is listed in Table 1, and the results obtained from the area report of the design compiler are shown in Fig.1.

As a result, the ratio of random logic to arithmetic logic is not constant across different applications. In addition, it shows that MUXs occupies a large area in every circuit and has different input ranges. Therefore, it is necessary to efficiently implement these functions within the logical blocks. A similar analysis and results has been presented for digital signal processing (DSP) applications in [4].

Table 1. Benchmark applications for analyzing the operation class

Circuits	Class	Overview
(A) Ac97	Controller	Audio codec controller
(B) Vga	Controller	VGA display controller
(C) aescipher	Cipher	Cipher codec core (128bit)
(D) Sha256	Cipher	Hash-function generator core
(E) Dct	DSP	Scanning and quantization circuit
(F) Biquad	DSP	Biquad IIR filter

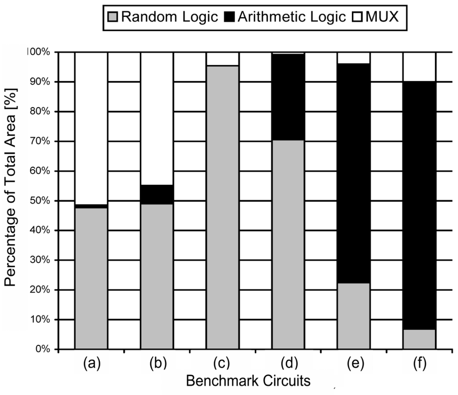


Fig. 1. Synthesis results

3 Proposed Logic Cell Architecture

As mentioned earlier, it is difficult to improve the performance of conventional RLDs which has the fixed granularity. Thus, we consider a logic cell that can change its operation granularity so that it can adjust to any application. This technique can simultaneously improve the operation performance and area efficiency. In this section, we propose a Hybrid Cell (HC) and VGLC (requiring four HCs), and describe their architectures and functions.

3.1 Hybrid Cell

The arithmetic computation is carried out by an adder, and the random logic functions can be efficiently expressed in a canonical form, such as LUTs. We conclude that it is necessary to implement both these techniques by means of logic cells, and thus, we focus on the common section of the circuit between a one-bit full-adder (FA) and two-input Reed-Muller canonical form (2-CF). The latter consists of four configuration memory bits; it can express any two-input function in the same manner as a 2-LUT. Further, 2-CF can be described by the following equation, (each part enclosed within the parentheses corresponds to a configuration memory bit).

$$\begin{aligned} F(x_0,x_1) = &\{F(0,0)\} \\ &\oplus x_0\{F(0,0)\oplus F(1,0)\} \\ &\oplus x_1\{F(0,0)\oplus F(0,1)\} \\ &\oplus x_0x_1\{F(0,0)\oplus F(0,1)\oplus F(1,0)\oplus F(1,1)\} \end{aligned} \quad (1)$$

On comparing (a) with (b) in Fig.2, common EOR and AND gates can be observed. We construct a HC, as shown in Fig.2(c), by appending four-bit configuration memories to a one-bit FA. HC can express FA and CF according to the desired computation. If we implement a one-bit FA using the traditional FPGA architecture consisting of 4-LUTs, one 4-LUT comprising 16 configuration memory bits and carry-logic are required. On the other hand, HCs require

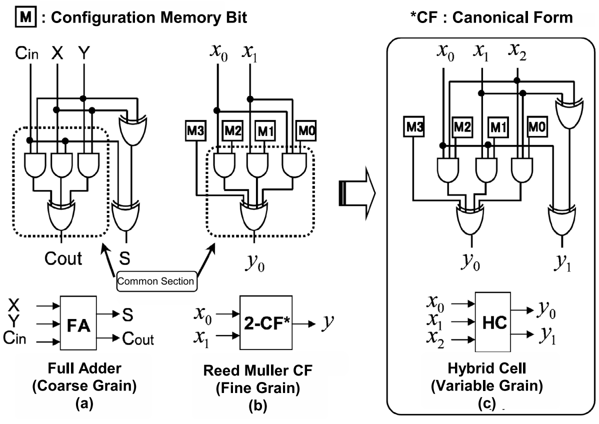

M : Configuration Memory Bit
*CF : Canonical Form
Cin X Y
Cout S
Common Section
M3 M2 M1 M0
x0 x1 x2
y0 y1
X Y Cin FA S Cout
Full Adder (Coarse Grain) (a)
x0 x1 2-CF* y
Reed Muller CF (Fine Grain) (b)
HC
Hybrid Cell (Variable Grain) (c)

Fig. 2. Basic structure of Hybrid Cell

only four configuration memories; therefore a large amount of configuration data can be reduced.

3.2 Variable Grain Logic Cell

Fig.3 shows the VGLC architecture. The structure within the flamed box is defined as the basic logic element (BLE). To select addition or substruction with signal AS, we add EXOR gate in BLE. It comprises a HC and MUXs and

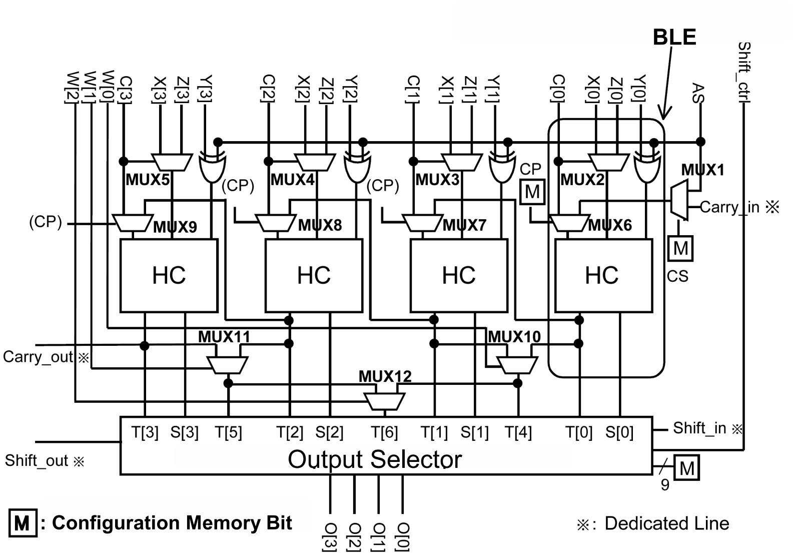


Fig. 3. VGLC Architecture

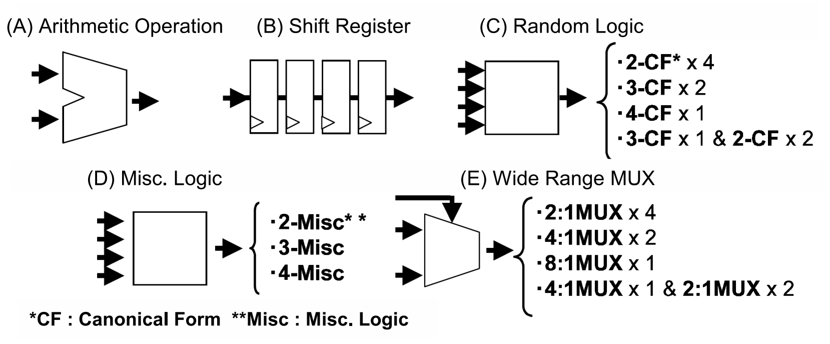


Fig. 4. Basic Operation Modes of VGLC

EXOR gates to expand the HC functionalities. VGLC consists of four BLEs, output selector, and 27-bit configuration memory. It has 21-bit input and 4-bit output. Carry_in, carry_out, shift_in, and shift_out signals are directly connected to neighboring VGLCs as dedicated lines. AS and CP (carry-select memory) signals are commonly inputted to the entire BLE in the VGLC.

3.3 Basic Operation Modes of VGLC

Fig.4 shows the five basic operation modes of VGLC. In the following subsection, they are described in detail.

(A) Arithmetic Operation Mode
In this mode, the VGLC performs four-bit addition or subtraction operations. Here, all the BLEs are set as FAs and are connected with the carry line (controlled by the CP signal). Carry propagation between the BLEs is fast because it occurs through a dedicated carry line. In order to dynamically control addition or subtraction in the VGLC, the AS signal is used. The carry path between the neighboring VGLCs is connected as dedicated lines so that the operation bit width can be increased.

(B) Shift Register Mode
The output selector of the VGLC consists of four flip-flop circuits and a few MUXs. It functions as a four-bit shift register; it can increase the operation bit width by using a dedicated line (Shift_in and Shift_out) similar to that in the arithmetic operation mode. In addition, if Shift_ctrl is appropriately controlled, we can make serial-to-parallel or parallel-to-serial converters can be developed.

(C) Random Logic Mode
When the BLE is used as a 2-CF, it can express any two-input function in the same manner as a 2-LUT. 3- or 4-CF can be expressed by using two or four BLEs by using Shannon expansion. According to the previous studies on FPGA architectures, it is known that a 4-LUT has the highest area efficiency[5];

therefore, VGLC comprises four BLEs. In the random logic mode, a VGLC can have a heterogeneous composition: (1)four 2-CFs, (2)two 3-CFs, (3)two 2-CFs and one 3-CF, and (4)one 4-CF. This implies that a VGLC can allocate minimum BLE resources required for implementing logic circuits, thereby achieving high area efficiency and reducing the number of configuration memories.

(D) Misc. Logic Mode
It is shown that VGLC can represent 2-, 4-input CFs; however, it still requires a larger area than the LUT. We utilize the Miscellaneous Logic (Misc. Logic) function, which applies its gate structure, in order to map the random logic.

The BLE used in the VGLC is four-input and two-output black box. It can represent a maximum of a four-input logic function. This means that one BLE can implement a limited three- or four-input logic function with configuration bits equal to that in a 2-CF.

A K-LUT can implement 2^{2^K} types of logic functions; thus, 65,536 and 256 functions can be implemented by using 4- and 3-CFs, respectively. Table 2 lists the logic patterns that can be expressed at the output of T and S. Since a BLE can be expressed using 2-CF, a two-input variable can be excluded. It is note worthy that the polarity of input Y is influenced by the AS signal. For example, since AS = 0 in the three-input variable, T and S are capable of expressing 120 and 43 logic patterns, respectively. In total, the Misc. Logic mode can represent

Table 2. Misc. Logic coverage using one BLE

	four-input variable		three-input variable	
AS	T	S	T	S
0	182/65,536	24/65,536	120/256	43/256
1	230/65,536	24/65,536	148/256	43/256
Total	446/65,536 (0.68%)		206/256 (80.47%)	

Table 3. Misc. Logic coverage by using multiple BLEs

	four-input variable (two BLEs)	five-input variable (four BLEs)
AS = 0	21.97%	4.83%
AS = 1	33.42%	11.17%
Total	41.91%	14.18%

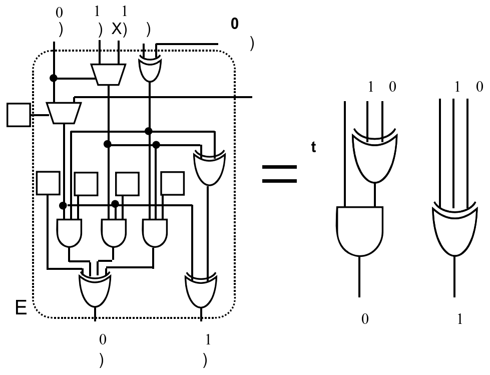


Fig. 5. Example of three-input Misc. Logic function

81.3% of all three-input logic patterns and 0.68% of all four-input logic patterns. Fig.5 shows an example of using the Misc. Logic mode.

According to Shannon expansion, VGLC can implement limited four- or five-input logic circuits in a manner similar to the random logic mode. Table 3 lists the total coverage of the Misc. Logic mode. It can be seen that four-input Misc. Logic can be used to implement 41.91% of all the four-input variables, and five-input Misc. Logic can be used to implement 14.18% of all the five-input variables. If we implement the typical random logic using the Misc. Logic mode, the number of required logic cells reduces by at least half when compared with random logic mode.

(E) Wide Range MUX mode
One VGLC can express some wide range MUXs, as shown in Fig.4(E). One BLE can essentially express one 2:1 MUX. By using MUX10-12 shown in Fig.3, it is possible to implement 4:1 or 8:1 MUX. It is a useful technique for implementing various applications mentioned in Section 2. For these reasons, VGLCs can efficiently implement applications by changing their operation modes and granularities for each operation. In particular, when the typical random logic operations are implemented, the area efficiency can be rapidly improved when the VGLC is preferentially in the Misc. Logic mode. The effect of Misc. Logic is described in Section 6.

4 Technology Mapping Method

4.1 Mapping Flow

First, in order to efficiently map using five modes in VGLC, arithmetic operations and multiple-bit MUXs are registered as macro blocks to the user library for mapping process. We can extract these circuits as a macro blocks during logic synthesis. The circuits, which are not covered, are mapped by using Misc. Logic and random logic mode. Finally, we include some information about the macro blocks and complete the mapping process.

4.2 HeteroMap

We need to implement typical random logic circuits using VGLC structure. FlowMap[6] is a well-known technology mapping algorithm used for FPGAs. However, it functions only with a homogeneous logic cell architecture such as 4-LUT-based FPGAs. In contrast, HeteroMap[7], developed by J.Cong et al. at UCLA, can function with a heterogeneous architecture that may comprise varying LUTs such as that in the Xilinx XC4000 series FPGAs.

The source code of the rapid system prototyping (RASP), —a mapping system based on the HeteroMap algorithm developed in UCLA, — is described in the C language and has been published[8]. We expand the code so that it can be employed with the VGLC architecture; it is referred to the VGLC-HeteroMap.

The key expansion elements are as fellows;

– adjusting the netlist to include the macro blocks and
– adding the Misc. Logic mapping mode

First, a HeteroMap is developed to map netlists using LUTs with only one output; however, macro blocks in VGLC yield outputs with multiple bits. In order to adjust this, we include a macro-block mapping mode in the VGLC-HeteroMap. Second, we map the netlists by using the VGLC in the Misc. Logic and random logic modes. The HeteroMap is expanded to adjust the Misc. Logic mapping mode. Further, the algorithm was not changed for the random logic mode because the processing is similar to that of LUT mapping. The pseudo-code is shown in Fig.6.

```
program VGLC_HeteroMap(network)
  for $n$ := each PI to PO do /* n is node */
    $min\_label := \infty$;
    if $n$ equal macro then
      Compute $Label(n)$ and $Cut(n)$ for each macro blocks.;
    else
      for Each granurality of CF and Misc. logic function do
        Compute $Label(n)$ and $Cut(n)$ with delay and # of input.;
        if Misc. logic function then
          if Mismatch the logic pattern. then continue;
        if $Label(n) < min\_label$ then
          $min\_label := Label(n)$; $min\_cut := Cut(n)$;
        end if
      end for
    $Label(n) := min\_label$; $Cut(n) := min\_cut$;
  end for
  Map with the selected function.;
end program
```

Fig. 6. Pseudo-code in random logic mapping

5 Evaluation Methodology

5.1 Target Architecture

We need to compare fairly both the coarse-grained and fine-grained types with VGLC. However, coarse-grained cells are not uniform and CAD tools are unavailable for free; therefore, in this study, we evaluate the fine-grained and VGLC architectures. The fine-grained cell, which consists of a 4-LUT, MUX and flip-flop circuit, is shown in Fig.7. This is the most popular architecture used in conventional FPGAs[9]. The number of configuration memory bits of the cell (NB) is $NB_{4-LUT} = 17$. In this evaluation, the logic delay (T) is set as $T_{4-LUT} = 1.6ns$. We referred to the data sheet of Xilinx XC4000[10], which is prepared using a 0.35-um technology process. In contrast, we designed a VGLC using a 3.3-V

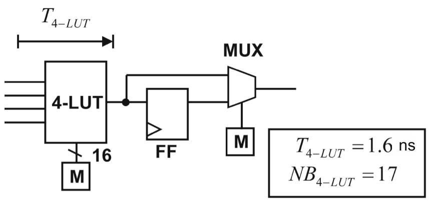


Fig. 7. LUT-based logic cell for comparison purpose

Table 4. Implementation parameter of VGLC

Function	$T_{func.}$[ns]
2-CF	0.77
3-CF	1.28
4-CF	1.35
3-Misc. Logic	0.80
4-Misc. Logic	1.29
5-Misc. Logic	1.34

transistor model referring to the 0.35-um technology. The delay is computed from a simulation performed using HSPICE, a circuit simulator. The results are listed in Table 4. The value of $T_{func.}$ in the table denotes the delay of each function; and they are used as the delay information in the VGLC-HeteroMap.

5.2 Evaluation Methodology

The mapping methods used in this evaluation are listed in Table 5, and the evaluation flow is shown in Fig.8. First, a benchmark circuit written in Verilog-HDL is synthesized by using Design Compiler 2003.03 and a gate level netlist is generated. Adder/subtractors and wide-range MUXs are extracted as macro blocks by using Design Ware. Second, we convert the generated netlist into the BLIF format by using a filter written in Perl. Then, this netlist is mapped by using the VGLC-HeteroMap with three methods— (A), (B), and (C)— shown in Table 5. In contrast, in method (D), we all the hierarchies are destroyed to

Table 5. Classification of mapping methods

	Mapping method
(A)	Only random logic mode
(B)	(A) + Misc. Logic mode
(C)	(B) + macro block
(D)	4-LUT(For comparison logic cell)

Table 6. MCNC benchmark circuits

MCNC circuits		
alu4	apex1	c1355
clip	cordic	e64
s1196	sbc	sct
table5		

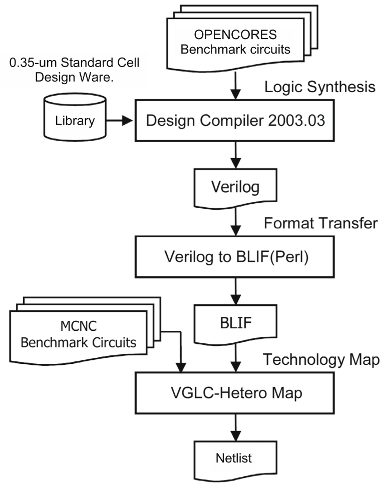


Fig. 8. Architecture evaluation flow

reveal a flat netlist when netlist is synthesized for the 4-LUT. Then, the result is converted into the BLIF format and the netlist is mapped onto the 4-LUT architecture by using FlowMap. Finally, we compare the results generated by VGLC-HeteroMap and FlowMap from the view point of the number of required logic cells, critical path delay, configuration data and implementation area.

5.3 Benchmark Circuits

In this evaluation, we use six circuits from OPENCORES listed in Table 1 and ten MCNC benchmark circuits[11] listed in Table 6. Although MCNC benchmark circuits were available in the electronic design interchange format (EDIF), we were unable to extract the macro blocks. Therefore, the mapping evaluation using macro blocks (C) was performed using only the OPENCORES benchmark circuits. While selecting the benchmark circuits, care was taken to balance the operation ratio of each application.

6 Evaluation Results

6.1 Number of Logic Cells and Critical Path Delay

The left-hand side of Table 7 lists the number of logic cells required to implement the benchmark circuits. (A)-(D) represent the mapping methods listed in Table 5. Columns (A)-(C) represent the required number of VGLC: the improvement ratio with respect to (A) is provided in the parentheses. Column (D) represents the required number of 4-LUT cells. Since macro blocks could not be extracted from the Vga, aes cipher, and MCNC benchmark circuits, they are not listed in column (C). The results of (B) and (C) are improved by a maximum of 46.9% and 50.9% and by an average of 29.4% and 41.6%, respectively. According to these results, larger circuits, such as sct and Dct, have better area efficiency.

The right-hand side of Table 7 lists the values of the critical path delays. Columns (A)-(C) list the values of critical path delays for VGLC and improvement ratios with respect to (D) are provided in the parentheses. Column (D) lists the value of the critical path delays of 4-LUT-based FPGAs. The critical path delays listed in (A), (B), and (C) are improved by a maximum of 40.4%, 49.7%, and 37.8% and by an average of 21.0%, 32.9%, and 23.1%, respectively, when compared with the results of (D). The delay of sha256 and Biquad in (C), which use macro blocks, are increased when compared with the result of (B). This is due to the differences in the mapping methodology. When the macro blocks are extracted, borders between the macro blocks and random logic operations are formed in the netlist. However, all the hierarchies are destroyed to reveal a flat netlist when the result of (B) are synthesized. Then, the borders are eliminated due to the integration of LUTs and the number of stages gets decreased. Therefore, the critical path delay of the application with macro blocks on the critical path increases in (C). It should be noted that this delay is the result of technology mapping because the wiring architecture considerable influence the delay. Hence, VGLC improves the circuit delay and the number of required logic cells.

Table 7. Number of logic cells and critical path delay

Circuit	Number of required logic cells (improvement rates from (A) [%])				Critical path delay [ns] (improvement rates from (D) [%])			
	(A)	(B)	(C)	(D)	(A)	(B)	(C)	(D)
alu4	1,135	818 (27.9)	—	2,163	9.17 (18.1)	8.69 (22.4)	—	11.20
apex1	501	417 (16.8)	—	911	9.04 (19.3)	8.07 (27.9)	—	11.20
c1355	72	62 (13.9)	—	74	6.19 (3.3)	5.39 (15.8)	—	6.40
clip	157	127 (19.4)	—	342	6.79 (29.3)	6.12 (36.3)	—	9.60
cordic	640	459 (28.4)	—	944	11.18 (22.3)	9.98 (30.7)	—	14.40
e64	223	149 (33.3)	—	440	5.13 (19.9)	3.68 (42.6)	—	6.40
s1196	183	122 (33.6)	—	294	9.37 (26.8)	7.97 (37.8)	—	12.80
sbc	249	181 (27.2)	—	437	6.67 (40.4)	5.64 (49.7)	—	11.20
sct	4,461	2,368 (46.9)	—	5,872	10.17 (20.6)	9.62 (24.8)	—	12.80
table5	368	309 (16.2)	—	610	8.55 (23.6)	7.59 (32.2)	—	11.20
Ac97	4,816	2,792 (42.0)	2,678 (44.4)	6,541	4.77 (25.4)	3.98 (37.8)	3.98 (37.8)	6.40
Vga	754	587 (22.2)	—	1,808	12.07 (16.2)	10.71 (25.6)	—	14.40
aescipher	7,678	5,071 (34.0)	—	10,466	10.65 (16.8)	8.95 (30.1)	—	12.80
Sha256	4,143	3,004 (27.5)	2,243 (45.9)	6,468	36.33 (15.9)	28.29 (34.5)	30.25 (30.0)	43.20
Dct	31,596	18,358 (41.9)	17,931 (43.3)	52,518	21.50 (20.9)	16.19 (40.5)	19.60 (28.0)	27.20
Biquad	1,071	648 (39.5)	525 (50.9)	1,612	22.54 (17.1)	16.95 (37.7)	28.11 (–3.4)	27.20
Average		(29.4)	(46.1)		(21.0)	(32.9)	(23.1)	

6.2 Configuration Data

Configuration data can be computed using following expression:
Configuration data = Configuration memory par cell × number of logic cells

The number of configuration memory bits required for VGLC and 4-LUT cells is 27 bits and 17 bits, respectively. Table 8 lists the improvement ratios of the configuration data from (D). Configuration data of (A) is improved by an average of 0.5%. The results of (B) and (C) are improved by a maximum of 48.5% and 47.6% and by an average of 30.0% and 43.3%, respectively. Hence, VGLCs require a smaller amount of configuration data than 4-LUT-based FPGAs; therefore, the reconfiguration cost can be drastically reduced.

6.3 Implementation Area

The area model used in this evaluation is shown in Fig.9: (a) shows one RLD logic tile comprising a logic block and wiring tracks. A_{route} and A_{LB} denote the areas of the wiring tracks and the logic blocks which include configuration memories, respectively. A_{FPGA} and A_{VGLC} in (b) denote the areas of the logic tiles of FPGA and VGLC, respectively. With regard to [12], the ratio of A_{route} and A_{LB} in LUT-based-FPGA is 1:4. In our previous study[2], the number of configuration transistors is three times greater in VGLC than FPGA, as demonstrated by A_{LB} in Fig.9(b). We assume that VGLCs require the same amount of wiring area as FPGAs. Therefore, the ratio A_{FPGA}:A_{VGLC} is defined as 5:7. We use this model as the area of one tile. To compute the implementation area, we multiply the area and the number of required logic cells listed in Table 7. The result is listed on the right-hand side of Table 8. By using the Misc. Logic mode, the results can be improved by a maximum of 54.6% and by an average of 38.3%. By using macro blocks, the result can be improved by a maximum of 54.4% and by an average of 50.2%. Therefore, VGLC improves the area efficiency.

Table 8. Improvement ratios of configuration data and implementation area from (D)

Circuit	Configuration memory bits [%]			Implementation area [%]		
	(A)	(B)	(C)	(A)	(B)	(C)
alu4	16.6	39.9	—	26.5	47.0	—
apex1	12.7	27.4	—	23.1	36.0	—
c1355	–54.5	–33.1	—	–36.2	–17.3	—
clip	27.1	41.2	—	35.7	48.2	—
cordic	–7.7	22.9	—	5.1	32.0	—
e64	22.5	48.3	—	29.0	52.6	—
s1196	1.1	34.4	—	12.9	42.1	—
sbc	9.6	34.2	—	20.3	42.0	—
sct	–20.7	36.0	—	–6.4	43.5	—
table5	4.2	19.7	—	15.5	29.2	—
Ac97	–16.8	32.2	35.0	–3.1	40.3	42.7
Vga	33.8	48.5	—	41.6	54.6	—
aescipher	–16.5	23.1	—	–2.7	32.2	—
Sha256	–1.7	26.2	44.9	10.3	35.0	51.5
Dct	4.4	44.5	45.8	15.8	51.1	52.2
Biquad	–6.9	35.3	47.6	7.0	43.7	54.4
Average	0.5	30.0	43.3	12.2	38.3	50.2

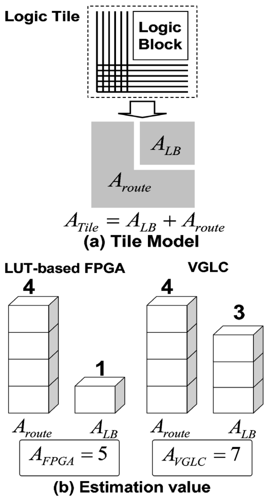


Fig. 9. Area model per tile

7 Conclusion and Future Works

In this paper, we describe our logic cell, VGLC which changes its operation granularity and operation mode depending on the application. We evaluate it with respect to area, delay, and configuration data by implementing benchmark circuits and technology mapping tools for VGLC. By comparing VGLC- and 4-LUT-based logic cells, the values of implementation area, critical path delay, and configuration data improved by a maximum of 54.6%, 49.7%, and 69.4%, respectively. We conclude that VGLC has the ability to improve the area efficiency and operation speed.

In the future, we would investigate the wiring architecture of VGLCs by using the results obtained from this evaluation. We will fabricate a VGLC chip and evaluate its performance using fine-grained and coarse-grained devices. The evaluation performed in this study used only two types of macro blocks: adders and MUXs. We will include multipliers in this list to further improve the VGLC performance. Finaly, we will develop clustering, placing, and rooting tools for VGLCs.

References

1. I. Kuon, J. Rose, “Measuring the Gap between FPGAs and ASICs,” Proc. of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays , pp.21-30, Feb. 2006.
2. M. Amagasaki, T. Shimokawa, K. Matsuyama, R. Yamaguchi, H. Nakayama, N. Hamabe, M. Iida, T. Sueyoshi, ”Evaluation of Variable Grain Logic Cell Architecture for Reconfigurable Device,” Proc. of The 14th IFIP International Conference on Very Large Scale Integration, pp.198-203(COct. 2006.

3. Opencores org, http://www.opencores.org/.
4. K. Leijten-Nowak, J.L. van Meerbergen, "An FPGA Architecture with Enhanced Datapath Functionality," Proc. of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, pp.195-204, Feb. 2003.
5. J. Rose, R. Francis, D. Lewis, P. Chow, "Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," IEEE J. Solid-State Circuits, Vol.25, No.5, pp.1217-1225, Oct. 1990.
6. J. Cong, Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs," Trans. on CAD, Vol.13, No.1, pp.1-12, Jan. 1994.
7. J. Cong, S. Xu, "Delay-Oriented Technology Mapping for Heterogeneous FPGAs with Bounded Resources," Proc. ACM/IEEE International Conference on Computer Aided Design, pp.40-45, Nov. 1998.
8. UCLA CAD-LAB, http://cadlab.cs.ucla.edu/.
9. V. Betz, J. Rose, and A. Marquardt, "Architecture and CAD for Deep-Submicron FPGAs," Kluwer Academic Publishers, 1999.
10. Xilinx, Inc., Xilinx XC4000 Data Sheet, 1999.
11. K. McElvain, "IWLS'93 Benchmark Set: Version 4.0," Distributed as part of the MCNC International Workshop on Logic Synthesis '93 benchmark distribution, May 1993.
12. A. Dehon, "Reconfigurable Architectures for General-Purpose Computing," MIT, Artificial Intelligence Laboratory, AI Technical Report No.1586, 1996.

The Implementation of a Coarse-Grained Reconfigurable Architecture with Loop Self-pipelining

Yong Dou, Jinhui Xu, and Guiming Wu

School of Computer Science, National University of Defense Technology, Changsha, Hunan, China, 410073
{yongdou,jinhuixu,guimingwu}@nudt.edu.cn

Abstract. Two critical points lie in applying loop pipelining in coarse-grained reconfigurable arrays. One is the technique of dynamically loop scheduling to achieve high pipeline throughput. The other is memory optimizing techniques to eliminate redundant memory accesses or to overlap memory accessing with computing. In this paper, we present the implementation techniques in LEAP, a coarse-grained reconfigurable array. We propose a speculative execution mechanism for dynamic loop scheduling with the goal of one iteration per cycle. We present the implementation techniques to support the decoupling between the token generator and the collector. We introduce implementation techniques in exploiting both data dependences of intra-iteration and inter-iteration. We design two instructions for special data reuses in the case of loop-carried dependences. The experimental results show the reduction in memory accesses reaches 72.8 times comparing with the approaches with no memory optimization in a RISC processor simulator.

1 Introduction

Reconfigurable computing appears to be a very promising solution for application-specific speedup, in which fine or coarse grained reconfigurable arrays are tightly coupled with general-purposed processors (GPP). While GPPs are responsible for tasks scheduling and data management, arrays of function units deal with computation-intensive procedures and loops. Since the hardware structure can be reorganized according to the data-flow graph (DFG) of programs, it has been shown that reconfigurable arrays have great performance potential in accelerating the scientific or multimedia applications. Much of research works in the area of reconfigurable computing focus on loop pipelining. The efforts roughly lay in certain of two broad fields. The first is the area of hardware pipeline, in which the research includes organizing the processing elements of array into a chain of pipelines in coarse-grained reconfigurable architecture, synthesizing loops into hardware pipeline based on software vectoring or translating imperative programs into hardware dataflow machines. The other field is software pipelining [4], closely related with VLIW architecture, in which different loop iterations are executed in parallel with precise scheduling of compilers. In both fields, the loop scheduling of DFG is the key not only for correct execution but also for high pipeline throughput.

Research works on the model of data driven computing, for example, Static DFG, FIFO DFG and Tagged DFG lay a firm foundation to describe the pipeline execution of DFG. Especially, in the work of spatial computing[5] at the Phonix project of CMU, they proposed a dataflow internal representation, called Pegasus for hardware synthesis from high level languages. The dataflow IR provides a virtual instruction set architecture for dataflow execution and program optimizations, such as, pipelining memory accesses in loops, inter-iteration register promotion and dataflow software pipelining. However, there is still space for tradeoffs and optimizations in implementing the virtual architecture in reconfigurable arrays.

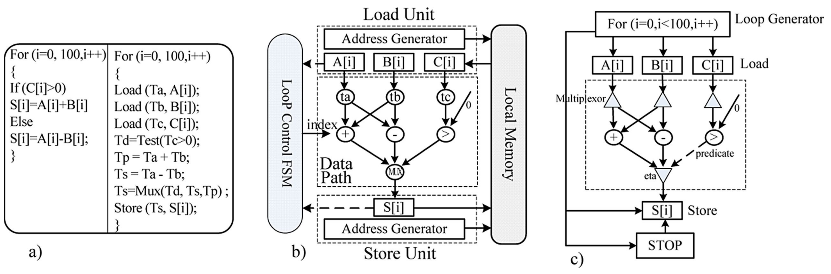
This paper focuses on the implementations of loop self-pipelining technique in the context of a coarse-grained reconfigurable architecture, LEAP (Loop Engine Array Processor)[6].We propose a new approach for automatically loop-iteration controlling with hardware supports, in which we distribute the functions of loop-index stepping into every memory accessing unit, called mPE (memory processing element). Combining the active index values, mPEs generate the address streams and drive the data stream into data path, composed of cPE (computing processing element), and finally into the result memory. We also investigate approaches of memory optimization to reduce the bandwidth requirements in both of external memory and local memory. The contributions of this paper are:

- We propose a speculative execution mechanism for dynamic loop scheduling with the goal of one iteration per cycle. We present the implementation techniques to support the decoupling between the token generators and collectors.
- We introduce implementation techniques in exploiting both data dependences of intra-iteration and inter-iteration. We design two instructions for special data reuses in the loop-carried dependences. The experimental results show the reduction in memory accesses reaches 72.8 times comparing with the approach with no memory optimization in a RISC processor simulator.

The remainder of this paper is organized as follows. In Section 2, some brief overview on LEAP architecture and dataflow representation is presented. We present our speculative execution technique in Section 3.1. The memory optimization techniques are introduced in Section 3.2. We present experimental results in Section 4.

2 LEAP Architecture Overview

The idea of LEAP architecture directly comes from the practice of transforming loop structures to custom hardware designs in handcraft. As shown in Fig 1-a, a program segment of loop, the left part is the source codes in C and the right part is the codes transformed in pseudo-assembly language. For hardware



```
For (i=0, 100,i++)
{
If (C[i]>0)
S[i]=A[i]+B[i]
Else
S[i]=A[i]-B[i];
}
```

```
For (i=0, 100,i++)
{
Load (Ta, A[i]);
Load (Tb, B[i]);
Load (Tc, C[i]);
Td=Test(Tc>0);
Tp = Ta + Tb;
Ts = Ta - Tb;
Ts=Mux(Td, Ts,Tp) ;
Store (Ts, S[i]);
}
```

Fig. 1. a) A sample loop program b) The conceptual structure of custom hardware for loop programs c)The dataflow representation of the sample loop program

implementation, a concept structure is presented in Figure 1-b, where the dataflow graph is transformed into an abstract hardware structure.

We adopt the IR of Pegasus to represent the dataflow execution of virtual hardware structure. Figure 1-c shows the data flow graph of the loop in Figure 1-a using IR of Pegasus. In addition to the nodes of Multiplexors, Eta, Arithmetic and Boolean, we add nodes of Load and Store to express the accesses to memory and nodes of Loop Generator and Stop to indicate the loop iterating and stopping. The rectangular node of Load operation has one input of address and one output of the returned data. The rectangular of Store operation has two inputs, address and data, and one output, the finished token. Since the computation for addresses and data are completely driven by Loop Generator, no explicit tokens edges are needed in the dataflow graph. The node of Loop Generator sends tokens, indicated by the value of loop iteration variable with a valid bit, to Load nodes and Store nodes to start the memory references continuously. The node of Stop also receives the tokens of valid index value from Loop Generator and checks the conditions for loop finishing. In the following sections of 3.1, we will introduce how loops are speculatively executed with the coordination of Loop Generator node and Stop node.

The architecture of LEAP modularizes the above concept structure to eliminate centralized control bottlenecks, and parameterize the loop controller, address generator and data path connections to execute all sorts of loops with different configurations, as shown in Figure 2. We have mPE for loop controlling and data accessing. It is the mPE that combines the functions of loop controlling and data accessing together to drive the execution of loops. Each of mPEs directly supports one load operation and one store operation of pseudo-assembly language, and has an independent, parameterized loop FSM. The function of the Loop Generator node and the Stop node are combined to one of mPEs. After mPE is fired, it can continually generate requests to read memory, while it can store results to memory. Logical and arithmetic statements within loop bodies are mapped to cPEs. Without loop FSM, cPE acts as a data driven computing units. Whenever the required data are ready, the cPE fetches and streams them to result FIFOs in mPE. There is a network of switch-matrix router connecting all PEs together. Since the topology of interconnection among mPE and cPE

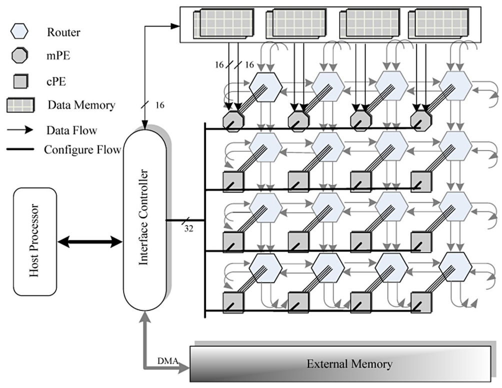


Fig. 2. The architecture of LEAP with host processor, external memory, distributed memory modules, 3 × 4 cPE and 1 × 4 mPE, connected by 4 × 4 routers

remains unchanged during the execution of a given loop, the network configurations are stored in a configuration memory.

3 Implementation of Loop Self-pipelining in LEAP

The virtual architecture works like its hardware counterpart in a pattern termed Loop Self-Pipelining. With a few cycles of starting delay, we hope the pipeline can achieve the performance of one loop-iteration per clock cycle. To reach the goal, such as the automatically loop scheduling, inter-iteration dependence resolving, effective data reusing and cost-effective implementation have to be explored. The following section will discuss the solution of above issues in implementing LEAP.

3.1 Speculative Execution of Loop Self-pipelining

The research work of Spatial Computing proposed dataflow software pipelining to accelerate the pipeline execution, but the throughput is still suffer from the latency of feedback edges in calculation of loop induction variables. Figure 3-a shows the example program discussed in [5].This program uses i as a basic induction variable to iterate from 1 to 10, and sum to accumulate the sum of squares of i. After several steps of optimization, the throughput is limited in 4 cycles per iteration, because the critical path lays in updating the value of i and producing the predicate to check the end of loop. To cope with this limitation, we propose a mechanism of speculative execution, a loop stepping FSM and a Loop Stop FSM (Finite States Machine).

We combine the functions of Loop Generator, Loop Stop, Load operation and Store operation in mPE. The number of mPE is configurable according to the hardware resources. Each mPE is able to support one Load operation and one Store operation to local memory respectively. The function of Loop Generator and Loop Stop are realized by Loop Stepping FSM and Loop Stopping FSM in

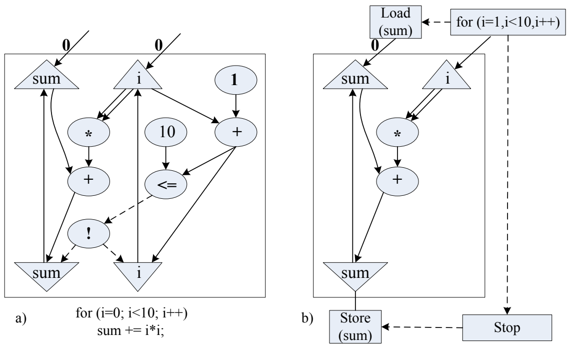


Fig. 3. a)The dataflow representation of the SUM loop b)The dataflow representation with speculative execution

each mPE respectively, but only one of mPEs is active in charge of the function of Loop Stop during the execution.

The Loop Stepping FSM works in a speculative mode, in which it does not check the conditions of loop ending at every cycle. To achieve one iteration per cycle, the Loop Stepping FSM updates the value of iteration variable per cycle without waiting the signal from Loop Stop FSM. Only when the receiving FIFO of its successors reaches the state of almost full, Loop Stepping FSM temporarily stops issuing tokens. As soon as Loop Stop FSM informs the end of loop, Loop Stepping FSM goes into the ending state, in which no further tokens are produced. Since the loop stepping is not synchronized with the checking information of loop end at every cycle, there are a few cycles of latency before the loop end information reaches the Loop Stepping FSM. Thus, after the loop end conditions are committed, several loop iterations have been activated. Measures have to be taken to guarantee that the computing results of escaped loop iterations are not stored to memory.

The Store operations of mPE have to obey the strict rules. Each mPE can receive one token for each loop iteration from Loop Stop FSM. The token includes the iteration number and a flag bit of loop ending. For example, the token of $\langle 7, True \rangle$ indicates that at iteration of 7, the loop reaches the condition of loop ending. In implementation of LEAP, the iteration number is replaced with a bit of enable signal, because the iteration number increases in sequence and the iteration is issued in order. Thus, the three conditions have to be all satisfied before mPE issuing the results of the loop iteration I to memory. The first condition is the token received from Loop Stop FSM indicates $\langle iteration\ I, False \rangle$. The second is the results of iteration I are ready, and the third is the addresses are ready and valid.

Loop Stop FSM plays an important role in the speculative execution of Loop Self-Pipelining. First, it receives predicate results per iteration from cPE or from Loop Stepping FSM. The predicates from Loop Stepping FSM indicate the end of normalized loops. The predicates from cPE are used to support Break operations of Loop programs. The Loop Stop FSM combines the two predicates to signal mPEs. We have to notice the scenario that after Loop Stop FSM have send signals of loop ending to mPE, there are still results waiting in Store FIFO of mPE. After sending the ending signal, the Loop Stop FSM enters the state of waiting all results written back to memory. The loop will not be completely finished to the state of loop ending until all mPE informs that Store FIFO is empty. After Loop Stop FSM reach the end state, there still exist unused results and intermediate data in cPE or mPE due to the speculative execution. Loop Stop FSM will send a signal to clear those unused data and reset all states.

As shown in Figure 3-b, the calculations of induced variable of I and predicate of loop ending are replaced with a Loop Stepping FSM $\langle for\ i = 1,\ i < 10,\ i{+}{+}\rangle$. The values of i are produced one by one until the loop reach the end conditions. If the accumulation of sum can be calculated in one cycle, the throughput of pipelines can reach one iteration per cycle. With the mechanism of speculative execution, the loop iterations can be triggered without the synchroning with the predicates of loop ending. Load requests are issued in pipelines and data are prefetched in advance. Whenever the data are ready, the throughput can reach one iteration per cycle.

3.2 Implementations of Data Reusing

Memory accessing is a great challenge to keep the loop pipelines in high throughput. To achieve the performance of one iteration per cycle, the memory controllers have to provide an amount of data per cycle. The bandwidth of memory becomes the bottleneck to the goal of high throughput. Exploiting data dependences of loop kernels, the scheme of data reusing can reduce the number of memory references or eliminate partial memory accesses. Below, we will introduce how data are reused in LEAP, according to the types of data dependency.

The intra-iteration flow dependence or input dependence indicates the data just written or read will be used in the same iteration. Both of dependences are represented by edges in IR. Such edges of IR are realized by physical channels from mPE to cPE or between cPE in LEAP. The data are transmitted directly from producers to consumers. To support the data delivery from one producer to multiple consumers, LEAP provides COPY instruction in cPE, which copy data from one input channel to two output channels. Multiple cPEs with COPY instructions form a broadcast tree.

The inter-iteration flow dependence indicates the data produced in the current iteration will be used in several iterations later. As shown an example program in figure 5, there is a flow dependence between A[i] and A[i-1] with distance of 1. The value produced by A[i] can be consumed in next iteration by A[i-1], but the first element of A[i-1] has to be fetched from memory. In cPE, we implement Select instruction $\langle C = select(A, B, n)\rangle$, the values of C are copied from A in the first n elements, then from B. The flow dependence in the example program can be represented by the instruction of $\langle S = select(T, R, 1)\rangle$ in cPE.

```
for (i=1，N，++)
{
   A[i]=A[i-1]+1;
}
```

```
for (i=1，N，++){
   <T=Load(A[i-1])>
   <S=select (T,R,1)>
   <R=S+1>
   <A[i]=Store(R)>
}
```

Fig. 4. A sample program with inter-iteration flow dependence and codes in psudo-assmbly lanuage

The first data element in the execution of Select instruction is copied from T, loaded from the memory address of A[i-1], then the following data are copied from R, which are results of last iteration in the instruction $\langle R = S + 1\rangle$. T, S and R are all temporary variables represented by edges of IR.

The inter-iteration input dependence indicates that data referenced in current iteration are reused in next several iterations. As shown in the bottom of Figure 5-a, there are 3 load operations from array A in the example program. The input dependence can be exploited to reduce the reference number from 3 to 1 per iteration. Such optimizations, called register promotion or scalar replacement, have been extensively explored in compiler technology. The goal of register promotion is to identify repeated references to the same memory location, and to replace the redundant memory accesses with registers. Classical register promotion algorithms use registers to rotate the temporary values [7]. [5] proposed an improved algorithm with valid flags to cope with dynamic circumstance. Both of their works have to evaluate the register pressure due to the limited number of registers, and insert a segment of program to rotate register values in the original programs.

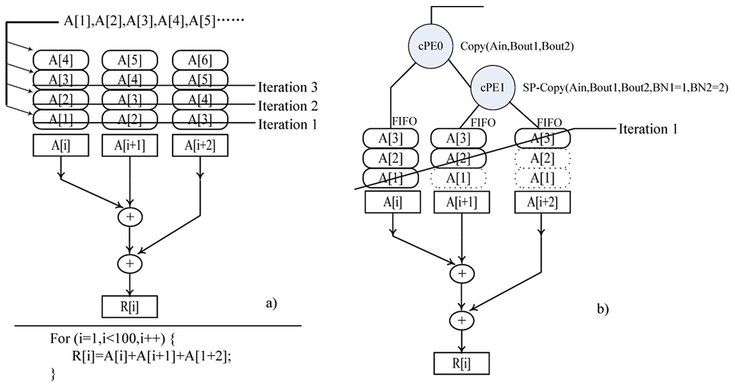


Fig. 5. a)An example program with inter-iteration input data dependence and the sequence of data reuse b)Partial copy instructions support data reuse

In LEAP, we provide a partial broadcast function in cPE to load dependent data into input FIFOs of multiple cPE in the sequence specified by data dependence distance. The push and pop operations of FIFOs naturally take the role of rotating register values without extra explicit register moving instructions.

A partial copy instruction $\langle SP - Copy(Ain, Bout1, Bout2, BN1, BN2)\rangle$ supports the special broadcast function. The function of SP-Copy is copy data elements from input channel of Ain to output channels of Bout1 and Bout2 simultaneously, but for the channel of Bout1, the elements of first BN1 are discarded, then the succeeding elements are stored in Bout1 channel; for channel of Bout2, the elements of first BN2 are discarded, then the succeeding elements are stored in Bout2 channel. When both of BN1 and BN2 are equal to 0, the SP-Copy is the same as normal Copy instructions.

As an example, Figure 5-b shows how SP-Copy instruction supports data reuse with inter-iteration input dependence. According to the program at the bottom of Figure 5-a, elliptical blocks at the top part of Figure 5-a shows the sequence in which the elements of A are used by A[i], A[i+1] and A[i+2] respectively. The data elements of A stream along the diagonal arrows to A[i], A[i+1] and A[i+2] in sequence of A[1], A[2], A[3], etc. We notice that the first element, A[1] is only used by A[i], the second element, A[2] is only used by A[i] and A[i+1], from the third element, succeeding data are reused by all of A[i], A[i+1] and A[i+2]. The horizontal lines across the elliptical blocks indicate the data elements used in the first loop iteration, the second iteration and the third respectively.

As shown in Figure 5-b, two cPE configured with special copy instructions can arrange input data elements in the shape specified by Figure 5-a. Receiving data elements of array A from Load unit of mPE, the first cPE, cPE0 moves the data from input channel to both of outputs with normal copy instruction. The left output represents A[i] and the right output flows into cPE1. cPE1, configured with SP-Copy instruction, copies data element from input to two output channels, discarding the first element in left channel Bout1 and the first 2 elements in right channel Bout2. Bout1 of cPE1 represents A[i+1] and Bout2 does A[i+2]. The blocks with dash lines indicate that the elements are discarded and are not in FIFOs. The slope line across the blocks in the Figure 5-b indicates the data elements residing at the top of FIFOs will be used in the first iteration. Over the succeeding iterations, the data elements in the FIFOs of cPEs are streamed out in sequence without register rotating.

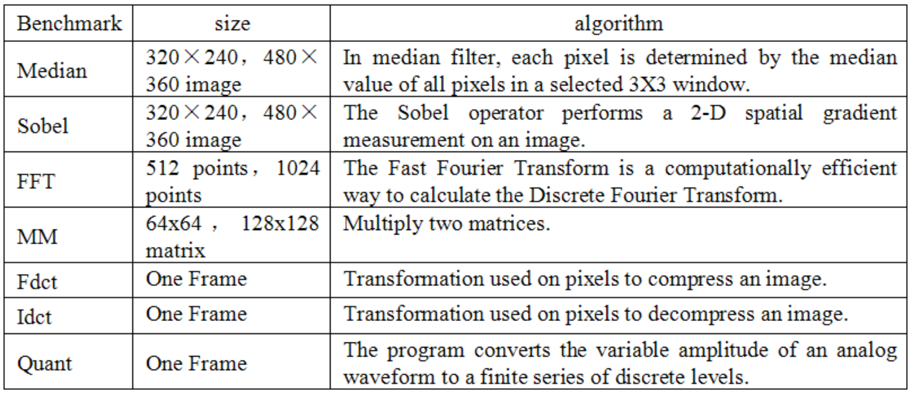
Under the assumptions of perfect dependence information by compiler, in LEAP we implement a patial copy instruction SP-Copy to arrange the initial shape of data elements. With the supports of FIFOs in cPE, the succeeding elements are naturally rotated without explicit register moving instructions.

4 Experimental Results

We have implemented all modules of the LEAP architecture in Verilog language. The module of cPE performs the function of normal 16bits ALU. Each module of mPE connects with a 2048 byte memory module as one bank of internal data memory. The module of static router connects mPE and cPE to form 8 by 8 torus network. The interface module as a bridge connects external SDRAM of 32MB with LEAP array. Before benchmark program execution, the configuration file and input data have been loaded into the external SDRAM. In addition to the simulation in the environment of ModelSim, we implemented 7 by 7 and 3 by 10 configuration in a FPGA development board, respectively. The board includes one FPGA device of Stratix EP1S80F1508C6 from Altera, 32 MB SDRAM with 64bits port width and a PCI controller connecting with a host computer. The synthesis tool adopts Quartus II 3.0. Table 1 lists the benchmark programs. The second column shows the size of input data. Programs of FDCT, IDCT and Quant are the most frequently executed kernels from MPEG2 program. The size of one frame is about 6M bytes, including 15840 macro blocks. All benchmarks are compiled and mapped to LEAP array in handcraft.

To provide the reader with some context, we compared the performance of memory optimization with a simulator of superscalar processor, called SimpleScalar. The simulator performs 4 issues, out of order superscalar RISC processing, with configuration of 16KB first level data cache, 256KB second level data cache, 4 integer units, 4 floating point units, and 2 ports to external memory.

Table 1. Benchmarks



Benchmark	size	algorithm
Median	320×240， 480×360 image	In median filter, each pixel is determined by the median value of all pixels in a selected 3X3 window.
Sobel	320×240， 480×360 image	The Sobel operator performs a 2-D spatial gradient measurement on an image.
FFT	512 points， 1024 points	The Fast Fourier Transform is a computationally efficient way to calculate the Discrete Fourier Transform.
MM	64x64 ， 128x128 matrix	Multiply two matrices.
Fdct	One Frame	Transformation used on pixels to compress an image.
Idct	One Frame	Transformation used on pixels to decompress an image.
Quant	One Frame	The program converts the variable amplitude of an analog waveform to a finite series of discrete levels.

We now analyse our experimental results. In the 6^{th} column of Table 2, #cPE represent the number of cPEs involved in computation. Since the loop body size in each kernel is different. The number of cPE is varied when mapped to LEAP array. In our experiment, the maximum number of cPE is 30, and the min number is 10. The number of cPE determines the maximum computing throughput of loop pipelines, indicated by #OP.

The rate issuing loop iteration is another factor that takes effects on #OP. With our speculative scheduling mechanism, each test achieves the issue rate of one iteration per cycle. As shown in the 6^{th} column, the number of #OP equals to the number of cPE. #OP indicates the maximum number of computing

Table 2. Experimental results

Benchmark	Data Size	LEAP						SS		
		AP (%)	#L	#T	#cPE /#OP	RTime (K)	#MAN (K)	#IPC	RTime (M)	#MAN (M)
Median	320x240	42.30	54	12.7	30/30	220.0	598	1.88	41.86	38.64
	480x360	41.20		12.4		478.8	1200	1.88	94.67	87.32
Sobel	320x240	41.47	54	6.6	16/16	217.0	598	1.85	9.94	7.55
	480x360	40.62		6.5		474.2	1200	1.84	22.52	17.05
FFT	512	40.26	34	4.0	10/10	6.7	25.6	1.36	3.69	0.21
	1024	43.49		4.4		12.8	51.2	1.38	7.35	1.90
MM	64x64	41.62	24	6.7	16/16	79.1	279	1.88	4.77	3.51
	128x128	82.60		13.2		580.3	4456	1.50	42.82	27.66
Fdct	one frame	39.17	24	7.1	18/18	2839.0	27394	1.68	2369.38	1292.50
Idct		39.17		7.1	18/18	2839.0	27394	1.56	2615.09	1289.46
Quant		32.58	64	5.5	17/17	2131.6	2395	1.05	373.40	157.34

operations issued from loop pipelines under the assumptions that all input data reside in internal memory.

But we notice that several factors prevent loop pipelines from achieving the goal of maximum computing throughput. The first factor is the start-up overhead of loop pipelines, indicating the latency from the time of first iteration trigged to the time of the first result reaching mPE. In LEAP, the start-up latency is determined by the stages in the path of ALU pipelines in cPE, internal FIFO in cPE and buffers in static routers. In the tests, the start-up latency, varying from 24 to 64, is relatively very short comparing with whole running time in the order of kilo cycles.

The most critical factor is the time for data movement between external memory and internal memory. #T represents the average throughput taking the overhead of memory accesses into account. #AP (active percentage) is another metrics indicating the busy percentage of cPE. Although there are overlaps between cPE computing and mPE memory accessing, the overhead of memory accesses decreases the actual throughput of loop pipelines. In our experiments, the Matrix Multiplication in the size of 128×128 achieves the maximum #AP of 82.6%. Because Matrix Multiplication is a computation intensive algorithm, loading data in $O(n^2)$ and computing in $O(n^3)$. Other loops are algorithms in $O(n)$. Thus, the minimal #AP is only 32.58% and the average #AP is 44.04%. We notice that memory optimizations become increasingly important to the efficiency of loop pipelining

The comparison between in the 8^{th} column and 11^{th} column, indicates the effects of memory optimization. Our scheme of memory optimization reach the goal of reduction in the number of memory accesses. Two columns of #MAN represents the number of memory accesses issued by mPE in LEAP and number of load/store instructions issued in SimpleScalar simulator, respectively. Since SimpleScalar simulator did not exploit the approach of register promotion and only took part of intra-iteration dependence into account. It produces much more memory accesses than our optimized approach. Comparing with SimpleScalar simulator, LEAP achieves about 72.8 times of reduction in memory accesses in the kernel median filter(480×360), and the average reduction reach 35.3 times. The reduction in memory accesses is the one of reasons why the gap of running time between SimpleScalar and LEAP, reaches one to two orders of magnitude in cycles.

5 Related Work

Several research projects applied pipelining techniques to reconfigurable architectures in order to obtain high performance. With respect to loops, software pipelining is a well-known technique to improve throughput. Due to the need of a statically-defined scheduler, one of the approaches uses Raus Iterative Modulo Scheduling (IMS) algorithm[2] to pipeline innermost loops in the Garp[3] architecture. Software pipelining suffers from the accurate resource usage modeling, some of which are hard to determine, such as unpredictable accessing latency due to memory contention.

More related to our work is dynamic loop pipelining [4], called self loop pipelining (SLP), a technique to map loop kernels in order that loop pipelining is dynamically achieved. Our work shares the same goal with the technique of SLP. The first difference resides in that we proposed a speculative execution mechanism to improve the throughput of loop pipelining, which can deal with not only intra-iteration dependence but also loop-carried dependence. The second is our work supports BREAK statement in normal FOR loops with the help of Loop Stop FSM in mPE. With BREAK statement, our scheme can apply loop pipelining to other types of loops, WHILE loops, tightly nested loops or loosely nested loops.

The work, Spatial Computing, in [5] presented a dataflow internal representation, which form a virtual instruction set architecture for data driven execution, and a series of compiler optimizations. By adopting their internal representation, our work focuses on the architectural implementation techniques that support mapping the internal representation to a reconfigurable array in the mode of Self-Loop-Pipelining. The unique implementation techniques, comparing with their work, include speculative execution mechanism of loop pipelining, partial broadcast instruction SP-COPY for register promotion and SELECT instruction for resolving inter-iteration flow dependence.

6 Conclusion

This paper introduces the implementation techniques in LEAP, a coarse-grained data-driven reconfigurable architecture, which supports loop self-pipelining.

Based on the extension of dataflow IR, we propose the mechanism of speculative execution for dynamically scheduling loop iterations without tightly synchronization between token generator and collector. The experimental results show our throughput of loop pipelines reaches one iteration per cycle with the condition of data residing in internal memory. Also, we explore the approaches of eliminating redundant memory accesses. We realize two special instructions, SELECT and SP-COPY, to exploit the inter-iteration data dependences. The experimental results strongly prove the efficiency of memory optimizations. The maximum decrease in the number of memory accesses reaches 72.8 times.

Acknowledgement

This work is sponsored by the National Science Foundation of China under the grant NO. 60633050.

References

1. B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. in Design, Automation and Test in Europe Conference and Exhibition (DATE'03), Munich, Germany, pages 10296-10301, March 3-7, 2003.
2. B. Rau. Iterative Module Scheduling: An Algorithm for Software Pipelining Loops. In Proceedings of the ACM 27th Annual International Symposium on Microarchitecture (MICRO-27), ACM Press, New York, pages 63-74, 1994.
3. T. Callahan and J. Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES00), San Jose, CA, USA, ACM Press, pages 57-64, November 17-19, 2000.
4. Joao M. P. Cardoso. Dynamic Loop Pipelining in Data-Driven Architectures. In: Proc. of the 2nd Conference on Computing Frontiers(CF'05), 2005. 106-115
5. M. Budiu. Spatial Computation. Ph.D. Thesis, CMU CS Technical Report CMU-CS-03-217, December 2003.
6. J. Xu, G. Wu, Y. Dou and Y. Dong. Designing a Coarse-Grained Reconfigurable Architecture Using Loop Self-Pipelining. ACSAC 2006, LNCS 4186, pp. 567-573, 2006.
7. S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. Software Practice and Experience, 24(1), January 1994.

Hardware/Software Codesign for Embedded Implementation of Neural Networks

Cesar Torres-Huitzil[1], Bernard Girau[2], and Adrien Gauffriau[2]

[1] Computer Science Department, INAOE
Apdo. Postal 51 and 216, Puebla, Mexico
ctorres@inaoep.mx
[2] CORTEX team, LORIA-INRIA Lorraine
Campus Scientifique, B. P. 239, Vandoeuvre-les-Nancy Cedex, France
Bernard.Girau@loria.fr, Adrien.Gauffriau@loria.fr

Abstract. The performance of configurable digital circuits such as Field Programmable Gate Arrays (FPGA) increases at a very fast rate. Their fine-grain parallelism shows great similarities with connectionist models. This is the motivation for numerous works of neural network implementations on FPGAs, targeting applications such as autonomous robotics, ambulatory medical systems, etc. Nevertheless, such implementations are performed with an ASPC (Application-Specific Programmable Circuits) approach that requires a strong hardware expertise. In this paper a high-level design framework for FPGA-based implementations of neural networks from high level specifications is presented but the final goal of the project is a hardware/software codesign environment for embedded implementations of most classical neural topologies. Such a framework aims at providing the connectionist community with efficient automatic FPGA implementations of their models without any advanced knowledge of hardware. A current developed software platform, NNetWARE-Builder, handles multilayer feedforward and graphically-designed neural networks and automatically compiles them onto FPGA devices with third party synthesis tools. The internal representation of a neural model is bound to commonly used hardware computing units in a library to create the hardware model. Experimental results are presented to evaluate design and implementation tradeoffs.

1 Introduction

To date the majority of neural network implementations, considered as computation models of the human brain [1], have been implemented in general purpose processors and software techniques. Despite their generally recognized performances, the high cost of developing ASICs (Application Specific Integrated Circuits) has meant that only a small number of hardware neural-computing devices have gone beyond the research-prototype stage in the past. With the appearance of large, dense, highly parallel FPGA circuits, it has now become possible to implement large-scale neural networks in hardware, with the flexibility and low cost of software implementations [2]. FPGA circuits are now considered as appealing devices for system implementations for several reasons. Their performance and complexity increase at a very fast rate, sometimes even faster than microprocessors [3]. State-of-the-art FPGA devices now include arithmetic units, large memories, or even DSPs (Digital Signal Processors). They may lead to systems where hardware updates would be as easy as software ones, and they offer hardware efficiency at a reduced cost when the cost of ASIC design ever increases with new technologies. Nevertheless, most implementations on FPGAs are performed with an ASPC approach, where the FPGA solution replaces the expensive design of an ASIC. ASPC requires a strong hardware expertise, even if it gets rid of some specific problems of ASIC design. This is why a new trend has appeared in FPGA-based system design, trying to avoid the time-consuming implementation work and its partial lack of flexibility (some design need to be deeply updated when changes are made in the application). These works aim at creating higher-level environments to handle FPGAs, so that these devices are more likely to be used as easily as microprocessors. They may be divided according to three main principles:

– *Hardware compilers based on standard software languages such as C/C++* [4]: they generate FPGA configurations from simple programs, so that FPGAs become target devices for standard programs. These solutions are not able to exploit neither the high degree of parallelism nor the area improvements of up-to-date FPGAs. More recent works try to adapt principles of automatic parallelization [5], but they are still unable to optimize implementations at a very fine grain level.
– *FPGA-embedded multiprocessor systems (based on in-house processor cores or on configurable processors developed by FPGA vendors)* [6]: such systems mix the parallelism of several optimized processors on a single FPGA, and the software programmability of each processor. Nevertheless, their use requires parallelization expertise, and they mostly handle applications whose parallelism lies in huge nested loops.
– *Specialized design environments* [7]: such tools may handle a limited number of computation models (such as neural networks for example), and they generate FPGA configurations from the description of each specific parameterized model. This approach appears as easier for the end-user, it is also able to take advantage of the parallelism of the implemented models, but its generality is strongly reduced to the models specified in the environment.

Our work follows the third approach, in the specific field of neural computing. Nevertheless, unlike in [8], we consider that our environment should not be some kind of additional high-level layer that generates neural networks descriptions using some standard software language: our claim is that efficient implementations have to take advantage of the so-called neural parallelism, so that they require the generation of an optimized hardware implementation that takes numerous constraints into account. On the other hand, our work also meets needs that emerge in the connectionist community. In the field of neural processing, several applications mix real-time or low-power constraints with a need for flexibility [9], so that FPGAs appear as a well-fitted implementation solution [10].

Their fine-grain parallelism shows great similarities with connectionist models composed of massively distributed elementary units interacting through dense interconnection schemes. These are the motivations for numerous works of neural network implementations on FPGAs. In spite of the interest in hardware implementations, a lack of an integrated design environment to handle neural networks from high level specifications and to cope with implementations constraints is a current problem to face [11]. Though easier than ASIC development, implementations on FPGAs still require a significant amount of work from the conception to the implementation of a neural model, especially for scientists who are not very familiar with tools such as Hardware Description Languages (HDLs), synthesis tools, etc. Therefore, we intend to develop a generic hardware/software methodology to fully and automatically specify, parameterize and implement neural networks according to various application and technological constraints, e.g. area consumption, required precision, etc. This project is our first attempt to simultaneously and automatically handle different aspects of neural implementations: technological choices to fit implementation constraints, generality and modularity of solutions, precise analysis of the relations between application data, device specifications, and performances.

This paper describes the software platform that has been developed to put our approach into a concrete form. Preliminary results on the implementation of feedforward neural networks are presented and discussed. The rest of the paper is organized as follows. Section 2 provides a general overview of our NNetWARE-Builder framework for embedded implementations of neural networks (http://nnetware.gforge.inria.fr). Section 3 describes the design flow for neural networks for the current implementation of the software tool. Section 4 presents results and a discussion on the FPGA implementation of feedforward neural networks for two different case studies.

2 NNetWARE-Builder Framework

Developing a tool for automatic implementation of neural networks onto FPGAs faces several conceptual issues, which are summarized below before describing the different parts of the software.

– *Models and genericity*: the variety of neural models is such that it is not possible to include them in the software in an exhaustive way. Very specific models are not in the scope of our platform. We mainly intend to address 4 wide families of neural models. The current version handles feedforward neural networks (as considered in [12]), that should be soon extended to recurrent networks. Advanced works have already been performed towards the inclusion of spiking models [13][14], and of bio-inspired models [15] (in the sense of cortical multimaps as defined in [16]).
– *Technological choices*: optimized implementations of neural networks often require the use of well-chosen arithmetic operators and data communication schemes [15]. Though the current version still relies on the user to specify such choices, the platform project aims at automating this process.

- *Parallelization methods*: the current version is fully based on the principle of connectionist parallelism (the neural architecture defines the hardware architecture), but more advanced techniques will be included (FPNA [17], pipelined PEs [15], etc.) so as to handle larger neural networks which direct parallel mapping does not fulfill hardware constraints. An automatic optimization of the choice of the different types of parallel levels and techniques is the final aim of the project.
- *Formalism for neural parallelism*: the automation mentioned above requires defining a formal description of neural computations that exhibits different levels of parallelism. This aspect may be seen as a neural adaptation of automatic parallelization techniques. The current version corresponds to a fully unfolded (parallelized) implementation.
- *I/O handling*: implementing neural networks is not useful if interactions with the environment are not considered. We had to define a set of general I/O handling policies that might cover most standard application needs combined with possible arithmetic choices. The current version may handle data as direct I/Os of the model, or as buffered I/Os, and patterns may be defined as rough data or possibly overlapping windowed partitions of input images.
- *Board dependency*: any such platform has to face technological issues when different boards are considered. As it will be described later, we have chosen to encapsulate neural processing and to use board-dependent wrapping libraries to minimize the effort required when targeting to a new board.
- *Coherence of the whole process*: even if each step and aspect of the implementation process raises specific problems, the main issue is to ensure a common and coherent development of the whole platform, so as to result in a functional chain from the creation of a network to the visualisation of results produced in the board.

2.1 General Architecture

The proposed high level synthesis framework, NNetWARE-Builder, for automatic generation of HDL models of neural networks from high level descriptions is shown in figure 1. The block diagram shows the interaction between the high level synthesis, the simulation tools and the hardware prototyping platform in an integrated framework. The synthesis framework also takes additional information as input, such as a hardware resource library and user directives to control the logic synthesis flow of specific CAD tools. The general framework includes capabilities for validating and testing, as shown in figure 1.

Briefly, the transformation of a high level specification into an HDL model is as follows. The synthesis framework accepts a block diagram description of a neural network, parses the description and creates an intermediate representation, then it runs an analysis pass to bind modules of the database, and finally generates an output in register-transfer level VHDL with a set of user constraints. The main component of the synthesis framework consists of a RTL VHDL Structural Model Generator, where a subset of VHDL components, synthesizable by commercial logic synthesis tools, is used from a library. The RTL VHDL model is mixed with

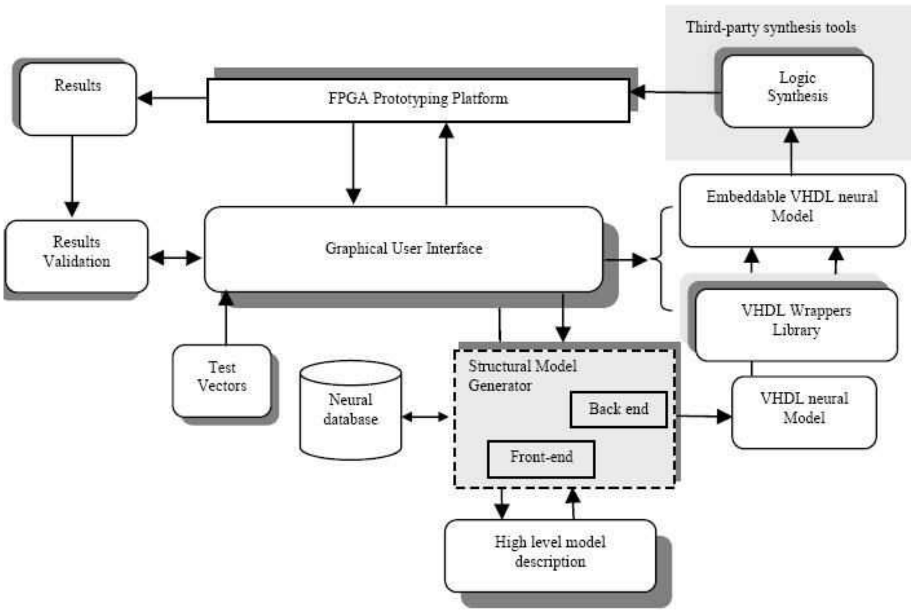


Fig. 1. Block diagram of the integrated framework architecture, NNetWARE-Builder, for digital implementations of neural models from high level specifications

a wrapper to make the model embeddable into a specific prototyping platform and then exported to FPGA CAD tools to finally generate the programming file.

2.2 High-Level Model Description and Parsing

The neural model is captured through a Graphical User Interface (GUI) that allows the user to define models in a block-diagram style. A wizard mode is also available. It is especially useful for multilayer feedforward networks definition. The front-end of the Structural Model Generator parses the high level description, analyzes its correctness, and extracts its general parameters, such as the network topology and number and type of neurons [18]. An internal structure representation for the model is generated for further utilization.

2.3 Neural Database

The neural database contains all the information about the optimized hardware building blocks which are designed to be reusable and parameterizable through both a precision-oriented and a feature-oriented hierarchical approach. Basic operators related to neural computations, such as adders, multipliers, comparators, and activation functions, are specified in the library [19]. Parameters of special interest for building blocks are its interface and arithmetic.

2.4 Structural Model Generator and Binding

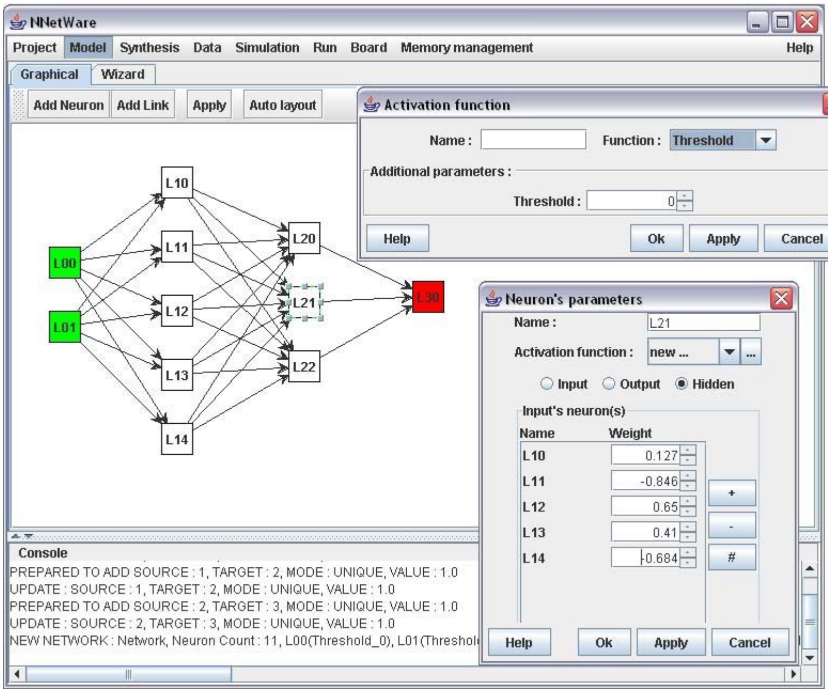
The back-end of the Structural Model Generator generates the HDL structural model, fully synthesizable for most CAD tools. Neural operations are bound to functional hardware units and all the interconnections among modules are performed taking into account the implementation constraints such as arithmetic and precision. The Structural Model Generator uses the information in the VHDL wrapper library to instantiate the appropriated memory controller, for instance, to read/write and to generate the final VHDL model that can be embedded into a specific prototyping platform. Control circuits are added to the model to implement the required control logic and access mechanisms to hardware resources for the prototyping platform. Platform information is provided in a VHDL wrapper library which is a collection of VHDL components to handle low level communication protocols directly in hardware. New wrapper components can be integrated for other prototyping platforms. The VHDL output files are integrated into the standard FPGA synthesis flow with additional constraints and optimizations to complete the path from architectural design and high level specification to a fully functional hardware implementation.

2.5 Simulation, Validation and Debugging

In order to verify the correctness of the VHDL model, NNetWARE-Builder provides mechanisms to generate a set of test vectors, and test benches automatically trough its GUI. Currently, for experimental evaluations, the embeddable VHDL neural module is generated considering that a single FPGA device is associated to external memories where the test patterns are stored. This approach was selected due to two main reasons. First, most FPGA platforms are built under this model and second because a large amount of data is usually used to test neural networks and thus data must be mapped to external memories. However, specific data accessing mechanisms are explored for more efficient implementations such as the use of FPGA internal registers and memory blocks for temporal storage and parallel access of data for visual perception oriented applications. In the following section, the design and implementation flow of neural network models in NNetWARE-Builder is shown through the presentation of the main capabilities of the associated GUI. The NNetWARE-Builder GUI was developed in Java using the Eclipse IDE and it was built both for Windows and Linux operating systems. Currently, only one FPGA prototyping platform is fully supported in the framework. More details on this aspect will be given in the next section. As the proposed tool is currently under development, the design flow will be described for feedforward multilayer neural networks.

3 Design Flow

The design flow involves four main steps in NNetWARE-Builder: a high level description of the neural network, the definition of implementation constraints,



Activation function

Name :

Function : Threshold

Additional parameters :

Threshold : 0

Neuron's parameters

Name : L21

Activation function : new ...

Input Output [✓ Hidden]

Input's neuron(s)

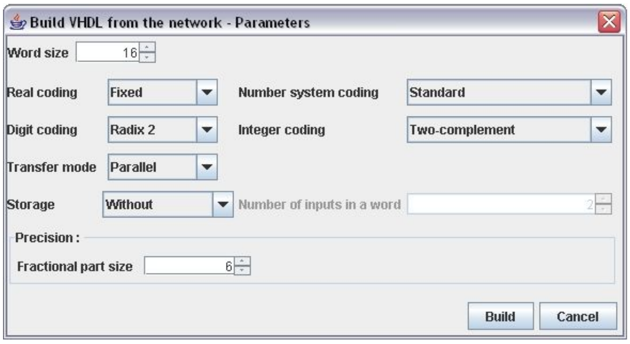
Name	Weight
L10	0.127
L11	-0.846
L12	0.65
L13	0.41
L14	0.684

```
PREPARED TO ADD SOURCE : 1, TARGET : 2, MODE : UNIQUE, VALUE : 1.0
UPDATE : SOURCE : 1, TARGET : 2, MODE : UNIQUE, VALUE : 1.0
PREPARED TO ADD SOURCE : 2, TARGET : 3, MODE : UNIQUE, VALUE : 1.0
UPDATE : SOURCE : 2, TARGET : 3, MODE : UNIQUE, VALUE : 1.0
NEW NETWORK : Network, Neuron Count : 11, L00(Threshold_0), L01(Threshol
```

Fig. 2. A screenshot of the GUI main window of NNetWARE-Builder with a two hidden-layer neural network. The dialog boxes for specifying neuron parameters, L21, and the activation function are shown on the right.

the generation of the VHDL model targeted to a specific platform, and the generation of testbenches for on-chip validation of the neural model.

Figure 2 shows a screenshot of the GUI main window of NNetWARE-Builder for a block-based graphical definition of a feedforward multilayer network. Neuron parameters in the network, such as the type of neuron, the activation function and the associated weights, can be customized through dialog boxes when a given neuron is selected. Implementations constraints can be established for the synthesis process through a dialog box as shown in figure 3. This dialog box invites the user to adjust arithmetic aspects of the model: arithmetic type, precision and representations. Several arithmetical choices are considered, parallel, serial, on-line and stochastic pulse-based. Currently a full validation for parallel, 2-complement and fixed point representation has been done. Implementation constraints also include the possible on-chip buffering of I/Os. The model internal representation is transformed into a structural descrip tion and the VHDL wrapper is generated as a top level unit to make the model embeddable in the specific platform. The wrapper includes the control logic to distribute to and to collect data from the neural model. For the current implementation, the VHDL wrapper library includes the control and communication protocols for the RC1000-PP FPGA


Build VHDL from the network - Parameters

Word size: 16

Real coding: Fixed

Number system coding: Standard

Digit coding: Radix 2

Integer coding: Two-complement

Transfer mode: Parallel

Storage: Without

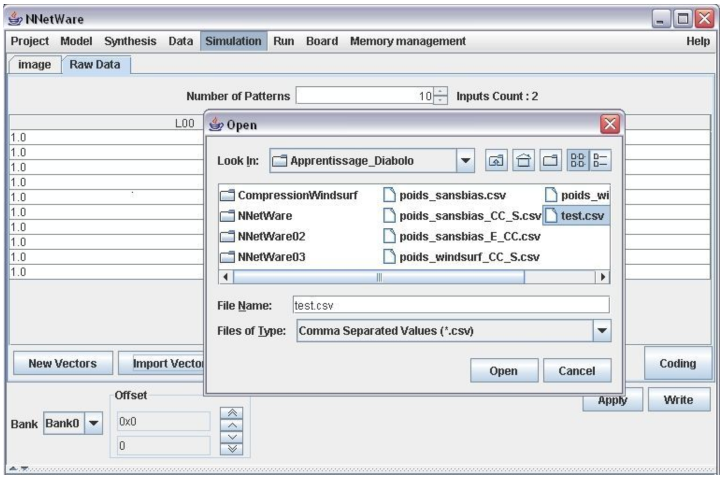
Number of inputs in a word: 2

Precision:

Fractional part size: 6

Fig. 3. Dialog box to choose the arithmetic for the hardware implementation of a neural network model and to generate the associated VHDL wrapper

prototyping platform from Celoxica. The FPGA programming file is generated through the invocation of the third party tools. For the current implementation, Xilinx Foundation ISE is called to generate the bit-stream configuration file. Once the FPGA is configured, the neural model may be tested (for implementation validation or model prototyping). The current version includes two main modes: computations performed on rough data vectors, as shown in figure 4, or on data extracted from images through configurable partitioning (the rebuilding of the output images are independently configurable), see figure 5.



Number of Patterns: 10 Inputs Count : 2

L00
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0
1.0

Open

Look In: Apprentissage_Diabolo

CompressionWindsurf
NNetWare
NNetWare02
NNetWare03
poids_sansbias.csv
poids_sansbias_CC_S.csv
poids_sansbias_E_CC.csv
poids_windsurf_CC_S.csv
poids_wi
test.csv

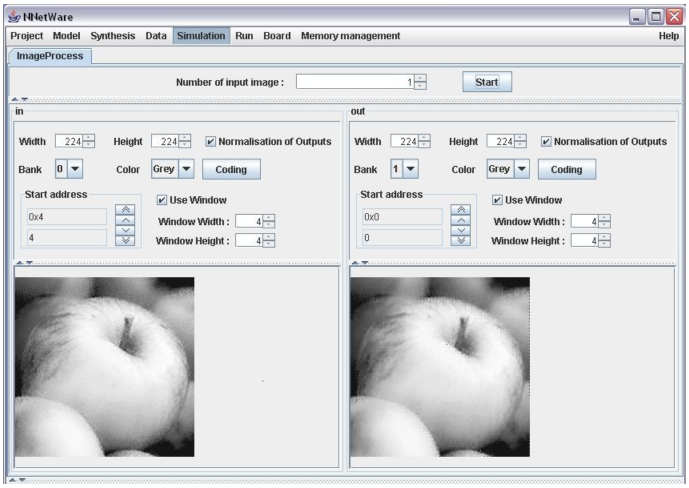
File Name: test.csv

Files of Type: Comma Separated Values (*.csv)

Bank: Bank0

Offset: 0x0, 0

Fig. 4. Screenshot showing the current capabilities to support simulation and validation of a hardware model. Test vector can be generated or imported from files.


NNetWare
Project Model Synthesis Data Simulation Run Board Memory management Help
ImageProcess
Start
in
out
Coding
Coding

Number of input image : 1

in: Width 224 Height 224 ✓ Normalisation of Outputs
Bank 0 Color Grey
Start address 0x4 4
✓ Use Window
Window Width : 4
Window Height : 4

out: Width 224 Height 224 ✓ Normalisation of Outputs
Bank 1 Color Grey
Start address 0x0 0
✓ Use Window
Window Width : 4
Window Height : 4

Fig. 5. Screenshot of the dialog boxes for image data transferring to and from the FPGA board for visual perception applications

4 Results

NNetWARE-Builder is a useful tool for connectionist researchers and engineers to evaluate their models on fully functional hardware systems and to analyze different implementations tradeoffs. In this section the results of the FPGA implementation for two case studies are presented and discussed.

4.1 An Example of 2D Vector Discrimination

A discrimination problem of three classes of points in the 2D plane is presented in this section. The patterns considered to train the network and some patterns

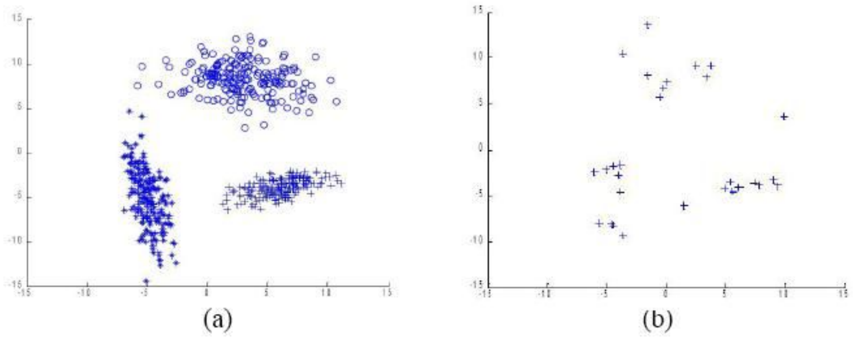


Fig. 6. (a) Training patterns for a 2D vector discrimination example and (b) the test pattern for network evaluation

Table 1. Hardware resource utilization for a neural network solving the 2D classification problem with two implementation options targeted to a Virtex XCV2000E-BG560-6 device

Resources	10-bit fixed point precision without saturated arithmetic	10-bit fixed point precision with saturated arithmetic
4-input LUTs	1234	1374
Flip-flops	30	30
Slices	755	838
Maximum clock freuqency	123 MHz	130 MHz

to test the network are shown in figure 6. The test patterns were correctly classified with the targeted network mapped onto the FPGA device. The network uses 2 neurons in the input layer, 15 sigmoid-like neurons in the hidden layer, and 3 neurons in the output layer. The neural network was previously trained in MATLAB. The precision study performed for this application showed that at least 12-bit values had to be chosen. Nevertheless, in order to test the possibility of unbuffered handling of I/Os, 10-bit words had to be chosen. The use of NNetWARE helped showing that a 10-bit coding was sufficient, at the cost of overflow detection in basic operators, resulting in a slightly increased resource utilization as shown in table 1.

4.2 An Example of Image Processing: A Diabolo Network for Image Compression/Decompression

A diabolo network [20] was implemented for lossy image compression. Such a network is trained to learn the identity function for all patterns that are extracted as non-overlapping sub-windows from a given image. After training, the compressed image lies in the outputs of the hidden layer for all given sub-windows. A network topology of 16x5x16 with a linear activation functions in the hidden

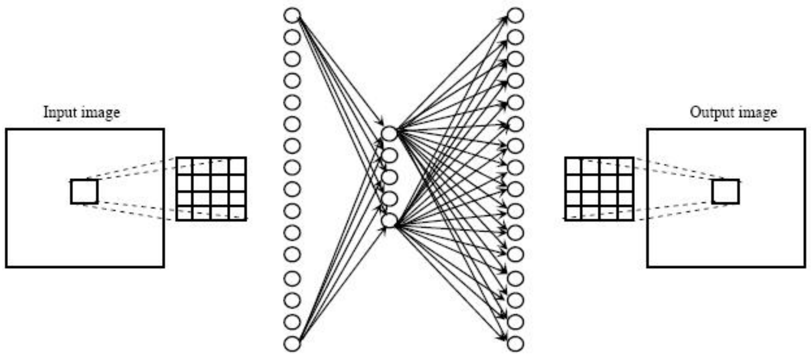


Fig. 7. General diabolo network architecture for image compression/decompression

Table 2. Hardware resource utilization for a diabolo neural network for image compression/decompression targeted to a Virtex XCV2000E-BG560-6 device

Resources	12-bit fixed point precision	11-bit fixed point precision	9-bit fixed point precision
4-input LUTs	14834	12701	7615
Flip-flops	286	269	233
Slices	9432	7921	5164
Maximum clock freuqency	90 MHz	95 MHz	95MHz

layer was used to reduce a 4x4 window of the input image, figure 7. An input image and the image decompressed by the network mapped onto the FPGA have already been shown in figure 5. A 12-bit fixed point precision for the neural network synthesis was used with 9 bits for the fractional part.

Additionally, the FPGA implementations of the network with other arithmetic precisions, 11-bit with 8-bit for the fractional part and 9-bit with 6-bit for the fractional part, and the corresponding obtained synthesis results are shown in table 2. The user takes advantage of NNetWARE to test the ratio between its precision choices and the rate of information loss during compression.

5 Conclusions

This work has presented a synthesis framework and a software platform that handle feedforward neural implementations on reconfigurable logic from high level specifications. Multilayer and graphically-designed neural networks are automatically "compiled" onto FPGA devices by this tool. NNetWARE-Builder is a high level tool for hardware-targeted neural models especially useful for users who are not familiar with the specific field of neural network implementations. The NNetWARE-Builder software is already functional and has been tested for multilayer perceptron-like networks for full parallel implementation options. Nevertheless, numerous technological aspects still have to be included, as well as advanced neural mapping methods. The final goal is to develop a hardware/software codesign environment for embedded neural network implementations onto reconfigurable devices. Future works will focus on automatic optimized parallelization for larger neural networks, through the partial or full exploitation of different levels of parallelism, and through the adaptation of neural-specific advanced parallelization techniques to our formalism.

References

1. B. Colwell, "Machine Intelligence Meets Neuroscience," IEEE Computer, Vol. 38, No. 1, pp. 12-15, January 2005.
2. Hammerstron Dan, "Digital VLSI Neural Networks," The Handbook of Brain Theory and Neural Networks, Michel A. Arbib Editor, Second Edition, pp. 349-352.

3. A. DeHon, "The Density Advantage of Configurable Computing," Computer, Vol. 33, No. 4, pp. 41-49, April 2000.
4. G. Lee, and G. Milne, "Programming Paradigms for Reconfigurable Computing," Microprocessors and Microsystems, Vol. 29, pp. 435-450, 2005.
5. L. Kaouane, M. Akil, Y. Sorel and T. Grandpierre, "From algorithm graph specification to automatic synthesis of FPGA circuit: a seamless flow of graph transformations," Proceedings of FPL'03, pp. 934-943, 2003.
6. Paulin P.G., Pilkington C., Langevin M., Bensoudane E., Lyonnard D., Benny O., Lavigueur B., Lo D., Beltrame G., Gagne V., Nicolescu G., "Parallel programming models for a multiprocessor SoC platform applied to networking and multimedia," IEEE Transactions on VLSI Systems, Vol.14, No.7, pp. 667- 680, July 2006
7. Banerjee P., Haldar M., Nayak, A., Kim V., Saxena V., Parkes S., Bagchi D., Pal, S., Tripathi N., Zaretsky D., Anderson R., Uribe J.R., "Overview of a compiler for synthesizing MATLAB programs onto FPGAs," IEEE Transactions on VLSI Systems, Vol.12, No.3, pp. 312- 324, March 2004.
8. E.M. Ortigosa, A. Caas, E. Ros, P.M. Ortigosa, S. Mota and J. Daz, "Hardware description of multi-layer perceptrons with different abstraction levels," Microprocessor and Microsystems, Vol. 30, Issue 7, pp. 435-44, November 2006.
9. B. Girau and K. Ben Khalifa, "FPGA-targeted neural architecture for embedded alertness detection," Proceedings of AIA, pp. 199-204, 2006.
10. B. Girau, "Neural networks on FPGAs: a survey," Proceedings of Neural Computation, 2000.
11. L. M. Reyneri, "Implementation Issues of Neuro-Fuzzy Hardware: Toward HW/SW Codesign," IEEE Transactions on Neural Networks, Vol. 14, No. 1, pp. 176-194, January 2003.
12. C. Gégout, B. Girau and F. Rossi, "A generalized feedforward neural network model," NeuroColt research report TR-95-041, 1995.
13. B. Girau and C. Torres-Huitzil, "FPGA implementation of an integrate-and-fire legion model for image segmentation," Proceedings of ESANN, pp. 173-178, 2006.
14. B. Schrauwen, and Jan M. Van Campenhout, "Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic," Proceedings of ESANN, pp. 623-628, 2006.
15. C. Torres-Huitzil and B. Girau, "FPGA implementation of an excitatory and inhibitory connectionist model for motion perception," Proceedings of FPT, pp. 259-266, 2005.
16. J.G. Taylor, "Neural bubble dynamics in two dimensions: foundations," Biological Cybernetics, 80:5167-5174, 1999.
17. B. Girau, "FPNA: Interaction between FPGA and neural computation," International Journal of Neural Systems, 10(3), pp. 243-259, June 2000.
18. S. Gupta, N. Savoui, N. Dutt, R. Gupta, and A. Nicolau, "Using Global Code Motions to Improve the Quality of Results for High-Level Synthesis," IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, 23 (2), pp. 302-312, February 2004.
19. S. Vassiliadis, M. Zhang, and J. Delgado-Frias, "Elementary Function Generators for Neural Network Emulators," IEEE Transactions on Neural Networks, Vol. 11, No. 6, pp. 1438-1449, November 2000.
20. H. Schwenk, "The Diabolo Classifier," Neural Computation, Vol. 10, No. 8, pp-2175-2200, November 1998.

Synthesis of Regular Expressions Targeting FPGAs: Current Status and Open Issues

João Bispo[1], Ioannis Sourdis[2], João M.P. Cardoso[1], and Stamatis Vassiliadis[2]

[1] IST/INESC-ID, Lisboa, Portugal
joaobispo@gmail.com,jmpc@acm.org
[2] Computer Engineering, TU Delft, The Netherlands
{sourdis,stamatis}@ce.et.tudelft.nl

Abstract. This paper presents an overview regarding the synthesis of regular expressions targeting FPGAs. It describes current solutions and a number of open issues. Implementation of regular expressions can be very challenging when performance is critical. Software implementations may not be able to satisfy performance requirements and thus dedicated hardware engines have to be used. In the later case, automatic synthesis tools are of paramount importance to achieve fast prototyping of regular expression engines. As a case study, experimental results are presented, for FPGA implementations of the regular expressions included in the rule-set of a Network Intrusion Detection System (NIDS), Bleeding Edge, obtained using a state-of-the-art synthesis approach.

1 Introduction

Regular expressions can be a heavy computational burden in some applications. For instance, the new generation of Network Intrusion Detection Systems (NIDS) relies heavily in regular expressions, a case where they represent a considerable amount of the total computing time [1]. The set of regular expressions used in those applications grows quickly. Table 1 shows the number of regular expressions in the available Snort [2] [3] and Bleeding Edge [4] rule-sets, all versions from October 2006, with exception of the November 2006 version of Bleeding Edge (last row). It is also shown the number of *constraint repetitions* (i.e., Exactly, AtLeast, and Between quantifiers) presented in the regular expressions of the rule-sets. As can be seen, the rule-sets include a large number of regular expressions and also many *constraint repetitions*. Those numbers are expected to grow since new rules are being continuously added. As an example, the number of regular expressions in the Snort 2.4 version has increased about 2.9× during 2006.

Pattern matching using regular expressions is distinct from static pattern matching, where the input string is matched against other literal strings. In regular expressions, meta-characters with special meaning are used, and a single regular expression can represent several strings. Regular expressions augment the challenges of static pattern matching (e.g., *overlapped matching*) with other ones, such as space explosion (regular expressions can represent very large strings in a

Table 1. Characteristics of Snort and Bleeding Edge rule-sets with respect to regular expressions

Rules	**Regular Expressions**			
	total	*Constraint Repetitions*		
		Exactly	**AtLeast**	**Between**
Snort 2.4 (Oct. 2006)	1,504	286	319	7
Snort 2.3 (Oct. 2006)	1,500	286	319	7
Snort 2.2 (Oct. 2006)	1,493	258	319	7
Snort 2.1 (Oct. 2006)	1,380	248	318	6
Bleeding Edge (Oct. 2006)	310	58	6	6
Bleeding Edge (Nov. 2006)	317	63	6	6

very compact form). Hardware solutions for regular expression pattern matching are already being used in order to achieve high performance demands. Since in most application domains using regular expressions (e.g., data mining, NIDS, email monitoring and inspection, etc.) periodical updates are required, FPGAs seem to be the preferable technology to maintain up-to-date and specialized hardware engines, able to achieve high-performance.

However, synthesis tools to generate the hardware engines from the regular expressions are required frameworks for fast generation of hardware engines. An example of a hardware regular expression engine approach targeted by the synthesis approach presented in [5] is shown in Fig. 1. The design consists of a character decoder (e.g., receives one character each clock cycle and flags the correspondent output) that outputs 256 flags (considering 8-bit ASCII codes) connected to the regular expression engines (one for each regular expression in the rule-set being synthesized). The synthesis tool can also take advantage of the sharing of some hardware blocks (e.g., responsible for prefix shared by more than one regular expression).

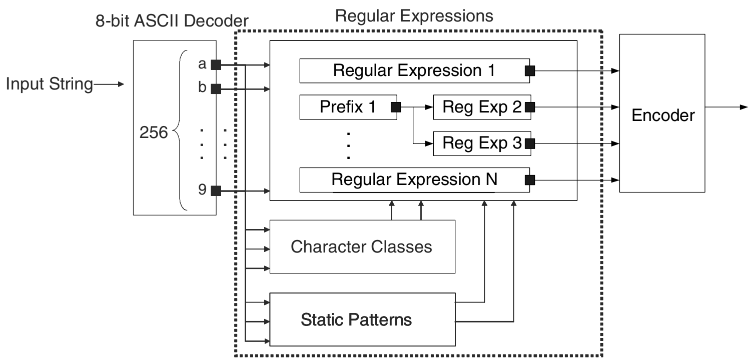


Fig. 1. Block Diagram of the architecture used in [5]

Although several contributions have been made, there are still open issues requiring further research. In addition to a brief explanation of current approaches, a number of those open issues are discussed in this paper.

This paper is organized as follows. Section 2 describes briefly the most relevant approaches to implement hardware regular expression engines. Section 3 discusses a number of open issues. Section 4 shows experimental results related to the implementation of hardware engines for the Bleeding Edge regular expressions. Finally, section 5 draws some conclusions.

2 Implementing Hardware Regular Expression Engines

There are two main approaches to implement regular expressions in hardware: using NFAs (Non-Deterministic Finite State Automats), or using DFAs (Deterministic Finite State Automats). The NFAs have been the solution initially used (see, e.g., the first known approaches to implement regular expressions in hardware [6] [7] [8]), and their inherent parallelism make them appealing for hardware implementations. DFAs are simpler and are the preferable model used in software implementations. Note, however, that DFAs need usually more nodes than NFAs and suffer from state explosion.

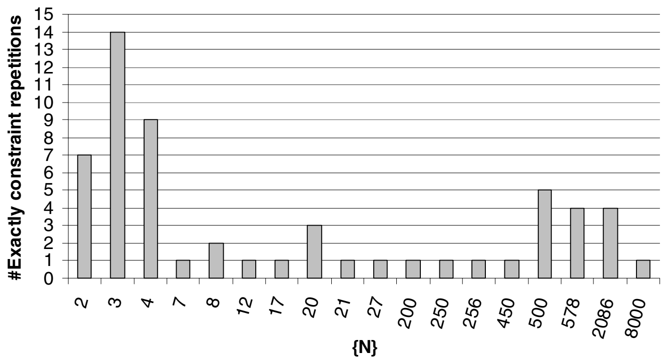
Both DFA and NFA based designs have problems when handling some of the regular expressions existent in NIDS (e.g., Snort), mostly because of the large amount of some kinds of quantifiers present (Exactly, AtLeast and Between – referred as *constraint repetitions*). Quantifiers as the previous ones, specifying repetitions of thousands of characters, are common (see Fig. 2 for the Bleeding Edge rule-set), and that is even more prominent in larger rule-sets (e.g., Snort). Most hardware solutions have to represent each of these characters individually (i.e., with full unrolling of repetitions) in order to achieve high-performance demands. From the number of repetitions presented in the new generation of NIDS rule-sets, it can be concluded that full unrolling is not an acceptable solution, because of the large hardware resources required.

NFA-Based Implementations

Regular expressions can be implemented using compound blocks. Sidhu and Prasanna [9] introduced, as far as we know, the NFA based block approach to implement regular expressions in hardware. They introduced the five fundamental blocks: Character, Kleene Star, Concatenation of Characters (Static String) Union and Parenthesis (see Table 2). With these five blocks, any kind of regular expression *defined by a regular language* can be built. Note, however, that designs based on those NFA building blocks implement *constraint repetitions* by unrolling the repeated expression and thus may lead to inefficient hardware engines.

In the paper by Franklin *et al.* [10], the same blocks introduced by Sidhu and Prasanna are used. They use a rule-set of Snort and implement all the static-pattern matching portion with an FPGA. Regular expressions were used as a

(a)



(b)

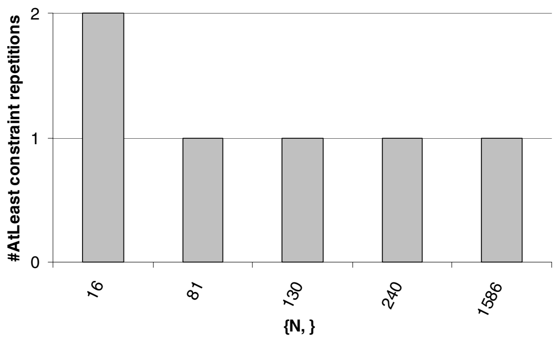


Fig. 2. Distribution of *constraint repetitions* of type: (a) Exactly; (b) AtLeast. Results are for the Bleeding Edge (Oct. 2006 version) presented in Table 1.

Table 2. Block primitives employed in the hardware engines as implemented in [5]

(a) Character	(abc...) Static String	(|) Union	(*) Kleene Star
( ) Parenthesis	(ˆ) Caret	($) Dolar	[ ] Character Class
(.) Dot	(?) Question Mark	(+) Plus	
{N} Exactly	{N,} AtLeast	{N,M} Between	

tool to represent *static strings*, and, to the best of our knowledge, none regular expressions present in the Snort rule-set were implemented.

Transition to DFAs

Another NIDS application, this time presenting a complete solution for a "Content-Scanning Module", is presented by Moscola *et al.* [11]. In terms of regular expressions, little is explained in the paper since the focus is on the complete system. To the best of our knowledge, the expressions implemented use the same blocks implemented by Sidhu and Prasanna, plus character classes (there is no mention to *constraint repetitions*). Their work goes a step further, and transform the NFAs extracted from the regular expressions into DFAs, to easily handle context switching (a DFA only has an active state at any time). They also claim that with the DFA approach, the number of states can be reduced most of the time, but the rule-sets used did not include Snort, and it is not explained in the paper what kind of regular expressions are prone to state reduction and state explosion.

The main focus of the work presented by Lin et al. [12] is area reduction, through reusing of common blocks. When implementing regular expressions, there are usually patterns that will be repeated (e.g., “tele” in patterns “telephone” and “television”). They proposed a scheme in order to share the logic of common prefixes, infixes and suffixes. As input, static patterns from an industrial NIDS application and static patterns and PCRE [13] regular expressions from Snort are used. While this is a step forward towards the implementation of more complex regular expressions in hardware, none of those blocks addressed two of the most used features in recent NIDS rule-sets: Character Classes and *Constraint Repetitions*. In addition the new blocks proposed in this scheme are not so compelling in terms of added performance achieved and/or space savings.

DFAs in an ASIC Implementation

Brodie *et al.* [14] presented high-throughput finite state machines (FSMs) for regular expression matching implemented on memory tables rather than logic. This design option was done because they focused on an ASIC implementation, where the memory approach is necessary if regular expressions are to be updated. Their design supports both static patterns and arbitrary regular expressions, and can scan multiple bytes per cycle for achieving high throughput. Although techniques for compression and redundancy minimization are employed, the proposed design architecture is, in terms of resources, prohibitive for FPGA implementations, and compared to the approach used in [5], it consumes much more resources.

DFAs with Microcontrollers

In Baker *et al.* [15], and as with the previous paper, a memory based approach is used. The idea is to make possible to update the regular expressions in the rule-set faster. Using memory, the regular expressions can be trivially updated (e.g., software-like), since the design in the FPGA does not need to be recompiled. The scope is an NIDS application. Also, static patterns and part of Snort regular expressions are both supported. They address the problem of *constraint repetitions*. To prevent state explosion due to the unrolling of certain regular expressions, the wildcards (*, +) and the *constraint repetitions* are handled separately by the microcontrollers, while the simpler patterns are done using DFAs implemented in glue logic. Their approach seems to have the same drawback as the one previously referred: it may require too much overhead when implemented in FPGAs. Basing the approach on microcontroller architectures, they inherit the same problem as the software counterpart – NFA to DFA conversion, because NFA execution is too complex when performed with sequential machines. It is also stated that this approach does not fully support *overlapped matching.*

Extended NFAs
Bispo *et al.* [5] use the same principle from [9] (NFAs in one-hot encoding and accepting one character per cycle), and includes some of the optimizations previously used: the central decoding of [16], and the prefix sharing of [12]. To save area, efficient blocks for *constraint repetitions* were introduced, sharing of other regular expressions components (see the Static Strings and Character Classes blocks in Fig. 1) was used, and the Xilinx SRL16 primitives were employed.

This approach also presented a synthesis methodology to automatically generate the hardware engines. It relies on transforming the regular expressions into a set of block primitives. The primitives used to generate hardware engines are summarized in the Table 2.

Implementations for the blocks referred in the last line of Table 2 have been introduced in [5]. Those implementations deal with *constraint repetitions* of single-cycle blocks (see the AtLeast implementation on Example 1) and implementations of repetitions for multi-cycle blocks without unrolling were identified as an open issue.

Note also that this work has been one of the first to show a fully implementation of a Snort rule-set using FPGAs.

Example 1. Single Character AtLeast Implementation
Fig. 3 shows the single character AtLeast Block proposed in [5]. It flags a match after detecting N or more equal characters. The output remains active until the first mismatch. Doing this, there is no need to store previous states. Even if new signals arrive, indicating subsequent matches, the output will not be affected. The AtLeast block can be implemented using only a counter (up to N) that keeps track of the number of matches.

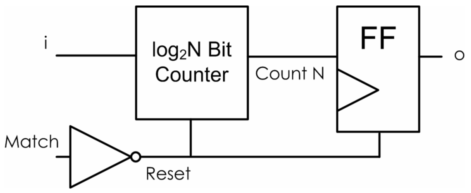


Fig. 3. AtLeast Block

Summary
As a brief summary, Table 3 shows the main characteristics of the previously introduced approaches. Although it is difficult to compare the approaches in terms of the performance achieved, in [5] a metric has been used. However, the metric fully depends on the number of characters in the regular expressions (this has been coined from comparisons with implementations of static strings) and that measure may not be the best one due to the quantifiers present in regular expressions. Unfortunately, a metric, such as the number of characters

Table 3. State-of-the-art summary

Authors	**Main Contributions**	**Target: RegExp / Static Patterns**
Sidhu *et al.* [9]	Introduction of NFA block approach.	RegExp
Franklin *et al.* [10]	Pattern Matching using regular expressions. Sharing of common prefixes.	Static Patterns
Lin *et al.* [12]	Sharing of prefixes, infixes and suffixes.	RegExp
Brodie *et al.* [14]	DFAs in memory tables for ASIC implementations. Multiple input bytes per cycle with high throughput.	RegExp
Baker *et al.* [15]	Micro-controllers for *constraint repetitions.* DFAs in FSMs with memory tables.	RegExp
Bispo *et al.* [5]	Extended NFA block approach	RegExpr

being scanned per second, is not always possible to be used, because different technologies are usually employed.

Concerning the hardware synthesis of the regular expressions, the tool presented in [5] uses a syntax tree-based approach to generate the structure of the hardware engines. That structure uses building blocks to implement the regular expression primitives. A structural-RTL VHDL code with components described in behavioral-RTL VHDL code is generated and logic synthesis, mapping, place and routing is then performed to create the bitstreams to program the target FPGA.

3 Open Issues

This section presents a number of open issues requiring research efforts in order to improve the hardware synthesis of regular expressions.

Overlapped Matching

One of the known problems is *overlapped matching.* Overlapped matches require new matching evaluations in every position of the input string. Considering as input the string “abc”, and overlapped matches is used, matching tests of “abc”, “bc” and “c” are performed. For a string with N characters, there will be N strings tested. Hence, *overlapped matching* is usually very inefficient in software, since all possible strings need to be tested. An approach such as the one presented by [9], takes advantage of the natural hardware concurrency and executes all *overlapped matching* cases concurrently without penalties. The reason behind this is because at each clock cycle, *every* block in the design sees the same character. In the end of the cycle, the character can be discarded, because all blocks have tested the mentioned character in every possible string position. This is the major reason why it has been difficult to propose a generic block for *constraint repetitions.*

Multi-character *Constraint Repetitions*
The *constraint repetitions e1*{*N*} (Exactly), *e1*{*N,*} (AtLeast) and *e1*{*N,M*} (Between), where *e1* represents a generic regular expression, can be implemented in hardware or software, using NFAs or DFAs. However, their non-unrolled hardware implementation with high throughput, *overlapped matching* and with all the blocks of the engines processing each character in one clock cycle still is an open issue.

For instance, the approach in [5] solves this problem when *constraint repetitions* are applied to regular expressions (*e1*) of the form single Character, single Dot or single Character Class. However, when *e1* is a more complex regular expression, unrolling has been used also as a way to enforce *overlapped matching.* As an example, a possible implementation of a *constraint repetition* evaluating *e1*{*N*} including *overlapped matching* and without requiring a single-cycle implementation is discussed in Example 2.

Example 2. *Constraint Repetitions* with *Overlapped Matching*
Consider the regular expression *(aba)*{*2*}. If the input string is "ababaaba", with *overlapped matching* it is equivalent to test as input the set "ababaaba", "babaaba", "abaaba"... Thus, a match of the sub-string "abaaba" should be flagged.

Note that when full unrolling the repetitions, all the states are explicitly available and *overlapped matching* is accomplished. However, applying full unrolling to all the *constraint repetitions* may require large amounts of FPGA resources. The approach presented in [5] implements, without full unrolling, repetitions of only one character (it should be noted that most *constraint repetitions* found in NIDS rule-sets are of this kind).

Fig. 4 presents an implementation with *overlapped matching* for the regular expression *(aba)*{*N*}. In this case, the 2-bit counter "001" counts sequences of the pattern "001" (the string matching for "aba" only happens after two clock cycles of un-matching) until it reaches N. For every '1' in the input without being in this sequence the counter is initialized to 1 (*overlapped matching*) and it is cleared in the other cases. In this way, *overlapped matching* is accomplished. However, this solution only deals with *e1* expressions (*e1*{N}) matching always the same number of characters per repetition (3 in this example), and a solution for the other cases needs further work.

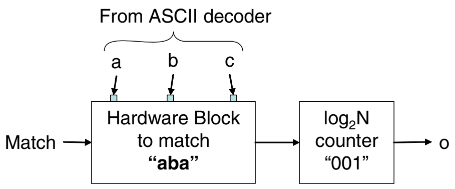


Fig. 4. Block diagram of a possible implementation of the regular expression *(aba)*{*N*} with *overlapped matching* and without unrolling

There is a trade-off between full unrolling and the use of hardware blocks of this kind. The special counters used only make sense for values of N above a certain limit. This is a feature that should be part of a synthesis tool for regular expressions.

Multi-character Input

There are two ways to improve throughput: higher clock frequency, and higher number of characters scanned per cycle. Higher clock frequency can only be achieved through improved hardware designs or FPGA technology upgrades. Increasing the number of characters scanned on each cycle is, whenever possible, an interesting option to improve performance. However, if *overlapped matching* is supported, that will lead to undesirable area overhead. This happens because all possible offsets must be taken into account [16]. In the paper by [16] it is not clear if it is advantageous to use a multi-character scheme in a one-hot encoding architecture, because when increasing the number of characters scanned per cycle, the number of implemented patterns decreases (due to the needed overhead). This is a problem, because current rule-sets (e.g., Snort) contain thousands of regular expressions. When scanning is done one character per cycle, all characters are position independent (since the blocks only see one character at a time: it does not matter in what position it has arrived). When unrolling is used, all states are explicitly implemented, and it is easier to accomplish *overlapped matching* with multi-character input. Due to the large hardware resources needed, these kind of approaches are now looking for techniques for space-saving [14].

Large Unions

In the NFA approach, one aspect of regular expressions that can have a negative impact in the clock frequency is the Union (same as the logic operator OR). In regular expressions unions of many elements can be used. Also, when sharing common parts of regular expressions union operations are used. With designs working strictly at one character per clock cycle, any union needs to perform its operation during a single clock cycle. The more elements a union has, the slower it may become (when using an LUT-based FPGA). If an union in a regular expression is responsible to a decrease in clock frequency, the synthesis tool should split the regular expression into two or more equivalent ones, using each one a subset of the unions.

Area Reduction

Area reduction in regular expressions is essential. When unrolling all the repetitions in a normal set of Snort regular expressions, up to 500,000 characters are needed to represent in the FPGA (due to the high repetition bounds). The proposed solution was to use special blocks that count the repetitions, instead of representing all the possible states. In repetitions such as AtLeast, a large amount of resources can be saved this way [5]. In repetitions like Exactly, where states are important to the output, they are compactly represented by shift registers implemented with Xilinx SRL16s primitives. Using prefix sharing also saves substantially hardware resources (∼12% less area for a version of the Snort 2.4 rule-set).

There is still work to be done to see if the infix-sufix sharing used by [12] can be supported by the approach presented in [5]. This is currently an open issue.

Hardware Virtualization

Another interesting issue would be the use of dynamic reconfiguration in order to accomplish hardware virtualization. This can be of paramount importance since the NIDS rule-sets are continuously increasing. Hardware virtualization can be exploited by temporal partitioning of the hardware engines executed by time-sharing the target device. At a first glance, temporal partitioning seems not easy to be applied, especially maintaining the one character per cycle processing rate.

Comparing Approaches

A metric often used for area, is the used resources by number of implemented characters. This makes sense for Static Pattern Matching, where all the elements are characters, but not for Regular Expressions. The expressions can be very different in terms of the structures they need (e.g., one expression may rely heavily on *character classes* and other on *constraint repetitions*), and many of the structures they need are independent of the number of characters that appear in the regular expression. A fair metric for regular expressions would be to compare results against fixed sets of regular expressions.

4 Experimental Results

This section illustrates the complexity of the regular expression hardware engines to implement rule-sets of current versions of NIDS. Table 4 shows the results obtained for the hardware engines responsible to implement the rule-set of the Bleeding Edge IDS (with characteristics presented in Fig. 2) with a Xilinx Virtex2 6000, speed grade -6, FPGA. The results were obtained after synthesis, mapping, place and route (Xilinx ISE 8.1 has been used) of the generated VHDL using the synthesis tool previously presented in [5]. The tool receives the regular expressions in the PCRE format and generates the VHDL-RTL (Register Transfer Level) code for the hardware engines.

The VHDL specifications generated consist of about 279,519 and 1,373,095 lines of code (including the character decoder module) for the version using counters and the version using full unrolling for the *constraint repetitions*, respectively.

As can be seen, a large number of FPGA resources is needed to implement the regular expressions of the Bleeding Edge rule-set used in this paper. The number of slices almost doubled when *constraint repetition* are unrolled (third column of Table 4) and surprisingly the maximum clock frequency achieved decreases. This is partially justified by the increase in the number of FFs and routing interconnections, when using unrolling instead of the SRL16 primitives. These

Table 4. Results when implementing the regular expressions of the Bleeding Edge rule-set (October 2006 version)

	non-unrolled (counter-based) *Constraint Repetitions*	fully unrolled *Constraint Repetitions*
Lines of VHDL	279,519	1,373,095
Execution time (RegExpr synthesis + Logic Synthesis + P&R)	∼ 1 hour and 24 min.	∼ 1 hour and 47 min.
#4-input LUTs	2,364	10,761
#FFs	12,533	29,290
#Slices	12,497	24,953
Maximum Frequency (MHz)	170.882	102.543
Maximum Throughput (Mchars/s)	170	102

preliminary results indicate that using the approach presented in [5], which uses building blocks for *constraint repetition* fully optimized with FPGA primitives, the full unrolling option might not be an interesting design decision.

The maximum clock frequencies of the designs permit to achieve a throughput of 170 and 102 Mega chars per second with the use of counters and with fully unrolling, respectively. Note, however, that the experimental results obtained do not take fully advantage of further optimizations that can be exploited such as the use of timing constraints.

Although a large number of FPGA resources are needed, the Bleeding Edge rule-set is relatively small and includes less *Constraint Repetitions* when compared with some heavier Snort rule-sets (see Table 1 for a comparison).

5 Conclusions

This paper introduced current solutions and a number of open issues related to the implementation of regular expressions on FPGAs. A special focus on approaches that can be systematically used to generate dedicated hardware regular expression engines, able to achieve high performance demands, has been addressed. One of the most challenging applications of hardware regular expression engines are the emergent Network Intrusion Detection Systems (NIDS). In order to show the complexity of the hardware engines needed to implement the regular expressions included in current NIDS, this paper includes experimental results for the Bleeding Edge regular expressions rule-set.

Ongoing work focuses on research efforts to circumvent some of the major limitations identified in this paper.

Acknowledgments

This work was partially supported by the European Commission in the context of the Scalable computer ARChitectures (SARC) integrated project #27648 (FP6).

João Bispo would like to thank INESC-ID for the PhD scholarship. João Cardoso would like to acknowledge the support by the Portuguese Foundation for Science and Technology (FCT) - FEDER and POSI programs - under the CHIADO project (POSI/CHS/48018/2002).

References

1. Sailesh Kumar, et al., “Algorithms to accelerate multiple regular expressions matching for deep packet inspection,” in *ACM SIGCOMM Computer Communication Review,* Volume 36 , Issue 4, October 2006, pp. 339-350.
2. Snort official web site, http://www.snort.org. Accessed last time on November 2006.
3. Martin Roesch, “Snort: Lightweight intrusion detection for networks,” In *Proc. 13th Systems Administration Conference (LISA)*, USENIX Association, November 1999, pp 229–238.
4. Bleeding Edge Threats web site, http://www.bleedingthreats.net. Accessed last time on November 2006.
5. João Bispo, Yiannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis, “Regular Expression Matching for Reconfigurable Packet Inspection,” in IEEE International Conference on Field Programmable Technology (FPT’06), Dec. 13-15, 2006, Bangkok Thailand, IEEE Computer Society Press, pp. 119-126.
6. R. W. Floyd and J. D. Ullman, “The Compilation of Regular Expressions into Integrated Circuits,” in *Journal of the ACM (JACM)*, vol. 29, no. 3, pp. 603–622, July 1982.
7. A. R. Karlin, H. W. Trickey, and J. D. Ullman, “Experience With A Regular Expression Compiler,” in *Proceedings of the ICCD: VLSI in Computers*, 1983.
8. R. McNaughton and H. Yamada, “Regular Expressions and State Graphs for Automata,” in *IEEE Transactions on Electronic Computers*, vol. 9, pp. 39–47, 1960.
9. R. Sidhu and V. K. Prasanna, “Fast Regular Expression Matching using FPGAs,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.
10. R. Franklin, D. Carver, and B. Hutchings, “Assisting Network Intrusion Detection with Reconfigurable Hardware,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.
11. J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, “Implementation of a Content-Scanning Module for an Internet Firewall,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2003.
12. C.-H. Lin, C.-T. Huang, C.-P. Jiang, and S.-C. Chang, “Optimization of regular expression pattern matching circuits on FPGA,” in *Proceedings of the conference on Design, automation and test in Europe (DATE’06)*, 2006, pp. 12–17.
13. Perl Compatible Regular Expressions website, http://www.pcre.org/. Accessed last time on November 2006.
14. B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching,” in 33rd International Symposium on Computer Architecture (ISCA’06), 2006, pp. 191–202.
15. Z. K. Baker, H.-J. Jung, and V. K. Prasanna, “Regular Expression Software Deceleration For Intrusion Detection Systems,” in *16th International Conference on Field Programmable Logic and Applications (FPL’06)*, 2006.
16. C. R. Clark and D. E. Schimmel, “Scalable Parallel Pattern-Matching on High-Speed Networks,” in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2004.

About the Importance of Operation Grouping Procedures for Multiple Word-Length Architecture Optimizations

Nicolas Hervé, Daniel Ménard, and Olivier Sentieys

IRISA – University of Rennes 1 – France
{first-name}.{name}@irisa.fr

Abstract. This paper presents an alternative approach for *multiple word-length* architecture synthesis and optimization. It is based on an iterative refinement process on *operation grouping*, word-length assignment and high-level synthesis. The focus is on the sub-problem of operation grouping before word-length assignment, and within iterations. Two algorithms are proposed and first results show the interest of the approach and invite for more investigations in iterative grouping procedures.

1 Introduction

This paper addresses the problem of High-Level Synthesis (HLS) of fixed-point architectures when arithmetic operators are shared and may have distinct Word-Lengths (WL). The objective is to minimize the cost (area and/or energy) while respecting accuracy and latency constraints. Unlike specialised DSP or general purpose processors, ASIC and FPGA implementations allow the number of arithmetic resources for each type (adder, multiplier, etc) and the input/output operation WL to be freely chosen, thus providing an important optimization potential (in terms of area, energy and latency).

The multiple word-length architecture synthesis problem is usually split into two sub-problems: data WL determination and HLS. Thus sub-problems have been processed separately. Firstly a dedicated resource is considered for operation WL determination [1,2]. The WL assignment has been demonstrated to be NP-Hard in [3], so heuristic methods have to be used for large scale problems such as those compared in [4]. Secondly the WL information so obtained is used as input for HLS techniques adapted for multiple WLs [5,6,7]. However, the two processes are highly inter-dependant and thus have to be coupled for more efficient results. In [8], the coupling is achieved through WL determination before and after HLS. Recently an optimal combined approach suited for new heuristic method evaluations has been presented in [9].

As the processes are inter-dependant and as suggested in [10], it could be interesting to use feedback in an iterative manner. To our knowledge, only work in [11] proposes an iterative refinement approach. The present paper explains more in detail the operation grouping objectives and mechanisms.

Operation grouping algorithms have been used in [6,8] before WL assignment or before HLS, both to limit the number of optimization variables and to guide HLS resource binding. In this paper, operation grouping information is used before WL assignment. This information is deduced from previous syntheses according to an iterative operation grouping algorithm. Thus, this article focuses on operation grouping algorithms.

The paper is organised as follows: section 2 presents the operation grouping procedure objectives and concepts to solve the problem. Section 3 proposes two algorithms to explore the design search space. Section 4 shows results for the first presented algorithm and section 5 concludes this paper.

2 Operation Grouping Procedures

The objective of a grouping procedure is to provide *a priori* information about future resource binding for the next global optimization step (i.e. WL optimization). Grouping procedures are, at the same time, both predictions on the resource number and constraints on the HLS.

A group is defined as a set of operations to be optimized with the same WL. A grouping represents a way to partition the set of operations among the groups. To achieve a grouping, the principle is first to determine the number of groups for each operation type, then to distribute the operation among the groups. For each operation type, the number of groups should correspond to the number of arithmetic operators used in the final architecture, so that grouping done *a priori* corresponds to the HLS resource binding.

2.1 Research Space

Let T, O and G be the sets of operation types, operations and groups respectively. Let t be a given operation type, O_t and G_t the sets of operations and groups of type t. The problem consists, for each operation type $t \in T$, of distributing the set of operations O_t among the groups G_t. So, let n be the number of operations and p the number of groups. This problem is equivalent to distributing n distinct elements in exactly p non-distinct sets, with $n \geq p$.

Proposition 1. *The number of solutions noted G_n^p is given by the following recurrent equation system where $A_n^p = \frac{n!}{(n-p)!}$ is the number of arrangements for p elements among n:*

$$\begin{cases} G_n^1 = 1 \\ G_n^p = \frac{p^n - \ {}_{k=1}^{p-1} A_p^k G_n^k}{p!} \qquad \forall p \leq n \end{cases} \tag{1}$$

Proof. The problem is equivalent to counting the way of arranging n elements into p distinct groups, possibly empty (p^n, because there are p choices for each n element) and to remove the number of arrangements with empty groups (1 empty group, 2 empty groups, etc., until $p-1$ empty groups) and to divide the result by the number of possible permutations ($p!$) of the p (non-empty) groups. The arrangement number with exactly k empty groups ($1 \leq k \leq p-1$) is equal to the number of ways to sorting the $p-k$ non-empty (thus distinct) groups (A_p^{p-k}) multiplied by the repartition of n elements in exactly $p-k$ groups (G_n^{p-k}).

The grouping search space is thus very important (in $O(p^n)$), so a maximum information is needed to both reduce and to explore it intelligently.

2.2 Exploring the Grouping Search Space

The targeted optimal grouping is among groupings for which each group will require only one operator. Let G_{1r} be the set of such groupings.

One way to reduce the architectural cost is to reduce the number of resources[1]. For G_{1r} groupings, this is equivalent to reducing the number of groups. The G_{1r} research space is considerably reduced compared to the previous research space. However it is not easy to stay in G_{1r} while searching for solutions. To check whether a grouping is G_{1r} or not, HLS must be used. When this is not the case, that is when any group needs at least 2 operators, some operations of this group shall be redistributed to return to G_{1r}.

Grouping corresponding to resource binding is necessarily G_{1r}. Thus from a non-G_{1r} grouping, the synthesis which allocates several operators to a group, proposes to split the initial group to obtain a G_{1r} grouping with more groups.

Taking into account information such as data dependencies and previous scheduling and binding, it is possible to deduce that the group choice for certain operations will directly depend on, be limited by or limit the group choice of other operations in order to keep G_{1r}. Thus, for the set of operations, a priority order to choose a group may be attributed. The choice of this priority order is important for the efficiency of algorithms with loop-back [12]. Highest priority operations should be those with the fewest degrees of freedom.

Prioritizing operations with a previous WL specification and allowing only group binding on immediately superior WL group, permits results from [6] where there is only $O(n^2)$ possible grouping solutions to be retrieved.

3 Proposed Algorithms

3.1 Simple Grouping Algorithm

For this approach, operation grouping is directly determined by the resource binding results. All operations assigned to an operator are specified to be in the same group for the next iteration. For the first iteration, the groups correspond to the different operation types. This approach is illustrated in figure 1.

Few iterations are needed to converge to a solution. In this method, groups do not constitute operation partitions, but just propose a common minimum WL which will be defined by the next phase of the global process and be available for

[1] One other way is to reduce the word-length of these resources.

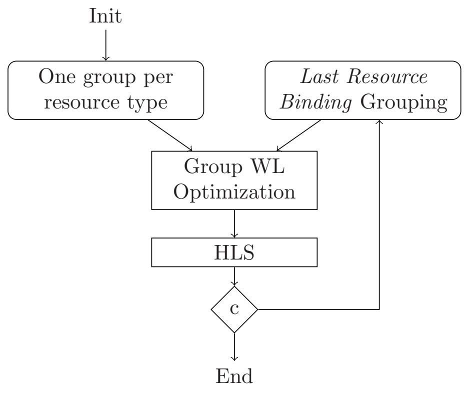


Fig. 1. Synopsis of Simple Grouping Algorithm

HLS. Indeed, the HLS could assign some operations to operators from another group with greater WL. Thus, the efficiency of this method is based on the ability of the HLS tool to share resources between groups. The algorithm finished when HLS resource binding coincides with data grouping or after a predefined number of iterations. Due to an absence of control and loop-back in this algorithm the last solution may not be the best explored solution.

3.2 Advanced Grouping Algorithm

To obtain a better solution, scheduling and binding should be performed in the same process. The present algorithm benefits from possible schedulings to direct grouping, and thus indirectly binding. It can be classified as a construction algorithm with loop-backs. The synopsis of the method is presented in figure 2.

We define the *operation mobility* as the set (interval) of possible execution dates for the operation and the *mobility index* as the number of distinct possible execution dates.

The minimum WL of each operation which permits the accuracy constraint to be respected is first determined. Then for a number of groups fixed for the iteration, the mobility indexes are computed and operations are processed by increasing order of mobility indexes. They either are added to an existing group, constitute a new group, or take the place of an operation already in a group.

Operation mobility is used to try and place operations in the most adequate group, that is the one with WL immediately superior or equal to the operation WL. The group WL is defined here as the maximum required WL for operations in the group. To be bound to a group, the other operations should not have execution constraints which overlap all scheduling possibilities offered by the mobility of the operation, otherwise the grouping will not be G_{1r}.

So if the operation cannot be added to the desired group, the algorithm first tries to swap this operation with a smaller WL operation in the group, before repeating this process with a larger WL group.

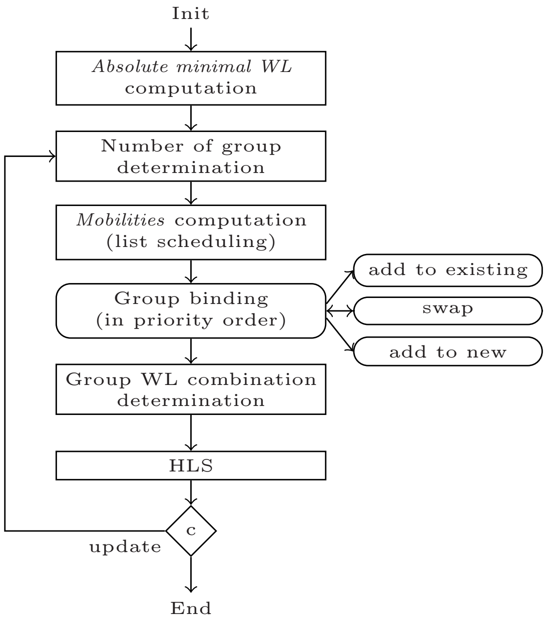

Init
Absolute minimal WL computation
Number of group determination
Mobilities computation (list scheduling)
Group binding (in priority order)
add to existing
swap
add to new
Group WL combination determination
HLS
c
update
End

Fig. 2. Synopsis of Advanced Grouping Algorithm

If the current operation WL is larger than all group WLs, there are two possibilities: either create a new group or add this operation to an existing group (priority to groups with highest WL), swapping if necessary with another operation (priority to lowest WL operations). The second choice is performed by default, thus new groups are always created with a WL as small as possible.

When an operation has its place stolen by another operation with a larger WL, it returns to the top of the priority list. Convergence of this algorithm is assured by the fact that operations can only take the place of smaller WL once and are bound one by one.

Once the grouping is performed, a WL optimization procedure is carried out on groups. Indeed, the combination of the obtained group WL usually does not respect the accuracy constraint (individual WLs are necessary but not sufficient regard to other WLs), so a HLS is finally achieved. As scheduling depends on operation latencies, which depends on operator WLs, new mobilities are computed at each iteration.

4 Experimental Results

Firstly, algorithm evolution over iterations is presented on a small application example for given latency and accuracy constraints. Secondly, caracterisation results in function of those constraints are presented on a 8-point radix 2 FFT algorithm.

Table 1. Iteration details for Simple Grouping Algorithm

```
float searcher(float inI, float inQ,
               float *cI, float *cQ)
{
  static float xI[3];
  static float xQ[3];
  float accI, accQ;
  int i;

  xI[0] = inI*K[0];
  xQ[0] = inQ*K[1];

  accI = xI[0]*cI[0] - xQ[0]*cQ[0];
  accQ = xI[0]*cQ[0] + xQ[0]*cI[0];

  for(i=1; i<3; i++) {
    accI += xI[i]*cI[i] - xQ[i]*cQ[i];
    accQ += xI[i]*cQ[i] + xQ[i]*cI[i];

    xI[i] = xI[i-1];
    xQ[i] = xQ[i-1];
  }

  return accI + accQ;
}
```

i	Opr.	Lat.	LUT	Operation Binding	Opt. Gr. WL
1	add19	2510	19	(43,81,38,33,28,23,14,5,3)	add19
	add19	2510	19	(76,71,66,61,56,51)	add18
	mul15x2	4072	34	(10,45,83,40,35,30,25,16,7)	mul14x2
	mul15x2	4072	34	(19,48,86,78,73,68,63,58,53)	mul16x2
	total		106	SNR : 60.8 dB	(105)
2	add19	2510	19	(43,81,38,33,28,23,14,5,3)	add18
	add18	2469	18	(76,71,66,61,56,51)	add17
	mul14x2	4009	32	(10,40,30,25,7)	mul14x2
	mul16x2	4133	36	(19,45,35,83,16,48,86,78,73, 68,63,58,53)	mul16x2
	total		105	SNR : 62.2 dB	(103)
3	add18	2469	18	(43,38,81,33,28,23,76,14,5, 71,66,61,56,51,3)	add23
	mul16x2	4133	36	(10,30,7,45,83,35,68,16,53)	mul14x2
	mul16x2	4133	36	(19,40,25,48,86,78,73,68,63)	mul15x2
	total		**90**	SNR : 63.0 dB	(89)
4	add23	2676	23	(43,81,38,33,28,23,14,5,51,3)	add20
	add23	2676	23	(76,71,66,61,56)	add18
	mul14x2	4009	32	(10,45,30,7,16)	mul13x2
	mul15x2	4072	34	(19,83,35,68,53,48,86,40,78, 73,25,63,58)	mul16x2
	total		112	SNR : 61.2 dB	(104)

The chosen application for detailing iterations steps, is part of the WCDMA application for the UMTS standard. It computes intercorrelations between received complex signal and a reference sequence. As the parameter coefficients for this application are either -1, 0 or 1, all coefficient WLs have been set to 2. The input data are specified in range [-1,1], the application latency has been set to 50 ns, the clock latency to 2.5 ns and the accuracy constraint to a signal-to-noise power ratio of 60 dB. The target architecture is a LUT based implementation on Xilinx Virtex II FPGA. The source code of the test application and the HLS results for 4 iterations are presented in table 1. The numbers reported in the "Operation Binding" column correspond to operation node numbers in our intermediate representation. The last column reports the group WL after optimization and the cost prevision if in G_{1r} (noted within parentheses). For this example, the best area cost is obtained at iteration 3 whose corresponding implementation will only need a single 18-bit adder and two 16x2-bit multipliers, resulting in a saving of 15% operator area compared to the initial solution.

Complete optimization characterization results are now presented for a 8-point radix-2 FFT algorithm. For this algorithm, roots of unity coefficients are all encoded on 12 bits for both the real an imaginary parts. The same SNR constraint is specified for all output signals.

Figure 3 compares the mutliple WL approach with a classical 16-bit implementation and a *per operation type* WL optimization (corresponding to the first iteration of the procedure). A 16-bit uniform WL implementation results in a SNR of 58 dB, so this value has been used as a fixed constraint for the optimizations. The LUT area is plotted for the classical solution and the two optimized solutions for different timing constraints. The bar plots show the percentage of area saved by the two optimized solutions compared to the classical 16-bit implementation and emphasize the extra saving provided by multiple WL solutions.

Figure 4 presents the total operator LUT area obtained after optimization for varying accuracy and timing constraints. Up to ten iterations on grouping

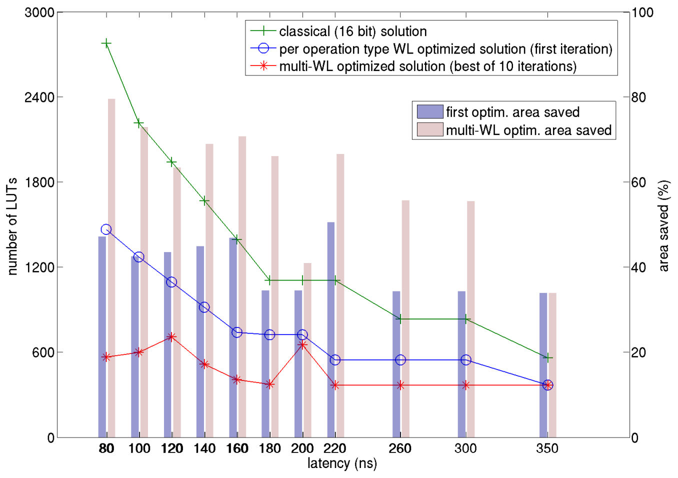


Fig. 3. Comparison with 16-bits classical implementation

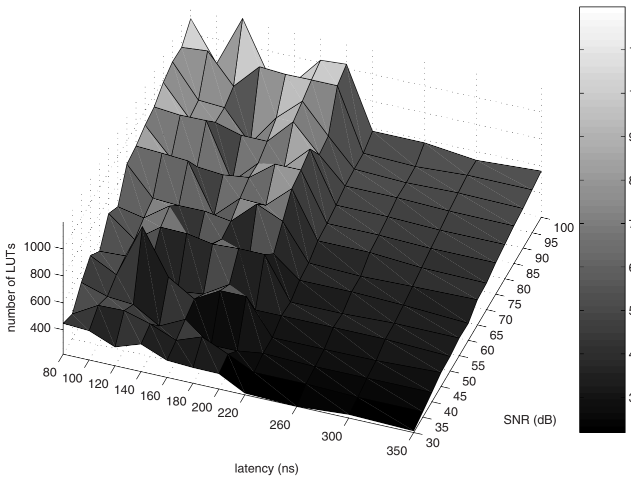


Fig. 4. Area Optimization Results for 8 points Radix-2 FFT algorithm

have been performed for each points. This 3D-graph presents some irregularities. The graph-equivalent function indeed is not monotonous in the constraint variables (SNR and latency), as expected for a characterization. This is mainly due to the fact that the simple grouping algorithm has no loop-back control, and so will always try to improve the last solution and not the last best one[2]. Indeed, from an iteration to another, area usually increases when the HLS resource binding leads to a non-G_{1r} grouping.

[2] This could however finally lead to a better solution, as simulated-annealing does.

Table 2. Numbers of best – over ten – iterations

SNR	latency (ns)										
(dB)	350	300	260	220	200	180	160	140	120	100	80
30	2	7	4	4	2	6	4	2	6	2	5
35	2	2	2	2	9	6	6	7	2	8	4
40	3	9	9	9	4	4	4	7	8	8	10
45	1	2	3	3	2	2	10	2	1	6	2
50	1	2	4	4	9	5	7	7	10	4	5
55	2	10	4	4	3	3	8	4	3	6	3
60	3	2	2	2	9	9	4	6	4	6	7
65	2	5	4	7	10	6	10	5	9	4	6
70	6	9	2	2	2	9	9	6	2	2	4
75	2	2	2	2	2	2	4	7	10	6	2
80	1	2	2	2	2	5	10	7	2	6	4
85	1	2	2	2	10	2	9	5	7	6	2
90	2	2	4	5	10	2	4	9	8	10	2
95	2	2	4	7	2	2	2	5	9	5	4
100	6	9	2	2	2	4	4	9	1	8	2

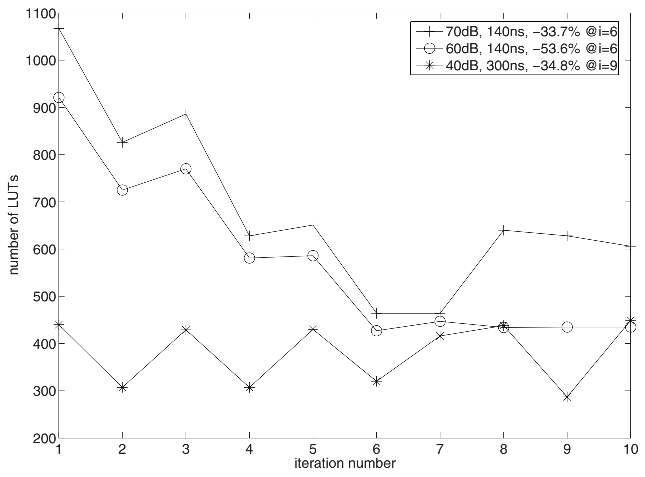


Fig. 5. Iteration details for 3 cases

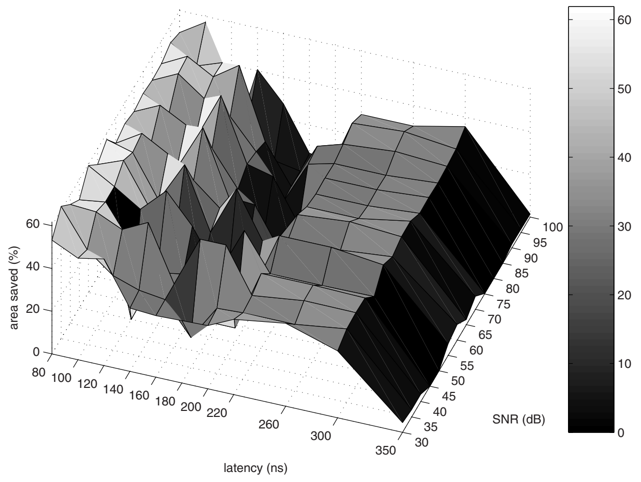


Fig. 6. Area saved compared to first optimized solution

Table 2 indicates the number of iterations performed to give the best solution. Still due to the absence of loop-back control in the iterative grouping procedure, no relation appears between the best iteration number and the constraints. Figure 5 shows as example the LUT number evolution through 10 iterations for 3 constraint sets. Figure 6 presents the percentage of LUT area saved compared to the initial solution (with WLs optimized for each operation type). The graph shows gain up to 60% area saving with an average of 28%.

Thus, the optimization efficiency of the simple grouping algorithm is unpredictable because it strongly depends on the HLS results at each iteration step, but those results clearly show the optimization potential one should expect from iteration procedures.

5 Conclusion

This paper has presented an alternative approach to the HLS of multiple WL architecture based on an iterative process. Simple iteration grouping procedures, that can be used with traditional HLS tools, permit certain architecture exploration and refinement. Thus, the multiple WL architecture optimization subject needs more investigations on iterative grouping procedures.

References

1. Constantinides, G.A., Cheung, P.Y.K., Luk, W.: Optimum and heuristic synthesis of multiple word-length architectures. IEEE Trans. on Very Large Scale Integr. Syst. **13**(1) (2005) 39–57
2. Cmar, R., Rijnders, L., Schaumont, P., Bolsens, I.: A Methodology and Design Environment for DSP ASIC Fixed Point Refinement. In: Design Automation and Test in Europe Conf. (1999) 271–276
3. Constantinides, G.A., Woeginger, G.J.: The complexity of multiple wordlength assignment. Applied Mathematics Letters **15**(2) (2002) 137–140
4. Cantin, M.A., Savaria, Y., Lavoie, P.: A comparison of automatic word length optimization procedures. In: IEEE Symp. on Circuits and Systems. Volume 2. (May 2002) 612–615
5. Choi, J.I., Jun, H.S., Hwang, S.Y.: Efficient hardware optimisation algorithm for fixed point digital signal processing asic design. Electronics Letters **32**(11) (May 1996) 992–994
6. Wadekar, S., Parker, A.: Accuracy Sensitive Word-Length Selection for Algorithm Optimization. In: IEEE/ACM Conf. on Computer Design. (November 1998) 54–61
7. Xu, J., Cong, J., Cheng, X.: Lower-bound estimation for multi-bitwidth scheduling. In: IEEE Symp. on Circuits and Systems. (2005) 696–699
8. Kum, K., Sung, W.: Combined Word-Length Optimization and High-level Synthesis of Digital Signal Processing Systems. In: IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems. (August 2001) 921–930
9. Caffarena, G., Constantinides, G., Cheung, P., et al.: Optimal Combined Word-Length Allocation and Architectural Synthesis of Digital Signal Processing Circuits. IEEE Trans. on Circuits and Systems **53**(2) (May 2006) 339–343

10. Constantinides, G.A., Cheung, P.Y.K., Luk, W.: 6. In: Synthesis and Optimization of DSP Algorithms. Kluwer Academic (2004)
11. Hervé, N., Ménard, D., Sentieys, O.: Data wordlength optimization for FPGA synthesis. IEEE Workshop on Signal Processing Systems (2005) 623–628
12. Glover, F.W., Kochenberger, G.A., eds.: Handbook of Metaheuristics. Springer (January 2003)

Switching Activity Models for Power Estimation in FPGA Multipliers

Ruzica Jevtic, Carlos Carreras, and Gabriel Caffarena

Dpto. de Ingeniería Electrónica, E.T.S.I. Telecomunicación, Universidad Politécnica de Madrid, Ciudad Universitaria s/n, Madrid, Spain
{ruzica, carreras, gabriel}@die.upm.es

Abstract. This paper presents a novel high-level analytical approach to estimate logic power consumption of multipliers implemented in FPGAs in the presence of glitching and correlation. The proposed methodology is based on: 1) an analytical model for the switching activity of the component, and 2) a structural analysis of the FPGA implementation of the component. The complete model is parameterized in terms of complexity factors such as word-lengths and signal statistics of the operands. It also accounts for the glitching introduced by the component. Compared to the other power estimation methods, the number of circuit simulations needed for characterizing the power model of the component is highly reduced. The accuracy of the model is within 10% of low-level power estimates given by the tool XPower and it achieves better performance than other proposed high-level approaches.

1 Introduction

A large number of routing switches needed for interconnections and a less efficient utilization of resources have made power consumption a constraining factor for FPGA designs to enter main-stream low-power applications [11].

The dominant sources of power consumption in CMOS circuits are the charge and discharge of the node capacitances. For each node, power consumption is

$$P = \alpha \cdot C_l \cdot V_{dd}^2 \cdot f \tag{1}$$

where α (referred to as the switching activity) is the average number of $0 \rightarrow 1$ transitions in one clock-cycle, C_l is the load capacitance at the given node, V_{dd} is the power supply voltage, and f is the clock frequency.

Many accurate techniques for power estimation already exist at the logic and circuit levels. As they all need transistor or gate level circuit descriptions, the power estimation occurs late in the design process, thus leading to severe penalties in design time when constraints are not met. On the other hand, the methods for estimating power consumption at higher levels consider extensive module simulations for different input statistics as a step prior to high-level synthesis. As the number of combinations for input word-lengths is infinite, a new set of simulations for the module characterization is necessary each time the module's parameters change.

In this paper, we present a methodology which has proven to overcome the above mentioned problems and is used for estimating power consumption of multipliers implemented in LUTs. Although the use of embedded multipliers has become more popular, multipliers implemented in LUTs are extensively used in high-level synthesis. Unlike other proposed approaches, which are completely based on circuit and signal simulations, this approach is based on an analytical model that uses the operand's word-length and the signal statistics as parameters. We propose three different power models for estimating power consumption. First, we consider an array multiplier. We derive an expression for the switching activity of this type of multiplier by using a word-level regional decomposition of the input signal based on its statistics. With the information about switching activity we construct a power model parameterized in terms of operand word-lengths and their signal statistics. Next, we include the FPGA implementation details of the multiplier into the expression for the switching activity to construct the second power model with enhanced characteristics. Finally, we improve this model by considering the effects of signal glitching. All three proposed models are capable of producing fast and accurate estimates of logic power consumption in multiplier components regardless of the word-lengths of the operands.

The paper is organized as follows. Section 2 highlights the previous work done in the area of high-level power estimation. Section 3 presents some preliminaries of the work used for estimating signal transition activity from word-level statistics. In Section 4, the transition activity for an array multiplier with and without glitching effects is computed. It is followed by the experimental results in Section 5. We conclude this paper in Section 6.

2 Related Work

The approaches based on the bit-level input signal statistics [5, 9] consider average bit level statistics which are found to be in direct relationship with power consumption. These statistics are introduced as variables in an equation which estimates the average power consumed by the module. Coefficients multiplying the variables in the equation are determined through extensive simulations. This model can be applied only to a specific component with fixed word-length, whereas in the approach proposed here the operand's word-lengths are used as variables in a single power model for that component.

Approaches based on word-level signal statistics [2, 6] consider the variation in power consumption caused by the variation of the input signal variance, mean and correlation coefficients. The approach used in [2] generates first-order and second-order equations for adders and multipliers respectively. The only variable introduced in an equation is the operands' word-length. Therefore, operands of different sizes cannot be modelled with this methodology.

In [6], the Dual-bit type method is presented, which describes a strategy for generating a black-box model of datapath power consumption at the architecture level. The technique accounts for a word-level signal broken into 3 regions: uncorrelated, correlated and sign data bits. Based on this methodology, a product

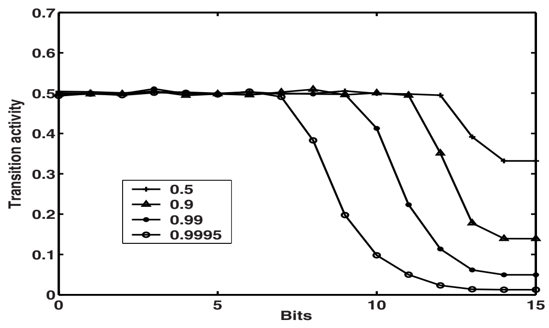


Fig. 1. Bit transition activity vs. bit position in a word for different autocorrelations

of the capacitance and the switching activity is calculated for every signal region by using extensive simulations. A disadvantage of this model lies in the fact that every specific black-box component needs to be simulated before determining its power consumption, thus, increasing design time significantly.

The approach that considers a signal division into correlated, uncorrelated and sign regions has also been used in the power estimation method proposed here, but instead of measuring the transition activity of the sign bits, it has been estimated from word-level signal statistics as presented in [7, 8].

3 Signal Model

In this analysis, signals at different points in the system are assumed stationary and considered to have zero-mean Gaussian distributions. It has been shown that dynamic power consumption in arithmetic components is affected to a greater extent by autocorrelation than by cross-correlation [2]. Therefore, we will consider only the effects of signals variances and autocorrelations on the power consumption models. As previously mentioned, we use the method described in [6] to divide each signal word into three regions referred to as MSB, linear and LSB region. To demonstrate the applicability of this approach, we have plotted the bit transition activity in signal word versus their bit position in the word for different autocorrelations (see Fig. 1). All the signals have Gaussian distribution of variance σ^2. It can be seen that the bits on MSB positions demonstrate a high correlation and have a constant switching activity up to a certain limit BP1 and that the bits on the lowest positions have a switching activity of 0.5 as they behave as random inputs. The region in between can be approximated as a linear region. Instead of a word division into regions based upon the transition activity, we will divide it based upon their bit-level correlation ρ_i following the methodology described in [7]. By definition $\rho_i = 0$ for $i < BP0$. It is assumed that $\rho_{BP1} = \rho$ where ρ represents the word-level temporal correlation. The expressions for the autocorrelation coefficient of all three regions are:

$$\rho_i = \begin{cases} 0 & i < BP0 \\ \frac{(i-BP0+1)\cdot\rho_{BP1}}{BP1-BP0} & BP0 \le i < BP1-1 \\ \rho_{BP1} & i \ge BP1-1 \end{cases} \tag{2}$$

In order to calculate the exact transition activity t_i we use the following relationship between the bit-level probability p_i (calculated as in [7]) and the bit-level autocorrelation ρ_i as it follows:

$$t_i = 2 \cdot p_i \cdot (1-p_i) \cdot (1-\rho_i) \tag{3}$$

The expression derived for a breakpoint BP1 is given by:

$$BP1 = [\log_2(6\sigma)] \tag{4}$$

The expression derived for a breakpoint BP0 depends on the coefficients of the ARMA (autoregressive moving average) signal model and is to be computed here for signals that have zero-mean Gaussian distributions. An (N,M)-order ARMA model can be represented as

$$x(n) = \sum_{i=0}^{N} d_i\gamma(n-i) + \sum_{i=1}^{M} a_i x(n-i) \tag{5}$$

where the signal $\gamma(n)$ is a white (uncorrelated) noise source with zero mean, and $x(n)$ is the signal being generated. It is also possible to transform this IIR model into one that depends only on the inputs as shown below

$$x(n) = \sum_{i=0}^{\infty} h_i\gamma(n-i) \tag{6}$$

where h_i can be computed according to the following recursion:

$$h_k = d_k + \sum_{i=1}^{N} a_i h_{k-i} \tag{7}$$

where $h_k = 0$ for $k < 0$, and $h_0 = d_0$. The breakpoint BP0, for signal $x(n)$ in (6) is estimated as the maximum of the BP0's of the signals $h_i\gamma(n-i)$.Hence,

$$BP0 = [\log_2(h_{\max}\sigma_\gamma)] \tag{8}$$

where $h_{\max} = \max(|h_i|)$. We will show the method for computing the ARMA-model coefficients in (8) when a signal has a zero-mean Gaussian distribution. Equations (6) and (7) have been used as our starting point in finding a relation between the signal statistics and the coefficients of the ARMA signal model. Based on the signal generation model presented in [2], it can be shown that any given signal with a zero-mean Gaussian distribution of variance σ^2 and autocorrelation coefficient ρ can be represented as an ARMA(0,1) model:

$$x(n) = d_0\gamma(n) + a_1 x(n-1) \tag{9}$$

Computing the variance and autocorrelation of signal which is presented as in (9) leads to a system of two equations. The coefficients d_0 and a_1 are then obtained by solving this system and the resulting expressions for them are given as follows:

$$d_0 \cdot \sigma_\gamma = \sqrt{(1 - a_1^2)} \cdot \sigma_x \tag{10}$$

$$a_1 = \rho \tag{11}$$

Next, we have computed the coefficients for the ARMA(0,1) model. Combining (7) and (11) we obtain that

$$\begin{aligned} &h_0 = d_0 \\ &h_1 = a_1 d_0 = \rho \cdot d_0 \\ &h_2 = a_1^2 d_0 = \rho^2 d_0 \\ &.... \end{aligned} \tag{12}$$

As the autocorrelation coefficient is always less than or equal to one, we obtain that the maximum of all h_i is h_0, i.e. d_0 . Finally, by replacing expressions (10) and (11) in (8), it becomes:

$$BP0 = \left[\log_2(\sqrt{1 - \rho^2} \cdot \sigma_x)\right] \tag{13}$$

We have used this expression for the breakpoint BP0 in our approach as it gives better results than the one proposed in [6].

4 Structural Model

4.1 Array Multiplier

We consider a standard array multiplier whose structure is shown in Fig. 2a. Operand x has N bits and y has M bits. The multiplier consists of basic elements, namely two-bit multipliers and half-adder and full-adder cells. We consider that each type of element is implemented into the slice section composed of one LUT and logic gates. As the LUT has the same structure regardless of the function performed in it, we assume that the capacitance being switched per each basic element is the same. This allows us to divide the dynamic power consumption of the array multiplier into 4 separate terms: V_{dd}^2, f, C_l and α. As the first three terms remain the same for each basic element of the multiplier, our goal is to analytically calculate the total switching activity and with one-time measurement of the logic power of the multiplier, include the first three terms as a constant multiplying the expression obtained for the switching activity. The module represented in Fig. 2a is regionally divided into 4 parts according to the input signal word decomposition as explained in Section 3. The linear region can be achieved by attributing the upper half of the bits in the linear region to the MSB region and the bottom half of the bits to the LSB region [6]. Hence, the number of bits in MSB region is obtained from:

$$x = [(BP1 - BP0)/2 + (N - BP1)] \tag{14}$$

where N is the number of bits in a signal word and [ ] is the rounding operation.

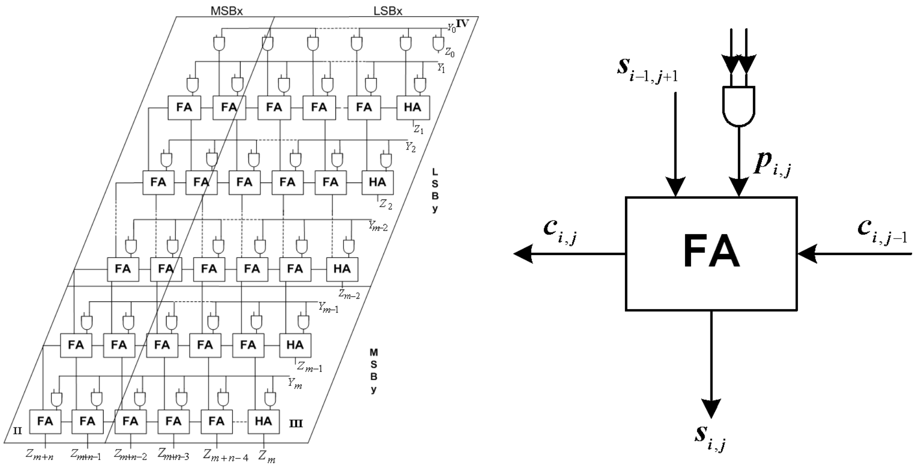


Fig. 2. a) Array multiplier, b) Full-adder cell

The four parts of the multiplier exhibit different switching activities. For each part, the switching activity on the outputs of its basic elements is to be computed as a function of their input switching probabilities. As the observed signals have zero-mean Gaussian distributions and are encoded in two's complement, it is assumed that the probability of each bit of being equal to '1' is the same as the probability of being equal to '0'. Although, these probabilities are assumed to be the same at the inputs of the multiplier, they change as the signals pass through the logic. With this effect taken into account, the switching activities of carry bits and outputs of the adder cells, as well as its probabilities of being '1' and '0', are to be computed as a function of the probabilities of the inputs and carry bits at the previous level. The methodology employed for computing the switching probability of the carry bit will be presented here as it is the most complex one. The calculation of the rest of the probabilities is obvious and only their expressions are provided. The list of notations used in the equations is given below:
- p is the transition probability at the output of the two-bit multiplier which goes to the one of the inputs of the full-adder cell
- q is the transition probability at the output of the full- adder cell from the previous level (see Fig. 2b) which goes to the other input of the full-adder cell
- c is the transition probability of the carry bit
- s is the transition probability of the output of the full-adder cell
- p^0 and p^1 are the probabilities of input p of being '0' and '1' respectively
- q^0 and q^1 are the probabilities of input q of being '0' and '1' respectively
- c^0 and c^1 are the probabilities of the carry bit of being '0' and '1' respectively
- s^0 and s^1 are the probabilities of the output bit of being '0' and '1' respectively

Index i is the row number and j is the column number of the full-adder cell in the array multiplier. The carry bit from a previous cell is the third input to a full-adder cell. As q represents the transition probability at the output of the full-adder cell from the previous level, it is clear that $q_{i,j} = s_{i-1,j+1}$.

Now we will show the methodology for computing the transition probability of the outgoing carry bit of the full-adder cell. As the full-adder cell has three inputs, there are 2^3 combinations for its inputs in one clock cycle. For each combination, there are four possible events which could occur in the following clock cycle. In the first case, neither of the inputs changes. Hence, there will be no transition in the carry. In the second case, only one of the inputs makes a transition. In this case, for the combinations "000" and "111" there will be no change at the outgoing carry bit. For the combinations "001","010" and "100", if any of the zeros changes, the transition will occur at the carry also. Hence, the transition activity for this case is:

$$\begin{aligned} c_{i,j}^{A1} &= p_{i,j}^0 \cdot q_{i,j}^0 \cdot c_{i,j-1}^1 \cdot (1 - c_{i,j-1}) \cdot (p_{i,j} + q_{i,j} - 2 \cdot p_{i,j} \cdot q_{i,j}) + \\ &p_{i,j}^1 \cdot q_{i,j}^0 \cdot c_{i,j-1}^0 \cdot (1 - p_{i,j}) \cdot (q_{i,j} + c_{i,j-1} - 2 \cdot q_{i,j} \cdot c_{i,j-1}) + \\ &p_{i,j}^0 \cdot q_{i,j}^1 \cdot c_{i,j-1}^0 \cdot (1 - q_{i,j}) \cdot (p_{i,j} + c_{i,j-1} - 2 \cdot c_{i,j-1} \cdot p_{i,j}) \end{aligned} \tag{15}$$

For the rest of the combinations "011","110"and "101" a change in the any of the ones, will produce a transition in the carry. Hence, the switching activity is

$$\begin{aligned} c_{i,j}^{A2} &= p_{i,j}^0 \cdot q_{i,j}^1 \cdot c_{i,j-1}^1 \cdot (1 - p_{i,j}) \cdot (q_{i,j} + c_{i,j-1} - 2 \cdot q_{i,j} \cdot c_{i,j-1}) + \\ &p_{i,j}^1 \cdot q_{i,j}^0 \cdot c_{i,j-1}^1 \cdot (1 - q_{i,j}) \cdot (p_{i,j} + c_{i,j-1} - 2 \cdot c_{i,j-1} \cdot p_{i,j}) + \\ &p_{i,j}^1 \cdot q_{i,j}^1 \cdot c_{i,j-1}^0 \cdot (1 - c_{i,j-1}) \cdot (p_{i,j} + q_{i,j} - 2 \cdot q_{i,j} \cdot p_{i,j}) \end{aligned} \tag{16}$$

In the third case, two inputs change and one remains the same. In the fourth case, all input bits change. The methodology for computing transition probability in these cases is the same as in the second case. The probabilities of the carry bit and the output bit of the full-adder cell of being '0' are computed as:

$$\begin{aligned} c_{i,j}^0 &= p_{i,j}^0 \cdot q_{i,j}^0 + c_{i,j-1}^0 \cdot (p_{i,j}^0 \cdot q_{i,j}^1 + p_{i,j}^1 \cdot q_{i,j}^0) \\ s_{i,j}^0 &= (p_{i,j}^0 \cdot q_{i,j}^0 + p_{i,j}^1 \cdot q_{i,j}^1) \cdot c_{i,j-1}^0 + (p_{i,j}^0 \cdot q_{i,j}^1 + p_{i,j}^1 \cdot q_{i,j}^0) \cdot c_{i,j-1}^1 \end{aligned} \tag{17}$$

The final expression for the carry is obtained by adding up the transition probabilities of all three cases.

Now, the only probability missing for the computation of the total switching activity is the probability at the output of the full-adder cell and which is given by:

$$\begin{aligned} s_{i,j} &= (p_{i,j} q_{i,j} + (1 - p_{i,j}) \cdot (1 - q_{i,j})) \cdot c_{i,j-1} \\ &+ (p_{i,j} \cdot (1 - q_{i,j}) + q_{i,j} \cdot (1 - p_{i,j})) \cdot (1 - c_{i,j-1}) \end{aligned} \tag{18}$$

The total switching activity of the array multiplier consists of the switching activities of the carry-bits and outputs of the adder and multiplier cells. Hence, the final result for the switching activity is obtained from the following expression:

$$SW = \sum_{i=1}^{M-1} \sum_{j=1}^{N} (s_{i,j} + c_{i,j} + p_{i,j}) \tag{19}$$

This methodology is similar to the methodology described in [3]. However, in [3], the method is applied to the whole module. Since the calculation of signal probability is NP-hard, in the case of very large circuits, a partitioning algorithm that limits the number of inputs to a module has to be employed. The approach proposed here applies the method for estimating switching activity only to a basic cell of a multiplier, thus providing a simple expression for the total switching activity of the module no matter its size.

The problem now reduces to the calculation of the other three power terms (C_l, V^2, f). As already mentioned, we consider the average capacitance switched per one basic element to be a constant. The other two terms, V_{dd} and f are fixed for a given FPGA architecture and clock period of the design. Hence, for a specific design, the product of all three parameters is considered to be a constant a which is introduced into the power model. It is obtained by solving the next equation:

$$P = a \cdot SW \tag{20}$$

P is the power obtained from the one-time low-level simulation and SW is the switching activity computed according to (19). Once the constant is computed, the power consumption of a multiplier with any given characteristics is estimated by using (20). It is clear that the model is parameterized in terms of operand's word lengths. Apart from being able to estimate power consumption of multiplier of any given size, the resulting model is also parameterized in terms of the input signal statistics. The statistics are expressed in terms of the transition probabilities of the input bits and the position of MSB-LSB breakpoint.

4.2 FPGA Multiplier Implementation

The above calculation takes into account the structure of the array multiplier according to its definition. However, there are many different ways to implement it into an FPGA. In particular, we consider the Virtex-2 family of FPGAs from Xilinx [10]. Xilinx IP Cores optimize the implementation of the array multiplier by transforming it into a row adder tree multiplier. This type of multiplier rearranges the adders of the array multiplier to equalize the number of adders that the result from each partial product must pass through [1]. As Virtex-2 and Spartan-III devices use 4-input LUTs, in the first optimization level the partial sum of two products is implemented into one LUT. This procedure optimizes two rows of the array multiplier by using only one row comprised of LUTs. The next level of the optimization compresses two LUT rows from the first optimization level into one using a similar methodology. Taking into account all these specific details of the multiplier implementation into the FPGA leads to a new expression for its switching activity. The methodology used for computing the switching activity is the same as in the Section 4.1. and will not be repeated here.

We give the final expression for the total switching activity when the word-length of the operand y is a power of two as:

$$SW = \sum_{i=1}^{\lceil \log_2 M \rceil} \sum_{j=1}^{l_j} \sum_{k=2^i-1}^{N+2^i-1} (s_{i,j,k} + c_{i,j,k}) \tag{21}$$

In other cases, the counters i and j of the two inner summations have slightly different values due to the parity of the number of LUT rows in each optimization level. This power model has the same features as the model described in the previous section as it is parameterized in terms of operands word lengths and input signal statistics.

4.3 Power Model with Glitching Effects

The switching activity at a node increases with glitching activity. The glitching activity is produced by the different signal delays entering the same logic component. As glitching represents additional activity at the output of the logic gate, it is directly proportional to the transition probabilities of its inputs. Therefore, we will consider that the most significant amount of glitching generated in the multiplier is produced at the LSB regions of its inputs. Hence, the glitching of the two-input multiplier can be expressed as the sum of glitching produced by each cell whose inputs belong to the LSB region of the multiplier (region IV in Fig. 2a). This leads us to the following expression:

$$\begin{aligned} &G = k \cdot ls_x \cdot \sum_{i=1}^{\lceil \log_2(ls_y) \rceil} m_i = k \cdot G' \\ &m_i = \lceil m_{i-1}/2 \rceil \\ &m_1 = \lceil ls_y/2 \rceil \end{aligned} \tag{22}$$

where G is the amount of glitching, ls_x and ls_y are the number of bits in the LSB region of the multiplicand and multiplier respectively and k is an empirically derived constant which represents the average glitching at the output of one LUT. When the position of the MSB-LSB breakpoint is known, the number of LSB bits in the operands is easily obtained. Hence, it is clear that the given model has the same features in terms of parameterization as the previous two. The final model for estimating the power consumption in the presence of glitching and the autocorrelation is given as follows:

$$P = b \cdot (SW + k \cdot G') \tag{23}$$

Constant b can be obtained together with constant k which has been introduced into the expression for glitching. Thus, the complete power estimation procedure consists of the following steps:

1) The number of optimization levels is determined according to the number of bits in the multiplier operand y;
2) The number of MSB bits for the each operand is computed according to (14);

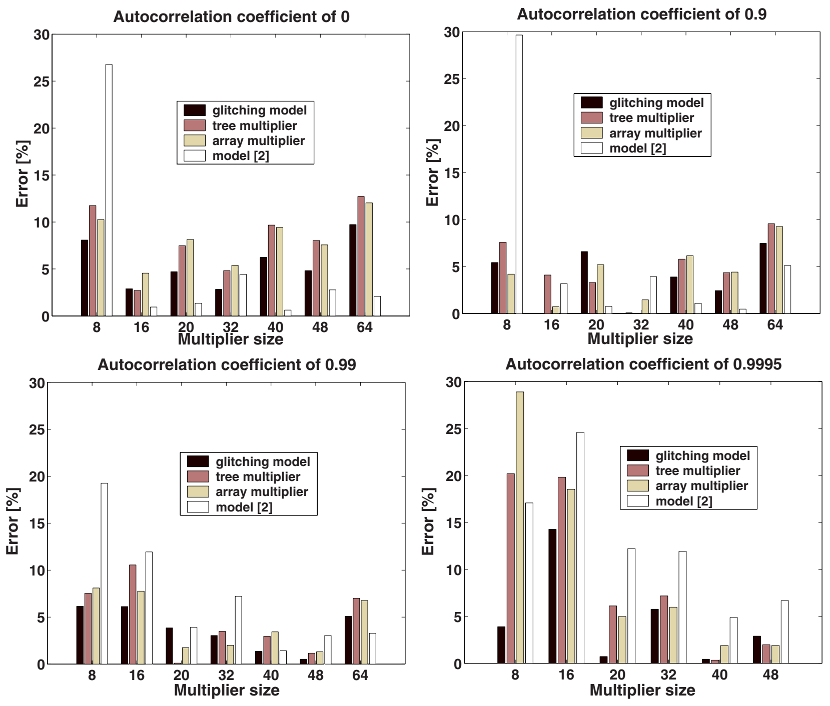


Fig. 3. Error performance for various autocorrelation coefficients considering four different approaches

3) The transition probabilities of all bits in the inputs are set to the values defined by (3);
4) For each optimization level, the switching activity is calculated on the outputs of the partial sum generators using (15)-(18) and is added as in the (21);
5) Glitching presented in (22) is introduced into a final power model;
6) Two low-level power measurements for different multiplier sizes using the same ρ are sufficient in order to determine coefficients b and k. As the factors SW, P and G' are known, the coefficients can be easily obtained.

5 Results

The following experiments have been performed to verify the proposed models of the word-level switching activities of multipliers in FPGAs. Two sets of experiments are performed - one where the word-lengths of both input signals are the same while the size of the component is varied together with the input signal statistics, and the other where one of the input word-lengths is varied while the signal statistics remain fixed. The experiments were performed on multipliers implemented as IP Cores in Xilinx Virtex-2 XC2V2000-5 devices. The design

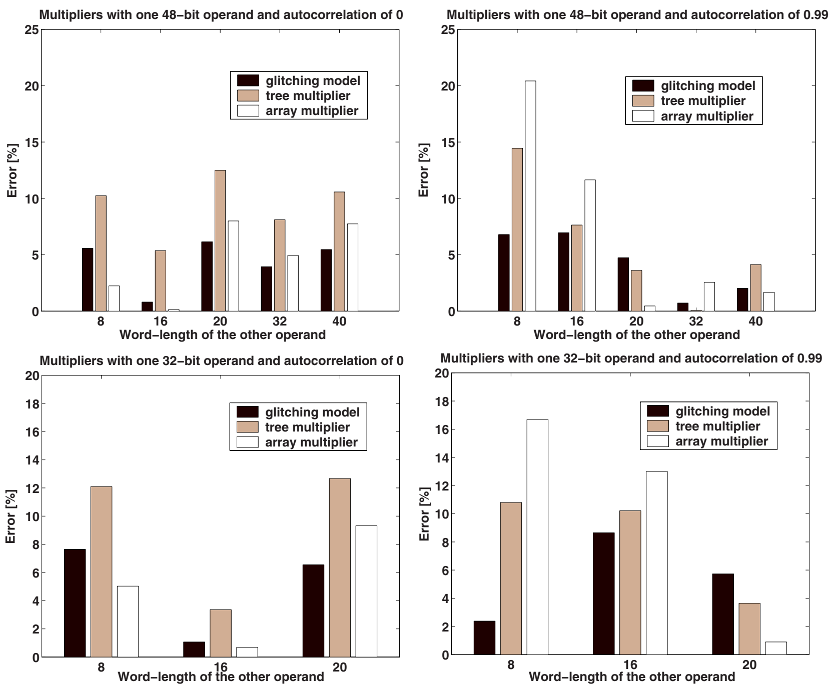


Fig. 4. Error performance for multipliers with operands of different sizes

frequency used in all experiments was set to 100 MHz. Input autocorrelation coefficients were varied from 0 to 0.9995. The signals used for experiments had zero-mean Gaussian distributions. The test-benches for multipliers with input word-lengths smaller than 55 bits were generated by Matlab. The highest number of bits that can be used for a multiplier implemented as a core is 64. As the numbers represented in Matlab have 55 bits at most, we have used functions provided in "The GNU GMP Multiple Precision Arithmetic Library" [4] to generate 64-bit numbers. Each operand signal word was divided into regions according to (4) and (13). All the estimation values are compared to low level power estimation values obtained from the Xilinx tool XPower. As the FPGA vendors are familiar with detailed internal reconfiguration mechanisms of LUTs which are basic cells of FPGA, we believe that the estimates provided by XPower are sufficiently accurate when power consumption of arithmetic blocks implemented in LUTs is considered. Therefore, the error performance of the power models is computed with respect to the values determined by this tool.

The first set of experiments assumes operands with the same word-lengths. Results are compared to those presented in [2], as they also refer to multipliers implemented in LUTs. The results in [2] relate to tables of coefficients used to obtain the power consumption, and a clock frequency is required to perform the appropriate computations. Since such information is not included in [2] we assume a frequency of 25MHz as it is the one providing the results that are closer to the values obtained from XPower. Therefore, four models are taken into consideration: 1) the model for an array multiplier; 2) the model for a row adder tree multiplier; 3) the model that considers both glitching effects and implementation details; and 4) the model described in [2]. The error performance is given in Fig. 3 for seven different multipliers with input word-lengths varying from 8 to 64 bits. It can be observed that in most cases the estimate provided by the model with glitching effects is accurate up to 10% of the value obtained by XPower. Besides, it clearly outperforms the models that do not consider glitching effects. The method described in [2] gives good results when considering large word-lengths, while the error is greater than 20% for 8-bit multipliers. We believe that this is due to the fact that the quadratic model in [2] is too simple for modeling the multipliers power consumption.

The second set of experiments evaluates the error performance of the three proposed models when the multipliers have operands of different sizes. The experiments were performed, first by setting one input to 48 bits and varying the other and then by setting one input to 32 bits and varying the other. Autocorrelation coefficients of 0 and 0.99 have been chosen for these experiments. As already mentioned, to our knowledge, this is the first work that provides estimations for multipliers with different operand sizes, without a need of a different power model construction. The estimates obtained were compared to the low-level estimates provided by XPower as in the previous case. The results are shown in Fig. 4. It can be seen again that for all cases, the model considering glitching effects is accurate within 10% of the value given by XPower.

6 Conclusion

We have presented a high-level analytical approach to estimate logic power consumption of multipliers implemented in FPGAs in the presence of glitching and correlation. The proposed methodology is based on an analytical model for the switching activity of the component and its structural description. We have constructed three different models for estimating power consumption using the proposed approach. The first model considers a standard array multiplier. The second one includes a structural model of the implementation of the component into an FPGA. The third model is derived from the second, by adding the glitching effects. All three models can estimate the power consumption for any given clock frequency, signal statistics and operands' word-lengths. We have shown that the accuracy of the proposed models is within 10% of low-level power estimates given by the tool XPower over a wide range of these parameters. The model that accounts for the glitching activity clearly has advantage compared to the models that do not consider this effect.

Acknowledgement. This work was supported in part by the Spanish Ministry of Education and Science under project TEC2006-13067-C03-03/TCM.

References

1. Andraka Consulting Group: Multiplication in FPGAs, available at http://www.fpga-guru.com/multipli.htm
2. Clarke, J., A., Gaffar, A., A., and Constantinides, G., A.: Parameterized logic power consumption models for FPGA-based arithmetic, Proc. FPL (2005) 626-629
3. Chou, T., Roy, K., and Prasad, S.: Estimation of Circuit Activity Considering Signal Correlations and Simulteneous Switching, Proc. of the 1994 IEEE/ACM Int. Conference on Computer-aided Design, (1994) 300-303
4. GNU MP library available at http://www.swox.com/gmp/
5. Gupta, S., and Najm, F., N.: Power Modeling for High Level Power Estimation, IEEE Trans. VLSI Syst., vol.8, (Feb. 2000) 18-29
6. Landman P., and Rabaey, J.: Architectural Power Analysis: The dual bit type method, IEEE Trans. On VLSI Systems, vol. 3, no.2, (1995) 173-187
7. Ramprasad, S., Shanbhag, N., R., and Hajj, I., N.: Analytical Estimation of Signal Transition Activity from Word-Level Statistics, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, vol. 16, no. 7, (July 1997) 718-733
8. Satyanarayana, J., H., and Parhi, K., K.: Theoretical Analysis of Word-Level Switching Activity in the Presence of Glitching and Correlation, IEEE Trans. On VLSI Systems, vol. 8, no. 2, (Apr. 2000) 148-159
9. Shang, L., and Jha, N., K.: High-level Power Modeling of CPLDs and FPGAs, in Proc. Of the Int. Conf. on Comp. Design. IEEE Computer Society, (2001) 46-53
10. Xilinx Inc. www.xilinx.com
11. Zhong, L., and Jha, N., K.: Interconnect-aware High-level Synthesis for Low Power, IEEE/ACM International Conf. on CAD, (Nov. 2002) 110-117

Multiplication over $\mathbb{F}_{p^m}$ on FPGA: A Survey[⋆]

Jean-Luc Beuchat, Takanori Miyoshi, Yoshihito Oyama, and Eiji Okamoto

Laboratory of Cryptography and Information Security
University of Tsukuba
1-1-1 Tennodai, Tsukuba
Ibaraki, 305-8573, Japan

Abstract. This paper aims at comparing multiplication algorithms over $\mathbb{F}_{p^m}$ on FPGA. Contrary to previous surveys providing the reader with an estimate of both area and delay in terms of XOR gates, we discuss place-and-route results which point out that the choice of an algorithm depends on the irreducible polynomial and on some architectural parameters. We designed a VHDL code generator to easily study a wide range of algorithms and parameters.

1 Introduction

Multiplication over $\mathbb{F}_{p^m}$ is a fundamental calculation in elliptic curve cryptography (ECC), pairing-based cryptography, and implementation of error-correcting codes. Field programmable gate arrays (FPGAs) have become more and more popular to implement hardware accelerators for such algorithms. This paper aims at comparing several multiplication algorithms in order to help engineers and researchers in selecting the most appropriate architecture for a given application on FPGAs.

Some papers survey multiplication over $\mathbb{F}_{p^m}$ from a theoretical point of view and provide the reader with an estimate of both area and delay of hardware operators in terms of XOR gates (see for instance [4, 6]). However, such results do not include routing delays and seem difficult to apply to FPGAs, whose architecture is usually based on look-up tables (LUTs). We propose here a comparison based on place-and-route results on Xilinx Spartan-3 FPGAs. In order to easily modify the irreducible polynomial and some architectural parameters (e.g. pipeline stages), we designed a tool which generates a structural VHDL description, as well as scripts to automatically place-and-route the operator and collect significant results.

There are several ways to encode elements of an extension field. In this paper, we will only consider the well-known polynomial representation. Let $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$ be a degree-m monic irreducible polynomial over $\mathbb{F}_p$, where p is a prime. Then, $\mathbb{F}_{p^m} = \mathbb{F}_p[x]/f(x)$, and an element $a(x) \in \mathbb{F}_{p^m}$ is a degree-$(m-1)$ polynomial with coefficients in $\mathbb{F}_p$: $a(x) = a_{m-1}x^{m-1} + \ldots + a_1x +$

⋆ This work was supported by the New Energy and Industrial Technology Development Organization (NEDO), Japan.

a_0. Three families of algorithms allow one to compute $a(x)b(x) \bmod f(x)$ (where $a(x)$ and $b(x)$ belong to $\mathbb{F}_{p^m}$). In parallel-serial schemes, a single coefficient of the multiplier $a(x)$ is processed at each step. This leads to small operands performing a multiplication in m steps. Parallel multipliers compute a degree-$(2m-2)$ polynomial and carry out a final modular reduction. They achieve a higher throughput at the price of a larger circuit area. Song and Parhi introduced array multipliers as a trade-off between computation time and circuit area [12]. The idea consists in processing D coefficients of the multiplier at each step. The parameter D is sometimes referred to as *digit size* and parallel-serial schemes can be considered as a special case with $D = 1$.

This paper focuses on array multipliers, which are often preferred to parallel architectures for hardware implementation of ECC or pairing-based cryptography [7, 5, 10, 11, 3]. Depending on the order in which coefficients of $a(x)$ are processed, multiplication modulo $f(x)$ can be performed according to two schemes: most-significant element (MSE) first and least-significant element (LSE) first. Sections 2 and 3 are respectively devoted to the study of MSE first and LSE first algorithms. We discuss experiment results in Section 4.

2 Most-Significant Element (MSE) First Algorithms

The celebrated Horner's rule allows the design of parallel-serial algorithms starting with the most-significant element (MSE). The simplest scheme requires m iterations to compute $p(x) = (a(x)b(x)) \bmod f(x)$. Recall that the equation

$$a(x)b(x) = ((\ldots(a_{m-1}b(x)x + a_{m-2}b(x))x + \ldots)x + a_1b(x))x + a_0b(x)$$

can easily be computed recursively. A register stores the intermediate result $p(x)$ which is initially set to zero. At step i, $m-1 \geq i \geq 0$, we compute $p(x) = xp(x)+a_ib(x)$. Note that $a_ib(x)$ is a degree-$(m-1)$ polynomial. Thus, it suffices to reduce $xp(x)$ at each step to perform a modulo $f(x)$ multiplication[1] (Algorithm 1 and Figure 1a).

Algorithm 1. MSE multiplication over $\mathbb{F}_{p^m}$.

```
Require: A degree-$m$ monic polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$ and
    two degree-$(m-1)$ polynomials $a(x)$ and $b(x)$.
Ensure: $p(x) = a(x)b(x) \bmod f(x)$
 1: $p(x) \leftarrow 0$;
 2: for $i$ from $m-1$ downto 0 do
 3:     $p(x) \leftarrow a_ib(x) + (xp(x)) \bmod f(x)$;
 4: end for
```

Several researchers proposed to process D coefficients of operand $a(x)$ at each clock cycle in order to reduce the computation time. Shu *et al.* introduced for

[1] Another implementation of this algorithm was proposed in [8]. Since it involves more hardware and a more complex control, we will not consider it in our experiments.

instance an algorithm which performs multiplication over $\mathbb{F}_{p^m}$ in $\lceil m/D \rceil$ clock cycles [11]. At step i, they compute a sum of D partial products reduced modulo $f(x)$:

$$t(x) = \sum_{j=0}^{D-1} a_{Di+j}(x^j b(x) \bmod f(x)),$$

where $t(x)$ is a degree-$(m-1)$ polynomial. A second degree-$(m-1)$ polynomial $p(x)$ accumulates these partial products. Algorithm 2 summarizes this multiplication scheme (see also Figure 1b).

Algorithm 2. MSE multiplication over $\mathbb{F}_{p^m}$ [11].

```
Require: A degree-$m$ monic polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1 x + f_0$ and
    two degree-$(m-1)$ polynomials $a(x)$ and $b(x)$. The algorithm requires a degree-
    $(m-1)$ polynomial $t(x)$ for intermediate computations.
Ensure: $p(x) = a(x)b(x) \bmod f(x)$
 1: $p(x) \leftarrow 0$;
 2: for $i$ from $\lceil m/D \rceil - 1$ downto 0 do
 3:     $t(x) \leftarrow \sum_{j=0}^{D-1} a_{Di+j}(x^j b(x) \bmod f(x))$;
 4:     $p(x) \leftarrow t(x) + (x^D p(x) \bmod f(x))$;
 5: end for
```

Song and Parhi suggested to compute at each step a degree-$(m + D - 2)$ polynomial $t(x)$ which is the sum of D partial products [12]:

$$t(x) = \sum_{j=0}^{D-1} a_{Di+j} x^j b(x). \tag{1}$$

A degree-$(m+D-1)$ polynomial $s(x)$, updated according to Horner's rule, allows to accumulate these partial products:

$$s(x) = t(x) + x^D(s(x) \bmod f(x)).$$

After $\lceil m/D \rceil$ iterations, $s(x)$ is a degree-$(m+D-1)$ polynomial congruent with $a(x)b(x)$ modulo $f(x)$. Song and Parhi included specific hardware to perform a modular correction [12]. A second approach, which does not require extra resources, was proposed in [3]. The idea consists in performing an additional iteration with $a_{-j} = 0$, $1 \le j \le D$. Since $t(x)$ is now equal to zero, we obtain $s(x) = x^D((a(x)b(x)) \bmod f(x))$ and

$$p(x) = s(x)/x^D = a(x)b(x) \bmod f(x).$$

Therefore, the m most-significant coefficients of $s(x)$ give the result (Algorithm 3 and Figure 1c). Note that a single modulo $f(x)$ reduction is needed, whereas Algorithm 2 requires D modular operations. Furthermore, from purely theoretical point of view, the critical path is shorter than the one of an operator based on Algorithm 2. Assume for instance that $p = 3$ and that elements of $\mathbb{F}_3$ are encoded with two bits. Addition and multiplication of two elements can thus be performed by means of four-input tables. On FPGAs featuring four-input LUTs, generation and addition of D partial products require $\lceil \log_2 D \rceil + 1$ stages of LUTs. The algorithm by Shu *et al.* involves a number of additional stages which depends on D and $f(x)$. However, since partial products are not reduced in Algorithm 3, their addition requires more hardware resources (degree up to $m + D - 2$ instead of $m - 1$). Our experiment results suggest that the choice between Algorithms 2 and 3 depends on D and $f(x)$ (Section 4).

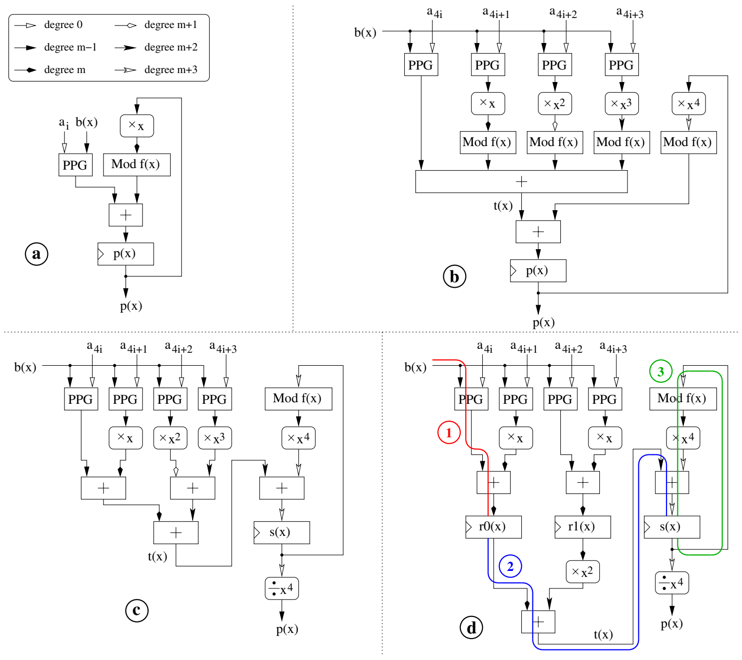


Fig. 1. MSE multiplication over $\mathbb{F}_{p^m}$. (a) Horner's rule. (b) Algorithm proposed by Shu *et al.* [11] ($D = 4$). (c) Algorithm proposed by Song and Parhi [12, 3] ($D = 4$). (d) Algorithm proposed by Song and Parhi with a pipeline stage ($D = 4$ and $R = 2$). Boxes with rounded corners involve only wiring.

Pipelining the computation of $t(x)$ (Equation (1)) sometimes allows one to shorten the critical path of a multiplier based on Algorithm 3 (Figure 1d). The depth of the pipeline stage is specified by a parameter R which indicates the number of registers inserted. Thus, $s(x)$ is the sum of $(R + 1)$ polynomials. Since the irreducible polynomial $f(x)$ is known at design time, all degree-$(m-1)$

Algorithm 3. MSE multiplication over $\mathbb{F}_{p^m}$ [3].

```
Require: A degree-$m$ monic polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$ and two degree-$(m-1)$ polynomials $a(x)$ and $b(x)$. We assume that $a_{-j} = 0$, $1 \leq j \leq D$. The algorithm requires a degree-$(m+D-1)$ polynomial $s(x)$ as well as a degree-$(m+D-2)$ polynomial $t(x)$ for intermediate computations.
Ensure: $p(x) = a(x)b(x) \bmod f(x)$
1: $s(x) \leftarrow 0$;
2: for $i$ from $\lceil m/D \rceil - 1$ downto $-1$ do
3:     $t(x) \leftarrow \sum_{j=0}^{D-1} a_{Di+j}x^j b(x)$;
4:     $s(x) \leftarrow t(x) + x^D \cdot (s(x) \bmod f(x))$;
5: end for
6: $p(x) \leftarrow s(x)/x^D$;
```

polynomials $x^i \bmod f(x)$, $m \leq i \leq m+D-1$, can be precomputed and the modulo $f(x)$ reduction of $s(x)$ is defined as follows:

$$s(x) \bmod f(x) = \sum_{i=0}^{m-1} s_i x^i + \sum_{i=m}^{m+D-1} s_i (x^i \bmod f(x)). \tag{2}$$

Both area and delay of a circuit implementing Equation (2) depend on D, $f(x)$, and FPGA family. Consider for example an FPGA embedding four-input LUTs, $\mathbb{F}_{3^{11}} = \mathbb{F}_3[x]/(x^{11} + 2x^8 + 1)$, and $D = 3$. Since $x^{11} \bmod f(x) = x^8 + 2$, $x^{12} \bmod f(x) = x^9 + 2x$, and $x^{13} \bmod f(x) = x^{10} + 2x^2$, we obtain

$$\sum_{i=11}^{13} s_i \cdot (x^i \bmod f(x)) = s_{13}x^{10} + s_{12}x^9 + s_{11}x^8 + 2s_{13}x^2 + 2s_{12}x + 2s_{11}.$$

Recall that, if each element $s_i \in \mathbb{F}_3$ is represented in radix-2 by two bits y_1 and y_0 (i.e. $s_i = 2y_1 + y_0$), multiplication by two is achieved by swapping y_0 and y_1. Therefore, the above equation involves only wiring and Equation (2) requires a single addition over $\mathbb{F}_{3^m}$. However, if we select $f(x) = x^{11} + x^{10} + x^8 + 2x^7 + x + 1$, we have:

$$\begin{aligned} x^{11} &= 2x^{10} + 2x^8 + x^7 + 2x + 2, \\ x^{12} &= x^{10} + 2x^9 + 2x^8 + 2x^7 + 2x^2 + 1, \\ x^{13} &= x^{10} + 2x^9 + x^8 + x^7 + 2x^3 + 2, \end{aligned}$$

and Equation (2) requires $D = 3$ adders over $\mathbb{F}_3$ to compute the degree-$(m-1)$ coefficient of the result. Thus, depending on $f(x)$, the number of operands ranges from 2 to $D+1$. Since four-input tables efficiently carry out addition over $\mathbb{F}_3$, the critical path of the circuit implementing Equation (2) contains up to $\lceil \log_2(D+1) \rceil$ LUTs. Consider now the operator described by Figure 1d. We need to study three paths to decide whether pipelining is efficient or not:

1. In characteristic three, computing a coefficient of a partial product requires a single LUT. Therefore, the longest path from input to pipeline stage includes $\lceil \log_2 \lceil D/R \rceil \rceil + 1$ LUTs.

2. Since computation of $s(x)$ involves $R+1$ operands, the path between the pipeline stage and the accumulator contains $\lceil \log_2(R+1) \rceil$ LUTs.
3. The accumulation loop includes a modulo $f(x)$ reduction and a single addition. The number of LUTs is therefore bounded by $\lceil \log_2(D+1) \rceil + 1$.

Let α denote the number of additions required to perform a modulo $f(x)$ reduction. Then, pipelining the computation of $t(x)$ shortens the critical path if:

$$\lceil \log_2 \alpha \rceil \leq \max(\lceil \log_2 \lceil D/R \rceil \rceil + 1, \lceil \log_2(R+1) \rceil) - 1. \tag{3}$$

Note that this equation does not include any information about routing, and practical results may differ. In our example, $D = 4$ and $R = 2$, thus $\lceil \log_2 \lceil D/R \rceil \rceil + 1 = \lceil \log_2(R+1) \rceil = 2$, and pipelining is therefore of interest only if computation of Equation (2) requires a single addition.

3 Least-Significant Element (LSE) First Algorithms

Least-significant element (LSE) first algorithms mainly consist of two loops. The first one computes the product $p(x)$, while the second one generates successive values of $(x^{iD}b(x)) \bmod f(x)$, $0 \leq i < \lceil m/D \rceil$ (Algorithm 4 and Figure 2a describe the case where $D = 1$). This family requires more hardware resources than MSE first algorithms because of the register, multiplexer, and modulo $f(x)$ reduction involved in the computation of $(x^{iD}b(x)) \bmod f(x)$. Nevertheless, it offers an effective way to compute $(a(x)b(x) + c(x)) \bmod f(x)$: it suffices to initialize register $p(x)$ with $c(x)$ instead of 0 in Algorithm 4. Note that this operation does not involve additional hardware (the control is however a little bit more complex): we can for instance load $c(x)$ in register $q(x)$ and multiply this polynomial by $a_i = 1$. MSE first schemes require an additional shift register to compute $(a(x)b(x)+c(x)) \bmod f(x)$: only D coefficients of $c(x)$ can be added at each step.

Algorithm 4. LSE multiplication over $\mathbb{F}_{p^m}$.

```
Require: A degree-$m$ monic polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1x + f_0$ and
    two degree-$(m-1)$ polynomials $a(x)$ and $b(x)$.
Ensure: $s(x) = a(x)b(x) \bmod f(x)$
1: $p(x) \leftarrow 0$; $q(x) \leftarrow b(x)$;
2: for $i$ in 0 to $m-1$ do
3:     $p(x) \leftarrow p(x) + a_iq(x)$;
4:     $q(x) \leftarrow xq(x) \bmod f(x)$;
5: end for
```

Bertoni *et al.* modified Algorithm 4 so that it processes D coefficients at each iteration [2]. This algorithm can be considered as a transposition of Song and Parhi's proposal to LSE schemes: at each iteration, the sum of D partial products forms a degree-$(m+D-2)$ polynomial $t(x)$ which is added to the intermediate result $s(x)$. After $\lceil m/D \rceil$ iterations, $s(x)$ is a degree-$(m+D+2)$ polynomial congruent with $a(x)b(x)$ modulo $f(x)$. A final modulo $f(x)$ reduction is therefore requested to get $p(x) = a(x)b(x) \bmod f(x)$. However, the delay of this operation may sometimes be added to the critical path of an operator connected to the output of such a multiplier. An additional output register avoids this drawback. Another solution consists in reducing $s(x)$ at each step and in performing an additional iteration with $a_j = 0$, $m \le j \le m+D-1$ (Algorithm 5 and Figure 2b). Since $t(x)$ is now equal to zero, $s(x) = (a(x)b(x)) \mod f(x)$ and it suffices to consider the m least-significant coefficients of $s(x)$ to get the result ($p(x) = s(x) \bmod x^m$).

Algorithm 5. LSE multiplication over $\mathbb{F}_{p^m}$ [2].

```
Require: A degree-$m$ monic polynomial $f(x) = x^m + f_{m-1}x^{m-1} + \ldots + f_1 x + f_0$ and
    two degree-$(m-1)$ polynomials $a(x)$ and $b(x)$. We assume that $a_j = 0$, $m \le j \le$
    $m + D - 1$.
Ensure: $p(x) = a(x)b(x) \bmod f(x)$
1: $p(x) \leftarrow 0$; $q(x) \leftarrow b(x)$;
2: for $i$ in 0 to $\lceil m/D \rceil$ do
3:     $t(x) \leftarrow \sum_{j=0}^{D-1} a_{Di+j} x^j q(x)$;
4:     $s(x) \leftarrow s(x) \bmod f(x) + t(x)$;
5:     $q(x) \leftarrow (x^D q(x)) \bmod f(x)$;
6: end for
7: $p(x) \leftarrow s(x) \bmod x^m$;
```

Kumar *et al.* further improved the algorithm described in [2] by introducing double accumulator multiplier (DAM) and n-accumulator multiplier (NAM) architectures [9, 6] (Figure 2c). They reduce the critical path by using $R \ge 2$ accumulators to carry out the sum of D partial products. At the end a multiplication, an adder tree combines these intermediate results and a final modulo $f(x)$ operation is performed. An output register should therefore be included in order to avoid adding this combinatorial logic to the critical path of another operator. Another solution consists in pipelining the computation of $t(x)$ (Figure 2d). This operator has the following advantages over DAM and NAM schemes: i) The modulo $f(x)$ reduction is in the accumulation loop. ii) It requires the addition of $(D+1)$ operands, whereas DAM and NAM architectures compute a sum of $(D+R)$ operands. Thus, a pipelined architecture should lead to smaller circuits than DAM or NAM.

Note that Equation (3) can also be applied to study the efficiency of pipelining. The only difference is that we have to consider the loop computing $x^{iD}b(x)$ mod $f(x)$ (Figure 2). If $p = 3$, the multiplexer involves five-input functions. However, several FPGAs embed dedicated multiplexers to efficiently combine two four-input LUTs. The delay of the multiplexer on Figure 2 is therefore equivalent

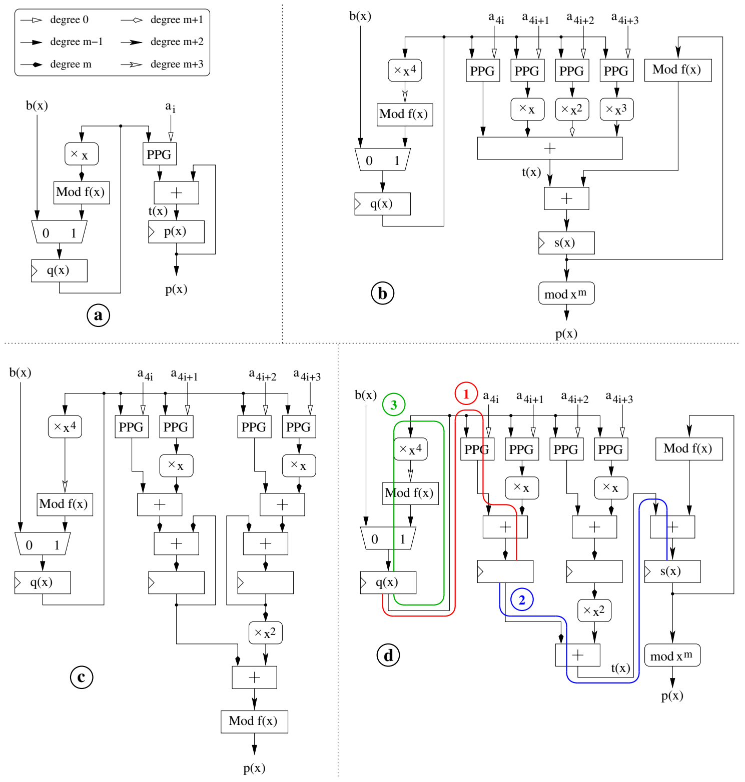


Fig. 2. LSE multiplication over $\mathbb{F}_{p^m}$. (a) Basic algorithm. (b) Modification of the algorithm proposed by Bertoni *et al.* [2] ($D = 4$). (c) Algorithm proposed by Kumar *et al.* [9] ($D = 4$ and $R = 2$). (d) Modification of the algorithm proposed by Kumar *et al.* [9] ($D = 4$ and $R = 2$). Boxes with rounded corners involve only wiring.

to the one of an addition over $\mathbb{F}_3$, and the longest path of this loop includes $\lceil \log_2(D + 1)\rceil + 1$ LUTs.

4 Results

In order to easily compare the algorithms described in previous sections, we wrote a VHDL code generator whose inputs are an irreducible polynomial, as well as the desired algorithm and its parameters. Our tool returns a structural VHDL description of the multiplier and some scripts to automatically place-and-route the design and extract its area and critical path. In the following, we discuss results of a series of experiments involving a Spartan-3 XC3S1500 device and ISE WebPACK 8.2.03i.

We considered the following NIST recommended polynomials for studying multiplication over $\mathbb{F}_{2^m}$: $\mathbb{F}_{2^{163}} = \mathbb{F}_2[x]/(x^{163} + x^7 + x^6 + x^3 + 1)$,

Table 1. Multiplication over $\mathbb{F}_{2^m}$ based on the Horner's rule (Algorithm 1)

$\mathbb{F}_{2^m}$	**Algorithm**	**Area [slices]**	**Delay [ns]**	**Multiplication time [ns]**	**Throughput [Mbits/s]**	**Throughput/ slice [Mbits/s]**
$\mathbb{F}_{2^{163}}$	Algorithm 1	85	4.9	798.7	204.1	2.4
	Algorithm 4	179	4.9	798.7	204.1	1.1
$\mathbb{F}_{2^{233}}$	Algorithm 1	122	4.9	1141.7	204.1	1.7
	Algorithm 4	270	5.3	1234.9	188.7	0.7
$\mathbb{F}_{2^{283}}$	Algorithm 1	146	5.3	1499.9	188.7	1.3
	Algorithm 4	322	4.9	1386.7	204.1	0.6
$\mathbb{F}_{2^{409}}$	Algorithm 1	216	6.2	2535.8	161.3	0.7
	Algorithm 4	473	4.9	2004.1	204.1	0.4
$\mathbb{F}_{2^{571}}$	Algorithm 1	311	4.9	2797.9	204.1	0.7
	Algorithm 4	643	4.9	2797.9	204.1	0.3

$\mathbb{F}_{2^{233}} = \mathbb{F}_2[x]/(x^{233} + x^{74} + 1)$, $\mathbb{F}_{2^{283}} = \mathbb{F}_2[x]/(x^{283} + x^{12} + x^7 + x^5 + 1)$, $\mathbb{F}_{2^{409}} = \mathbb{F}_2[x]/(x^{409} + x^{87} + 1)$, and $\mathbb{F}_{2^{571}} = \mathbb{F}_2[x]/(x^{571} + x^{10} + x^5 + x^2 + 1)$. Our first experiment aimed at comparing Horner's rule (Algorithm 1) and the basic LSE first scheme (Algorithm 4). Whereas both algorithms have almost the same critical path, MSE first approach requires less hardware and offers therefore a higher throughput per slice (Table 1).

We then considered several values of D (number of coefficients of $a(x)$ processed at each clock cycle) for Algorithms 2, 3, and 5. Table 2 summarizes the multiplication time measured after placing-and-routing the operators. Recall that Algorithms 2 and 3 respectively require $\lceil m/D \rceil$ and $\lceil m/D \rceil + 1$ iterations. This additional clock cycle explains why Algorithms 2 is often slightly faster. Table 3 reports the throughput per slice of these three algorithms. We conclude from this experiment that MSE first schemes (Algorithms 2 and 3) are almost always more efficient than LSE first algorithm. However, it seems that the choice between these two MSE architectures depends on the irreducible polynomial and parameter D. Note that we obtained slightly different results with an older version of ISE WebPACK. Design tools should also be considered as a parameter when selecting an algorithm. In order to increase the throughput per slice of Algorithm 1, the parameter D should be smaller than 16.

Our next experiment aimed at studying pipelining, DAM, and NAM approaches (Table 4). We selected $D = 32$ and $R = 4$, and generated VHDL descriptions of one MSE first operator (Figure 1d) and two LSE first operators (Figure 2c and 2d). From a theoretical point of view, this choice of parameters should significantly improve the throughput of multiplication over $\mathbb{F}_{2^{283}}$ and $\mathbb{F}_{2^{409}}$: since each modulo $f(x)$ reduction involves a single addition, the multi-operand addition (sum of D partial products) should include the critical path of the circuit. Although experiments carried out for $\mathbb{F}_{2^{233}}$ confirm these assumptions, results obtained for $\mathbb{F}_{2^{409}}$ are somewhat surprising: adding a pipeline stage

Table 2. Multiplication over $\mathbb{F}_{2^m}$ (1). Computation time (in nanoseconds) of Algorithms 2, 3, and 5 on a Spartan-3 XC3S1500 FPGA. Bold numbers highlight the fastest algorithm for each experiment setup.

$\mathbb{F}_{2^m}$	**Algorithm**	**D = 4**	**D = 8**	**D = 12**	**D = 16**	**D = 20**	**D = 24**	**D = 28**	**D = 32**
$\mathbb{F}_{2^{163}}$	Algorithm 2	**203.7**	115.8	79.1	**66.7**	**58.8**	**54.7**	**45.7**	**46.2**
	Algorithm 3	294.4	116.8	**76.2**	72.7	79.1	60.3	51.4	53.6
	Algorithm 5	209.2	**109.7**	89.4	69.3	64.0	58.0	57.4	72.7
$\mathbb{F}_{2^{233}}$	Algorithm 2	**294.4**	**154.2**	119.8	**87.4**	**80.3**	73.8	**93.1**	**95.3**
	Algorithm 3	298.7	155.7	111.7	89.6	81.6	**72.5**	130.4	108.8
	Algorithm 5	345.8	222.4	**111.5**	112.6	95.7	113.6	110.3	104.8
$\mathbb{F}_{2^{283}}$	Algorithm 2	474.6	**201.4**	152.6	138.0	**103.2**	122.0	**121.2**	**94.2**
	Algorithm 3	423.1	303.5	140.7	**119.2**	108.6	**87.6**	123.3	115.3
	Algorithm 5	**363.4**	253.4	**132.1**	161.0	106.8	130.1	131.5	120.9
$\mathbb{F}_{2^{409}}$	Algorithm 2	**518.9**	**272.7**	210.9	**157.8**	201.3	**185.3**	173.4	153.5
	Algorithm 3	535.0	323.7	207.4	177.3	**165.8**	194.8	**168.2**	**150.1**
	Algorithm 5	515.9	348.4	**202.2**	175.6	167.8	196.8	185.0	157.6
$\mathbb{F}_{2^{571}}$	Algorithm 2	**713.9**	444.7	432.9	338.5	**293.9**	267.8	**247.1**	295.8
	Algorithm 3	878.8	**428.1**	**325.9**	**232.9**	347.2	**265.8**	261.8	226.4
	Algorithm 5	779.2	502.4	334.1	307.8	354.4	271.0	251.5	**224.7**

Table 3. Multiplication over $\mathbb{F}_{2^m}$ (2). Throughput/slice [Mbits/s] of Algorithms 2, 3, and 5 on a Spartan-3 XC3S1500 FPGA. Bold numbers indicate a throughput/slice greater than or equal to the one of Horner's rule.

$\mathbb{F}_{2^m}$	**Algorithm**	**D = 4**	**D = 8**	**D = 12**	**D = 16**	**D = 20**	**D = 24**	**D = 28**	**D = 32**
$\mathbb{F}_{2^{163}}$	Algorithm 2	**2.9**	**2.5**	2.0	1.9	1.4	1.2	1.2	1.1
	Algorithm 3	1.9	**2.4**	**2.8**	2.1	1.1	1.2	1.2	1.0
	Algorithm 5	2.1	2.2	2.0	2.0	1.3	1.2	1.0	0.7
$\mathbb{F}_{2^{233}}$	Algorithm 2	**2.1**	**1.9**	**1.7**	**1.7**	1.1	1.0	0.7	0.6
	Algorithm 3	**1.8**	**1.9**	**1.9**	**1.7**	1.1	1.0	0.5	0.5
	Algorithm 5	1.3	1.1	**1.7**	1.3	0.9	0.6	0.5	0.5
$\mathbb{F}_{2^{283}}$	Algorithm 2	**1.3**	**1.4**	1.1	1.0	0.8	0.6	0.5	0.5
	Algorithm 3	**1.3**	**1.0**	**1.5**	**1.3**	0.8	0.8	0.5	0.5
	Algorithm 5	1.2	1.0	**1.4**	0.9	0.8	0.5	0.5	0.4
$\mathbb{F}_{2^{409}}$	Algorithm 2	**1.2**	**1.1**	**1.0**	**0.9**	0.4	0.4	0.4	0.3
	Algorithm 3	**1.0**	**0.9**	**1.0**	**0.8**	0.5	0.4	0.4	0.4
	Algorithm 5	**0.8**	**0.7**	**0.9**	**0.8**	0.5	0.3	0.3	0.3
$\mathbb{F}_{2^{571}}$	Algorithm 2	**0.8**	0.6	0.4	0.4	0.3	0.3	0.2	0.2
	Algorithm 3	0.6	0.6	0.6	0.6	0.3	0.3	0.2	0.2
	Algorithm 5	0.6	0.5	0.5	0.4	0.2	0.3	0.2	0.2

Table 4. Multiplication over $\mathbb{F}_{2^m}$ (3). Comparison of architectures including pipeline stages for $D = 32$ and $R = 4$.

$\mathbb{F}_{2^m}$	Operator	Area [slices]	Delay [ns]	Multiplication time [ns]	Throughput [Mbits/s]	Throughput/ slice [Mbits/s]
$\mathbb{F}_{2^{163}}$	Figure 1d	2158	5.9	47.2	3453.4	1.6
	Figure 2c	2637	6.6	52.8	3087.1	1.2
	Figure 2d	2269	5.9	47.2	3453.4	1.5
$\mathbb{F}_{2^{233}}$	Figure 1d	3458	5.8	58.0	4017.2	1.2
	Figure 2c	3917	7.1	71.0	3281.7	0.8
	Figure 2d	3504	6.2	62.0	3758.1	1.1
$\mathbb{F}_{2^{283}}$	Figure 1d	3649	6.7	73.7	3839.9	1.1
	Figure 2c	4163	7.8	85.8	3298.4	0.8
	Figure 2d	3754	7.7	84.7	3341.2	0.9
$\mathbb{F}_{2^{409}}$	Figure 1d	5406	10.2	153.0	2673.2	0.5
	Figure 2c	6061	9.8	147.0	2782.3	0.5
	Figure 2d	5489	10.6	159.0	2572.3	0.5
$\mathbb{F}_{2^{571}}$	Figure 1d	8329	10.3	206.0	2771.8	0.3
	Figure 2c	10053	10.7	217.0	2631.3	0.3
	Figure 2d	8730	10.1	202.0	2826.7	0.3

does not shorten the critical path. The problem is that our theoretical framework does not include routing delays. This suggest the design of incremental code generators which take advantage of timing information in order to automatically pipeline an arithmetic operator. We also obtained an interesting result with multiplication over $\mathbb{F}_{2^{163}}$. Operators depicted by Figures 1d and 2c have roughly the same throughput as MSE first or LSE first multiplication without pipelining. However, the throughput per slice is improved. It seems that hardware design tools were able to meet timing constraints in both cases. However, adding a pipeline stage made this task easier, thus allowing to build a smaller operator. Table 4 also indicates that pipelining an operator leads to a smaller circuit than DAM or NAM approaches.

We also conducted a series of experiments in $\mathbb{F}_{3^m}$. They involved four pairing-friendly polynomials[2] proposed in [1]: $\mathbb{F}_{3^{97}} = \mathbb{F}_3[x]/(x^{97} + x^{16} - 1)$, $\mathbb{F}_{3^{167}} = \mathbb{F}_3[x]/(x^{167} + x^{92} - 1)$, $\mathbb{F}_{3^{193}} = \mathbb{F}_3[x]/(x^{193} + x^{64} - 1)$, and $\mathbb{F}_{3^{239}} = \mathbb{F}_3[x]/(x^{239} + x^{26} - 1)$. Results are very similar to those obtained in characteristic two. MSE first schemes are almost always more attractive than LSE first algorithms and the choice between Algorithms 2 and 3 depends again on $f(x)$ and D.

5 Conclusion

We studied eight operators performing multiplication over $\mathbb{F}_{p^m}$. Our approach consisted in writing a VHDL code generator in order to investigate a wide

[2] With such polynomials, the cost of computing cube roots over $\mathbb{F}_{3^m}$ is $O(m)$.

solution space. Our results indicate that the choice of an algorithm depends on several parameters, like the irreducible polynomial or the number of coefficients processed at each clock cycle. We plan to improve our tool to automatically pipeline an operator. This task should be based on theoretical results (e.g. Equations 3) and place-and-route information including wire delays. We could for instance generate a VHDL description of a modulo $f(x)$ reduction or an addition tree, place-and-route this operator, and extract timing information, which would then allow one to optimize the architecture of the multiplier.

References

1. P. S. L. M. Barreto. A note on efficient computation of cube roots in characteristic 3. Cryptology ePrint Archive, Report 2004/305, 2004.
2. G. Bertoni, J. Guajardo, S. Kumar, G. Orlando, C. Paar, and T. Wollinger. Efficient GF(p^m) arithmetic architectures for cryptographic applications. In M. Joye, editor, *Topics in Cryptology – CT-RSA 2003*, number 2612 in Lecture Notes in Computer Science, pages 158–175. Springer, 2004.
3. J.-L. Beuchat, M. Shirase, T. Takagi, and E. Okamoto. An algorithm for the η_T pairing calculation in characteristic three and its hardware implementation. Cryptology ePrint Archive, Report 2006/327, 2006.
4. S. E. Erdem, T. Yamk, and Ç. K. Koç. Polynomial basis multiplication over GF(2^m). *Acta Applicandae Mathematicae*, 93(1–3):33–55, September 2006.
5. P. Grabher and D. Page. Hardware acceleration of the Tate Pairing in characteristic three. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, number 3659 in Lecture Notes in Computer Science, pages 398–411. Springer, 2005.
6. J. Guajardo, T. Güneysu, S. Kumar, C. Paar, and J. Pelzl. Efficient hardware implementation of finite fields with applications to cryptography. *Acta Applicandae Mathematicae*, 93(1–3):75–118, September 2006.
7. T. Kerins, W. P. Marnane, E. M. Popovici, and P.S.L.M. Barreto. Efficient hardware for the Tate Pairing calculation in characteristic three. In J. R. Rao and B. Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, number 3659 in Lecture Notes in Computer Science, pages 412–426. Springer, 2005.
8. T. Kerins, E. Popovici, and W. Marnane. Algorithms and architectures for use in FPGA implementations of identity based encryption schemes. In J. Becker, M. Platzner, and S. Vernalde, editors, *Field-Programmable Logic and Applications*, number 3203 in Lecture Notes in Computer Science, pages 74–83. Springer, 2004.
9. S. Kumar, T. Wollinger, and C. Paar. Optimum digit serial GF(2^m) multipliers for curve-based cryptography. *IEEE Transactions on Computers*, 55(10):1306–1311, October 2006.
10. R. Ronan, C. Ó hÉigeartaigh, C. Murphy, M. Scott, T. Kerins, and W.P. Marnane. An embedded processor for a pairing-based cryptosystem. In *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG'06)*. IEEE Computer Society, 2006.
11. C. Shu, S. Kwon, and K. Gaj. FPGA accelerated Tate pairing based cryptosystem over binary fields. Cryptology ePrint Archive, Report 2006/179, 2006.
12. L. Song and K. K. Parhi. Low energy digit-serial/parallel finite field multipliers. *Journal of VLSI Signal Processing*, 19(2):149–166, July 1998.

A Parallel Version of the Itoh-Tsujii Multiplicative Inversion Algorithm

Francisco Rodríguez-Henríquez[1], Guillermo Morales-Luna[1], Nazar A. Saqib[2], and Nareli Cruz-Cortés[1,⋆]

[1] Computer Science Department
Centro de Investigación y de Estudios Avanzados del IPN
Av. Instituto Politécnico Nacional No. 2508, México D.F.
{francisco,gmorales}@cs.cinvestav.mx,
nareli@computacion.cs.cinvestav.mx
[2] Centre for Cyber Technology and Spectrum Management
(CCT & SM), NUST, Islamabad-Pakistan
nabbas@computacion.cs.cinvestav.mx

Abstract. In this contribution, we derive a novel parallel formulation of the standard Itoh-Tsujii algorithm for multiplicative inverse computation over $GF(2^m)$. When implemented in a Virtex 3200E FPGA device, our design is able to compute multiplicative inversion over $GF(2^{193})$ after 20 clock cycles in about 0.94μS.

1 Introduction

Binary extension finite fields $GF(2^m)$ are extensively used in many modern applications, such as public and secret key cryptosystems, etc. Among customary finite field arithmetic operations, namely, addition, subtraction, multiplication and inversion of nonzero elements, the computation of the later is the most time-consuming one. Multiplicative inversion computation of a nonzero element $a \in GF(2^m)$ is defined as the process of finding the unique element $a^{-1} \in GF(2^m)$ such that $a \cdot a^{-1} = 1$.

The *Itoh-Tsujii Multiplicative Inverse Algorithm* (ITMIA) was presented in [1]. Originally, ITMIA was proposed to be applied over binary extension fields with *normal basis* field element representation. Since its publication however, several improvements and variations of it have been reported [2, 3], showing that it can be used with other field element representations too.

In this contribution, considering a special class of irreducible trinomials, namely, $P(X) = X^m + X^k + 1$, with m and k odd integers, we derive a novel version of the Itoh-Tsujii algorithm which uses field multiplication, field squaring and field square root operators as main building blocks. We also show how this version of the algorithm can be parallelized when implemented in hardware platforms.

The rest of this paper is organized as follows. In Section 2 some basic mathematical concepts are reviewed. In Section 3, the standard version of the Itoh-Tsujii algorithm combined with the concept of addition chains is presented. Then, in Section 4, a novel parallel formulation of the ITMIA procedure is presented. In Sections 5 and 6, design

⋆ Work partially supported by CONACyT project 45306, México.

details of our algorithm hardware implementation are given together with the main performance figures of the block designs and comparison with other reported designs. Finally, in Section 7 some conclusive remarks are drawn.

2 Mathematical Preliminaries

Let $GF(2)$ be the prime field of characteristic 2: $GF(2) = \{0, 1\}$, addition is XOR and multiplication is the logical AND. Let $P(X) \in GF(2)[X]$ be an irreducible polynomial of degree $m > 1$, $m \in \mathbb{N}$, and let α be a root of $P(X)$ in a finite extension of $GF(2)$. Then the Galois field $GF(2^m)$ is isomorphic to the finite extension $GF(2)[\alpha]$, it has 2^m elements and the powers of α form a basis of $GF(2^m)$ over $GF(2)$. The set $\{1, \alpha, \alpha^2, \dots, \alpha^{m-1}\}$ is called the *polynomial basis* of $GF(2^m)$ corresponding to α. Thus, for each $a \in GF(2^m)$ there exists a polynomial $A(X) \in GF(2)[X]$ of degree at most $m-1$ such that $a = A(\alpha)$ and this is the *polynomial representation* of a. Addition in $GF(2^m)$ is performed by adding corresponding coefficients using polynomial representations. Multiplication corresponds to polynomial multiplication reduced modulus the irreducible polynomial $P(X)$: If $a \in GF(2^m)$ is represented by the polynomial $A(X)$ and $b \in GF(2^m)$ is represented by the polynomial $B(X)$ then $c = ab$ is represented by the polynomial $C(X) = A(X)B(X) \bmod P(X)$. The multiplicative group of $GF(2^m)$ is cyclic. If the irreducible polynomial $P(X)$ is also primitive, then any of its roots α is a generator of the multiplicative group of $GF(2^m)$ and has order $2^m - 1$. Thus for any nonzero $a \in GF(2^m)$, if $i \equiv j \bmod (2^m - 1)$ then $a^i = a^j$ in $GF(2^m)$. Since the field characteristic is 2, the squaring operator is linear:

$$a = A(\alpha) = \sum_{j=0}^{m-1} a_j \alpha^j \quad \Rightarrow \quad a^2 = \sum_{j=0}^{m-1} a_j \alpha^{2j} = A(\alpha^2). \tag{1}$$

Moreover, using just the representing polynomial, we can compute the reduction step $C(X) = A(X)^2 \bmod P(X)$ by adding four terms as,

$$\begin{aligned} C = {} & A'_{[0,m-1]} + A'_{[m,2m-1]} + A'_{[m,2m-1-n]}X^n \\ & + \left(A'_{[2m-n,2m-1]} + A'_{[2m-n,2m-1]}X^n\right) \end{aligned} \tag{2}$$

where $A'_{[i,j]}$ denotes the list of coefficients in $A(X)$ with indexes ranging from i to j. For instance, in the case of the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$, field squaring operation can be computed as,

$$c_i = \begin{cases} a_{i/2} & i \text{ even}, i < 15, \\ a_{i/2} + a_{i/2+89} + a_{178+i/2} & i \text{ even}, 15 < i < 30, \\ a_{i/2} + a_{i/2+89} & i \text{ even}, i \geq 30, \\ a_{(m+i)/2} + a_{193-(15-i)/2} & i \text{ odd}, i < 15, \\ a_{(m+i)/2} & i \text{ odd}, i \geq 15. \end{cases}$$

Above equation can be implemented at a cost of 96 two-input XOR gates and two XOR gate delays, $2T_x$.

The computation of the square root $\sqrt{a}$ of an arbitrary element can be obtained as follows. Considering that the squaring map $a \mapsto a^2$ is linear, if we denote by A_1 and A_2 the polynomial representation of a and a^2 respectively, there exists a square matrix M_2 of order $m \times m$ with coefficients in $GF(2)$ such that $A_2 = M_2 A_1$. If we are interested in calculating the square root $d = \sqrt{a} \in GF(2^m)$, with polynomial representation $D(\alpha)$, then we must have $M_2 D = A_1$, or $D = M_2^{-1} A_1$ which gives the alternative procedure for computing the field square root.

For instance, for $P(X) = X^{193} + X^{15} + 1$, one can solve the corresponding square matrix based on (2). The following set of equations are then obtained for $d = \sqrt{a}$,

$$d_i = \begin{cases} a_{2i} & i < 8, \\ a_{2i} + a_{2i-15} & 8 \leq i < 97 \\ a_{2i-15} + a_{2i-193} & 97 \leq i < 104, \\ a_{2i-193} & 104 \leq i \end{cases}$$

Above equation can be implemented at a cost of 96 two-input XOR gates and one T_x gate delay.

3 Itoh-Tsujii Multiplicative Inversion Algorithm (ITMIA)

Since the multiplicative group of the Galois field $GF(2^m)$ is cyclic of order $2^m - 1$, for any nonzero element $a \in GF(2^m)$ we have $a^{-1} = a^{2^m-2}$. Clearly,

$$2^m - 2 = 2(2^{m-1} - 1) = 2 \sum_{j=0}^{m-2} 2^j = \sum_{j=1}^{m-1} 2^j.$$

The right-most component of the above equalities allow us to express the multiplicative inverse of a in two ways:

$$\left[a^{2^{m-1}-1}\right]^2 = a^{-1} = \prod_{j=1}^{m-1} a^{2^j} \tag{3}$$

Let us consider the sequence $\left(\beta_k(a) = a^{2^k-1}\right)_{k \in \mathbb{N}}$. Then, for instance,

$$\beta_0(a) = 1 \quad , \quad \beta_1(a) = a,$$

and from the first equality at (3), $[\beta_{m-1}(a)]^2 = a^{-1}$.

It is easy to see that for any two integers $k, j \geq 0$,

$$\beta_{k+j}(a) = \beta_k(a)^{2^j} \beta_j(a). \tag{4}$$

Namely

$$\begin{aligned}\beta_{k+j}(a) &= a^{2^{k+j}-1} = \frac{a^{2^{k+j}}}{a} = \frac{\left(a^{2^k}\right)^{2^j}}{a} \\ &= \left(\frac{a^{2^k}}{a}\right)^{2^j} \frac{a^{2^j}}{a} = \left(a^{2^k-1}\right)^{2^j} a^{2^j-1} \\ &= \beta_k(a)^{2^j}\beta_j(a)\end{aligned}$$

In particular, for $j = k$,

$$\beta_{2k}(a) = \beta_k(a)^{2^k}\beta_k(a) = \beta_k(a)^{2^k+1}. \tag{5}$$

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse $a^{-1} \in GF(2^m)$ of a can be found by repeatedly applying eq's. (4) and/or (5). Hence, given $\beta_{u_0}(a) = a^{2^1-1} = a$, for each $u_i, 1 \le i \le t$, compute

$$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \beta_{u_{i_2}}(a) = \beta_{u_{i_2}+u_{i_1}}(a) = \beta_{u_i}(a) = a^{2^{u_i}-1}$$

A final squaring step yields the required result since,

$$[\beta_{u_t}(a)]^2 = \left(a^{2^{m-1}-1}\right)^2 = (a^{2^m-2}) = a^{-1}.$$

It is easy to see that that computation can be accomplished at a computational cost of t field multiplication plus a total of $m-1$ field squaring computations.

Table 1. $\beta_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\beta_{u_{i_1}}(a)\right]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_i}(a) = a^{2^{u}_i-1}$
0	1	–	–	$\beta_{u_0}(a) = a^{2^1-1}$
1	2	$2u_{i-1}$	$[\beta_{u_0}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_1}(a) = a^{2^2-1}$
2	3	$u_{i-1}+u_{i-2}$	$[\beta_{u_1}(a)]^{2^{u_0}} \cdot \beta_{u_0}(a)$	$\beta_{u_2}(a) = a^{2^3-1}$
3	6	$2u_{i-1}$	$[\beta_{u_2}(a)]^{2^{u_2}} \cdot \beta_{u_2}(a)$	$\beta_{u_3}(a) = a^{2^6-1}$
4	12	$2u_{i-1}$	$[\beta_{u_3}(a)]^{2^{u_3}} \cdot \beta_{u_3}(a)$	$\beta_{u_4}(a) = a^{2^{12}-1}$
5	24	$2u_{i-1}$	$[\beta_{u_4}(a)]^{2^{u_4}} \cdot \beta_{u_4}(a)$	$\beta_{u_5}(a) = a^{2^{24}-1}$
6	48	$2u_{i-1}$	$[\beta_{u_5}(a)]^{2^{u_5}} \cdot \beta_{u_5}(a)$	$\beta_{u_6}(a) = a^{2^{48}-1}$
7	96	$2u_{i-1}$	$[\beta_{u_6}(a)]^{2^{u_6}} \cdot \beta_{u_6}(a)$	$\beta_{u_7}(a) = a^{2^{96}-1}$
8	192	$2u_{i-1}$	$[\beta_{u_7}(a)]^{2^{u_7}} \cdot \beta_{u_7}(a)$	$\beta_{u_8}(a) = a^{2^{192}-1}$

Example 1. Let us consider the binary field $GF(2^{193})$ using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, using the addition chain shown in Table 1, we can compute the sequence of $\beta_{u_i}(a)$ coefficients. Notice that after having computed the coefficient $\beta_{u_8}(a)$, the only remaining step is to obtain a^{-1} which can be achieved as $a^{-1} = \beta^2_{u_8}(a)$. □

3.1 A Square Root-Then-Multiply Recursive Sequence of Field Elements

For any nonzero $a \in GF(2^m)$, we have $a^{\frac{1}{2}} = a^{2^{m-1}}$, and

$$\forall j \leq m-1: \quad a^{2^{-j}} = a^{2^{m-j}} \tag{6}$$

just because $2^{j(m-1)} \equiv 2^{m-j} \bmod (2^m - 1)$. Using (3) and (6), we obtain

$$a^{-1} = \prod_{j=m-1}^{1} a^{2^{m-j}} = \prod_{j=m-1}^{1} a^{2^{-j}} = a^{\sum_{j=m-1}^{1} 2^{-j}}$$

(observe that all indexes are varying in descending order). Noticing now that,

$$\sum_{j=m-1}^{1} 2^{-j} = \frac{1-\left(\frac{1}{2}\right)^m}{1-\left(\frac{1}{2}\right)} - 1 = \frac{2^m - 1}{2^{m-1}} - 1 = \frac{2^{m-1}-1}{2^{m-1}},$$

we get,

$$a^{-1} = a^{\frac{2^{m-1}-1}{2^{m-1}}}$$

Let a be any arbitrary nonzero element in the field $GF(2^m)$. Let us consider an addition chain U of length t for $m-1$ and its associated sequence V. Then the multiplicative inverse of a, $a^{-1} \in GF(2^m)$, can be found by repeatedly applying eq's. (9) and (10). Hence, given $\gamma_{u_0}(a) = a^{1-2^{-1}} = \sqrt{a}$, for each u_i, $1 \leq i \leq t$, compute

$$\left[\gamma_{u_{i_1}(a)}\right]^{2^{-u_{i_2}}} \gamma_{u_{i_2}}(a) = \gamma_{u_{i_2}+u_{i_1}}(a) = \gamma_{u_i}(a) = a^{1-2^{-u_i}}$$

Where $\gamma_{\{u_t=m-1\}} = a^{1-2^{-(m-1)}} = a^{-1}$ gives the required result. It is easy to see that that computation can be accomplished at a computational cost of t field multiplication plus a total of $m-1$ square root computations.

Example 2. Following with our running example, let us consider the binary field $GF(2^{193})$ generated using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Let $a \in GF(2^{193})$ be an arbitrary nonzero field element. Then, Table 2 shows how to compute the sequence of $\gamma_{u_i}(a)$ coefficients. The multiplicative inverse is then given as $\gamma_{u_8} = a^{-1}$. □

Table 2. $\gamma_i(a)$ Coefficient Generation for m-1=192

i	u_i	rule	$\left[\gamma_{u_{i_1}}(a)\right]^{2^{-u_{i_2}}} \cdot \gamma_{u_{i_2}}(a)$	$\gamma_{u_i}(a) = a^{1-2^{-u_i}}$
0	1	–	–	$\gamma_{u_0}(a) = a^{1-2^{-1}}$
1	2	$2u_{i-1}$	$[\gamma_{u_0}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_1}(a) = a^{1-2^{-2}}$
2	3	$u_{i-1}+u_{i-2}$	$[\gamma_{u_1}(a)]^{2^{-u_0}} \cdot \gamma_{u_0}(a)$	$\gamma_{u_2}(a) = a^{1-2^{-3}}$
3	6	$2u_{i-1}$	$[\gamma_{u_2}(a)]^{2^{-u_2}} \cdot \gamma_{u_2}(a)$	$\gamma_{u_3}(a) = a^{1-2^{-6}}$
4	12	$2u_{i-1}$	$[\gamma_{u_3}(a)]^{2^{-u_3}} \cdot \gamma_{u_3}(a)$	$\gamma_{u_4}(a) = a^{1-2^{-12}}$
5	24	$2u_{i-1}$	$[\gamma_{u_4}(a)]^{2^{-u_4}} \cdot \gamma_{u_4}(a)$	$\gamma_{u_5}(a) = a^{1-2^{-24}}$
6	48	$2u_{i-1}$	$[\gamma_{u_5}(a)]^{2^{-u_5}} \cdot \gamma_{u_5}(a)$	$\gamma_{u_6}(a) = a^{1-2^{-48}}$
7	96	$2u_{i-1}$	$[\gamma_{u_6}(a)]^{2^{-u_6}} \cdot \gamma_{u_6}(a)$	$\gamma_{u_7}(a) = a^{1-2^{-96}}$
8	192	$2u_{i-1}$	$[\gamma_{u_7}(a)]^{2^{-u_7}} \cdot \gamma_{u_7}(a)$	$\gamma_{u_8}(a) = a^{1-2^{-192}}$

4 A Parallel Version of the Itoh-Tsujii Algorithm

In an analogous way as we did in Subection 3 for the sequence $(\beta_k(a))_{k\in\mathbb{N}}$, let us define now the sequence $\left(\gamma_k(a) = a^{1-2^{-k}}\right)_{k\in\mathbb{N}}$. First of all, let us remark that, from (6), $\forall k \in \mathbb{N}$:

$$\gamma_k(a) = a^{1-2^{-k}} = a \cdot \left[a^{2^{-k}}\right]^{-1} = a \cdot \left[a^{2^{m-k}}\right]^{-1} = [\beta_{m-k}(a)]^{-1}. \tag{7}$$

In particular, for $k = m-1$,

$$a^{-1} = \gamma_{m-1}(a). \tag{8}$$

As in (4), for any two integers $k, j \geq 0$,

$$\gamma_{k+j}(a) = \gamma_k(a)^{2^{-j}} \gamma_j(a). \tag{9}$$

and, as in (5), for $j = k$,

$$\gamma_{2k}(a) = \gamma_k(a)^{2^{-k}} \gamma_k(a) = \gamma_k(a)^{2^{-k}+1} \tag{10}$$

The sequence $(\gamma_k(a))_{k\in\mathbb{N}}$ is also periodic with period m. Furthermore, using eq's. (8), (9) and (6), $\forall k \leq m-2$:

$$a^{-1} = \gamma_{m-1}(a) = \gamma_{m-1-k}(a)^{2^{-k}} \gamma_k(a) = \left[\beta_{k+1}(a)^{2^{-k}} \beta_{m-k}(a)\right]^{-1}.$$

Let us remark also that if k, j are such that $k+j = m-j$, then, according with (4) and (7),

$$\begin{aligned}\beta_k(a)\left(\beta_j(a)^{2^k}\gamma_j(a)\right) &= \left(\beta_j(a)^{2^k}\beta_k(a)\right)\gamma_j(a)\\ &= \beta_{k+j}(a)\,\gamma_j(a)\\ &= \beta_{m-j}(a)\,\gamma_j(a)\\ &= 1\end{aligned}$$

and consequently $\beta_k(a)^{-1} = \beta_j(a)^{2^k}\gamma_j(a)$. In summary

$$m \equiv k \bmod 2 \implies \beta_k(a)^{-1} = \beta_{\frac{m-k}{2}}(a)^{2^k}\gamma_{\frac{m-k}{2}}(a), \tag{11}$$

Thus, in particular by letting $k = 1$ in eq. (11) we get,

$$m \equiv 1 \bmod 2 \implies a^{-1} = \beta_{\frac{m-1}{2}}(a)^2\gamma_{\frac{m-1}{2}}(a) = \left[a^{2^{\frac{m-1}{2}}-1}\right]^2 a^{1-2^{-\frac{m-1}{2}}}.$$

Our modification of Itoh-Tsujii algorithm uses now an addition chain for $\frac{m-1}{2}$ and relations (9) and (10) to compute $a^{-1} = \gamma_{m-1}(a)$. Above result can be summarized as follows,

Theorem. *Let $a \in GF(2^m)$ be an arbitrary nonzero field element, with m odd. Then, its multiplicative inverse, a^{-1}, can be found as,*

$$a^{-1} = \left[\beta_{\{\frac{m-1}{2}\}}(a)\right]^2 \gamma_{\{\frac{m-1}{2}\}}(a) = \left[a^{(2^{\frac{m-1}{2}}-1)}\right]^2 \cdot a^{1-2^{-\frac{(m-1)}{2}}}. \tag{12}$$

Example 3. Consider the binary finite field *GF*(2^3), then according to (12) the multiplicative inverse of an arbitrary nonzero field element would be,

$$\begin{aligned} a^{-1} &= \left[\beta_{\{\frac{3-1}{2}\}}(a)\right]^2 \gamma_{\{\frac{3-1}{2}\}}(a) = \left[a^{(2^{\frac{3-1}{2}}-1)}\right]^2 \cdot a^{1-2^{-\frac{(3-1)}{2}}} \\ &= a^2 \cdot a^{1-2^{-1}} = a^2 \cdot a \cdot \left[a^{2^{-1}}\right]^{-1} = a^3 \cdot \left[a^{2^{3-1}}\right]^{-1} = a^3 \cdot \left[a^4\right]^{-1} = a^{-1}. \end{aligned}$$

Notice that in virtue of (6), the equality $a^{2^{-1}} = a^{2^{3-1}}$ used above holds. □

Example 4. Let us consider once again the binary field *GF*(2^{193}) using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. We can reuse coefficients β_7 and $\hat{\beta}_7$ defined in Tables 1 and 2, respectively. Then the multiplicative inverse of a can be found as, $a^{-1} = \beta^2{}_{u_7}\gamma_{u_7}$. □

5 Reconfigurable Hardware Architecture for Multiplicative Inversion in *GF*(2^{193})

In this Section, a description of our proposed architecture in reconfigurable hardware is presented. [1]

[1] Throughout the rest of this Section, we assume that the binary extension field *GF*(2^{193}) was generated using the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$.

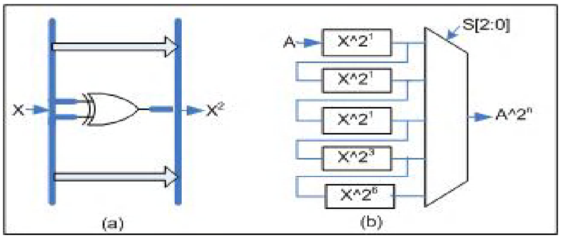


Fig. 1. Squarer $GF(2^{193})$ (a) for X^{2^1} (b) for X^{2^n} implementation

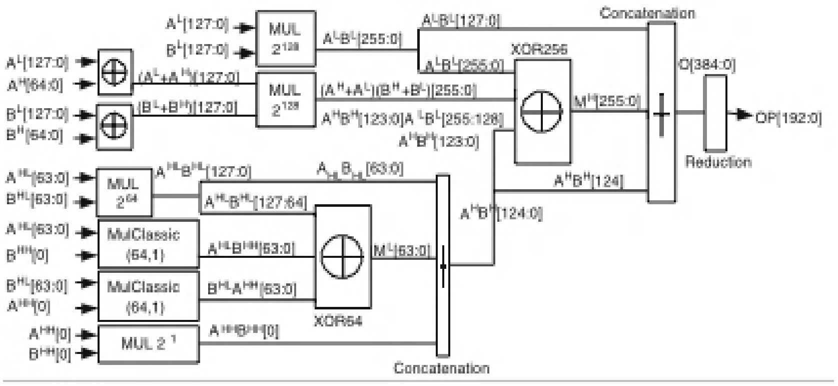


Fig. 2. $GF(2^{193})$ binary Karatsuba multiplier

Field Squaring and Square Root Blocks

Figure 1.a shows our strategy for implementing field squaring trying to use as few clock cycles as possible. In the case of the binary extension field $GF(2^{193})$, field squaring can be obtained using XOR gates only, as it was summarized in (2).

Figure 1.b shows the $GF(2^{193})$ field squarer block used in this work. Referring to the addition chain described in Table 1, we must compute up to 96 squaring operations in a row.

That is why we took the compromise decision of cascading 12 field squarer blocks back to back and then, by the appropriate usage of multiplexer blocks, obtain the corresponding outputs after 1, 3, 6, and 12 squarer blocks as shown in Figure 1.b. As an example, the $X^{2^{24}}$ field operation can be accomplished in just two clock cycles by taking the output after the last squarer block (12 squarers) in the first clock cycle and then repeating this operation in a second clock cycle so that we get the required 24 field squarings. Similarly, we implemented multiple square root operations trying to save as many clock cycles as possible. The five inputs of the multiplexer are formed by replicating

1, 1, 1, 3, and 6 square root blocks which can perform 1, 2, 3, 6 and 12 square root operations respectively, in just one clock cycle.

Field Multiplier

Our strategy for multiplication is based on the binary Karatsuba-Ofman multiplier which is a variation of normal Karatsuba-Ofman multiplier as it was presented in [4]. Figure 2 shows a $GF(2^{193})$ binary Karatsuba-Ofman multiplier which is a hybrid approach that utilizes Karatsuba-Ofman multiplication with a combination of the classical school-method whenever it is useful. In this design, two 193-bit operands A and B are multiplied by first dividing each operand into two parts: upper part (say A^H and B^H of 128 bits each) and lower part (say A^L and B^L of 65 bits each). For 128-bit multiplications, two Karatsuba-Ofman multipliers were used. However, for 65 bit multiplication, instead of using three 64-bit Karatsuba-Ofman multipliers, only one 64-bit Karatsuba-Ofman multiplier plus circuitry was used. Therefore, the time delay of the 193-bit multiplier is equal to the time delay of the biggest multiplier only (time delay of the 128-bit multiplier block).

Table 3. Algorithm Dataflow

Clock	AdrA‖AdrB‖$S_0S_1S_2$‖SQR‖SQROOT	Write Reg1	Write Reg2
1	0000‖0000‖000‖000‖000	Loading input data	
2	0001‖0000‖000‖000‖000	$t_1 = [\beta_{u_0}(a)]^{2^1}$	
3	1001‖0001‖001‖000‖000	$\beta_{u_1}(a) = t_1 \cdot \beta_{u_0}(a)$	$t_2 = [\gamma_{u_0}(a)]^{2^{-1}}$
4	0010‖0000‖000‖000‖000	$t_1 = [\beta_{u_1}(a)]^{2^1}$	$\gamma_{u_1}(a) = t_2 \cdot \gamma_{u_0}(a)$
5	1010‖0000‖001‖010‖000	$\beta_{u_2}(a) = t_1 \cdot \beta_{u_1}(a)$	$t_2 = [\gamma_{u_1}(a)]^{2^{-1}}$
6	0011‖0010‖000‖000‖010	$t_1 = [\beta_{u_2}(a)]^{2^1}$	$\gamma_{u_2}(a) = t_2 \cdot \gamma_{u_1}(a)$
7	1011‖1010‖001‖011‖000	$\beta_{u_3}(a) = t_1 \cdot \beta_{u_2}(a)$	$t_2 = [\gamma_{u_2}(a)]^{2^{-1}}$
8	0101‖0011‖000‖000‖011	$t_1 = [\beta_{u_3}(a)]^{2^6}$	$\gamma_{u_3}(a) = t_2 \cdot \gamma_{u_2}(a)$
9	1100‖1011‖001‖100‖000	$\beta_{u_4} = t_1 \cdot \beta_{u_3}(a)$	$t_2 = [\gamma_{u_3}(a)]^{2^{-6}}$
10	0110‖0101‖000‖000‖100	$t_1 = [\beta_{u_4}(a)]^{2^{12}}$	$\gamma_{u_4} = t_2 \cdot \gamma_{u_3}(a)$
11	1101‖1100‖001‖100‖000	$\beta_{u_5}(a) = t_1 \cdot \beta_{u_4}(a)$	$t_2 = [\gamma_{u_4}(a)]^{2^{-12}}$
12	DC‖ DC ‖10DC‖100‖100	$t_1 = [\beta_{u_5}(a)]^{2^{12}}$	$\gamma_{u_5}(a) = t_2 \cdot \gamma_{u_4}(a)$
13	0111‖0110‖010‖000‖100	$t_1 = (t_1)^{2^{12}}$	$t_2 = [\gamma_{u_5}(a)]^{2^{-12}}$
14	0111‖1101‖001‖100‖000	$\beta_{u_6}(a) = t_1 \cdot \beta_{u_5}(a)$	$t_2 = (t_2)^{2^{-12}}$
15	DC ‖ DC ‖10DC‖100‖100	$t_1 = [\beta_{u_6}(a)]^{2^{12}}$	$\gamma_{u_6}(a) = t_2 \cdot \gamma_{u_5}(a)$
16	DC ‖ DC ‖11DC‖100‖100	$t_1 = (t_1)^{2^{12}}$	$t_2 = [\gamma_{u_6}(a)]^{2^{-12}}$
17	DC ‖ DC ‖11DC‖000‖100	$t_1 = (t_1)^{2^{12}}$	$t_2 = (t_2)^{2^{-12}}$
18	1000‖0111‖011‖000‖000	$t_1 = (t_1)^{2^{12}}$	$t_2 = (t_2)^{2^{-12}}$
19	1111‖1110‖000‖000‖000	$\beta_{u_7}(a) = t_1 \cdot \beta_{u_6}(a)$	$t_2 = (t_2)^{2^{-12}}$
20	DC ‖1111‖001‖000‖000	$t_1 = [\beta_{u_7}(a)]^{2^1}$	$\gamma_{u_7}(a) = t_2 \cdot \gamma_{u_6}(a)$
21	INV‖INV‖000‖000‖000	$\gamma_{u_8}(a) = t_1 \cdot \gamma_{u_7}(a) = a^{-1}$	

General Architecture

The proposed architecture for multiplicative inversion includes a square root, squarer and multiplier blocks as shown in Figure 3. We can calculate arbitrarily multiplicative inverses over $GF(2^{193})$ by computing in parallel the $\beta_{u_i}(a)$ and $\gamma_{u_i}(a)$ coefficients, discussed in the previous Section. Both sequences are independent and can be processed in parallel provided that hardware resources meet up design requirements. A direct approach would be to use two multipliers with squarer and square root blocks operating separately. That would, however, be more expensive as our multiplier block consumes a large amount of hardware resources. A more reasonable architecture can be obtained with a single multiplier by introducing a multiplexer for squarer and square-root blocks as shown in Fig. 3. The intermediate results required for next stages of the algorithm are read/written in a Block select RAM (BRAM).

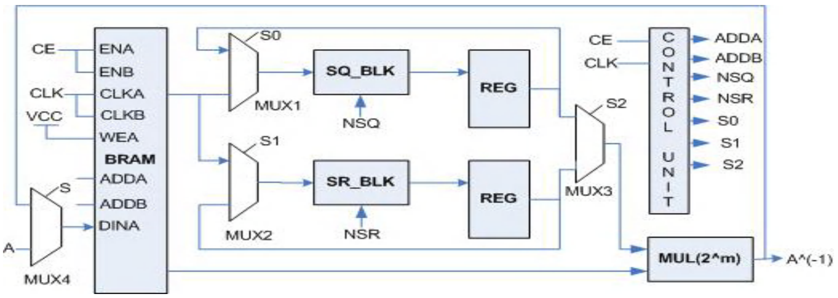


Fig. 3. General Architecture for Multiplicative Inversion over $GF(2^{193})$

We use Xilinx's dual-port BRAMs built-in memory modules in order to achieve higher parallelism. Hence, first port was configured as a RAM in order to write the multiplier outputs, while the second one was configured as a ROM, responsible to read the second multiplier operand (already stored from previous iterations). A single BRAM has a size of 4K (4096 bits), which is sufficiently large for storing all intermediate results generated by our algorithm. Additionally, an array of 12 BRAMs was needed for managing a 193-bit data bus.

The multiplier I/Os for writing/reading to/from BRAM are governed by an address scheme. Data paths for squaring, square root and then multiplication are adjusted by providing selection bits for the three multiplexers MUX1, MUX2, and MUX3. MUX4 is used for switching external data during the first cycle and then to feedback data until the final calculation of inversion is obtained. The NSQ and NSR control signals select data path for performing a number of squaring and square root operations. This is done by providing address bits to the multiplexers available inside the SQ_BLK and SR_BLK blocks.

For instance, NSQ=000 selects the first input of the multiplexer which is connected to the output of a single squarer unit, whereas NSQ=101, selects the fifth multiplexer input, which is connected to the 12 squarer unit. A total of 17 bits (4 bits for BRAM port A, 4 bits for BRAM port B, 1 bit for MUX1, 1 bit for MUX2, 1 bit MUX3, 3 bits for NSQ, 3 bits for NSR) are used for controlling and synchronizing the whole circuitry. The 17-bit control word for each clock cycle is filled in the ROM block, and then they are extracted at the rising edge of each clock cycle. A short description of the control unit is given below.

As shown in Fig. 3, Control Unit block orchestrates and synchronizes data flow for the whole design. A 4- bit counter and a ROM constitute the control unit. The ROM block is filled with a total of twenty 17-bit control words, whose address bits are timely incremented by a 4-bit counter. Those control words are used at each one of the 20 clock cycles required for completing the execution of our algorithm.

Table 3 shows the algorithm dataflow. In the first cycle, the field element a, whose multiplicative inverse is required, is written into the BRAM. Then, starting at cycle 2, our architecture of Fig. 3 computes both of them, $\beta_{u_i}(a)$ and $\gamma_{u_i}(a)$ for $i = 0, 1, \cdots, 7$ in parallel. At clock 21, a final computation, namely, $\gamma_{u_8}(a) = [\beta_{u_7}(a)]^{2^1} \cdot \gamma_{u_7}(a)$, is performed, which according to theorem 4 yields the required multiplicative inverse, i.e., $\gamma_{u_8}(a) = a^{-1}$.

Notice that the control word that commands all the operations to be performed in the next rising edge of the master clock, is set at the rising edge of the previous clock cycle. For example at cycle 8, the control word: $0101||0011||000||000||011$ selects the operations for cycle 9 as follows: address 0101 commands to write data at port A; address 0011 orders to read data from port B; address 000 selects the first input of all the three multiplexer blocks; address 000 commands to perform a single squaring; and finally the address 011 orders to perform six square root operations that will be stored in the register t_2. [2] If more than one multiplicative inverse computations are required, it is possible to compute a single multiplicative inverse calculation in just 20 clock cycles.

Table 4. Specifications for inversion in $GF(2^m)$

Reference	Platform	Field	Cycles	Freq (MHz)	*timings*
Gutub et. al. [5]	Virtex II	$GF(2^{256})$	5000	50	$100\mu S$
Goodman et. al. [6]	0.25μ CMOS	$GF(2^{256})$	3712	50	$74.24\mu S$
Bednara et. al. [7]	Xilinx Virtex	$GF(2^{191})$	274	36	$7.4\mu S$ (est.)
Lutz [8]	Xilinx Virtex	$GF(2^{163})$	259	50	$5.18\mu S$
This work (Standard)	Xilinx VirtexE	$GF(2^{193})$	28	21.2	$1.32\mu S$
This work (Parallel)	Xilinx VirtexE	$GF(2^{193})$	20	21.2	$0.943\mu S$

6 Comparison and Results

A summary of the implementation results obtained for each individual building block as well as for the whole system, i.e., inversion over $GF(2^{193})$ is given below. Xilinx Foundation Tool F4.1i was used for design synthesis, implementation and verification of results. The Binary Karatsuba-Ofman multiplier block occupied 8753 CLB slices executing one field multiplication in 43.1ηS. The field Squarer and square root in $GF(2^{193})$ took a total of 47 and 46 CLB slices for a single block respectively. The architecture was implemented in a XCV3200efg1156 (VirtexE device) occupying a total of 11131 (34.3%) CLB Slices and 12 (5 %) BRAMs. One inversion in $GF(2^{193})$ consumes 0.943μS in 20 clock cycles at a rate of 21.2 MHz (47.16 ηS).

[2] The code word "DC" denotes the *Don't Care* condition.

Table 4 shows the computational cost of several reported designs for the computation of multiplicative inversion over $GF(2^m)$ in hardware platforms. Furthermore, we show also that an implementation of the *standard* Itoh-Tsujii algorithm using our architecture requires 28 clock cycles. [3]

7 Conclusions

In this paper, a novel derivation of the standard Itoh-Tsujii algorithm that offers a significant speedup when implemented in hardware platforms was presented. We implemented the proposed algorithm in a reconfigurable hardware device for the computation of multiplicative inverses of nonzero field elements in the finite field $GF(2^m)$ generated by the irreducible trinomial $P(X) = X^{193} + X^{15} + 1$. Our experimental results show that the parallel version of the Itoh-Tsujii algorithm implementation yields a speedup of about 30% when compared with the standard version of it.

References

1. T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in GF(2^m) using normal basis," *Information and Computing*, vol. 78, pp. 171–177, 1988.
2. J. Guajardo and C. Paar, "Itoh-tsujii inversion in standard basis and its application in cryptography and codes," *Designs, Codes and Cryptography*, vol. 25, pp. 207–216, 2002.
3. F. Rodríguez-Henríquez, N. A. Saqib, and N. Cruz-Cortés, "A fast implementation of multiplicative inversion over GF(2^m)," in *International Symposium on Information Technology (ITCC 2005)*, vol. 1, Las Vegas, Nevada, U.S.A., April 2005, pp. 574–579.
4. F. Rodríguez-Henríquez, N. A. Saqib, and A. Díaz-Pérez, "A fast parallel implementation of elliptic curve point multiplication over $GF(2^m)$," *Elsevier Journal of Microprocessors and Microsystems*, vol. 28, no. 8, pp. 329–339, Aug. 2004.
5. A. A.-A. Gutub, A. F. Tenca, E. Savas, and Ç. K. Koç, "Scalable and unified hardware to compute montgomery inverse in GF(p) and GF(2)," in *LNCS*, vol. 2523. Springer, 2003, pp. 485–500.
6. J. Goodman and A. P. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor," *IEEE Journal of Solid-State Circuits*, vol. 36, no. 11, pp. 1808–1820, Nov. 2001.
7. M. Bednara et al., "Reconfigurable implementation of elliptic curve crypto algorithms," in *Proc. of The 9th Reconfigurable Architectures Workshop (RAW-02)*, April 2002.
8. J. Lutz, "High Performance Elliptic Curve Cryptographic Co-processor," Master's thesis, University of Waterloo, 2004.

[3] Notice that this implies that the computation of the step 8 in either Table 1 or Table 2 that would normally imply a cost of 9 clock cycles in a standard implementation can be accomplished by our architecture in only 1 clock cycle yielding a speedup factor of S=9. Since the cost of step 8 represents $P = 9/28 = 0.3214$ of the whole algorithm, then using Amdahl's law, our architecture can achieve a speedup of $\frac{1}{1-P+\frac{P}{S}} = 1.4$.

A Fast Finite Field Multiplier*

Edgar Ferrer[1], Dorothy Bollman[2], and Oscar Moreno[3]

[1] PhD. CISE Program, University of Puerto Rico, Mayagüez, PR 00681
eferrer@cs.uprm.edu
[2] Department of Mathematical Sciences,
University of Puerto Rico, Mayagüez, PR 00681
bollman@cs.uprm.edu
[3] Department of Computer Science,
University of Puerto Rico, Rio Piedras, PR 00931
moreno@uprr.pr

Abstract. We present a method for implementing a fast multiplier for finite fields $GF(2^m)$ generated by irreducible trinomials of the form $\alpha^m + \alpha^n + 1$. We propose a design based on the Mastrovito multiplier which is described by a parallel/serial architecture that computes a multiplication in m clock cycles by using only bit-adders (XORs), bit-multipliers (ANDs), and shift registers. This approach exploits symmetries and subexpression sharing in Mastrovito matrices in order to reduce the number of operations, and hence computation time in our FPGA implementation. According to preliminary performance results, our approach performs efficiently for large fields and has potential for a variety of applications, such as cryptography, coding theory, and the reverse engineering problem for genetic networks.

1 Introduction

A Field-Programmable Gate Array or FPGA is a silicon chip containing an array of configurable logic blocks (CLBs). Over the last years new technologies have been developed in order to increase speed and the amount of CLBs in FPGAs, therefore they are becoming more feasible for implementing a wide range of applications. Nowadays FPGAs are becoming attractive in the High Performance Computing (HPC) community, because they can accelerate critical segments in high performance applications. HPC hardware vendors have begun to offer solutions that incorporate FPGAs into HPC systems where they can act as co-processors, accelerating key kernels within an application.

This work is concerned with accelerating finite field arithmetic in a high performance application by using FPGAs. Our motivation is an important problem in computational biology: the reverse engineering problem for genetic networks that are modeled by finite fields [1],[3]. However, our work has applications in other areas such as cryptography and error correction coding. These applications rely on intensive arithmetic computations over finite fields. Addition and

* This research is supported by grant NIH-MBRS (SCORE) S06-GM08102.

multiplication are two basic operations. Addition is easily realized at very low computational cost, but other arithmetic operations on finite fields such as inversion, squaring, exponentiation and divisions are typically performed by repeated multiplications.

2 Background

An element in the finite field $GF(2^m)$ can be represented as a sequence of m bits in $GF(2)$ describing the coefficients of a binary polynomial. This representation is useful for manipulating finite field elements via bitwise operations, so we can exploit the hardware architecture of computers by carrying out finite field arithmetic by means of bit-level operations. In essence, the arithmetic computation over $GF(2^m)$ is suitable for FPGA implementations. We take advantage of reconfigurable hardware resources with the aim of accelerating computations considerably.

Multiplication is an expensive operation. Many solutions have been proposed for efficient multiplication over finite fields. Solutions are based on purely software approaches, purely hardware approaches, and more recently, on hardware/ software using reconfigurable computing.

The representation of the field elements distinguishes the particular features of a finite field multiplier. The most common representations are dual basis, normal basis, and standard basis. In this work we deal with finite field elements represented in standard (or polynomial or canonical) basis, such that the finite field $GF(2^m)$ consists of a finite set of all binary polynomials of degree less than m. For example $GF(2^2) = \{0, 1, \alpha, \alpha + 1\}$, where α is a root in $GF(2^2)$ of the irreducible polynomial $\alpha^2 + \alpha + 1$.

A very natural approach for standard basis multiplication in $GF(2^m)$ is to multiply two elements in the field as polynomial multiplication modulo an irreducible polynomial. This operation is typically accomplished in two stages: polynomial multiplication and modular reduction.

Let $A(\alpha)$, $B(\alpha)$, $C(\alpha)$ elements in $GF(2^m)$ and $f(\alpha)$ the irreducible polynomial generating $GF(2^m)$. Then the finite field multiplication $C(\alpha) = A(\alpha)B(\alpha)$ is accomplished by calculating

$$C(\alpha) = A(\alpha) * B(\alpha) \mod f(\alpha) \tag{1}$$

where $*$ denotes polynomial multiplication. In a first stage the product $A(\alpha) * B(\alpha)$ is calculated, resulting in a polynomial $Q(\alpha)$ of degree at most $2m - 2$.

$$Q(\alpha) = A(\alpha) * B(\alpha) = \left(\sum_{i=0}^{m-1} a_i \alpha^i\right)\left(\sum_{i=0}^{m-1} b_i \alpha^i\right) \tag{2}$$

In a second stage the modular reduction is performed on $Q(\alpha)$, that is, $C(\alpha) = Q(\alpha) \mod f(\alpha)$, resulting in the polynomial $C(\alpha)$ of degree at most $m - 1$.

It is easy to show that the expansion of equation (2) can be expressed as a matrix-vector product $Q = MB$, where Q is a vector of dimension $2m - 1$, which consists of the coefficients of $Q(\alpha)$. In the same way B is an m dimensional vector which consists of the coefficients of $B(\alpha)$, while the $(2m-1)\times m$ matrix M involves coefficients of $A(\alpha)$ (see for example [11]).

Notice that the last $m-1$ components of the vector Q (i.e. $[q_m,\dots,q_{2m-2}]$) contain terms with degree greater than $m-1$. These terms must be reduced modulo the irreducible polynomial $f(\alpha)=\alpha^m+g(\alpha)$ in order to express them as polynomials in the field $GF(2^m)$. This reduction is obtained by using the reducing identity $\alpha^m=g(\alpha)$, so all the terms with degree greater than $m-1$ will be reduced to terms with degree in the proper range $[0,m-1]$. Each reduced term is added to the respective terms in $[q_0,\dots,q_{m-1}]$, and so we get $C(\alpha)$. A particular term may need to be reduced several times. The maximum number of reductions is determined by:

$$N[m,n]=\left\lceil\frac{m-1}{\Delta}\right\rceil$$

where $\Delta=m-n$ [6].

For example, let $m=3$ and $f(\alpha)=\alpha^3+\alpha^2+1$, thus $\alpha^3=\alpha^2+1$ and $\alpha^4=\alpha^3+\alpha$. Using these identities the term $q_3\alpha^3$ is reduced only once: $q_3\alpha^3=q_3\alpha^2+q_3$, while $q_4\alpha^4$ is reduced twice: $q_4\alpha^4=q_4\alpha^3+q_4\alpha=q_4\alpha^2+q_4\alpha+q_4$, and so we get $C(\alpha)=q_4\alpha^4+q_3\alpha^3+q_2\alpha^2+q_1\alpha+q_0=(q_4+q_3+q_2)\alpha^2+(q_4+q_1)\alpha+(q_4+q_3+q_0)$. Notice that the maximum number of reductions is $N[3,2]=2$.

An alternative to the two-stage method, described above, for computing C is to perform the reduction directly on the matrix M, obtaining an already reduced $m\times m$ dimensional matrix Z, such that $C=ZB$. Z is called the Mastrovito matrix [8].

3 Multiplication Algorithm

A common approach to the design of multipliers that are based on the Mastrovito matrix Z is to compute Z and then do the multiplication in $GF(2^m)$ by means of matrix-vector multiplication. In our approach, we exploit the symmetry of Z without actually computing Z.

A method for constructing the Mastrovito matrix is proposed in [11], [6]. According to this method if $GF(2^m)$ is defined by the trinomial $\alpha^m+\alpha^n+1$ then Z is given by

$$Z=\begin{bmatrix}U\\L\end{bmatrix}$$

where U and L are Toeplitz matrices defined as follows:

Let $F=[\,0\;a_{m-1}\;a_{m-2}\;\dots\;a_1\,]$ and for each $i=0,1,\dots,m-1$, let $F[i\rightarrow]$ be the result of shifting F i positions to the right (vacated positions on the left are filled with zeros). Also let $G=[\,a_n\;a_{n-1}\;\dots\;a_1\;a_0\;a_m\;\dots\;a_{n+1}\,]$

U is $n \times m$, its first column is $[\,a_0\ a_1\ \ldots\ a_{n-1}\,]^T$, and its first row is

$$[a_0]||\sum_{i=0}^{N-1} F[i\varDelta \rightarrow]$$

where $\varDelta = m - n$, $||$ represents concatenation, and N is a short notation for $N[m, n]$.

L is $\varDelta \times m$, its first column is $[\,a_n\ a_{n+1}\ \ldots\ a_{m-1}\,]^T$, and its first row is

$$G + \sum_{i=0}^{N-1} F[i\varDelta \rightarrow]$$

Although the previously described method is used for constructing the entire Mastrovito matrix Z, in this work we construct only one row of Z which is sufficient in our approach for carrying out multiplications in $GF(2^m)$. By constructing the n-th row Z_n (where rows are numbered $0, 1, \ldots$), the remaining rows of Z can be obtained by means of right-shifts and concatenations over Z_n.

Example: If $GF(2^7)$ is defined by $\alpha^7 + \alpha^4 + 1$, then $\varDelta = 3$, $N = 2$, and

$$G = [\,a_4\ a_3\ a_2\ a_1\ a_0\ a_6\ a_5\,]$$

$$\sum_{i=0}^{N-1} F[i\varDelta \rightarrow] = F + F[\varDelta \rightarrow] = [\,0\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\,] + [\,0\ 0\ 0\ 0\ a_6\ a_5\ a_4\,]$$

and so L_0 is

$$Z_4 = [\,a_4\ a_3 + a_6\ a_2 + a_5\ a_1 + a_4\ a_0 + a_3 + a_6\ a_6 + a_2 + a_5\ a_5 + a_1 + a_4\,]$$

The construction method previously described allows us to devise a template for the n-th row of the Mastrovito matrix(Z_n), which needs to be pre-computed as a first step in the multiplication algorithm (see Algorithm 1). In the for-loop, one output bit of C is obtained in each cycle by multiplying (via inner product) the current row Z_i by B. Each row is obtained as a result of right-shifting the previous row and filling the vacated position on the left with a_i.

Algorithm 1

```
Input: $A(\alpha), B(\alpha)$,template for $Z_n$; $A(\alpha), B(\alpha) \in GF(2^m)$
Output: $C(\alpha) = A(\alpha)B(\alpha)$; $C(\alpha) \in GF(2^m)$

Compute $Z_n$
$S \leftarrow Z_n$
for $i = 0$ to $m - 1$
    $c_{(i+n) \bmod m} \leftarrow S \cdot B$
    $S \leftarrow$ right-shift($S$)
    $s_0 \leftarrow a_i$
end for
return(C)
```

4 FPGA Implementation

The proposed multiplier is implemented in a parallel/serial architecture, which computes a multiplication in m clock cycles. One output bit of C is obtained in each cycle by multiplying (inner product) the current row Z_i by B, thus achieving the matrix-vector product C = ZB. According to this method, the finite field multiplication is carried out by means of m inner products. Hence, inner product is the main operation in our finite field multiplier

Since all the inner products are performed over fields of characteristic 2, they could be done by means of FIR (Finite Impulse Response) filters. FIR filters are widely used in various Digital Signal Processing (DSP) applications. A comprehensive treatment of the fundamentals of FIR filters and FPGA implementations can be found in [9].

A traditional architecture of a FIR filter is shown in figure 1. The m-bit input is shifted through m bit-registers (known as taps). Each output stage of a particular register is multiplied by a known factor. The resulting outputs of the multipliers are then summed to produce the filter output. A conventional FIR filter implementation consists basically of multiplication units and summation units. Inner product operations are carried out by a Multiply-and-Accumulate model, which compute and accumulate the binary partial products in a bit-register. This implementation requires m multiplications and $m - 1$ additions to compute an inner product over an m-bit input. Thus, m cycles are required before the next inner product can be processed.

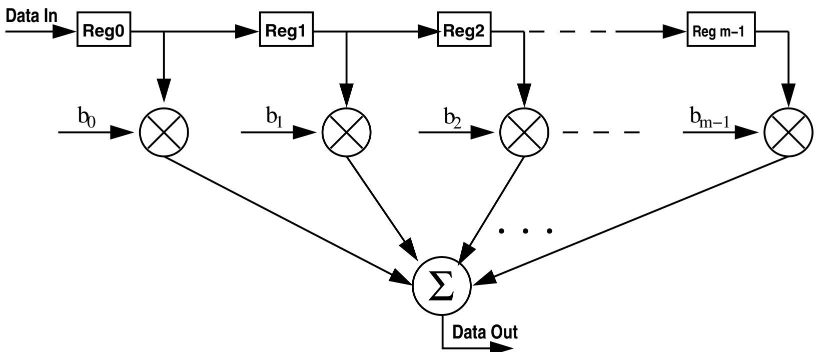


Fig. 1. An m-Tap FIR filter traditional architecture

Instead of using the aforementioned Multiply-and-Accumulate model, in this work we use a Multiply-and-Add design. By using this approach, two bit sequence can be multiplied in parallel, and afterwards, the sum of the resulting bit sequence has to be computed. This parallel implementation can speed-up the performance of the inner product. Here parallel multiplication is possible because the input vectors, namely, the current row of the Mastrovito matrix Z_i

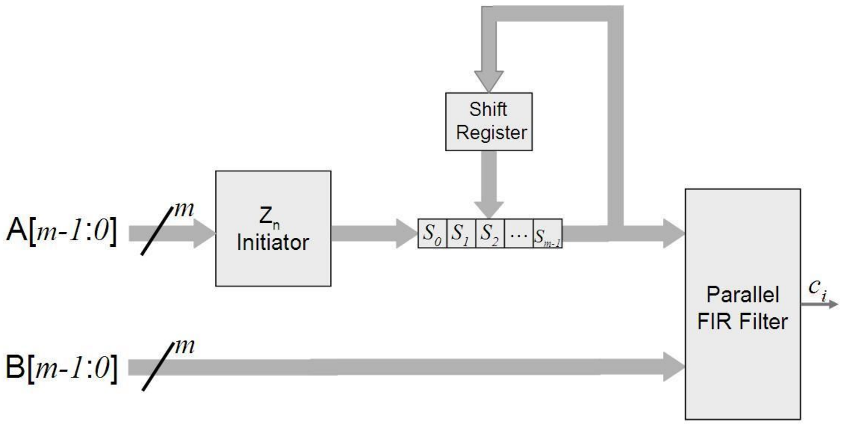


Fig. 2. Block diagram of the proposed multiplier

and the field element B are accessible at the same time. With this approach each inner product can be completed in one clock cycle.

In addition to the inner product, the entire multiplier implementation also includes the action of a shift register in each clock cycle, as is shown in figure 2. The initial row Z_n already includes the reductions required for the finite field multiplication, thus avoiding the modular reduction stage in the finite field multiplier. A template for Z_n is precomputed and then hardwired, taking advantage of subexpression sharing in order to reduce the number of operations.

One output bit of C is obtained in each cycle by multiplying (inner product performed by a FIR filter) the current row stored in register S by B. The current row is obtained as a result of right-shifting the previous row and filling the vacated position on the left with a_i.

5 Experimental Results

In this section we present a performance comparison between our finite field multiplier and other efficient FPGA implementations of finite field multiplication over $GF(2^m)$. Each of these multipliers represent elements in the polynomial basis.

Naturally, a finite field multiplication consists of a polynomial multiplication followed by a modular reduction. In [4], an efficient multiplier architecture of the type serial/parallel is presented where the modular reduction is made concurrently over each partial product, and finally all the partial products are added to obtain the final result. A similar, but more flexible architecture, is proposed in [7], where the value of the field degree can be changed and the irreducible polynomial can be configured and programmed; this feature can be achieved by implementing demultiplexers in the architecture design. In [5], the authors consider a hybrid-Karatsuba multiplier based on the Karatsuba multiplication method which reduces the number of multiplication but at the cost of increasing the number of additions and the total propagation delay. To achieve a tradeoff between area and propagation delay, a hybrid model using Karatsuba formulas combined with the classical polynomial multiplication method is proposed.

In Table 1 we compare our approach with the mentioned polynomial basis multipliers reported in [4,5,7]. The field sizes used in this experiment are the same as those used in the cited references, the only suitable benchmarks for comparisons that are known to us. However, our approach can be implemented for larger finite fields.

The works cited in Table 1 are implemented on different platforms. We have synthesized our implementation over the same target devices used in the cited references for the purpose of comparison. However our work is focused on accelerating finite field arithmetic in a high performance hardware/software environment. Our target platform is a Cray XD1 system which includes six FPGAs units tightly integrated to 12 2.2 GHz Opteron AMD processors through a high bandwidth interconnection system. FPGA units are Xilinx Virtex II-Pro xc2vp50-7.

The times in Table 1 have been measured using FPGA synthesis results reported by Xilinx tool XST (Xilinx Synthesize Technology) included in the package ISE Foundation 7.1. Our implementations are synthesized without area and timing constraints.

Table 1. Comparison of multipliers

Field	Target FPGA	Implementation	Time (μs)	Space (slices)
$GF(2^{210})$	Xilinx Virtex xcv-300-6	Reference [7]	12.30	343
		This work	2.21	334
$GF(2^{233})$	Xilinx xc2v-6000-4	Reference [5]	2.58	not reported
		This work	2.42	415
$GF(2^{239})$	Xilinx Virtex xcv-300-6	Reference [4]	3.10	359
		This work	2.47	385

According to the given results, our implementation exhibits the best time performance, whereas the area is not the most favorable for some cases. However our main goal is to achieve very fast computation using reasonably the physical devices.

Higher acceleration rates are obtained using the Cray XD1 FPGA (see Table 2). According to our results, there are significant opportunities for speed up on the Cray XD1 using reasonably the FPGA's physical space, however the communication time between CPU and FPGA becomes an obstacle. The communication model that we have used is a simple push-model in which the CPU pushes the input data to the FPGA's registers, and reads the output data from a destination register on the FPGA. Our experimental results indicate that this is a costly communication model, for example the direct multiplier for $GF(2^{63})$ spent 2.77 μs for communications and 0.62 μs for computations. Other works such as [2] have reported similar communication problems with the Cray XD1.

However, in [10] it is shown that a savvy decision about the workload assigned to the FPGA can vastly improve the communication performance on the Cray Hyper-transport I/O bus over other communication interfaces technologies.

Table 2. Comparison of multipliers on the Cray XD1 FPGA

Field	Time (μs)	Space (slices)	Space Utilization
$GF(2^{210})$	1.85	305	1.29%
$GF(2^{233})$	2.02	369	1.56%
$GF(2^{239})$	2.04	363	1.53%

6 Conclusions and Future Work

We have presented a high performing FPGA implementation of a $GF(2^m)$ multiplier. The structure of our multiplication algorithm, has allowed us to enhance performance by exploiting Mastrovito matrix symmetries while avoiding modular reductions in the multiplication process. The main operation in our multiplier is the inner product which is accelerated by bit-level parallelism, this simple architecture uses a small amount of space, and does not present opportunity for performance improvement in terms of alternative implementations.

Although our approach has shown to be fast for the finite fields reported in the previous section, it promises more favorable results for multipliers on larger fields. In order to deal with high performance reconfigurable applications such as reverse engineering genetic networks, our FPGA implementation could be used as a co-processor for accelerating a CPU-based application. This requires a judicious partitioning of the problem between high performance CPUs and FPGAs. Here the communication overhead is an important issue and we have to consider alternative ways of communication in order to improve the overall performance. The most promising approach is to shift more of the computational burden to FPGAs by embedding our multiplier into a more complex FPGA-based high performance application. This is possible since our implementation uses only from 1 to 2 percent of the FPGA area. Communication overhead can also be reduced by taking advantage of FPGA Transfer Region of Memory using the I/O subsystem proposed by [2].

References

1. D. Bollman, E. Orozco, O. Moreno, "A Parallel Solution to Reverse Engineering Genetic Networks", in Gavrilova et al (eds), Lecture Notes in Computer Science, Springer-Verlag, Part III, 3045, pp. 490-497, 2004.
2. J.Fernando, D. Dalessandro, A. Devulapalli, and A. Krishnamurthy, "Enhancing FPGA Based Encryption", Ninth Workshop on High Performance Embedded Computing (HPEC). Sept. 2005.

3. E. Ferrer, D. Bollman, and O. Moreno, “Toward a Solution of the Reverse Engineering Problem Using FPGAs,” to appear in Proceedings of the International Euro-Par Workshops, Lecture Notes in Computer Science, Springer-Verlag, Dresden, Germany, 2006.
4. M.A. Garcia-Martinez, R. Posada-Gomez, G. Morales-Luna, F. Rodriguez-Henriquez. “FPGA implementation of an efficient multiplier over finite fields $GF(2^m)$”, Proceedings of International Conference on Reconfigurable Computing and FPGAs, 2005 (ReConFig’05), September 2005.
5. C. Grabbe, M. Bednara, J. Shokrollahi, J. Teich and J. Von Zur Gathen, “FPGA Designs of parallel high performance Multipliers”, Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS-03), volume II, 268-271. Bangkok, Thailand.
6. A. Halbutogullari, Ç. Koç, “Mastrovito Multiplier for General Irreducible Polynomials”, IEEE Transactions on Computers, volume 49, number 5, pp. 503-518, 2000
7. P. Kitsos, G. Theodoridis, and O. Koufopavlou, “An efficient Reconfigurable Multiplier Architecture for Galois Field $GF(2^m)$”, Microelectronics Journal, volume 34, pp.975-980, 2003.
8. E.D. Mastrovito, “VLSI Architectures for Computation in Galois Fields”, PhD thesis, Dept. of Electrical Eng., Linkvping Univ., Linkvping, Sweden, 1991.
9. U. Meyer-Baese, “Digital Signal Processing with Field Programmable Gate Arrays”, Second Edition. Springer Verlag, Berlin, 2004
10. D. Strenski, “Computational Bottlenecks and Hardware Decisions for FPGAs”. FPGA and Programmable Logic Journal, Nov 14, 2006.
11. B. Sunar and Ç. K. Koç “Mastrovito Multiplier for All Trinomials”, IEEE Transactions on Computers”, volume 48, number 5, pp. 522-527, May 1999.

Combining Flash Memory and FPGAs to Efficiently Implement a Massively Parallel Algorithm for Content-Based Image Retrieval

Rayan Chikhi[1], Steven Derrien[2], Auguste Noumsi[3], and Patrice Quinton[4]

[1] ENS Cachan, antenne de Bretagne – Bruz Cedex, France
[2] IRISA/Université de Rennes 1 – 35042 Rennes Cedex, France
[3] IRISA/Université de Douala – Daouala, Cameroun
[4] IRISA/ENS Cachan, antenne de Bretagne – Bruz Cedex, France

Abstract. With ever larger and more affordable storage capabilities, individuals and companies can now collect huge amounts of multimedia data, especially images. Searching such databases is still an open problem, known as content-based image retrieval (CBIR). In this paper, we present a hardware architecture based on FPGAs which aims at speeding-up visual CBIR.Our architecture is based on the unique combination of reconfigurable resources combined to Flash memory, and allows for a speed-up of 45 as compared to existing software solutions.

1 Introduction

With large storage devices becoming more and more affordable, individuals and companies can collect huge amounts of information, e.g. multimedia data. Many activities such as journalism, medical diagnosis or crime prevention rely on large multimedia (mostly images) databases. To use such databases efficiently, users must be able to browse and query their images according to their *content*. These types of search operations are usually referred to as Content-Based Image Retrieval (CBIR) and they are a very active research area [9,1,3].

There are mainly two types of CBIR methods. The first one is based on the semantic content of the image. It consists in finding images in a database that match a user query (e.g. keywords) which describes the semantic content of the image sought. Such an approach requires each image to be annotated with its semantic content. This annotation can be either done manually or automatically. Most approaches for automated annotation usually rely on textual data which is associated to the image, for example, text in the enclosing web page.

The second method is based on the visual content of the image. It relies on complex image processing algorithms which extract *image descriptors* summarizing the image visual content. A typical use of this approach is digital content copyright enforcement, which is a big concern for image database copyright owners such as photo agencies. In this case, the goal of the copyright owner is to retrieve – typically from the web – all unregistered uses of its images. In such a context, the retrieval technique must be very robust to image transformations, as the original image might have undergone several transformations such as cropping, compression, color change, etc.

In this work we focus on visual content based image retrieval methods. These methods share characteristics which make them very interesting candidates for hardware acceleration :

– They suffer from prohibitive execution time. For example, searching a 30,000 image database requires 15 minutes of execution time on a standard PC workstation.
– They are computationally intensive since they rely on euclidian (i.e. L_2) distance calculation in higher dimension vectors.
– They involve very large databases: typical image databases range from a few thousands to tens of millions of images.

We propose an application-specific parallel architecture for speeding-up visual CBIR methods. This architecture is designed as a target application for the ReMIX machine, a reconfigurable accelerator aiming at content processing for large databases. The specificity of the ReMIX machine is its unique combination of high throughput, very large storage resource based on Flash technology, and of high performance FPGA technology in order to speed-up search problems.

The remaining of this paper is organized as follows. Section 2 provides background information regarding the type of CBIR algorithms we are interested in. Section 3 presents the ReMIX platform, both at the system and at the architectural level. Section 4 presents our hardware implementation strategy. Results are given and discussed in Section 5. Conclusion and future work directions are sketched in Section 6.

2 Content-Based Image Algorithms

2.1 A Short Introduction to Visual CBIR Methods

Visual Content-Based Image Retrieval consists in searching an image database to retrieve images which are visually similar to a given query image. This type of operation is based on the notion of *image descriptor*. An image descriptor can be seen as a *signature* which is computed directly from visual features of the image such as colors, shapes, contrast, etc.

Two types of descriptors can be used: *global descriptors* encode an overall property of the image such as its color histogram, while *local descriptors* are only related to specific points of interest in the image (see Fig. 1). Using such *descriptors* simplifies the use of image databases, since it allows the user to search among the descriptor database rather than the whole image database.

In this paper, we are interested in *local descriptors*, since it has been shown that they provide a more robust approach to CBIR.

2.2 Extracting Local Descriptors

Retrieving an image consists first in computing the set of descriptors of the reference image – typically, a few hundred vectors of n real components. These descriptors are extracted from small regions of the image that contains specific visual features (these

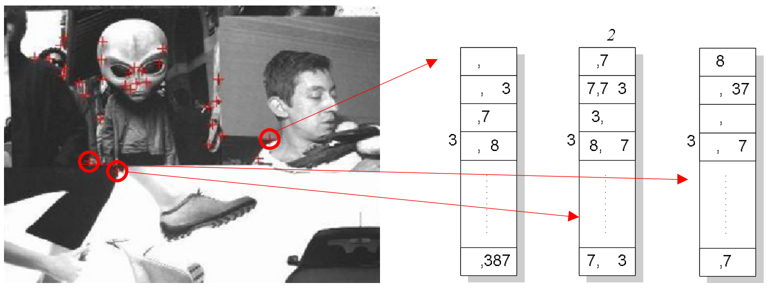


Fig. 1. Point of interest in an image and their associated local descriptors

regions are called interest points), using complex image processing algorithms (see for example Mikolajczyk et al. [10] for an overview of these techniques).

In addition to offering a concise description of the image content, these descriptors also need to be *robust* to image transformations. In other words, descriptors should be able to identify an image, even though it underwent several image transformations, such as cropping, constrast/light change, rotation, compression, etc. In our case an image is typically represented by a set of 50 to 1500 descriptors, each one being a 24-dimension real vector.

2.3 CBIR with Local Descriptors

Once extracted, the reference image descriptors (q_i) are compared to the image database descriptors (b_i) according to the metric $d(b_j, q_i)$ which corresponds to the euclidian distance (*distance calculation stage*):

$$d(b_j, q_i) \triangleq \sum_{n=0}^{24} |b_{j,n} - q_{i,n}| \quad .$$

For each reference descriptor q_i, a k-NN sorting (k-NN stands for k-nearest neighbors) selects the k database descriptors, the distances $d(b_j, q_i)$ of which are the smallest (*selection stage*). Finally, votes are assigned to images depending on their occurrences in the k-nearest neighbor lists (*election stage*): the image that has the largest number of votes is considered the best match.

As mentioned in the introduction, retrieving an image in a 30,000 image database requires about 15 minutes on a standard workstation. This is impractical for most applications of CBIR, since they often require a low response time. Research on smarter algorithms, based on clustering techniques for example, although very active, has not lead to definitive results because of a phenomenon called *curse of dimensions* which affects large databases operating on higher-dimensional data sets [1,3]. Very recently, search methods based on list-ranking have been proposed [9]. Although they offer approximate results, these methods happen to be very efficient in terms of response time. For example, using this approach, searching a 20,000 image database takes less than twenty seconds.

It is therefore questionable whether there is any interest in speeding-up the original sequential scan which is based on exhaustive search. In practice, the sequential scan search is of great use for the community that studies descriptors and search algorithms. When introducing a new type of descriptor extraction or encoding, there is a need for validating its efficiency and robustness. To obtain unbiased results, researchers need to benchmark their descriptors by using large databases which must be scanned completely to obtain exhaustive results. This is usually a very time consuming process since for each image of the database, a large number (a few hundred) of image variants are generated. Each variant is to be matched against the whole database, and this operation is repeated for a significant subset of the database. This represents a huge volume of computation (in the order of weeks or months) since the larger the database is, the more valuable the search results are.

2.4 Related Work

Accelerating CBIR on a parallel machine is the most natural choice, and has already been studied by a few authors, among whom Robles et al [14]. There has been only some work on special-purpose hardware for this type of application [7,15,11]. However, most of this work either did not address the problem in the context of a *real-life* hardware system (i.e. with its communication interface, I/O bandwidth constraints, etc.), or considered a very naive algorithm which has no interest in practice. In a previous work [12], we proposed to accelerate the CBIR algorithm on a FPGA based smart-disk architecture [5]. While this approach provided interesting performance improvement, its efficiently was greatly affected by the limited sustained hard-disk throughput. While modern hard-drive interface such as SATA offer data transfer bandwidth up to 133 MBps, the hard disk internal I/O rate when performing sequential scan is much lower. To overcome this difficulty, we propose an improved architecture which can take advantage of a very high data throughput, by handling both *distance computation* and *selection* in hardware.

3 The ReMIX Platform

3.1 Overview

The ReMIX machine is a reconfigurable accelerator targeted at content processing for very large unstructured and indexed databases. The idea behind ReMIX is to benefit simultaneously from the very high data throughput and the short access-time that can be obtained with the parallel use of Flash memory devices on the one hand, and from the high computing density of a high-end FPGA on the other hand. As such, this principle follows the philosophy of *intelligent memory* proposed by Patterson et al. in the context of the IRAM project [13].

The goal of the ReMIX architecture was to design a reconfigurable accelerator that could easily be integrated within a host system and would have the largest possible storage capability and the smallest possible random access time. We considered SRAM, DRAM, magnetic storage and FLASH memory as possible candidate storage technologies. Table 1 summarizes the characteristics of these technologies with respect to density, cost, throughput and random access time in late 2006. All numbers shown assume a 64 gigabyte memory with a 64 bit width data bus.

Table 1. Memory and storage technology in late 2006

Technology	# of chips for 64 GB	Cost	Access Time	Bandwidth	Total power
SRAM	7280	$ 123,500	5 ns	800 MBps	5250 W
SDRAM	512	$ 4,115	10 ns	2 GBps	30 W
Flash -NAND	64	$ 1,030	25 us	160 MBps	450 mW
Flash -NOR	4096	$ 72,500	100 ns	320 MBps	550 mW

These figures tell us that SRAM technology is obviously not suited to build large size memory systems. SDRAM could be a good candidate; However integrating 64 GB of SDRAM memory on a single PCB device remains a very challenging problem because of power signal integrity issues. On the other hand, NAND-Flash technology is probably the best solution: it offers storage densities above those of DRAM (in late 2006 32 Gb NAND-Flash are available, while only 1 Gb for SDRAM), and this gap is expected to grow in favor of Flash memory in the forthcoming years. Nevertheless, NAND-Flash exhibits significant differences with standard memories in the way data is accessed. In NAND-Flash data is addressed at the *page* level, each page containing between 512 byte and 2 KB. Additionally, each access suffers from a relatively important latency ($20\mu s$), three orders of magnitude higher than SRAM or DRAM (but still three orders of magnitude better than an HDD).

3.2 The ReMIX Architecture

The ReMIX system is based on a PCI board which integrates a Xilinx Virtex-II Pro FPGA coupled to 64 GB of NAND-Flash memory. This memory is organized in multiple parallel banks as illustrated in Fig. 2. Our prototype system is fully operational, and several applications in the field of bio-informatics have already been successfully ported to it [8]. A simple file system allows the user to transparently manage the content of the 64 GB Flash memory, while guaranteeing optimal throughput when accessing data during processing stage.

Porting an application to the ReMIX machine consists in designing a hardware filter which follows a simple data-driven FIFO interface. The data throughput at the filter input (e.g. at the Flash output) is approximately 640 MBps, while the filter output (e.g the host PCI bus) throughput is restricted to 5 MBps (the target PCI board only supports slave I/O). The following Section describes our hardware filter architecture, and how we accounted for these constraints during its design.

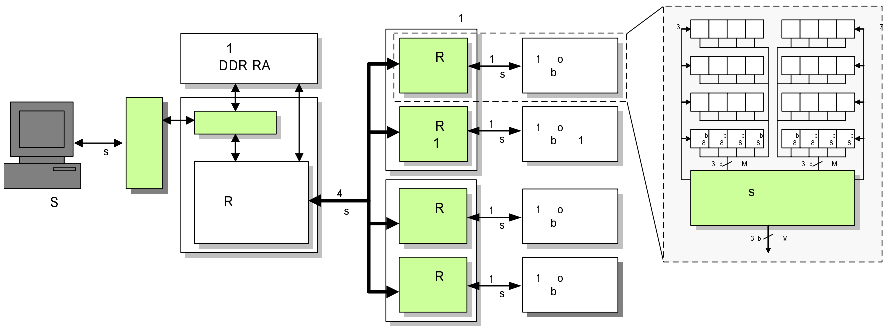


Fig. 2. RMEM card architecture

4 A Hardware Filter Architecture for CBIR

Profiling data shows that the most time-consuming step in the CBIR algorithm is the distance computation: it takes more than 98 % of the total execution time. This is not surprising: searching an image database containing B descriptors with a query image from which Q query descriptors are extracted requires $B.Q$ distance computation steps. It therefore seems natural to try to speed-up this part of the application with a dedicated hardware architecture.

4.1 Accelerating Distance Computation

The distance computation algorithm can be seen as a triple nested loop with data dependencies limited to the most inner loop (distance accumulation). This algorithm can be very easily parallelized as a 2D-systolic architecture. However, such an architecture requires accessing 24 descriptor components per cycle, while the Flash memory can only produce 8 bytes per cycle when the hardware filter is clocked at 80 MHz. Instead of this pure systolic implementation, we use a partitioned systolic linear array which is represented in Fig. 3. This architecture allows for parallel distance computation between Q fixed query descriptors q_i and every single descriptor b_j of the database, the B database descriptors being read from the Flash memory.

When a database descriptor b_j is read, distances are computed and accumulated for each descriptor component and eventually, a distance $d(b_j, q_i)$ is computed for each query descriptor q_i. In the initial software implementation of our algorithm, distances were computed using floating-point arithmetic. However, floating-point in FPGAs has major drawbacks in terms of performance and resources usage [6,4]. We have shown in [12] that using 8-bit fixed-point arithmetic for descriptor distance computation preserves the accuracy of the results. Similarly we have replaced euclidian distance by Manhattan distance since the latter allows multiplication to be replaced by a simple compare-and-add instruction. Table 2 summarizes resource usage and performance (in MHz) of a single array Processing Element for various bitwidth.

The throughput of a linear processor array of 24 PE is 74 MBps: in other words it processes a new descriptor approximately every 26 cycles, and produces 24 distance scores every 26 cycles.

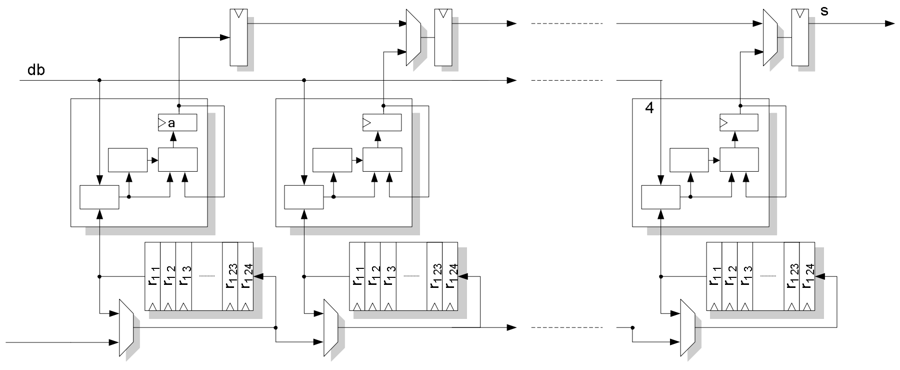


Fig. 3. Distance computation component

Table 2. Resource and performance for a distance computation PE as a function of bitwidth

Bitwidth	24 bits	16 bits	12 bits	8 bits	3 bits
Resource (Slices)	62	42	32	22	11
Frequency (MHz)	147	161	161	161	168

As we now accelerate the distance computation, it is the selection which is likely to become a performance bottleneck. Indeed, because of the limited output bandwidth available at the filter IP block output port, it is not reasonable to forward all distance scores to the embedded processor or to the host CPU, since our IP would be slowed down by I/O stalls. We thus propose to implement the selection entirely in hardware, so that it can be integrated within the filter IP. With this approach, the filter simply forwards the content of the k-NN lists to the host for the *election step*.

4.2 Accelerating Selection Using Hardware

The selection consists in sorting distance scores and in retaining the k best distances. In the software implementation, selection is implemented as a periodic sorting: all distance scores that may be part of the final list, – that is to say, all the distance scores that are below at least one item of the current list – are stored in a buffer. Once full, this buffer is sorted, and is merged with the previous list to form the updated list. This approach is very efficient in practice and profiling data show that selection represents less than 1% of the total execution time.

Implementing sorting in hardware is a well-studied problem, and several highly parallel solutions have been proposed, ranging from sorting networks [2] to systolic arrays.

- Sorting networks are very efficient to sort data that enter the sorter in parallel: networks structure of spatial complexity $O(n \log n)$ can sort n tokens every cycle.
- Systolic sorting is more suited to sort data-streams that enter the processor array sequentially. Sorting is then done in $O(n)$ time on a systolic array with n processors.

As distance scores are produced by the distance computation component at a rate of one score per cycle, selection must be done on the fly. Moreover, we are only interested in the k lowest distance scores, where k is very small as compared to the total number of scores n produced by the distance computation step. These observations suggest that neither sorting networks nor systolic sorters are appropriate solutions. One could think of using a modified systolic sorting array with only k processors, however this approach still requires important hardware resources. Another important observation is that, once the steady regime is reached, and since $n \gg k$, only very few distance scores would walk past the first processor of a systolic sorter array[1]. This would result in a highly inefficient architecture in which processors would remain idle most of the time.

Instead of using systolic sorting, we therefore propose to implement selection as a simple insertion sorting. Fig. 4 represents the insertion sorting datapath, which consists of a dual-port on-chip memory associated to a simple comparator. This architecture can handle up to 32 lists, which is enough since the output flow of our distance stage consists in distance scores related to 24 distincts query descriptors.

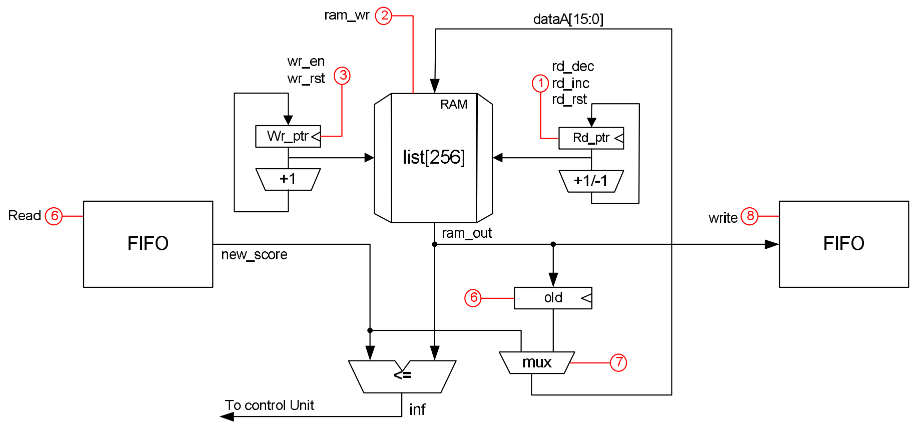


Fig. 4. Structural view of the sorting element

This solution is highly inefficient from a theoretical point of view, since its time complexity for obtaining the k-NN from a set of n distance scores is $O(n.k)$. However, its practical complexity remains very close to $O(n)$: the vast majority of scores (more than 99%) are not to be inserted into the k-NN list and would just pass through the insertion sorting step with an overhead of a few cycles. Even so, this overhead is still unacceptable, since it happens for each distance score. To remove this overhead, we perform a preliminary filtering step which buffers potential matches into a FIFO, as described in Fig. 5.

The resulting selection component uses 135 FPGA slices and one RAM block and its maximum operating frequency is 200 MHz. It allows each new score to be inserted at the n-th position in the k-NN list in $n + 3$ cycles.

[1] Note that the software implementation takes advantage of this property to skip distance calculation whenever the current score is above its corresponding k-NN list threshold score.

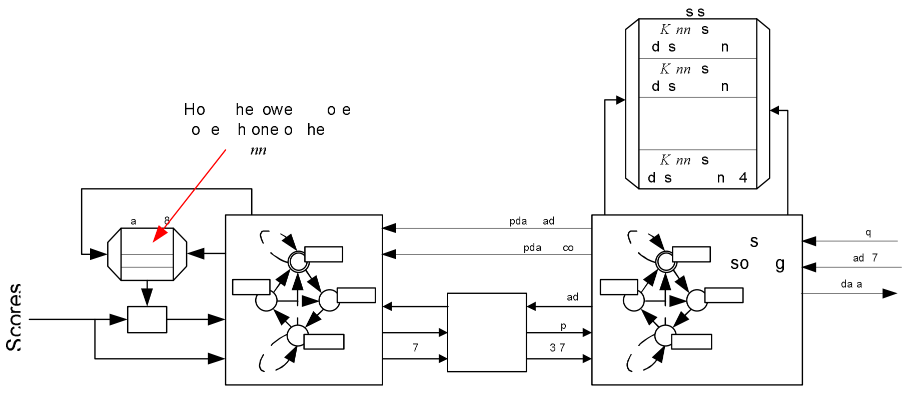


Fig. 5. Structural view of the selection component

4.3 Putting It Altogether

The Xilinx Virtex-II Pro FPGA used in the ReMIX machine has 15,000 slices and approximately 50% of them are used by the ReMIX controller. It is therefore possible to take advantage of the available space by implementing several *lines* of distance computation and selection units which operate in parallel, as shown in Fig. 6. In this execution scheme, each line is given a different subset of query descriptors, and they all process the same database subset in parallel.

Each line contains 24 query image descriptors. Therefore, each time the entire database is read, the filter computes in parallel the nearest neighbors of 240 query descriptors, i.e. in average $1/3$ of an image descriptors set. When the whole database has been processed, the lists of query descriptors are flushed out separately through a simple pipeline mechanism.

Table 3 compares various bitwidth implementations, in terms of maximum number of lines, with the maximum frequency and slices usage for each case. This shows that with 8 bits descriptors and a ReMIX architecture clocked at 80 MHz, 10 such lines can be instantiated.

5 Experimental Results

In this Section, we present the performance model that guided our design choices, and then we compare the estimations obtained from this model to the actual performance results.

5.1 Theoretical Performance Model

Let N_d be the number of descriptors in our database, T_c the filter clock period (set to 12.5 ns in our tests), N_{PE} the total number of distance processors in the design (here $N_{PE} = 240$) and k the number of items in each k-NN list (we have $k = 32$). Knowing

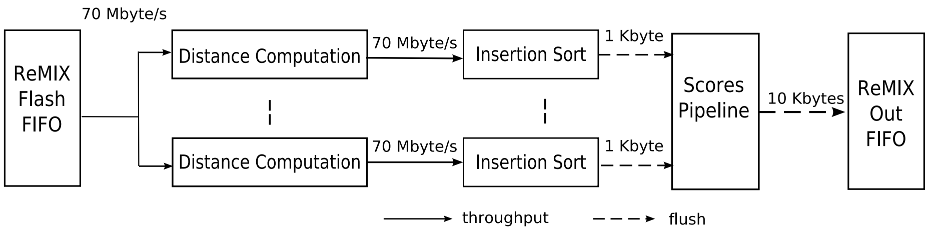


Fig. 6. Filter as integrated in ReMIX

Table 3. Resource and performance for the maximum number of processor lines as a function of bit-width, with Synplify used for synthesis and Xilinx ISE8.1 used for back-end

Bitwidth	24 bits	16 bits	12 bits	8 bits	3 bits
Nb lines	4	5	7	10	16
Resource (Slices)	7037 (98%)	6206 (93%)	6865 (97%)	7180 (99%)	7078 (99%)
Frequency (MHz)	124	129	131	84[2]	125

that our implementation of distance computation proceeds at the rate of 1 descriptor every 26 cycles, the time T_{calc} required to perform the distance computation step is given by $T_{calc} = 26N_dT_c$. However, our performance model must also account for several performance overheads.

For example, the whole descriptor database cannot be read from the Flash memory in a single pass, it is therefore necessary to split the database into N_c chunks. This induces a chunk access overhead T_{chunk} the value of which can only be determined experimentally (our performance model will therefore ignore this overhead).

On the other hand, whenever a FIFO reaches its maximum capacity, the distance computation component must be stalled to avoid data losses. Quantifying the overhead due to these stalls (we write T_{stalls} this overhead) would require to have a precise model for the FIFO usage over execution time. However, a simple probabilistic reasoning can help us to obtain a higher bound for T_{stalls}.

Let us consider a scenario in which there is no FIFO between the pre-filtering step and the insertion sort. In such a situation, each *match* causes the distance computation component to stall until the *match* is correctly inserted in its corresponding k-NN list.

We write p_{match} the probability for a distance score to not be pre-filtered before insertion. We know from software profiling that this value is close to $p_{match} = 2.10^{-5}$. We also know that our insertion sort component has a worst-case execution time of $k+3$ cycles. The percentage of stall cycles in this scenario can therefore be bounded by:

$$p_{stall} = \frac{p_{match}}{p_{match}(k+3)+1} \quad .$$

The overhead (that we can write as $T_{stall} = p_{stall}T_{calc}$) in this scenario can be considered as a higher bound of what we would observe in practice, since we use a 256-slot FIFO buffer between pre-filtering and sorting steps.

[2] Given 10 lines, any frequency optimization overflows the number of available slices.

Finally, we must also account for the time spent flushing out the k-NN lists (we write T_{flush} this overhead). This step has a fixed impact on the global execution and can be written as :

$$T_{flush} = k.\frac{N_{PE}}{B_{PCI}}$$

where B_{PCI} stands for the actual output PCI bandwidth (here we have $B_{PCI} =$ 5 MBps). The total search time T_{search} for a query image containing N_q descriptors can then be written as :

$$T_{search} = \left\lceil \frac{N_q}{240} \right\rceil (T_{calc} + T_{stalls} + T_{flush})$$

5.2 Measured and Projected Performance

We have benchmarked our design over a real-life 30,000 images database consisting in 20,868,278 descriptors. This database normally requires 2 GB of storage, however, thanks to the use of 8 bit fixed-point arithmetic instead of single precision floating-point, this size was reduced to 650 MB.

Using the performance model obtained, we estimate that searching a 30,000 image database with a 720 descriptors query leads to a search time of $T_{search} = 19,68\,s$. Running the same search on the actual ReMIX system lead us to an observed search time of 20.43 seconds, that is within a 4 % error margin of the predicted performance.

The actual speed-up factor over the original software implementation is 45. In other words a single ReMIX system is as efficient as a 45 PCs cluster. While this acceleration factor only holds for descriptors encoded as 8 bits integers, we estimated the corresponding results for different bitwidth by implementing as many processor lines as possible on the FPGA and use this result to estimate the corresponding speed-up. These performance projections are summarized in Table 4.

Table 4. Estimated search time and corresponding speed-up for varying bitwidth for a 30.000 images database

Bitwidth	24 bits	16 bits	12 bits	8 bits	3 bits
Query Time (sec)	50	40	28.5	20	12.5
Speed-up factor	18	22	31	45	72
Number of lines	4	5	7	10	16

6 Conclusion

In this paper we have proposed a hardware accelerator for Content-Based Image Retrieval based on local descriptors. Our architecture performs both the distance calculation and selection in hardware, and was experimentally validated on the ReMIX machine using a real-life image database.

A single RMEM PCI board provides similar performance to a 45 node PC cluster, at $1/20$ the price (without accounting for the host PC). An interesting point is that our hardware architecture can easily be targeted at different descriptors (e.g. for different precision, or higher number of dimension), with moderate design efforts.

Directions for future work include studying the implementation of CBIR on GPUs which appear to be an interesting platform to speed-up this type of algorithms.

References

1. Laurent Amsaleg and Patrick Gros. Content-Based Retrieval using Local Descriptors: Problems and Issues from a Database Perspective. *Pattern Analysis and Applications*, 2001.
2. K.E. Batcher. Sorting Networks and their Applications. In *Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968.
3. Christos Faloutsos. *Searching Multimedia Databases by Content*. Kluwer Academic Publishers, 1996.
4. Gokul Govindu, Ling Zhuo, Seonil Choi, and Viktor Prasanna. Analysis of High-performance Floating-Point Arithmetic on FPGAs. In *Reconfigurable Architecture Workshop*, 2004.
5. Stéphane Guyetant, Mathieu Giraud, Ludovic L'Hours, Steven Derrien, Stephane Rubini, Dominique Lavenier, and Frédéric Raimbault. Cluster of Reconfigurable Nodes for Scanning Large Genomic Banks. *Parallel Computing*, 2005.
6. Walter B. Ligon III, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.
7. L. Kostoulas and I. Andreadis. Parallel Local Histogram Comparison Hardware Architecture for Content-Based Image Retrieval. *Journal of Intelligent and Robotic Systems*, 2004.
8. Dominique Lavenier, Xinchun Liu, and Gilles Georges. Seed-Base Genomic Sequence Comparison Using a FPGA/FLASH Accelerator. In *To appear in Proceedings of EEE International Conference on Field Programmable Technology*, 2006.
9. Herwig Lejsek. A case-study of scoring schemes for the pvs-index. In *CVDB '05: Proceedings of the 2nd international workshop on Computer vision meets databases*, pages 51–58, New York, NY, USA, 2005. ACM Press.
10. Krystian Mikolajczyk and Cordelia Schmid. A Performance Evaluation of Local Descriptors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(10):1615–1630, 2005.
11. Koji Nakano and Etsuko Takamichi. An Image Retrieval System Using FPGAs. In *Proceedings of ASPDAC*, 2003.
12. Auguste Noumsi, Steven Derrien, and Patrice Quinton. Acceleration of a Content-Based Image-Retrieval Application on the RDISK Cluster. In *International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006.
13. David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17(2):34–44, 1997.
14. Oscar D. Robles, José L. Bosque, Luis Pastor, and Angel Rodríguez. Performance Analysis of a CBIR System on Shared-Memory Systems and Heterogeneous Clusters. In *IEEE International Workshop on Computer Architectures for Machine Perception (CAMP'05)*, 2005.
15. Constantinos Skarpathiotis and K.R. Dimond. A Hardware Implementation of a Content-Based Image Retrieval Algorithm. In *International Conference on Field Programmable Logic and Application (FPL)*, January 2004.

Image Processing Architecture for Local Features Computation

Javier Díaz[1], Eduardo Ros[1], Sonia Mota[2], and Richard Carrillo[1]

[1] Dep. Arquitectura y Tecnología de Computadores, Universidad de Granada, Spain
[2] Dep.Informática y Análisis Numérico, Universidad de Córdoba, Spain
{eros, jdiaz, rcarrillo}@atc.ugr.es, smota@uco.es

Abstract. Quadrature filters are widely used in the Computer Vision community because of their biological support and also because they allow an efficient coding of local features. They can be used to estimate local energy, phase, and orientation or even allow classifying image textures. The drawback of this image decomposition is that requires performing intensive pixel-wise computations which makes it impossible to use in most real-time applications. In this contribution we present a high performance architecture capable of extracting local phase, orientation and energy at a rate of 56.5 Mpps. Taking into account that FPGA resources are constrained, we have implemented a steerable filters bank (instead of Gabor filters bank) based on Second Order Gaussian derivatives. This method has the advantage that the filters are 2-D separable and each image orientation can be extracted from a basic set of seven filters. We present in this paper the proposed architecture and analyze the quantization degradation error generated by using fixed point arithmetic. We show the resources consumption, the performance and finally, we present some results from the developed system.

Keywords: Real-time image processing, quadrature filters, local image phase, orientation and energy, fixed point arithmetic.

1 Introduction

The analysis of image features such as colour, edges, corners, phase, orientation or contrast provides significant cues in the process of image understanding. They are used as base for higher level task such as object segmentation, object recognition, texture analysis, motion and stereo processing, image enhancement and restoration or special effects [1], [2].

In this contribution we focus on three basic image properties, phase, energy and orientation. They have been extensively used on computer vision [3], [4] and, as we will see, they can be obtained from the same set of image operations.

Local orientation is an early vision feature that encodes the geometric information of the image. It has an ambiguity in the representation of local orientation coming from the two possible directions, for each orientation because it is a 180º periodic measurement. We solve this uncertainty using the double angle representation proposed by G. H. Granlund in [5], which is to simply double the angle of each orientation estimate.

Energy and phase information has been widely used at the literature. Energy informs us about the areas of higher contrast (and therefore is a luminance-depending feature). Phase information, as it is stated in the literature [6], is more stable against change on contrast, scale, or orientation. It can be used to interpret the kind of contrast transition at its maximum [7]. It has been applied to numerous applications, especially for motion [8], [9], and stereo processing [10], [11].

These three features can be extracted using the same preliminary operations using Quadrature filters. They are based on symmetric and antisymmetric pairs such as Gabor filters [12] or Gaussian derivatives [13] and allow decomposing the image into structural and luminance information.

This contribution has been structured as follows. Section 2 presents the computer vision theory required for the development of the presented architecture. Section 3 focuses on the evaluation of the accuracy degradation due to the utilization of fixed point arithmetic. Section 4 will cover the hardware architecture details, the resources and the performance evaluation. Finally, Section 5 describes our main conclusions and future work.

2 Quadrature Filters Based on Steerable Gaussian Derivatives for Local Orientation, Energy and Phase Computation

A Quadrature filter is a complex filter that allows decomposing each output as phase and magnitude. If we note ***h****(x,*k*)* for a complex filter tuned to a spatial frequency f_0 along the *x* axe then:

$$h(x; f_0) = c(x; f_0) + js(x; f_0) \tag{1}$$

Where *c* and *s* respectively represent the even and odd components of the filters, fulfilling the condition that the even and odd filters are Hilbert transforms of each other. The convolution with the image *I(x)* is expressed in equation (2):

$$I * h = \int I(\xi) h(x-\xi; k_0) d\xi = C(x) + jS(x) = \rho(x) e^{j\phi(x)} \tag{2}$$

Where ρ(x) denotes its *amplitude (*that we will also note *as magnitude* and *energy* when we refer to its square value), ϕ(x) is the phase and the components C(x) and S(x) will represent respectively the responses of the even and odd filters.

Although local energy can be easily extended to a bidimensional signal, phase information is intrinsically one-dimensional. Therefore, phase information only can be computed for image areas which can be locally approximated as lines or edges and therefore, their main orientation needs to be estimated. An extensive analysis of the available theories can be found in [14]. As conclusion, orientation needs to be computed to properly estimate phase. This relation between features makes Quadrature filters based on Gabor or Gaussian derivatives a good approach because they are oriented filters which analyze the image at discrete orientations.

Furthermore, Quadrature filters are based on convolutions which are hardware friendly operations which fit quite well on FPGA devices.

2.1 Steerable Gaussian Derivatives

Concerning the constraints of real-time embedded systems we will focus on Gaussian derivatives because they represent a good trade-off between accuracy and resources consumption. Gaussian derivatives allow the computation of any particular orientation based on a basic set of separable kernels, this is usually referenced as stereability [13].

In our architecture we use the second derivative order with a 9 taps kernel. The peak frequency is f_0=0.21 pixels^{-1} and bandwidth β= 0.1 pixel·s^{-1}.As presented on [13], this derivative order is able to provide high accuracy at a reasonable system cost. Kernel equation can be found in [13] and its shape is presented in Fig. 1. Note that one of the advantages of Gaussian derivatives (compared to Gabor filters) is that they are computed from a set of 2-D separable convolutions which significantly reduces the required resources.

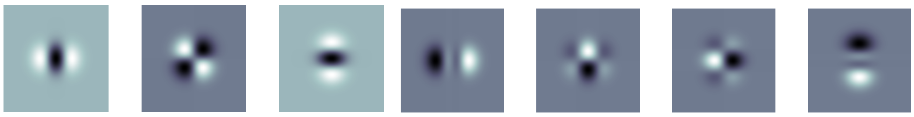


Fig. 1. Second order Gaussian derivatives separable base set. From left to right, filter G_{xx}, G_{xy}, G_{yy}, H_{xx}, H_{xy}, H_{yx}, H_{yy}. Subindexes stand for the derivative variables, G indicates Gaussian filter and H their Hilbert transforms. With this set of filters, we can estimate the output at any orientation just combining the base set output. This allows building oriented Quadrature filters banks at any desired orientation.

After convolving the image with the filters presented in Fig. 1, we steer this filter to estimate the Quadrature filters at some predefined orientations. In our system, we compute the Quadrature filters output at 8 different orientations which properly sample the orientation space (further details c.f. [14]). The filters steering is done following equation (3) where θ stands for the orientation.

$$\begin{aligned} c_\theta &= \cos^2(\theta)G_{xx} - \cos(\theta)\sin(\theta)G_{xy} + \sin^2(\theta)G_{yy} \\ s_\theta &= \cos^3(\theta)H_{xx} - 3\cos^2(\theta)\sin(\theta)H_{xy} - 3\cos(\theta)\sin^2(\theta)H_{yx} + \sin^3(\theta)H_{yy} \end{aligned} \quad (3)$$

The computation stages described in this section can be summarized as follows:

1. Second order Gaussian derivatives G_{xx}, G_{xy} and G_{yy} and their Hilbert transforms H_{xx}, H_{xy}, H_{yx} and H_{yy} computation, using separable convolutions.
2. Linear filters steering at 8 orientations using equation (3).

After these two stages for each pixel we have 8 complex numbers corresponding to the different orientations and even-odd filter outputs pairs. Section 2.2 describes how to interpolate this data to properly estimate the orientation and then the phase and energy at this local image orientation.

2.2 Features Interpolation from a Set of Oriented Filters

If we consider 8 oriented filters (computed using Gabor or Gaussian Derivatives), is likely that the local orientation of some features do not fit this discrete number of orientations. Under this circumstance, we require to interpolate the feature values computed from this set of outputs. In this paper we will focus in the tensor-based Haglund's approach [15]. Based on a local tensor that projects the different orientations, information can be computed as described in equations (4-6). Note that these equations, as described in [14], are a modification of the presented ones in [15] in order to improve the hardware feasibility. In these equations, j stands for the complex unit and *i* for the orientation.

$$E_{local} = \frac{\sum_i E_i}{N} \tag{4}$$

$$\theta_{local} = \frac{1}{2}\arg\left(\sum_i \frac{4}{3}\left(c_i^2 + s_i^2\right)\exp\{j2\theta_i\}\right) \tag{5}$$

$$\begin{aligned} c &= \sum_i c_i \\ s &= \sum_i s_i \\ P_{local} &= \arctan\left(s/c\right) \end{aligned} \tag{6}$$

3 Bit-Width Analysis and Evaluation of the Accuracy Using Fixed Point Arithmetic

The effect of the arithmetic representation and bit-width quantization on the embedded system design is addressed in this section. On one hand, current standard processors use floating point representation which has very large data range and precision. On the other hand, digital circuits usually use fixed point representation with a very restricted number of bits. Unfortunately, embedded systems with specific circuitry for each computation can only afford floating representation at critical stages [16] and most of the computation should be done using fixed point arithmetic. The quantization effects should be properly studied to ensure that the data dynamic range is properly adjusted and the bit quantization noise is kept at a small value. Furthermore, we need to reduce the data variables bit-width to limit resources consumption.

Each of the algorithm main stages requires a careful substages bit-width design but here we are only interested on highlighting the main designing decisions. Therefore, on this section we focus on the required bit-width of the following stages:

1. Convolution kernel weights of the G_{xx}, G_{xy}, G_{yy}, H_{xx}, H_{xy}, H_{yx} and H_{yy}.
2. Convolution outputs results of this filter bank.

3. Oriented quadrature filters after steering operations.
4. Trigonometric functions (*sin*, *cos* and their powers or multiplications) that are used at the steering and interpolation stages.
5. Main non linear operations: division and *arctan*.
6. Estimated goal features: local energy, phase and orientation.

This analysis has been performed using the tool *MCode for DSP Analyzer* described in [14]. The error metric for evaluating the system degradation has been the *Signal to Quantization Noise Ratio* SQNR.

Relative to the first point of the list, convolution kernel weights, the methodology is the following. We keep all the computation using double precision floating point representation but the kernel values and we compare the results with the complete software version. The results are shown on Fig. 2.

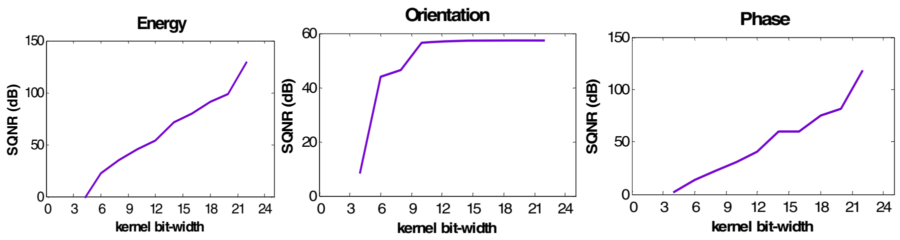


Fig. 2. System SQNR vs. Gaussian derivatives kernels quantization. Energy and phase have approximately linear behavior. For orientation, bit-widths larger than 11 bits do not produce any significant accuracy improvement. This is due to the fact that, on the hardware simulator we use energy instead of magnitude as weight for equation (6) that makes not possible to achieve higher SQNR than 57.4 dB. Nevertheless, there is not significant error degradation and the quality of the system is very high already using 11 bits in this stage.

Our first conclusion is that, values higher than 11 bits for the kernels do not improve the accuracy on orientation but they do for energy and phase features. The next stage is the analysis of the bit-width used to store the output produced after convolution with these quantized kernels. These results are illustrated in Fig. 3. Different kernels bit-widths are represented with different curves for each image feature. On this experiment we have considered (without loss of generality) 8 bits for the integer part and a variable number of bits for the fractional part of the convolution outputs (represented in the x axis). Different plots represent different bit-widths used for the kernel weights mask. This is a general method because a proper scaling was used.

The main conclusion is that, in order to take full advantage of larger bit-widths of the convolution kernels, a larger bit-width should be used for the convolutions output.

At this point we need to define some accuracy goals. We consider acceptable SQNR values larger than 30 dB. On Table 1 we show two SQNRs of our system for

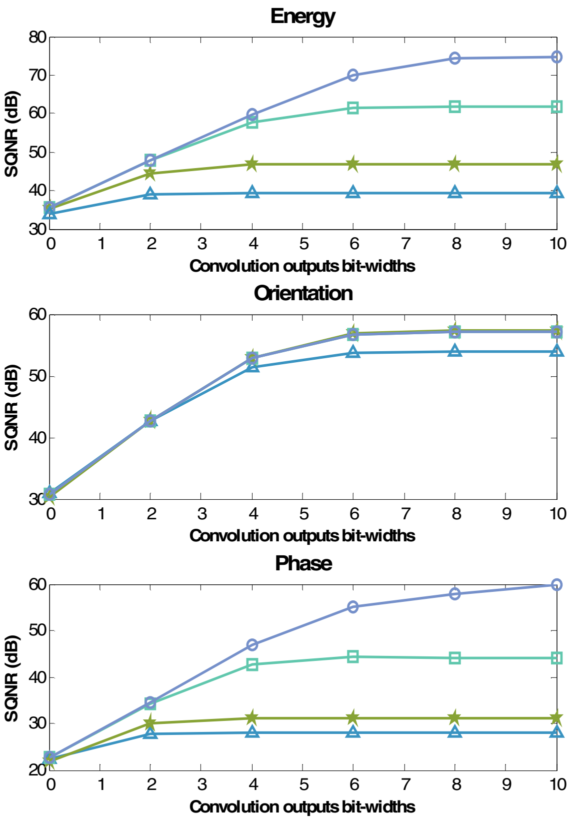


Fig. 3. SQNR (db) evolution for different bit-widths configuration for kernel and convolution outputs quantization. The y-axe represents the SQNR values and x-axe the convolution outputs fractional part bit-widths. The integer part is fixed to eight bits. Four different curves are presented at each plot. They represent different kernels bit-width configurations: triangles represent 9, stars 11, squares 13 and circles 15 bits to store the convolution outputs. Note that in the orientation plot top curve combine the starts, squares and circles data.

two well balanced low area decision cases. Already in Fig. 2 was shown that the phase estimation is the most demanding (in terms of bit-width) to achieve high SQNR (phase requires larger bit-width to achieve the same SQNR). For instance with 11 bits we obtain approximately SQNR=40 dB.

The results presented in Table 1 show that the most sensible stages are the kernel and convolution outputs quantization. Trigonometric, non linear function as well as the image features themselves can be quantized with a relatively small number of bits without extra accuracy degradation as can be deduced from the SQNR results. From this table we also conclude that fixed point arithmetic can be properly used for the system design on embedded systems due to the high SQNR obtained.

Table 1. SQNR at different stages for two well defined configuration examples. Example A uses 11 bits for the kernel quantization and 9 bits for the convolution stage (only one fractional bit). Example B uses 13 bits for the kernels and 11 for the convolution outputs. Each stage (0 to 3) represents the SQNR of the Energy, Orientation and Phase after consecutive bit-widths quantizations. At stage 0, only the kernels and convolution bits have been quantized. Stage 1 represents the SQNR after the previous quantization plus the quantization of the oriented quadrature filters with the same number of bits than the convolution outputs. Stage 2 shows the results with the addition of the next quantization stage, the trigonometric and non linear functions (*arctan* and division). Trigonometric functions require only 9 bits. *Arctan* function uses (2·bit-width-convolution outputs+2) bits to avoid degrading the system. Finally, stage 3 shows the SQNR of the whole system quantized using fixed-point data representation. Orientation and phase are angles and therefore their dynamic range is well defined (0 to π for orientation and -π to π for phase). We have used 9 bits to represent their values. For the energy, the dynamic range depends on the convolution outputs bit-widths. We have used (2·bit-width-convolution outputs) bits to achieve satisfactory results.

Stage	**SQNR (dB)**	**Energy**	**Orientation**	**Phase**
A .0	*Kernel and convolution outputs quantization*	40.739	36.323	27.036
A. 1	*Orientation Quadrature filters quantization*	39.853	35.404	26.061
A. 2	*Trigonometric and non linear functions quantization.*	39.849	35.403	26.061
A. 3	*Image features results quantization.*	39.842	35.027	26.008
B. 0	*Kernel and convolution outputs quantization*	53.234	48.238	39.194
B. 1	*Orientation Quadrature filters quantization*	48.560	46.923	40.512
B. 2	*Trigonometric and non linear functions quantization.*	48.557	46.922	40.512
B. 3	*Image features results quantization.*	48.550	46.272	40.163

4 System Architecture

We have designed a very fine grain pipeline architecture whose parallelism grows along the stages to keep our throughput goal of one pixel output for clock cycle. The main circuit stages are described on Fig. 4, each of them finely pipelined to achieve our throughput goal. Note that the coarse parallelism level of the circuit is very high (for example state S_1 requires 16 parallel paths, eight for the orientation and 2 filter types, even and odd each). The stage S_0 is basically composed of 7 separable convolvers corresponding to the convolutions with G_{xx}, G_{xy}, G_{yy}, H_{xx}, H_{xy}, H_{yx} and H_{yy} kernels. The separability property of the convolution operation allows computing first the convolution by rows and after that by columns. This configuration requires only 8 double port memories shared by all the convolvers. They are arranged storing one image row at each memory to allow parallel data access at each column. This stage S_0 is finely pipelined in 24 microstages as indicated in brackets in the upper part of Fig. 4.

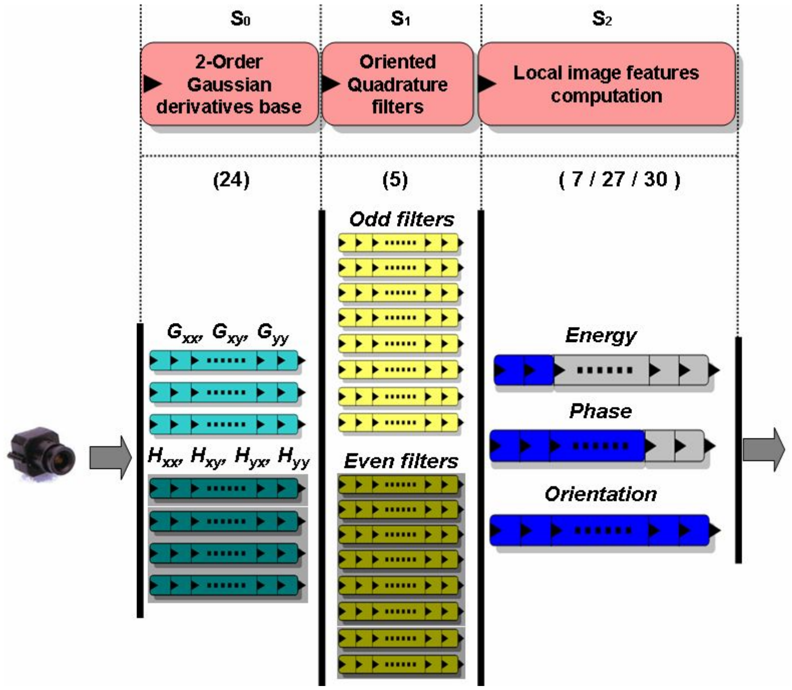


Fig. 4. Image features processing core. Coarse pipeline stages are represented at the top and superpipelined scalar units at the bottom. The number of parallel datapaths increases according to the algorithm structure. The whole system has more than 59 pipelined stages (without counting other interfacing hardware controllers such as memory or video input/output interfaces). This allows computing the three image features at one estimation per clock cycle. The number of substages for each coarse-pipeline stage is indicated in brackets in the upper part of the figure.

Stage S_1 is used to compute the oriented Quadrature filters. This is achieved by multiplying each output of stage S_0 by some interpolation weights as shown on equation (3). These coefficients are stored using distributed registers for parallel access. In a sequential approach they could be stored using embedded memory but the required data throughput makes necessary full parallel access to all of them making hard to implement this configuration.

Stage S_2 computes the image features from this set of even and odd complex filters. Figure 4 shows the three different computations which calculate the local features of orientation, phase and energy.

Note that the last stage S_2 has different latencies (7, 27 and 30) for each feature (Energy, phase and orientation). Nevertheless, since we need all of them at the same time we use delay buffers to synchronize the outputs.

4.1 Hardware Resources Consumption Analysis

Based on the analysis of Section 3, we have chosen the bit-widths described in Table 1, configuration B. The study described in previous sections can be summarized as follows. We have used 13 bits for the kernels coefficients and 11 for the convolution outputs. The trigonometric LUT values use 9 bits. Intermediate values are properly scaled to keep the data accuracy. The final orientation and phase results use 9 bits and 22 bits for the energy. The final achieved SQNR of the Energy, Orientation and Phase after the full quantization process are 48, 46 and 40 respectively. This accuracy fulfils the requirements to utilize the system as coprocessor without any significant degradation. Figure 3 shows that the hardware resources for this approach represent a good trade-off between accuracy and resources consumption. The whole core has been implemented on the RC300 board of Celoxica [17]. This prototyping board is provided with a Virtex II XC2V6000-4 Xilinx FPGA as processing element including also video input/output circuits and user interfaces/communication buses. The final required hardware consumption and performance measures of our implementation are shown in Table 2.

Table 2. Complete system resources required for the local image features computing circuit. The circuits have been implemented on the RC300 prototyping board [17]. The only computing element is the Xilinx FPGA Virtex II XC2V6000-4. The system includes the image features processing unit, memory management unit, camera Frame-grabber, VGA signal output generation and user configuration interface. Results obtained using the Xilinx ISE Foundation software [18]. (*Mpps*: *mega-pixels per second* at the maximum system processing clock frequency, *EMBS*: embedded memory blocks).

Slices / (%)	EMBS / (%)	Embedded multipliers / (%)	Mpps	Image Resolution	Fps
9135 (27%)	8 (5%)	65 (45%)	56.5	1000x1000	56.5

Detailed information about the image features core and the coarse pipeline stages are shown in Table 3. Note that the sum of the partial stages is not the total number of resources. This can be easily explained based on the synthesis tools. For the whole system is more likely that the synthesizer engine can share some resources and reduce hardware consumption. This also explains why the whole system has lower clock frequency than the slowest subsystem.

The previous argument does not explain why the number of multipliers grows on the complete system with respect to the sum of the multipliers used at each substage. We have used automatic inference of multipliers on the system. Because the number of multiplications by constants is quite high, it is difficult to manually determine when is better to use an embedded multiplier or compute it using FPGA logic. Synthesis tools define cost functions that are able to evaluate this based on technological parameters for each multiplication which is the best option. We have compared the manual and the automatic inference of multipliers and, though differences are not large, automatic inference usually achieves higher clocks rates than manual generation with slightly less logic. This automatic inference changes depending on the circuit that is being synthesized and produces the differences presented in Table 3.

The final qualitative results for a test image are shown in Fig. 5. Though there are significant differences in the bit-width of the software and hardware data, the high SQNR guarantees high accuracy with imperceptible degradation.

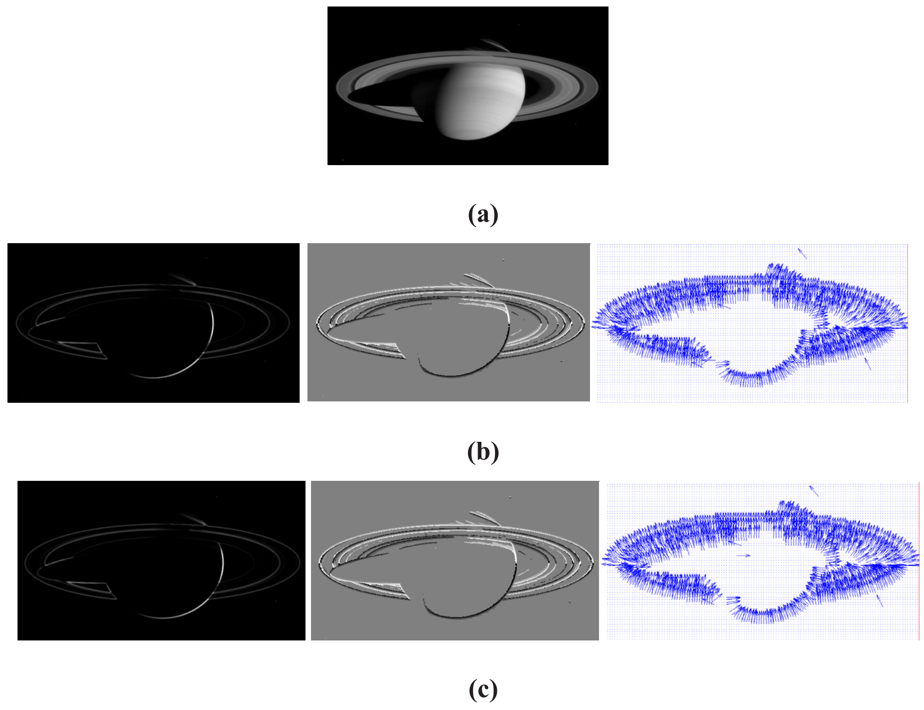


Fig. 5. Example of system results for Saturn image. (a) Original image. (b) From left to right, local energy, phase and orientation computed with double floating point precision. Energy and phase are encoded using gray levels. Orientation is represented by oriented arrows. (c) From left to right, local energy, phase and orientation computed with our customized system. Note that the differences are practically indiscernible though hardware uses much more constrained data bit-width.

Table 3. Partial system resources required on a Virtex II XC2V6000-4 for the coarse pipeline stages described for this circuit. Results obtained using the Xilinx ISE Foundation software [18]. (*EMBS* stands for embedded memory blocks). The differences between the sum of partial subsystems and the whole core are explained in the text.

Circuit stage		Slices / (%)	EMBS / (%)	Embedded multipliers / (%)	f_{clk} (MHz)
S_0	*Gaussian base convolutions*	4,170 (12)	8 (5)	50 (34)	85
S_1	*Oriented quadrature filters*	1,057 (3)	0	0	69
S_2	*Features Energy, Phase and Orientation computation*	2,963 (8)	0	6 (4)	89
	Whole processing core	7627 (22)	8 (5)	65 (45)	58.8

5 Conclusions

We have presented a customized architecture for computation of local image features: energy, phase and orientation. The presented implementation combines a well justified bit-width analysis with a high performance computing architecture. The circuits SQNR is very high (more than 40 dB) which provides a high accuracy computation.

The computation performance allows processing more than 56 Mpixels per second, where the image resolution can be adapted to the target application. For instance, using VGA resolution of 640x480 pixels, the device can compute up to 182 fps. This is achieved thanks to the fine grain pipeline computing architecture and to the highly parallelism employed. This outstanding power performance is required for the target specifications addressed for instance on complex vision projects such as DRIVSCO [19]. Note that, even current standard processors running at more than 3 GHz, 60 times faster than the presented FPGA (which runs at 50 MHz) hardly process this model in real time. The superscalar and superpipelined proposed architecture fulfils our requirements and illustrates that, like biological systems, even with a low clock rate, a well defined architecture can achieve an outstanding computing performance unviable with current processors.

Future work will cover the implementation of this system using image analysis at different resolutions, trying to properly sample the spatial frequency spectrum of the images. We also plan to evaluate different accuracy vs. resources consumption trade-off depending on the target application to properly tune the proposed architecture for specific application fields.

Acknowledgment

This work has been supported by the Spanish project National Project DEPROVI (DPI2004-07032) and by the EU grant DRIVSCO (IST-016276-2).

References

[1] Gonzalez and Woods: Digital Image Processing. 2nd Edition, Prentice Hall, (2002).

[2] Granlund, G. H. and Knutsson, H.: Signal Processing for Computer Vision. Norwell, MA: Kluwer Academic Publishers, (1995).

[3] Krüger, N. and Wörgötter, F.: Multi Modal Estimation of Collinearity and Parallelism in Natural Image Sequences. *Network: Computation in Neural Systems*, vol. 13, no. 4, (2002) pp. 553-576.

[4] Krüger, N. and Felsberg, M.: An Explicit and Compact Coding of Geometric and Structural Information Applied to Stereo Matching. *Pattern Recognition Letters*, vol. 25, Issue 8, (2004) pp. 849-863.

[5] Granlund, G. H.: In search of a general picture processing operator. *Computer Graphics and Image Processing*, vol. 8, no. 2, (1978) pp. 155-173.

[6] Fleet, D. J. and Jepson, A. D.: Stability of phase information. *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 15, issue 12, (1993) pp. 1253-1268.

[7] Kovesi, P.: *Image Features From Phase Congruency*. Videre [Online]. 1 (3) (1999). Available: http://mitpress.mit.edu/e-journals/Videre/001/v13.html.

[8] Fleet, D. J.: Measurement of Image Velocity. *Norwell, MA: Engineering and Computer Science*, Kluwer Academic Publishers, (1992).

[9] Gautama, T. and Van Hulle, M. M.: A Phase-based Approach to the Estimation of the Optical Flow Field Using Spatial Filtering. *IEEE Trans. Neural Networks*, vol. 13, issue 5, (2002) pp. 1127-1136.

[10] Solari, F., Sabatini, S. P., Bisio, G. M.: Fast technique for phase-based disparity estimation with no explicit calculation of phase. *Elec. Letters*, vol. 37, issue 23, (2001) pp. 1382 -1383.

[11] Fleet, D. J., Jepson, A. D., Jenkin, M. R. M.: Phase-Based Disparity Measurement. CVGIP: Image Understanding, vol. 53, issue 2, (1991) pp. 198-210.

[12] Nestares, O., Navarro, R., Portilla, J., Tabernero, A.: Efficient Spatial-Domain Implementation of a Multiscale Image Representation Based on Gabor Functions. *Journal of Electronic Imaging*, vol. 7, no. 1, (1998) pp. 166-173.

[13] Freeman, W. and Adelson, E.: The design and use of steerable filters. *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 13, issue 9, (1991) pp. 891-906.

[14] Díaz, J.: Multimodal bio-inspired vision system. High performance motion and stereo processing architecture. PhD Thesis, (2006), University of Granada.

[15] Haglund, L.: Adaptive Multidimensional Filtering. Ph.D. thesis, nº 284, Linköping University, Sweden, SE-581 83 Linköping, Sweden, (1992).

[16] Díaz, J., Ros, E., Pelayo, F., Ortigosa, E. M., Mota, S.: FPGA based real-time optical-flow system. *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 16, no. 2, (2006) pp. 274-279.

[17] Celoxica, RC300 board information. Last access December 2006. Available at: http://www.celoxica.com/products/rc340/default.asp.

[18] Xilinx, ISE Foundation software. Last access January 2007. Available at: http://www.xilinx.com/ise/logic_design_prod/foundation.htm.

[19] DRIVSCO, project web page, Last access January 2007. Available at: http://www.pspc.dibe.unige.it/~drivsco/

A Compact Shader for FPGA-Based Volume Rendering Accelerators

G. Knittel

WSI/GRIS, Tübingen University
72076 Tübingen, Germany
knittel@gris.uni-tuebingen.de

Abstract. We describe a map-based shading unit for volumetric raycasting architectures. This is a critical part of volume rendering accelerators with respect to image quality, achievable performance, and resource consumption. The typical problem with map-based shaders is that highlights might not be rendered correctly on highly glossy surfaces due to the limited resolution of the maps. As a solution to this problem we introduce the use of detail light maps.

1 Introduction

Image synthesis from three-dimensional data, a.k.a. *Volume Rendering*, has become a routine task in fields such as medical diagnosis or non-destructive testing. Typical data set sizes approach and surpass 512^3 elements. At the same time, rendering algorithms are becoming more and more sophisticated, making hardware-acceleration mandatory. A popular approach is to use GPUs for this task [4],[6],[12], but their distinct architecture can impose severe restrictions on such non-native applications. Likewise, ASIC-based solutions [10] can hardly accommodate the rapid algorithmic evolution in this field. Thus, the decision was made to design an FPGA-based volume rendering accelerator for maximum flexibility. After VIZARD [5] and VIZARD-II [9], this represents our third-generation project in this field. Other projects to mention are VGE [3], Sepia-2 [8], and RACE-II [11].

The rendering method we use is perspective raycasting [7]. Starting from the viewpoint, a viewing ray is shot through the volume data set for each pixel on a virtual screen. Each ray consists of a sequence of equidistant raypoints, which in general do not coincide with grid points. At each raypoint location, data set properties are derived from the surrounding data elements (*voxels*), transformed into optical properties (color, opacity etc.), and as such assigned to the raypoint. The visual contributions of all raypoints along the ray are blended in some way to give the final pixel color. In first approximation, the number of raypoints to process is in the same order of magnitude as the number of voxels.

One of the main tasks of volume rendering is to identify and display surfaces of interest within the data set. Such surfaces can be iso-value surfaces or material boundaries. If a raypoint is on or close to such a surface, it is considered a surface element whose orientation (surface normal) is given by the local gradient. It can now be shaded by applying a local illumination model, assuming a number of (point) light sources in the scene. It is desirable to render surfaces within different materials with unique parameters.

A commonly used illumination model includes ambient, diffuse and specular reflected light. The ambient part represents a constant level of brightness. The diffuse part depends on the gradient **G** and the direction to the light source **L**. The specular part (in Blinn's notation, [2]) depends on the so-called halfway vector **H** and again the gradient. The halfway vector in turn is the normalized sum of the vector to the viewpoint **V** and the vector to the light source.

We make a number of simplifications. The number of light sources is limited to four. They are placed at infinity, emit white light and have unit strength. We further assume that **V** is constant throughout the scene, irrespective of perspective projection. The illumination equation can then be written as:

$$I = ka_m \cdot C + kd_m \cdot C \cdot \sum_p \mathbf{G} \cdot \mathbf{L_p} + ks_m \cdot \sum_p (\mathbf{G} \cdot \mathbf{H_p})^{e_m} \ . \qquad (1)$$

In (1), I is the reflected light, C is the raypoint color, both quantities consisting of red, green and blue components. The sums run over all light sources p. ka, kd and ks are ambient, diffuse and specular reflection coefficients, respectively. e is the specular reflection exponent, which defines the "shininess" of a surface, and which typically ranges from 2 to 255. The latter four parameters are material-dependent to provide an individual appearance for different materials, and are user-supplied. Our system supports up to 16 different materials m in a data set.

By our assumption, **V** and $\mathbf{L_p}$, and thus $\mathbf{H_p}$ are constant throughout the scene (for all possible raypoints). The only variables during raypoint shading are C, which is typically read from a look-up table, and **G**.

Still it is very expensive to evaluate (1) for each potential raypoint. A total of eight dot products must be performed. More problematically, the gradient is typically precomputed for each voxel, and interpolated at the raypoint location from the eight surrounding voxels. Thus, it does not have unit length and must be normalized first. The exponentiation poses further problems. Especially on an FPGA-implementation, these operations can be prohibitively expensive. Even more so if it becomes necessary to place more than one volume rendering pipeline on the FPGA to meet performance targets.

A common approach therefore is to use *pre-computed light maps* (as done in [3] as well as [9]), most often in the form of *cube maps* [13],[14]. A cube map is a collection of six two-dimensional tables, which form the faces of the unit cube. Each table contains the values of the sum terms in (1) for a discrete set of gradients (surface orientations). A newly computed gradient is normalized to its largest component (L_∞-norm), which selects one of the tables. The remaining two components are indices into that table, and the final intensity is obtained by bilinear interpolation between the four surrounding light map elements. A light map needs to be re-computed after each observer or light source movement.

This method is relatively inexpensive, but has a number of severe disadvantages. These include:

- A large exponent e causes narrow highlights, which require a high resolution in the light maps.
- If e is material-dependent, a separate light map would be required for each material.
- The number of light maps (six) is impractical, and might cause memory resources to be waisted.

In the remainder of this paper, we address these issues. The first problem is solved by the use of *detail maps*. An approximate solution to the second can be found by *reversing the order* of operations. The third disadvantage is removed by using the concept of *octahedron-derived square maps*. These are the contributions of this paper.

2 Hardware Platform

Our development system is a commercial PCI-card with one Xilinx Virtex-II 4000 FPGA, speed grade -4 (slowest version). The hardware units this FPGA has to offer are:

- 120 18×18-bit two's complement multipliers,
- 120 18kbit BlockRAMs, e.g. organized as 512×36,
- 46,080 function generators (LUTs), each also configurable as a 1-bit adder, a 16×1bit memory, or as a 16-bit shift register, and
- 46,080 flipflops.

3 Design Principles

3.1 General Shader Requirements

As stated above, the shader must provide the following features:

- compact design,
- support for sixteen surface types (materials),
- support for four light sources,
- ability to produce sharp highlights,
- one shading operation per clock.

We'll start the discussion with the specular term.

3.2 The Specular Term

The specular term I_s in our system is conceptually related to Blinn's model using halfway vectors, given by

$$\sum_p (\mathbf{G} \cdot \mathbf{H_p})^{e_m} \text{ , where } m \text{ refers to the surface type.} \tag{2}$$

Instead of using a cube map, we project a gradient onto an octahedron by forming the L_1-norm. Then, the z-component except for the sign is redundant, and so the octahedron can be flattened into two squares. A coordinate transform gives the final tables. This is shown in Fig. 1. Each table has 32×32 entries. Each entry corresponds to a distinct surface orientation, for which an I_s-related quantity is precomputed.

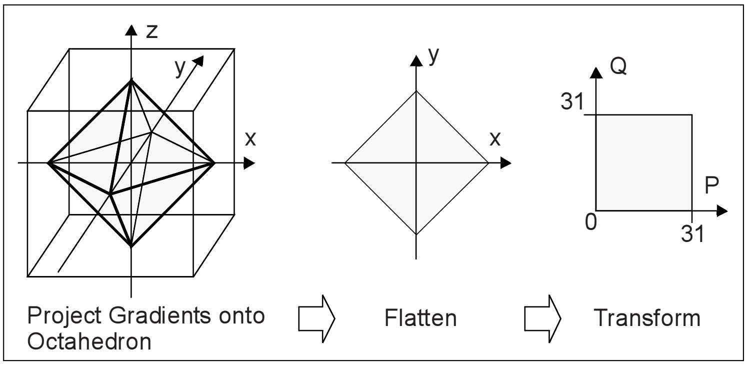


Fig. 1. Octahedron Light Map, transformed into Square Maps

By using these octahedron-derived maps, the impractical handling of six maps can be avoided. However, the properties of each version must be compared. One quantity to look at is angular resolution. In both versions, the angle between any two neighboring sample points (gradients) is not constant. For three configurations, the minimum and maximum values (using an 8-connected neighborhood) along with the map size are shown in Table 1. The distribution of angles between neighboring samples is shown in Fig. 2, for a 16×16 cube map (Fig. 2b), and our 32×32 square map (Fig. 2a). The square maps appear to be an acceptable alternative.

Table 1. Minimum and maximum Angles in Degrees between neighboring Gradients

	Cube Map	Cube Map	Square Map
Size	6×16×16 = 1536 entries	6×32×32 = 6144 entries	2×32×32 = 2048 entries
Max.	10.77	5.22	9.02
Min.	3.76	1.78	1.91

During rendering, an incoming gradient is transformed into the square map according to

$$P = \frac{(G_x + G_y) \cdot a}{|G_x| + |G_y| + |G_z|} + b \, , \quad Q = \frac{(G_y - G_x) \cdot a}{|G_x| + |G_y| + |G_z|} + b \, , \quad a, b = 15.5 \, . \quad (3)$$

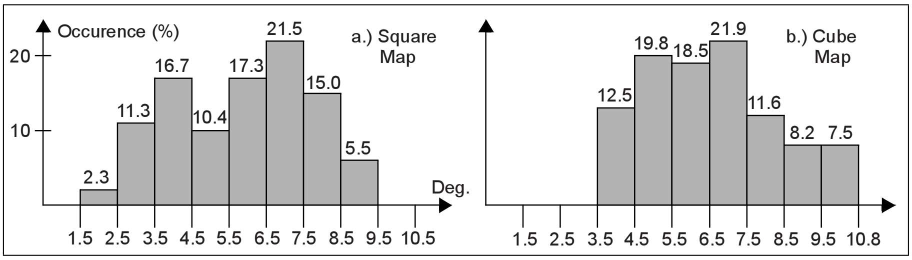


Fig. 2. Distribution of Angles between neighboring Gradients

The factor

$$F = \frac{a}{|G_x| + |G_y| + |G_z|} = \frac{a}{L}, \quad \text{with } 0 \leq L \leq 768 \text{ in our design,} \tag{4}$$

is precomputed and stored in a table addressed by L (which is the L_1-norm). This memory is organized as 1k×18 bits and consumes one BlockRAM. One of the two square maps is selected by the sign of the z-component of the gradient.

However, the angular resolution in the light map is still too low, and so highlights might be lost due to undersampling. Thus, we introduce the use of a *detail map* for each light source (halfway vector) for locally increased resolution. The extent of each detail map is one cell in the light map (dimension is 1×1). The cells in the light map cover different angular regions, so one detail map for all light sources would not suffice. Each detail map has a resolution of again 32×32 entries, and is centered around its halfway vector in light map coordinates. In order to limit memory costs, we limit the number of light sources to four. A diagram depicting the superposition of light and detail maps is shown in Fig. 3.

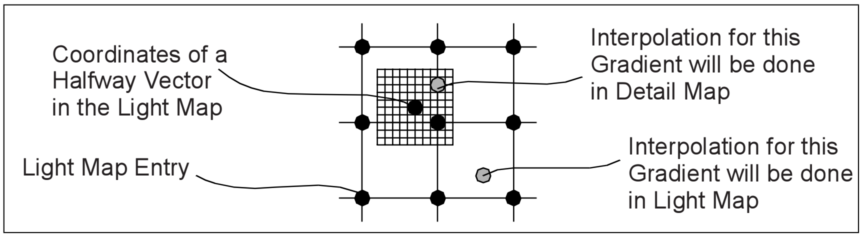

Coordinates of a Halfway Vector in the Light Map
Light Map Entry
Interpolation for this Gradient will be done in Detail Map
Interpolation for this Gradient will be done in Light Map

Fig. 3. A Detail Map centered around its Halfway Vector in the Light Map

Observer or light source movements cause a halfway vector to move in the light map, and along with it, the detail map as well. A halfway vector can move from the light map for gradients with positive z-components (”upper light map”) to the light map for negative z-components (lower light map) and vice versa. This creates the problem of detail maps having to wrap around an edge or a corner of the light map. The solution is to provide two separate detail maps for each light source, one for the upper and one for the lower light map. Both detail maps are simultaneously needed only if the halfway vector is close to the $z = 0$-plane. Otherwise, one of them is marked invalid by means of a software-controlled flag. Note that, if using cube maps, this method would be more expensive.

If an incoming gradient is within a detail map, interpolation is done exclusively within the detail map, otherwise within the light map. Each detail map must be computed such that there will be no C_0-discontinuities along its edges with respect to the light map.

The entries in both detail and light maps are dependent on the specular exponent, which is material-dependent. Thus, a separate set of maps for each material would be required. However, one set of maps for each surface type is not affordable. Thus we propose to fill the maps with an auxiliary function, and to perform material-dependent exponentiation *after* map access.

Accordingly, exponentiation is applied to the interpolated value from the light or detail map, using a set of one-dimensional look-up tables, one for each of the 16 surface types. The entries in the light or detail map are therefore indices, which should span a sufficiently large range in the exponentiation table for small angular deviations from a halfway vector. Therefore, we define an auxiliary reflectance function $A(\alpha_p)$, α_p being the angle between $\mathbf{G}$ and $\mathbf{H_p}$, of shape like that shown in Fig. 4. The light maps finally contain some (normalized) function σ of the auxiliary reflectance functions. For the rendering in Fig. 8b we used

$$\sigma = scale \cdot \sum_p A(\alpha_p) \ , \ \text{with } A(\alpha) = 255 \cdot e^{-0.1 \cdot \alpha} \ . \tag{5}$$

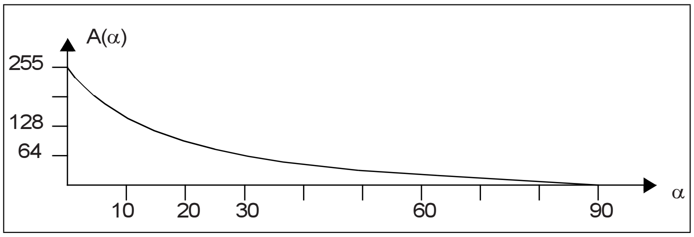


Fig. 4. Auxiliary Reflectance Function

Light map examples are shown in Fig. 5. A version without detail maps is shown in Fig. 5a. Due to the coarse sampling, maximum values are missed for some peaks. Fig. 5b shows the same map complemented by the detail maps (sitting on top of the peaks), which now reproduces the full range.

If using 8-bit quantities, the values in a detail map will not fall below 128, and so the MSB is always one. Thus, detail map entries need only 7 bits. Light and detail maps are stored in an interleaved way in four banks. The eight detail maps plus the two light maps can be combined into four BlockRAMs, each organized

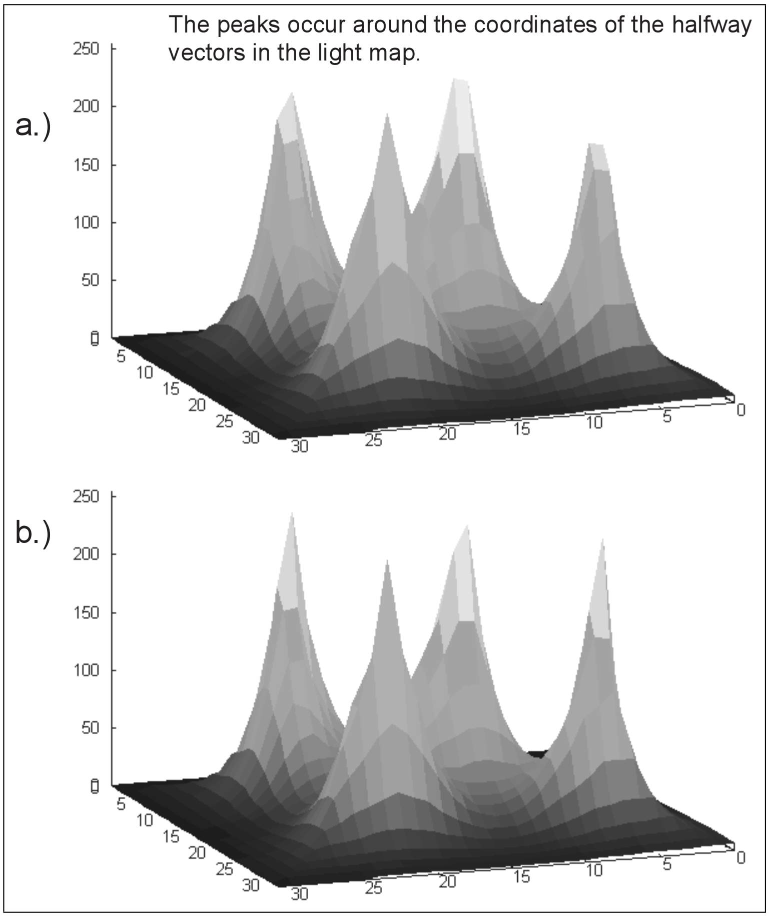


Fig. 5. a.) Light Map, b.) Light and Detail Maps. Light Source Directions are approximately {1,0,0}, {0,1,0}, {0,-1,0}, and {0,0,1}.

as 2×256×36, where each entry provides 8 bits for the light map plus 4×7 bits for the detail maps.

Each exponentiation table contains 256 entries computed as

$$I_s[m][\sigma] = 255 \cdot \cos^{e_m}(\alpha)\,, \text{ with } \alpha = -\frac{\pi}{180 \cdot 0.1} \cdot ln\left(\frac{\sigma}{255}\right) \ . \qquad (6)$$

3.3 Diffuse Term

The diffuse term I_d depends on the gradient and the light vectors, as opposed to the halfway vectors for the specular term, and would therefore require an additional entry in the light map, and an additional bi-linear interpolator. However, due to the slow decrease of the cosine, directional information from the diffuse term is relatively subtle. Thus, as a further approximation, we use

$$I_d[\sigma] = 255 \cdot \cos(\alpha), \quad \text{(with } \alpha \text{ computed as in (6))} \tag{7}$$

instead, σ being obtained from the same maps as I_s. This is inaccurate in the fact that the direction of the "diffuse light sources" is then not $\mathbf{L_p}$ but $\mathbf{H_p}$. The effects of this simplification are shown in Fig. 6. Here, two solid spheres have been rendered with one light source, coming from 45° above the viewer. In Fig. 6a, the illumination has been rendered as in (1), with a faint highlight. Fig. 6b uses the aforementioned approximation. Most notably, the maximum of the diffuse term coincides here with the specular highlight (see Fig. 6c and d for an exaggerated view). For this project the decision was made to accept this inaccuracy, and so to save an additional map and bi-linear interpolator.

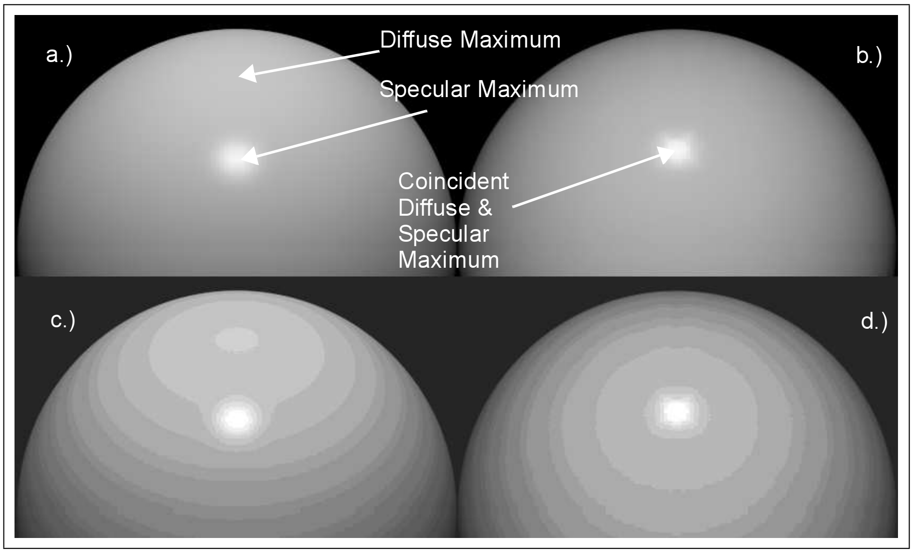


Fig. 6. Comparison of Diffuse Terms. a.) Computed as in (1). b.) Approximation. In c.) and d.), Brightness, Contrast and Gamma Correction of the 2D Images have been strongly (but identically) adjusted to make the Differences visible.

3.4 Illumination Equation

The exponentiation tables occupy just two BlockRAMs, organized as 2×256×72, for both I_s and I_d. Once the diffuse and specular terms are available, the shader computes

$$I_R = C_R \cdot (ka_m + kd_m \cdot I_d) + ks_m \cdot I_s \tag{8}$$
$$I_G = C_G \cdot (ka_m + kd_m \cdot I_d) + ks_m \cdot I_s \tag{9}$$
$$I_B = C_B \cdot (ka_m + kd_m \cdot I_d) + ks_m \cdot I_s \tag{10}$$

3.5 Shader Block Diagram

A simplified block diagram of the shader can now be given, as shown in Fig. 7. Input to this stage are the gradient components at the raypoint location, as well as the assigned color components C and the material identifier m of the raypoint. Output is the reflected light intensity I, using reflection coefficients stored in small local memories.

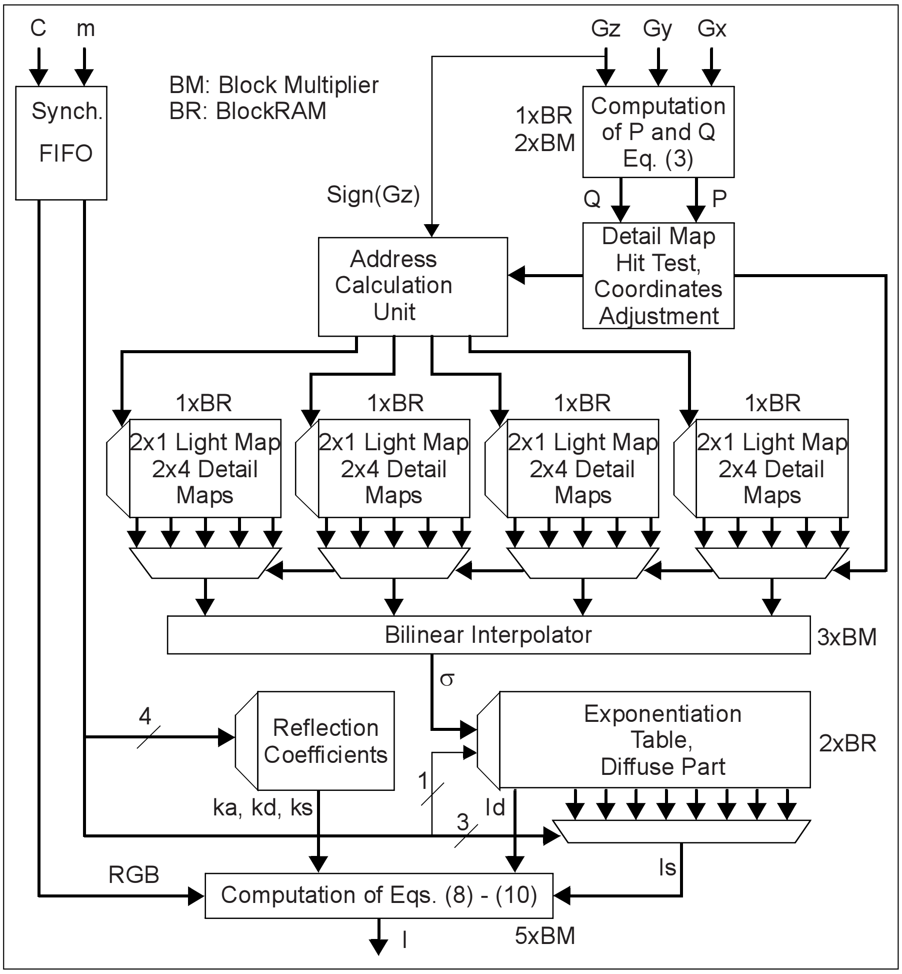

C
m
Synch.
FIFO
BM: Block Multiplier
BR: BlockRAM
Gz
Gy
Gx
1xBR
2xBM
Computation of P and Q Eq. (3)
Sign(Gz)
Q
P
Address Calculation Unit
Detail Map Hit Test, Coordinates Adjustment
1xBR
2x1 Light Map 2x4 Detail Maps
1xBR
2x1 Light Map 2x4 Detail Maps
1xBR
2x1 Light Map 2x4 Detail Maps
1xBR
2x1 Light Map 2x4 Detail Maps
Bilinear Interpolator
3xBM
σ
4
Reflection Coefficients
Exponentiation Table, Diffuse Part
2xBR
1
ka, kd, ks
Id
3
RGB
Is
Computation of Eqs. (8) - (10)
I
5xBM

Fig. 7. Shader Block Diagram

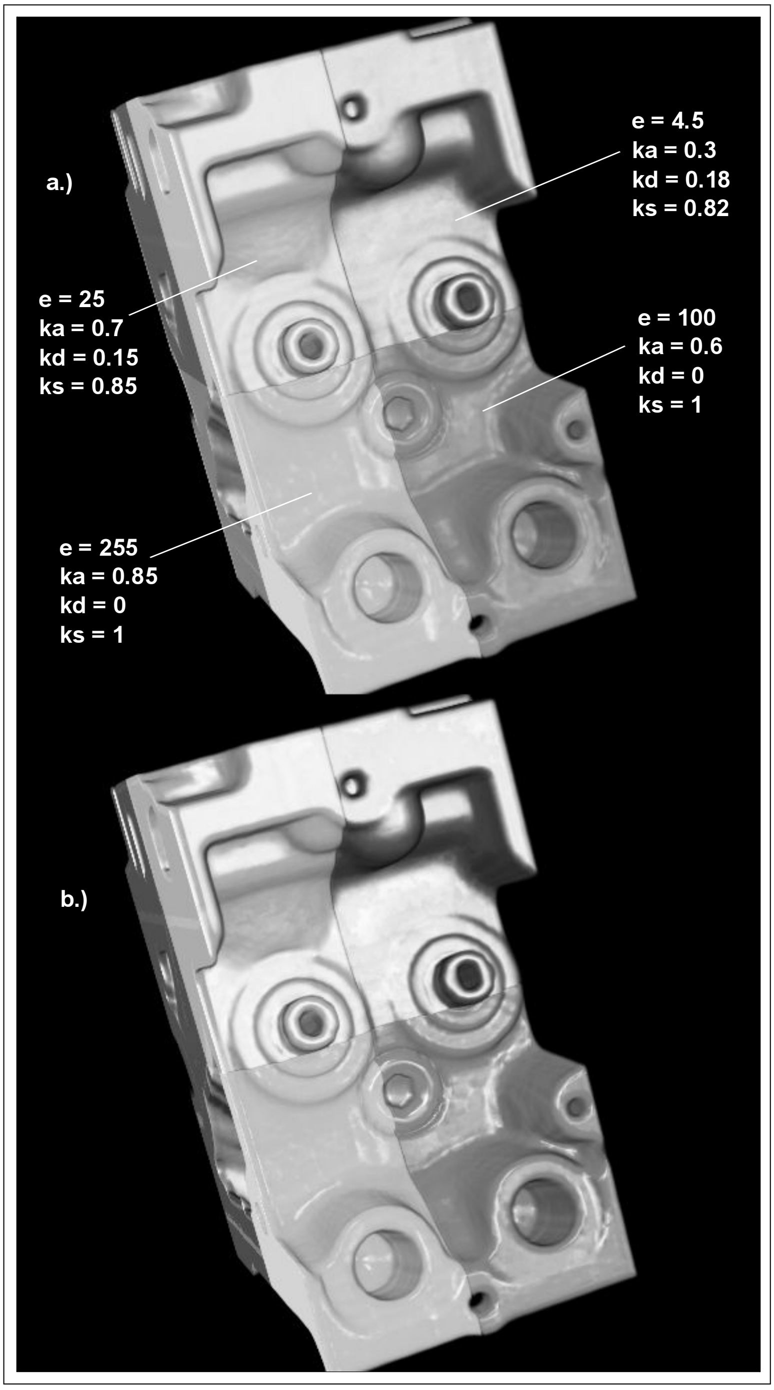


Fig. 8. Shading Quality. a.) Computed as in (1). b.) Compact Shader Simulation.

4 Implementation

The shader was implemented as a cycle-accurate C++ software model, which was used to compute the images in Fig. 6b and Fig. 8b. Also, a VHDL implementation was made, which was placed and routed on the Xilinx Virtex-II 4000 FPGA. To meet our target clock frequency, the shader was divided into 20 pipeline stages. Hardware consumption is listed in Table 2. Five multipliers are needed for evaluation of (8) – (10), irrespective of how the diffuse and specular terms have been obtained.

This compact shader is expected to produce images as shown in Fig. 8b. The object (an engine part) has been (artificially) divided into eight materials, each of them being rendered with different parameters. In Fig. 8a the reference image is shown, which was computed using the conventional model (1) in full floating-point precision. As can be seen, the method of using detail maps can produce the look of highly glossy surfaces with strong curvature.

The achieved clock frequency of the latest Place&Route was 115MHz, for 115M shaded raypoints per second. It is difficult to translate this number into application performance, e.g., frames per second. For the rendering in Fig. 8b, a total of 3,313,970 raypoints entered the shader. Shader performance alone would then give about 35 frames per second for renderings like this.

Table 2. Resource Consumption of the Shader

Resource	Total Number	Percentage of Chip Resources
LUT	1073	2.3%
FlipFlop	1341	2.9%
Multiplier	10	8.3%
BlockRAM	7	5.8%

5 Conclusion and Future Work

We have presented a shading unit for volumetric raycasting accelerators which was designed with the goal of ultimate compactness. It is table-based, and thus minimizes the number of multipliers. The typical disadvantage of table-based shaders, the inability to produce sharp highlights on glossy surfaces, has been eliminated by the use of detail maps. Furthermore, an approximate solution has been presented for the handling of multiple materials (objects) with individual surface properties. While it is undisputed that a number of coarse approximations have been used, image quality is still acceptable at very low hardware expenses.

Further efforts will be spent on increasing the rendering speed, most likely at the cost of additional pipeline stages. For the FPGAs we use, the upper limit is most probably given by the path flipflop $\rightarrow$ wire $\rightarrow$ pipelined multiplier. Assuming a wire delay of 1.5ns, this path would need roughly 6ns [1], for a maximum clock frequency of 167MHz.

Acknowledgment

This work was funded by the Deutsche Forschungsgemeinschaft under grant SFB 382.

References

1. Anonymous: Virtex-II Platform FPGAs: Complete Data Sheet, DS031, v3.4, March 2005, Xilinx, Inc.
2. Blinn, J.: Models of Light Reflection for Computer Synthesized Pictures, Proc. ACM SIGGRAPH 77, pp. 192-198
3. Chen, H., Vettermann, B., Hesser, J., Maenner, R.: Innovative computer architecture for real-time volume rendering, Computers & Graphics, Vol. 27, No. 5, Oct. 2003, pp. 715-724
4. Hadwiger, M., Berger, C., Hauser, H.: High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware, Proc. 14th IEEE Visualization Conference, Oct. 19-24, 2003, Seattle, USA, pp. 301-308
5. Knittel, G., Straßer, W.: VIZARD - Visualization Accelerator for Realtime Display, Proceedings of the 1997 ACM Siggraph/Eurographics Workshop on Graphics Hardware, Los Angeles, CA, August 3-4, 1997, pp. 139-146
6. Krueger, J., Westermann, R.: Acceleration Techniques for GPU-based Volume Rendering, Proc. 14th IEEE Visualization Conference, Oct. 19-24, 2003, Seattle, USA, pp. 287-292
7. Levoy, M.: Display of Surfaces from Volume Data, IEEE Computer Graphics & Applications, 8(5):29-37, 1988
8. Lombeyda, S., Moll, L., Shand, M., Breen, D., Heirich, A.: Scalable Interactive Volume Rendering Using Off-the-Shelf Components, Proc. IEEE PVG 2001, pp. 115-122
9. Meissner, M., Kanus, U., Wetekam, G., Hirche, J., Ehlert, A., Strasser, W., Dogget, M., Forthmann, P., Proksa, R.: VIZARD II: A Reconfigurable Interactive Volume Rendering System, Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware 2002, Saarbruecken, Germany, pp. 137-146
10. Pfister, H., Hardenbergh, J., Knittel, J., Lauer, H., Seiler, L.: The VolumePro Real-Time Ray-Casting System, Proc. ACM SIGGRAPH 99, pp. 251-260
11. Ray, H., Silver, D.: The RACE II Engine for Real-Time Volume Rendering, Proc. SIGGRAPH/Eurographics Workshop on Graphics Hardware 2000, Interlaken, Switzerland, pp. 129-136
12. Roettger, S., Guthe, S., Weiskopf, D., Ertl, T., Strasser, W.: Smart Hardware-Accelerated Volume Rendering, Proc. Eurographics - IEEE TCVG Symposium on Visualization, Grenoble, France, 2003, pp. 231-238
13. Terwisscha van Scheltinga, J., Smit, J., Bosma, M.: Design of an On-Chip Reflectance Map, Proc. 10th Eurographics Workshop on Graphics Hardware, 1995, pp. 51-55
14. Voorhies, D., Foran, J.: Reflection Vector Shading Hardware, Proc. ACM SIGGRAPH 94, pp. 163-166

Ubiquitous Evolvable Hardware System for Heart Disease Diagnosis Applications

Yong-Min Lee[1], Chang-Seok Choi[1], Seung-Gon Hwang[1], Hyun Dong Kim[2], Chul Hong Min[2], Jae-Hyun Park[1], Hanho Lee[1], Tae Seon Kim[2], and Chong-Ho Lee[1]

[1] School of Information and Communication Engineering, Inha University, Incheon, 402-751, Korea
{hhlee, jhyun, chlee}@inha.ac.kr

[2] School of Information, Communication and Electronics Engineering, Catholic University of Korea, Bucheon, Korea
tkim@catholic.ac.kr

Abstract. This paper presents a stand-alone ubiquitous evolvable hardware (u-EHW) system that is effective for automated heart disease diagnosis applications. The proposed u-EHW system consists of a novel reconfigurable evolvable hardware (rEHW) chip, an evolvable embedded processor, and a hand-held terminal. Through adaptable reconfiguration of the filter components, the proposed u-EHW system can effectively remove various types of noise from ECG signals. Filtered signals are sent to a PDA for automated heart disease diagnosis, and diagnosis results with filtered signals are sent to the medical doctor's computer for final decision. The rEHW chip features FIR filter evolution capability, which is realized using a genetic algorithm. A parallel genetic algorithm evolves FIR filters to find the optimal filter combination configuration, associated parameters, and the structure of the feature space adaptively to noisy environments for adaptive signal processing. The embedded processor implements feature extraction and a classifier for each group of signal types.

1 Introduction

The electrocardiogram (ECG) signal is a tracing of electrical activity signal generated by rhythmic contracting of the heart, and affords crucial information to detect heart disease [1]. Since the ECG signal varies from patient to patient, and according to measured time and environmental conditions, an adaptable heart disease diagnosis algorithm based on context-aware computing has been proposed in which a genetic algorithm (GA) finds the optimal set of preprocessing, feature extraction, and classifier for each group of signal types [1].

This paper presents a stand-alone ubiquitous evolvable hardware (u-EHW) system to process the adaptable heart disease diagnosis algorithm and detect heart disease. The u-EHW system consists of a reconfigurable evolvable hardware (rEHW) chip, an evolvable embedded processor, and a PDA. The rEHW chip

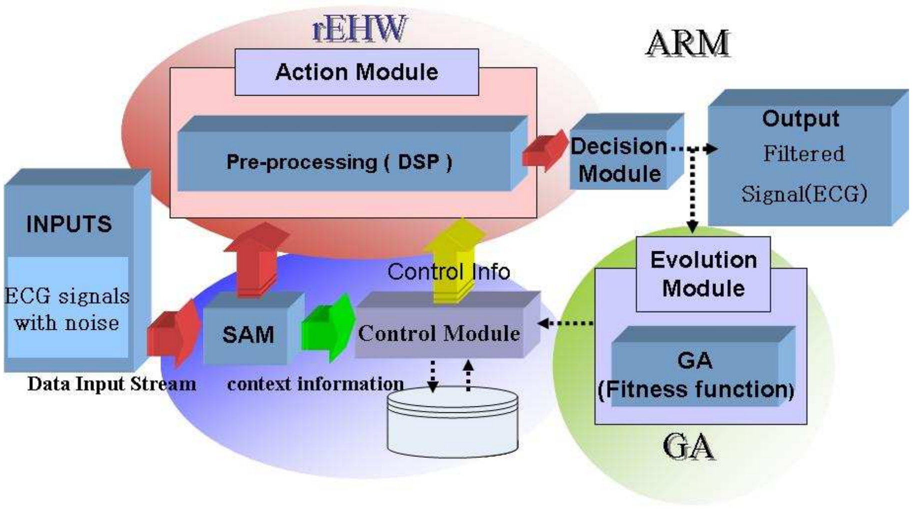


Fig. 1. Block diagram of adaptable heart diseases diagnosis system

can effectively find the optimal set of preprocessing, and processes the adaptive digital signal processing (DSP). That is, the rEHW chip processes the low-pass, band-pass, and high-pass FIR filter algorithms with various frequencies. The evolvable embedded processor operates GA, feature extraction, and a classifier for each group of signal type. Currently, many DSP and smart health care applications are implemented on digital signal processors and embedded processors by software. Effective DSP algorithms require computing architectures that are less complicated, highly flexible, and more cost-effective for smart health care applications. Currently, complex and fast computation can be performed by dedicated hardware instead of digital signal processor since dedicated hardware can operate in parallel. The concept of reconfigurable hardware and evolvable hardware has been studied actively [2-4]. Since evolvable hardware can evolve in real time, it can maintain optimal performance in a variety of unexpected environments. The configurations that optimize the device output responses are combined so as to make better configurations until an optimal architecture can be realized [6]. Evolvable hardware continues to reconfigure itself in order to achieve better performance.

In contrast to the conventional ECG signal measurement system, the proposed u-EHW system can measure and analyze the ECG signal in a ubiquitous environment. For this, a GA based rEHW chip effectively reconfigures the filter components to remove noise components. Filtered signals are sent to a PDA for automated heart disease diagnosis, and diagnosis results with filtered signals are sent to the medical doctor's computer where a final decision can be made. Through this, patient customized healthcare service in ubiquitous environments can be realized.

2 Adaptable Heart Disease Diagnosis Algorithm

The adaptable heart disease diagnosis algorithm mainly consists of six modules; a noise context estimation module, a control module for filter block design at running mode, a rEHW module for a 3-stage filter block, a GA module for filter design at evolution mode, a decision module for fitness calculation at evolution mode, and a disease diagnosis module, as shown in Figure 1.

Accurate estimation of ECG signal noise in a dynamic environment is impossible since signal artifacts from respiration, motion activity, and their coupled signal noise are not predictable. For this reason, noisy signals are grouped into several categories by environmental context estimation. Baseline wander noise and muscle noise are then quantized based on environmental context information.

In order to identify the adaptable filter composition, the outputs of the noise context estimation module are fed to the inputs of the control module. The control module consists of two parts, an evolutionary knowledge accumulator (EKA) and a neural network for the filter design. The EKA stores the optimal filter design results for six-clustered dynamic measurement environments. For network construction, two types of outputs from the context estimation module are used as network inputs and six outputs are used to cluster the dynamic measurement environment into six groups. The 6 best filter combinations for each cluster are a 0.8 45 Hz band pass filter, a 0.8 Hz high pass filter, a 50 Hz low pass filter, a combination of a 0.8 Hz high pass filter and a 50 Hz low pass filter, a combination of a wavelet interpolation filter (WIF) and a 0.8 45 Hz band pass filter, and a combination of WIF, a 0.8 Hz high pass filter and a 50 Hz low pass filter. For clustering of environmental noise, a three layered feed-forward error back-propagation neural network (BPNN) is applied and network weights are updated after each training vector (by "training by vector" method).

For a continuous performance update, every filtered output from the 3-stage filter block is evaluated by fitness function at decision module. The fitness function is defined as the detection percentage of five ECG signal features, i.e., P-R interval, QRS duration, Q-T interval, R-R interval, and S peak point, using So & Chan and Kim's feature extraction method [1].

3 Ubiquitous Evolvable Hardware System

The overall u-EHW system consists of a rEHW chip, an evolvable embedded processor, and a hand-held terminal, as shown in Figure 2. The input data of the rEHW and evolvable embedded processor is the measured ECG signal data, which may contain muscle noise, baseline noise, in-coupling noise, and 60 Hz power line noise.

3.1 Reconfigurable Evolvable Hardware Chip

The rEHW chip processes the DSP algorithms, i.e., low- pass, band-pass, and high- pass filter algorithms with various frequencies. This chip consists of a

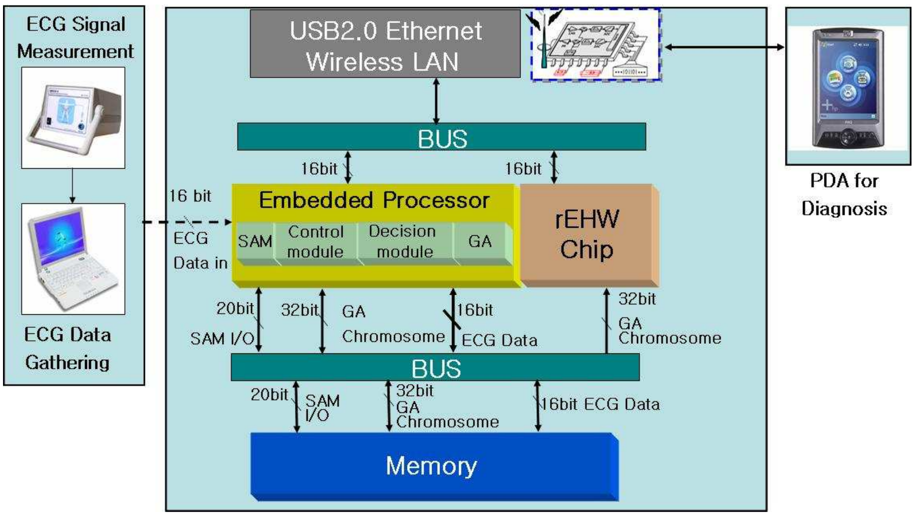


Fig. 2. Block diagram of ubiquitous evolvable hardware system

reconfigurable processing unit, a configuration manager, and coefficient memory, as shown in Figure 3. The reconfigurable processing unit has a 3-stage reconfigurable filter block in which the optimal FIR filter function can be searched and selected by GA chromosome data obtained using a GA running on an embedded processor. Each reconfigurable filter block includes 20 reconfigurable processing modules (RPMs).

The GA has been used to evolve digital circuits. A chromosome represents a component of order. The 30-bit GA chromosome data has 3-stage 30-bit chromosome data, which defines the type of optimal filter, cutoff frequency, and filter order, as shown in Fig. 4. The first 2-bit data decides the optimum filter type, such as a low-pass, band-pass, or high-pass filter. If a high-pass filter or low-pass filter is selected by the GA, cutoff frequency#1 or #2 has the data, respectively. If the band-pass filter is selected, cutoff freqeuncy#1 and #2 have the data. The last 3-bit data defines the filter order, which decides the number of filter taps, from 6- to 20-taps. The configuration manager consists of the order distributor, stage selector, and memory address decoder. The order distributor analyzes the filter order information and the stage selector selects the filter stage to be used. Coefficient memory stores the coefficients, which are used in each filter.

The rEHW chip is responsible for providing the best solution for realization and autonomous adaptation of FIR filters, and is used to process the optimal DSP algorithms for noise removal operation prior to feature extraction and classification steps. Generally, low-pass, band-pass, and high-pass filters are used separately in many preprocessing algorithms [1].

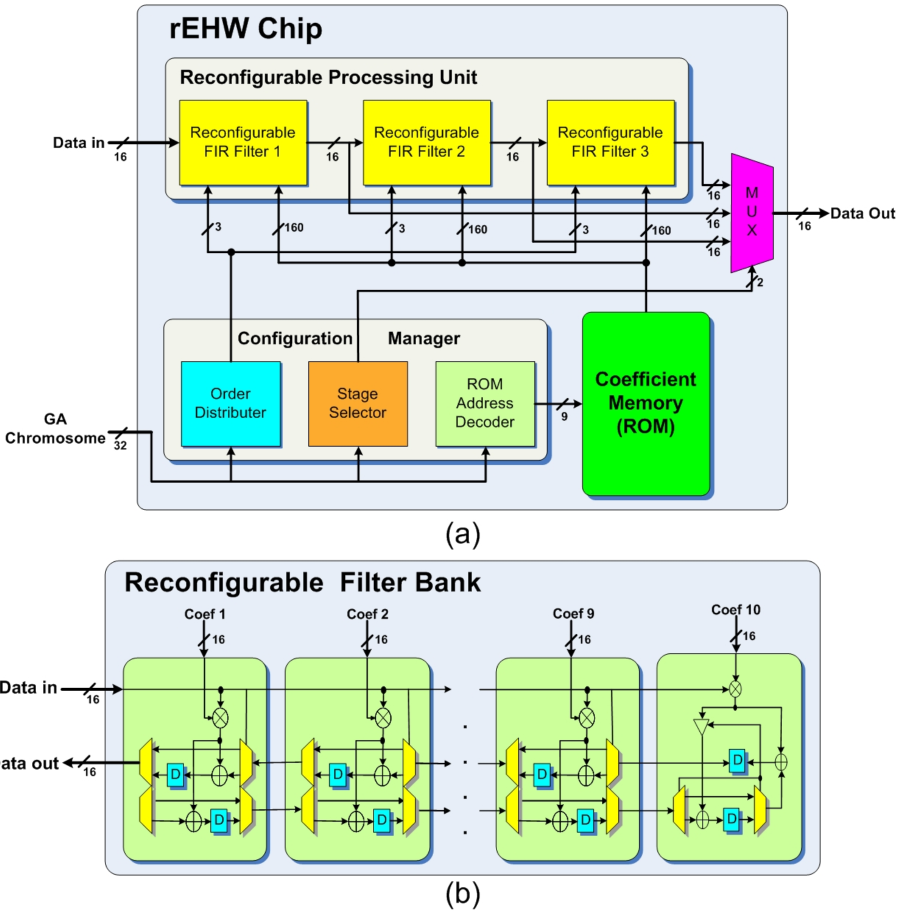


Fig. 3. Block diagram of (a) reconfigurable evolvable hardware and (b) reconfigurable filter block

3.2 Evolvable Embedded Processor

The evolvable embedded processor implements a noise context estimation module, decision module, and control module to process the feature extraction and classifier algorithms by software. Also, embedded processor processes the GA, which is a search procedure inspired by populating genetics, and has excellent search capabilities for finding a good solution to a problem without a priori information about the nature of the problem [7].

For a continuous performance update, every filtered output from the rEHW chip is evaluated by a fitness function at the decision module. To determine the adaptable filter composition, the outputs of the noise context estimation module are fed as inputs of the control module. The fitness function is defined as the detection percentage of five ECG signal Features; P-R interval, QRS duration,

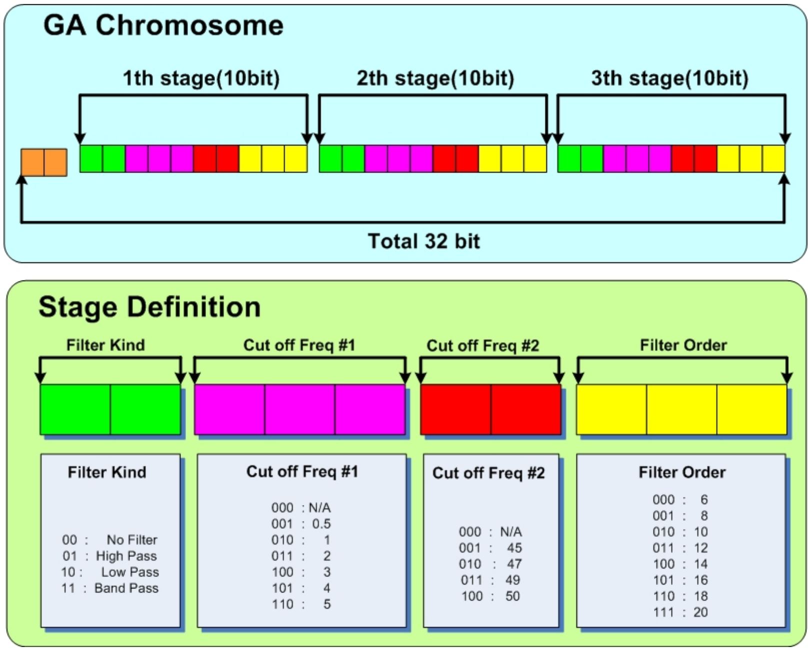


Fig. 4. GA chromosome data definition

Q-T interval, R-R interval, and S peak point, using So & Chan and Kim's feature extraction method [1]. In order to confirm the fitness function, the filtered signals are fed into the noise context estimation module again. If the fitness value is greater than 0.8, the ECG feature extraction results are considered to be acceptable values. The noise context estimation module can estimate the noise context using a neural network for filter design. For this, pre-determined optimal filter design results for six-clustered dynamic measurement environments are stored. For network construction, two types of outputs from the noise context estimation module are used as network inputs and six outputs are used to cluster the dynamic measurement environment into six groups. For clustering of environmental noise, a three layered feed-forward error back-propagation neural network (BPNN) is applied and network weights are updated after each training vector (by the "training by vector" method).

4 Implementation and Result

The rEHW architecture was modeled in Verilog HDL and functionally verified using a ModelSim simulator. The output data from the Verilog coded architecture was validated against a bit-accurate MATLAB model. The rEHW chip was implemented using standard 0.18-μm CMOS technology. Figure 5 shows the rEHW chip layout and Table 1 shows the rEHW chip summary.

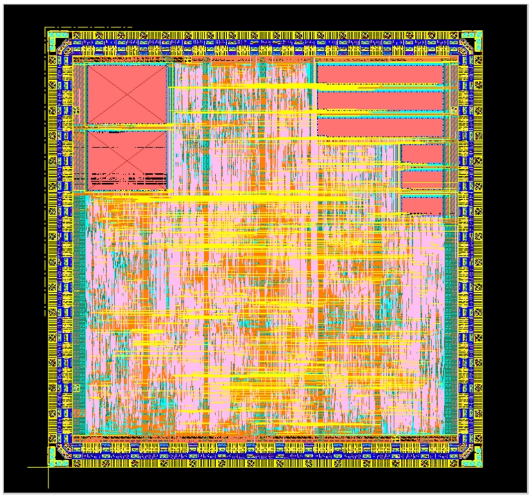


Fig. 5. rEHW chip layout

Table 1. rEHW chip summary

	Chip Summary
Technology	Standard 0.18-μm CMOS 1.8V core, 3.3V I/O
Die size	3.75 $\times$ 3.75 mm^2
Package	64-pin LQFP
Gate Count	582,182
Clock Speed	80 MHz

The rEHW chip consists of 582,182 gate excluding memories and the operating clock frequency is about 80 MHz. There are five control/status registers in the rEHW chip, a control register, interrupt status register, interrupt clear register, and two chromosome registers. Besides these control/status registers, 4Kbytes of input and output buffer memory are allocated for data transfer between the rEHW chip and embedded processor. Each ECG data consists of 1024 samples and they are processed by the rEHW chip. The chromosome used by the rEHW chip is selected by the GA algorithm in the embedded processor and set in the chromosome register. The rEHW chip processes 1024 items of data every 15 sec. After processing data, the rEHW chip generates interrupt the embedded processor in order to update processed data.

The complete stand-alone u-EHW system for adaptable heart disease diagnosis was tested on an embedded system test-bed including the rEHW chip and evolvable embedded processor. The noise context estimation module, control module, decision module, and GA have been implemented by a bit-accurate C-model and implemented on an embedded processor.

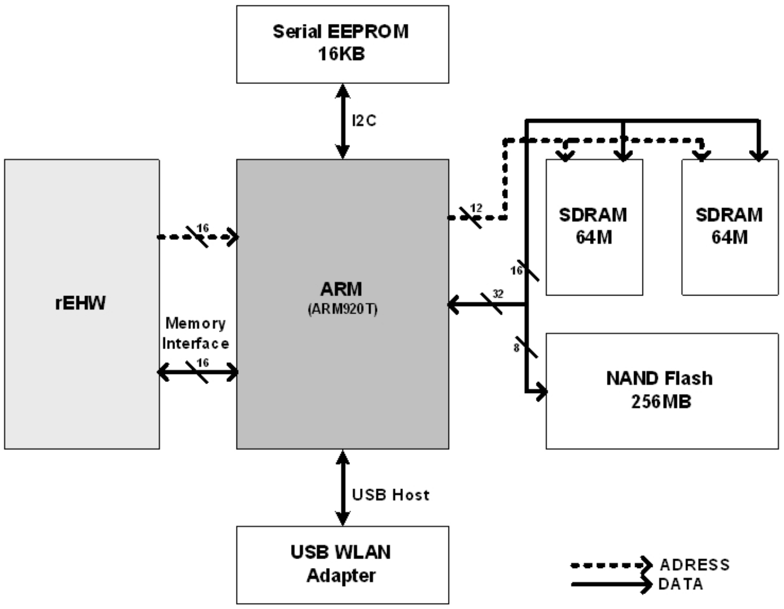


Fig. 6. Block diagram of u-EHW test-bed

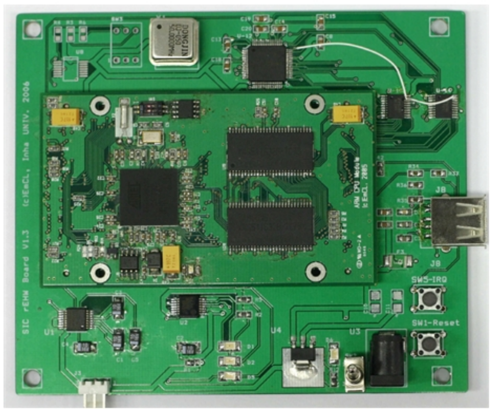


Fig. 7. Implemented u-EHW system test-bed

The implemented test-bed consists of an ATMEL AT91RM9200 ARM processor, of which the performance is 200 MIPS, a rEHW chip, external memories, and a LAN interface, as shown in Figure 6. To avoid coupling noise when running at a high speed of 200 MHz, the whole system is divided into two sections of a 6-layer PCB; a CPU board and a communication/rEHW chip board. Figure 7 shows the implemented u-EHW test-bed including the embedded processor

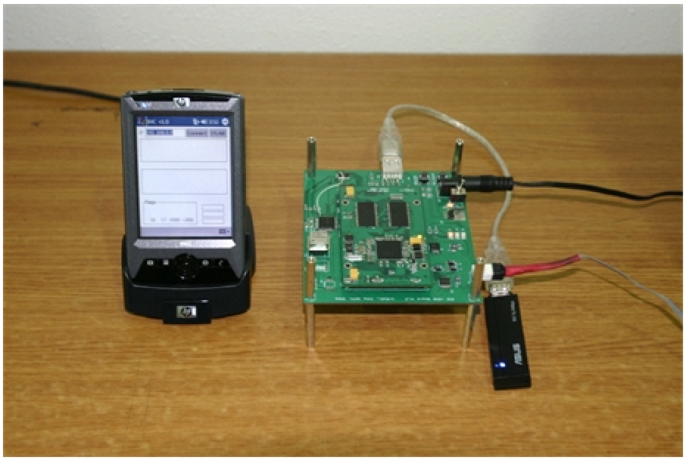


Fig. 8. u-EHW system platform for automated heart disease diagnosis applications

board and rEHW chip board. Embedded Linux version 2.6.12 is ported to support application software such as the noise context estimation module, control module, and decision module. A standard Linux-hosted Pentium PC is used as a development environment. GCC 3.4 is used for cross compiling and targeting to the embedded processor system. A 100 Mbps LAN network is used for the software development and debugging and a wireless LAN provides connectivity to the handheld application devices.

To demonstrate the u-EHW system, a set of sample ECG data were gathered from various patients and stored in the embedded processor board. The raw data and diagnosis results were transmitted to the hand-held terminal (PDA) in real-time over a wireless LAN. The PDA is used to display ECG data and diagnostic results. It displays the raw data wave and the processed wave as well as one out of four diagnostic results; SB, ST, RBBB, and LBBB. Figure 8 shows the stand alone prototype system of u-EHW.

5 Conclusion

In this paper, we present a ubiquitous evolvable hardware (u-EHW) system, which implements a stand-alone adaptable heart disease diagnosis system. The u-EHW system consists of a rEHW chip, an evolvable embedded processor, and a PDA. The rEHW chip is used for adaptive digital signal processing and the embedded processor implements feature extraction and a classifier for each group of signal type. The proposed stand-alone u-EHW system performs well, especially in changing noisy environments, since it can adapt itself to the external environment. The proposed u-EHW system is a promising solution for various applications such as DSPs, communications, and smart healthcare systems in a ubiquitous environment.

Acknowledgement

This research was supported by the MIC under the IT Research Center support program, Korea.

References

1. Kim, T. S., Kim, H-D.: Context-Aware Computing Based Adaptable Heart Diseases Diagnosis Algorithm. KES 2005, LNAI3682. (2005) 284-290.
2. Higuchi, T., Iwata, M., Liu, W.: Evolvable Systems: From Biology to Hardware. Tsukuba, Springer. (1996).
3. Stoica, A., et. al.: Reconfigurable VLSI Architectures for Evolvable Hardware: From Experimental Field Programming. Transistor Arrays to Evolution-Oriented Chip. IEEE Trans. on VLSI Systems. vol. 9, no. 1, (2001).
4. Tsuda, N.: Fault-Tolerant Processor Arrays Using Additional Bypass Linking Allocated by Graph-Node Coloring. IEEE Trans. Computers. vol. 49, no. 5, (May 2000) 431-442.
5. Marshall, A., Stansfield, T., Kostarnov, I., et. al.: A Reconfigurable Arithmetic Array for Multimedia Applications. ACM/SIGDA International Symposium on FPGAs. (1999) 135-143.
6. Faugman, J.: Uncertainty relation for resolution in space, spatial frequency, and orientation optimization by two-dimensional cortical filters. Journal Opt. Soc. Amer. 2(7), (1985) 675-676.
7. Sakanashi, H., Iwara, M., Higuchi, T.: Evolvable hardware for lossless compression of very high resolution bi-level images. IEE Proc.-Comput. Digit. Tech., vol. 151, no. 4, (July 2004) 277-286.

FPGA-Accelerated Molecular Dynamics Simulations: An Overview

Xiaodong Yang, Shengmei Mou, and Yong Dou

School of Computer Science,
National University of Defense Technology, Changsha 410073, P.R. China
proudwind@hotmail.com,hpcs_msm@163.com,yongdou@nudt.edu.cn

Abstract. Molecular Dynamics (MD) simulation, supported by parallel software and special hardware, is widely used in materials, computational chemistry and biology science. With advances in FPGA capability and inclusion of embedded multipliers, lots of studies steer to focus on FPGA-accelerated MD simulations. In this paper we make an overview on the background of MD simulations and latest FPGA-based research programs. Characteristics, major approaches and problems are introduced, and possible solutions are also discussed. Finally, some future directions of research are given.

1 Introduction

Molecular Dynamics (MD) Simulation is a technique for modeling the motion and interaction of atoms or molecules using the equations of classical Newtonian mechanics. During the simulation, force calculation, position/velocity update and other tasks will be performed, among which the non-bonded force calculation is very computationally intensive and thus good candidate for hardware acceleration. In this part we give a general introduction to MD background. In doing so, we reference [1].

The most common non-bonded interaction is modeled by the Lennard-Jones (LJ) potential, defined by $u_{LJ}(r) = 4\varepsilon[(\sigma/r)^{12} - (\sigma/r)^6]$, where ε and σ are particle-specific constants and r is the distance between the two interacting particles. Using LJ potential, the force between two particles can be calculated by $F = -\nabla_r u_{LJ}(r)$.

The potential V, is calculated for every possible pair of interacting particles as $V(r_1, \ldots, r_N) = \sum_{\mathrm{i}} \sum_{j>i} u(|\boldsymbol{r}_i - \boldsymbol{r}_j|)$, which is used to obtain the net force acting on any given particle, $F_i = -\nabla_{r\ i} V(r_1, \ldots, r_N)$. The largest contribution to the potential and forces comes from neighbors close to the molecule of interest, and for short-range forces we normally apply a spherical cutoff, beyond which two particles are no longer considered to interact.

Velocity Verlet (VV) algorithm, shown in Fig.1, is used to update the position and velocity, where Δt is the time-step. Clearly the force computation presents a significant challenge as that portion of the algorithm is O(n^2). The VV need only be performed once per particle and, thus, does not result in a significant

$$r(t+\Delta t) = r(t) + v(t)\Delta t + (1/2)a(t)\Delta t^2$$
$$v(t+\Delta t/2) = v(t) + (1/2)a(t)\Delta t$$
$$a(t+\Delta t) = F/m = -(1/m)\nabla V(r(t+\Delta t))$$
$$v(t+\Delta t) = v(t+\Delta t/2) + (1/2)a(t+\Delta t)\Delta t$$

Fig. 1. Velocity Verlet Update algorithm

bottleneck. Once a time step is complete, the process can continue until the desired length of time has been simulated.

The rest of the paper is organized as follows. Section 2 discusses the granularity of FPGA acceleration. Section 3 elaborates on the FPGA acceleration techniques in MD simulation. Section 4 concludes the work and discusses future directions.

2 Overview of Hardware/Software Approach

A great deal of prior research has focused on improving software performance of molecular dynamics simulation with fast algorithms or parallelism algorithms through atom decomposition, spatial decomposition or force decomposition. Currently there have some sophisticated highly-modular Short-Range Potential MD software package, such as NAMD and GROMACS. Limited by the performance of general purpose processor, some research has turned to special purpose hardware acceleration of the MD simulation.

An ASIC known as MODEL [2] was developed to calculate both the LJ potential and the Coulombic force. The system uses 76 MODEL chips and is approximately 50 times faster than the equivalent software implementation on a 200 MHz Sun Ultra 2. Another prominent example of hardware acceleration for MD simulations is MD-GRAPE [3], which uses a 1024-piece, fourth order polynomial to approximate the calculation of the force or potential. The coefficients in this polynomial determine which force or potential is calculated. MD-GRAPE only accelerates the force and potential calculation, leaving the rest of the MD simulation to a host processor.

Due to its long development time and little configurability, special purpose hardware is gradually replaced by high density FPGAs. Hardware/software codesign trades off the flexibility and high performance, and it is ideal for executing applications that contain both control-intensive portions and data-intensive portions, such as MD simulations.

3 FPGA Acceleration of Molecular Dynamics Simulation

Recent advances have made FPGAs a viable platform for accelerating MD simulation. The goal of this work is to give an overview on FPGA-based MD Simulations, and to explore the feasibility of FPGA-accelerated MD simulations.

As we know, not all tasks in MD simulation, are suitable for FPGA acceleration. Control-intensive tasks and those executed very efficiently on general purpose processors should be left in software. Prior studies have concentrated on acceleration of different parts of the MD simulation, such as [8] mapping the velocity and position update to FPGA, [6,7] computing Lennard-Jones potentials and forces of a single time step with FPGA, while only a few ones move all tasks into FPGA [4, 5]. In following subsections we describe the acceleration of primary tasks in MD simulation in detail, and make discussion on the precision, performance and improvement.

3.1 Precision and Representation of Variables

Before hardware implementation, the necessary precision must be determined. While software implementations typically rely on the standard IEEE double precision floating-point arithmetic, in hardware such precision is not always necessary. Limited by capacity of FPGA, it is desirable to trade off the data-path width for an increased number of functional units.

An experimental approach to determine the necessary precision [9] found varying amounts of precision were needed for different stages of computation. The stability of the simulation was observed for varying bit widths and the minimum for a stable simulation in terms of energy was considered to be the minimum required precision. There are two classical metrics for simulation quality. One metric is the fluctuation of physical quantities that should stay constant, such as energy, and the other one is the ratio of the fluctuation between total energy and kinetics energy $R = \Delta E_{total}/\Delta E_{kinetic}$. According to this, research in [5] shows that a 40-bit mantissa results in a similarly low energy fluctuation as a full 53-bit mantissa when the time-steps were set to E-15 seconds and E-16 seconds. As to the second metric, it shows that 31 bits are sufficient for time-step of E-15 seconds and 30 bits for time-step of E-16 seconds. The issue of how much precision is required for which kind of MD simulation has not been well studied. But it really is the most important thing should be considered in FPGA design to trade the precision with performance.

3.2 Pair Generation

Only a small fraction of researchers move all tasks of the simulation into hardware, and most studies just focus on the calculation of the force between two particles or the position and velocity update between two time-steps.

Pair generation is the premise of force calculation between particles. In this step, the address of two particles are generated and used to fetch two position vectors from memory. Then the squared distance r^2 are calculated. What should be taken into consideration is the periodic boundary conditions, namely, the distance calculated must be the shortest between any mirror image of each particle.

Suppose the position coordinates of particles are limited to [0, L]. As shown in Fig.2, the minimum distance between A and B in axis x is 3 instead of 5.

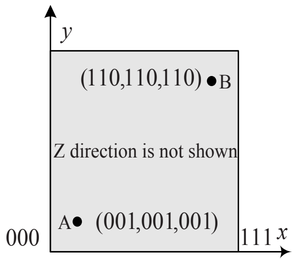


Fig. 2. Minimum mirror effect

How to compute it efficiently? Navid et al [4] gives us a hint. They perform two unsigned subtractions which rely on the natural roll-over of modular arithmetic to achieve the mirrored effect. The lesser one of $X_1 - X_2$ and $X_2 - X_1$ is what we want. But there is a limitation in it, that is, the largest range of position coordinate should equal to L, which is the size of simulation box. In another word, L should be the power of 2. For example, in Fig.1, L=8, the coordinate of position should lie between 000 and 111. Suppose the coordinate of particles A and B are (001, 001, 001) and (110,110,110) respectively, then the lesser value of 101 and 011 is 011. So the squared distance between A and B is 27 in decimal format.

In order to reduce the complexity of control logic, the research of [4, 5] does not make use of neighbor list or cell linked list mechanics. So during the step of pair generation, great deals of squared distances are calculated but not used later because of less than squared cut-off distance. Ronald et al [6] introduce the cell linked list into their design. They use hardware/software approach to perform the MD simulation. The hardware is only responsible for nonbonded force calculation of each particle, and other tasks such as initialization, building neighbor list and Verlet update are performed on a PC. In such method, most of the calculated distances are used to calculate force, which improves the utility of FPGA logics greatly.

3.3 Lennard-Jones Force Calculation

The time-consuming part of a typical MD simulation is the potential and force calculation. Intermolecular forces are usually expressed in terms of formulas containing transcendental functions and other complex operations. Some researchers suggest using look-up tables to interpolate the potential and force values. The simplest interpolation method is linear interpolation such as Newton Forward difference method, which is easy to compute, but needs a sufficiently fine grid to satisfy the accuracy of calculation. An improvement to this method has been suggested by Andrea et al [10], which approximates the potential function with a fifth-order polynomial, and expresses the correspondingly force with the derivative of the potential function. The main idea is shown below.

Suppose the potential function is $u(r)$, and the force associated with this potential is the negative of the gradient of u, which can be written as $\nabla u(r) = \boldsymbol{r}[u'(r)/r]$,where $u'(r)$ is the derivative of u with regard to the scalar distance r. The potential and force calculation problem then is: given the function form of u and a value for r^2, calculate $u(r)$ and $u'(r)/r$. Let $z = r^2$ and $w(z) = u(z^{1/2}) = u(r)$,then $w'(z) = u'(r)/2r$. Thus the problem is converted to: Given functional w and value z, compute $w(z)$ and $w'(z)$. Divide z into a set of grid points z_k. In each interval (z_k, z_{k+1}), the function $w(z)$ and $w'(z)$ are expressed as:

$w(z) = c_0 + c_1\delta + c_2\delta^2 + c_3\delta^3 + C_4\delta^4 + C_5\delta^5$
$w'(z) = c_1 + 2c_2\delta + 3c_3\delta^2 + 4C_4\delta^3 + 5C_5\delta^4$, where $\delta = z_{k+1} - z_k$.

The six coefficients are determined by the exact value of $w(z_i)$,$w'(z_i)$,$w''(z_i)$, $w(z_{i+1})$,$w'(z_{i+1})$ and $w''(z_{i+1})$, and can be calculated beforehand and stored in the table. The author also mentioned that if storage permits it is also useful to store $2c_2$, $3c_3$, $4C_4$ and $5C_5$. In the following part we'll show that even without these values, we still can complete the calculation with FPGA without bringing extra stalls.

Using Horner's algorithm we can transform the formulas above into

$$\begin{aligned}
w(z) &= c_0 + c_1\delta + c_2\delta^2 + c_3\delta^3 + C_4\delta^4 + C_5\delta^5 \\
&= c_0 + \delta(c_1 + \delta(c_2 + \delta(c_3 + \delta(C_4 + C_5)))) \\
w'(z) &= c_1 + 2c_2\delta + 3c_3\delta^2 + 4C_4\delta^3 + 5C_5\delta^4 \\
&= \delta(C_4 + C_5)\delta^2 + \delta(c_3 + \delta(C_4 + C_5))\delta \\
&+\delta(c_2 + \delta(c_3 + \delta(C_4 + C_5))) + \delta(c_1 + \delta(c_2 + \delta(c_3 + \delta(C_4 + C_5))))
\end{aligned}$$

The calculation of $w'(z)$ makes full reuse of the intermediate results of $w(z)$. The pipeline is shown in Fig.3. Because the calculation of $w'(z)$ is not on the critical path, there is no need storing the value of $2c_2$, $3c_3$, $4C_4$ and $5C_5$.

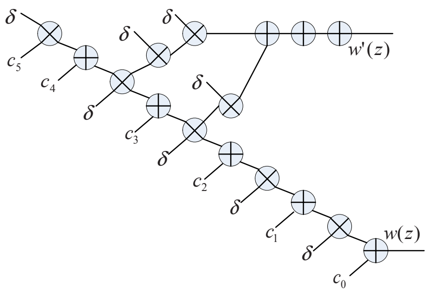


Fig. 3. Shown is a simple pipeline to compute the potential and force simultaneously. The variable δ should be registered to flow in the pipeline.

When simulating systems with more than one species of atoms, larger memory is needed. For two-body potentials, the number of tables is $C_n^2 + n$,where n is the number of atomic species. Compared to linear interpolation, Andrea's method needs more computation but can achieve higher precision.

There is no general comparison between hardware table look-up and FPGA accelerated direct calculation for each potential function, but [11] compare them under software environment. It provides approximate ratios of time for direct calculation versus table lookups for three classes of potentials. The ratio is 1/1 for Lennard Jones, 7/4 for 6-exp, 4/1 for generalized, and 5/4 for Stillinger Webber. If the potential function is not complex enough, it is recommended it be computed by direct calculation or table look-up with linear interpolation; however, if the potential is expressed by transcendental functions, we can use Andrea's method to satisfy the precision and storage requirements while gaining advantage of short latency.

3.4 Verlet Update

The Verlet update module reads the position, acceleration and velocity information from memory and writes them back after update. In conventional software implementations, the Velocity Verlet algorithm is typically performed in two separate steps as shown in Fig 1. In hardware, to implement these two steps by two Verlet modules would be clearly inefficient. So researchers take some measures to simplify the hardware implementation, such as updating the position and velocity to a full time-step in the same step [5], or updating the position and velocity to a full or half time-step respectively in a single step [4].

The first method sacrifices precision for simplicity, while the second one deals with complexity and precision perfectly. In such way, the velocity stored in hardware is not the velocity at a given time-step but the velocity at a half time-step ahead of the current time. The velocity is adjusted back a half time-step by software. The author of [4] did not give the exact deduction and ideas in detail; here we put forward a new method to reorganize and combine the two different steps of consecutive time-steps into one step, shown in Fig.4. In this way, the velocity is a half time-step behind the current time, which also can be adjusted by software.

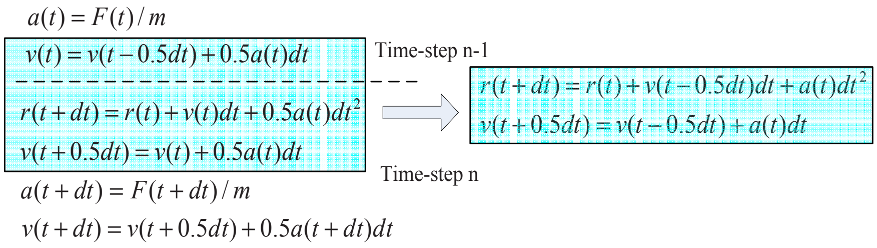

$a(t) = F(t)/m$
$v(t) = v(t-0.5dt) + 0.5a(t)dt$
Time-step n-1
$r(t+dt) = r(t) + v(t)dt + 0.5a(t)dt^2$
$v(t+0.5dt) = v(t) + 0.5a(t)dt$
Time-step n
$a(t+dt) = F(t+dt)/m$
$v(t+dt) = v(t+0.5dt) + 0.5a(t+dt)dt$
$r(t+dt) = r(t) + v(t-0.5dt)dt + a(t)dt^2$
$v(t+0.5dt) = v(t-0.5dt) + a(t)dt$

Fig. 4. Reorganizing steps between two consecutive time-steps

In order to reduce the number of multiplications, [4] integrates the time-step δt into lookup tables of force. That is to say, the product of force and δt is gotten from tables, which can be adjusted by software. There are seldom other techniques to optimize the FPGA acceleration of Verlet update algorithm to a greater extent.

3.5 Memory Access and Pipeline Stalls

During force calculation, the position vectors of particle i and particle j should be read to compute r^2, and the new partial values of force are accumulated to $\boldsymbol{f}_i$ and $\boldsymbol{f}_j$ until all the neighbors of particle i are processed. Because of the pipeline latency in accumulation, the design is susceptible to a read-after-write hazard. The hazard caused by force accumulation on particle j can be reduced by without using Newton's third law, which requires twice as many force calculations. Notice that using such method requires a new neighbor list, which is symmetrical if expressed as a neighbor matrix. However such method still can not remove the hazards caused by force accumulation on particle i. Up to now we have not seen discussions on such problem in FPGA-accelerated MD simulation. But there is related research on accumulation in matrix multiplication and we can use it for reference. Lin Zhuo et al[12] introduce a method for developing scalable floating-point reduction circuits that run in optimal time while requiring only O(lg (n)) space and a single pipelined floating-point unit. Coupled with the O(n) optimal time performance, it is an ideal candidate for floating-point reduction.

3.6 Simulation and Performance

Most research shows that significant speedups can be obtained with FPGA implementation. Due to limits in on-board memory, almost all the systems perform typical 8192-molecule simulations. The design in [4] is targeted on an FPGA platform named TM3 [13], scaled to modern FPGAs running at 100MHz, a speedup over 20× can be achieved over a PC implementation. The design in [5] is implemented on a Wild-star II-Pro board and can obtain a speed-up from 31× to 88×. The scalability of designs in [4, 5] is limited by particle numbers. They do not use techniques like Verlet neighbor list or cell linked-list, so with the increase of particle number, the time consumed in each step increase rapidly. The design in [6] uses hardware/software approach and achieves a 2× speed-up over the corresponding software-only solution. According to our knowledge, the design is the only one that introduces Verlet neighbor list in to design. Although the speed-up it achieves is only 2×, compared to software simulation without using Verlet neighbor list, the speedup will be more than 2×.

Speedup is not the only element to evaluate the design; FPGA resource utilization, area and simulation accuracy should also be taken into consideration. By reducing the area cost and increase the resource utilization, more pipelines can be mapped onto FPGAs, and the speedup can still be improved.

4 Conclusion

In this paper we give an overview on the background and latest FPGA-based research of MD simulations. FPGA development offers great flexibility. With the adjustment of a few parameters, an entirely new FPGA system can be generated to cope with a different amount of particles, with different particle types, and most importantly with a varying amount of precision.

There are several future directions to pursue. One is to investigate possibilities for implementing double-precision simulations. With the development of FPGA technology, it may be possible to realize double-precision simulations instead of single-precision or fix-point arithmetic on future reconfigurable computers that have more on-board memory banks and larger FPGAs. Another direction is to study parallel implementations containing multiple FPGAs. When MD simulations are parallelized over multiple FPGAs, the linked-cell list should be used to reduce the communication cost between processing elements.

Acknowledgement

This work is sponsored by the National Science Foundation of China under the grant NO. 60633050.

References

1. M. Allen, D. Tildeseley, Computer Simulation of Liquids. New York: Oxford University Press, 1987
2. Shinjiro Toyoda, Hiroh Miyagawa, Kunihiro Kitamura, Takashi Amisaki, Eiri Hasimoto, Hitoshi Ikeda, Akihiro Kusumi, and Nobuaki Miyakawa. Development of md engine: High-speed acceleration with parallel processor design for molecular dynamics simulations. Journal of Computational Chemistry, 20(2):185–199, 1999.
3. Toshiyuki Fukushige, Makoto Taiji, Junichiro Makino, Toshikazu Ebisuzaki, and Daiichiro Sugimoto. A highly parallelized special-purpose computer for many-body simulations with an arbitrary central force: Md-grape. The Astrophysical Journal, 468:51–61, 1996.
4. N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable molecular dynamics simulator. In Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, pages 197–206, April 2004.
5. Y. Gu, T. VanCourt, and M. C. Herbordt. Accelerating molecular dynamics simulations with configurable circuits. In Proceedings of the 2005 International Conference on Field Programmable Logic and Applications, August 2005.
6. Ronald Scrofano, Maya Gokhale, Frans Trouw, and Viktor K. Prasanna. A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers. In Proceedings of the 2006 IEEE Symposium on Field-programmable Custom Computing Machines
7. R. Scrofano and V. K. Prasanna. Computing Lennard-Jones potentials and forces with reconfigurable hardware. In Proceedings of the International Conference on Engineering Reconfigurable Systems and Algorithms, June 2004.
8. C.Wolinski, F. Trouw, and M. Gokhale. A preliminary study of molecular dynamics on reconfigurable computers. In Proceedings of the 2003 International Conference on Engineering Reconfigurable Systems and Algorithms, June 2003.
9. Takashi Amisaki, Takaji Fujiwara, Akihiro Kusumi, Hiroo Miyagawa, and Kunihiro Kitamura. Error evalutation in the design of a special-purpose processor that calculates nonbonded forces in molecular dynamics simulations. Journal of Computational Chemistry, 16(9):1120–1130, 1995.

10. T. A Andrea, W. C Swope, H.C Andersen. The role of long ranged forces in determining the structure and properties of liquid water. J. chem. Phys. 79,4576-85, 1983
11. D. Wolff, W.G. Rudd. Tabulated potentials in molecular dynamics simulations. Computer Physics Communications, vol. 120, Issue 1, pp.20-32. 1999
12. L. Zhuo, G. R. Morris, and V. K. Prasanna. Designing scalable FPGA-based reduction circuits using pipelined floating point cores. In Proceedings of the 12th Reconfigurable Architectures Workshop, Denver, CO, April 2005.
13. TM-3 documentation.http://www.eecg.utoronto.ca/∼tm3/.

Reconfigurable Hardware Acceleration of Canonical Graph Labelling

David B. Thomas[1], Wayne Luk[1], and Michael Stumpf[2]

[1] Department of Computing, Imperial College London
{dt10,wl}@doc.ic.ac.uk
[2] Centre for Bioinformatics, Imperial College London
m.stumpf@imperial.ac.uk

Abstract. Many important algorithms in computational biology and related subjects rely on the ability to extract and to identify sub-graphs of larger graphs; an example is to find common functional structures within Protein Interaction Networks. However, the increasing size of both the graphs to be searched and the target sub-graphs requires the use of large numbers of parallel conventional CPUs. This paper proposes an architecture to allow acceleration of sub-graph identification through reconfigurable hardware, using a canonical graph labelling algorithm. A practical implementation of the canonical labelling algorithm in the Virtex-4 reconfigurable architecture is presented, examining the scaling of resource usage and speed with changing algorithm parameters and input data-sets. The hardware labelling unit is over 100 times faster than a quad Opteron 2.2GHz for graphs with few vertex invariants, and at least 10 times faster for graphs that are easier to label.

1 Introduction

Computational biology is a key part of modern research into biological processes and drug development, allowing for virtual lab-work, large-scale biological simulation, and computer driven search processes. This reduces the number of traditional wet experiments required, as well as broadening the scope of experiments that can be considered. One of the enabling factors in computational biology has been the rapid increase in software processing speed over previous decades.

Now that single processor speed-increases are beginning to slow, it is necessary to consider technologies such as multi-processor computers and multi-node clusters. Unfortunately these solutions are large, expensive, and require large amounts of power, as well as presenting problems of application scaling. Another possibility is to use custom hardware, such as FPGAs, to provide acceleration, making each node more powerful, and so reducing the cost, power, and degree of scaling needed per cluster.

This paper presents an architecture, implementation and evaluation of matrix based canonical labelling in hardware. This is a key building-block for many graph based bioinformatics algorithms, and a computational bottleneck in software implementations. The key contributions of this paper are:

– an architecture for the implementation of canonical labelling in hardware;
– resource usage and performance evaluation in the Virtex-4 platform;
– a comparison of the performance of software and hardware canonical labelling units, with an xc4vlx60 showing a 100 times speed-up over a quad Opteron software implementation.

2 Motivation

Many biological and chemical processes are represented using graphs. Protein Interaction Networks (PINs) are one example, which represent biological processes using protein-protein interactions as paths between proteins [1]. These interactions are recovered experimentally, and the functions of different interactions within the graph are initially unknown. One approach for extracting information from these PINs is to find recurring motifs within the network [2].

Frequent Sub-Graph Mining [3] is one technique for looking for these motifs, by enumerating all sub-graphs less than a certain size. After all sub-graphs within the biological network have been counted, the frequency of each sub-graph is compared with the probability of its random occurrence. Any graphs that occur more often than chance would predict may indicate a unit of functionality, or building block, within the biological network. Attempting to find such graphs through human inspection would be almost impossible, and it is only through the use of large amounts of computation power that such techniques are possible.

One problem that occurs when searching for motifs is that a given sub-graph may be counted more than once, as it may be encountered at a number of different points during the search, and the in-memory representation of the sub-graph may not appear the same as when it was encountered before. One way in which the accuracy of searches can be improved is to use canonical labelling to uniquely identify the structure of each node. However, this represents a considerable cost in software, as billions of sub-graphs may be encountered during each search. The approach proposed in this paper is to use software to generate candidate sub-graphs, then to perform the canonical labelling in hardware.

3 Canonical Labelling Algorithm

Canonical labelling is the process of attaching a unique label to graphs, such that any two graphs with the same structure will receive the same label, and any two graphs with different structures will not receive the same label. It has many applications, such as in algorithms for finding graph-isomorphisms [4], and for chemistry and biological applications where repetitions of structures such as molecules and biological structures need to be identified [2].

In this paper the graphs will be considered to be undirected, vertex-labelled, and edge-labelled. The finite set of all vertex and edge labels are L_V^+ and L_E^+ respectively. Each graph is defined as a tuple $G = (V, E, \varphi_V, \varphi_E)$, where V is a finite set of vertices, and $E \subset V \times V$ is a finite set of edges. Graphs must be connected, i.e. for each graph there is a path between every vertex in V by traversing one or more edges in E. The two total functions $\varphi_V : V \mapsto L_V^+$ and $\varphi_E : E \mapsto L_E^+$ assign labels to each vertex and node.

This definition reflects the most complex type of graph commonly encountered in applications, for example where distinct vertices within a molecular graph can represent different instances of the same atom (i.e. have the same vertex label), or where distinct edges between vertices represent one of a number of types of bonding. If $|L_V^+| = 1$ then there is only one edge label, so the graph edges are effectively unlabelled, and similarly if $|L_E^+| = 1$ then the edges are unlabelled. The mappings can also be made injective, in which case the label will uniquely identify each edge or node within the graph.

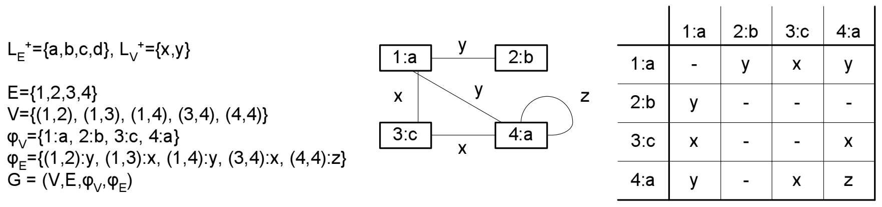
Canonical labelling is a function $\varphi_G : G \mapsto L_G^+$ that takes a graph G, and returns a label L_G for that graph, such that $\varphi_G(G_1) = \varphi_G(G_2)$ if and only if there exists an isomorphism between G_1 and G_2 [5]. For example, if G describes a protein interaction network, where L_V^+ identifies proteins, and L_E^+ describes different types of interactions between proteins, then the canonical label of G uniquely identifies that interaction structure. If the same canonical label is then observed in a different graph, then the two protein interaction networks must be the same. If the same interaction graph is observed in many different places then it is possible the structure is important in its own right, or forms a key building-block for larger structures.

One method for calculating the canonical label is use graph operations, by constructing sets of sub-graphs with certain characteristics, or partitioning the graph in some way. NAUTY [6] is a software package that uses this approach, and is able to label large graphs with many thousands of vertices.

This approach is effective for large graphs, as the asymptotic behaviour of the algorithms is good. However, there is a significant overhead involved in maintaining and manipulating the data structures, so for small graphs the cost of these algorithms is high. In applications searching for motifs the sub-graphs are generally small, only requiring graphs of size less than 24 to be labelled. However, the number of examined sub-graphs may be extremely high, thus the canonical labelling algorithm must be as fast as possible for small graphs.

A different way of calculating the canonical label is to first convert the graph into a matrix form, then to use matrix operations to calculate the canonical label. Figure 1 shows a matrix represented in three different ways, with the set based description on the left, the familiar graphical representation in the middle, and a matrix representation on the right. To allow easy comparison between the representations, the vertexes have been given a unique id number, so node $1 : a$ has the id 1 and label a. Note that the matrix representation is symmetric, as the graph is undirected.

Once the graph is in matrix form, it is possible to define a label for each matrix. The top left of Fig. 2 shows the example matrix, and highlighted in grey are the elements used to define the label. A textual version of the label is shown below, consisting of first the vertex labels, followed by the lower triangle of the edge labels. The canonical label will be defined using the lexicographical minimum


L_E^+={a,b,c,d}, L_V^+={x,y}
E={1,2,3,4}
V={(1,2), (1,3), (1,4), (3,4), (4,4)}
φ_V={1:a, 2:b, 3:c, 4:a}
φ_E={(1,2):y, (1,3):x, (1,4):y, (3,4):x, (4,4):z}
G = (V,E,φ_V,φ_E)
1:a
2:b
3:c
4:a
y
x
y
x
z

	1:a	2:b	3:c	4:a
1:a	-	y	x	y
2:b	y	-	-	-
3:c	x	-	-	x
4:a	y	-	x	z

Fig. 1. Representation of matrix as a set, as a diagram, and in matrix form

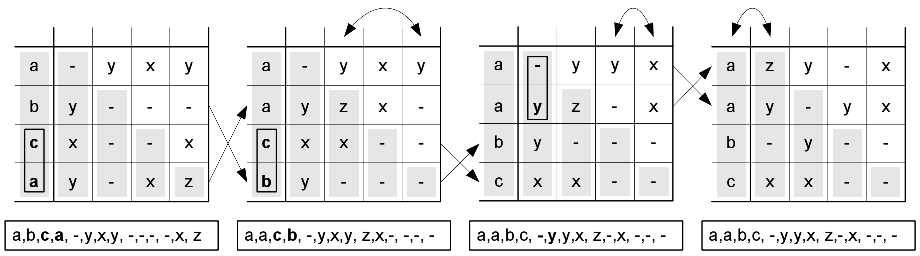

a,b,**c**,**a**, -,y,x,y, -,-,-, -,x, z
a,a,**c**,**b**, -,y,x,y, z,x,-, -,-, -
a,a,b,c, **-**,**y**,y,x, z,-,x, -,-, -
a,a,b,c, -,y,y,x, z,-,x, -,-, -

Fig. 2. Matrix method for canonical labelling

as an ordering. The canonical label is the label that comes first amongst all structure preserving orderings of the graph matrix.

Looking at the label for the top-left matrix it is clearly not the minimum label, as *a* occurs after *c*. A lower label can be reach by swapping these two vertexes, but to maintain the interconnection structure it is also necessary to swap the ordering of the edge matrix, by symmetrically swapping the rows and columns containing *a* and *c*. Note that after swapping both rows and columns the *z* from the lower right corner has moved diagonally within the matrix. Another swap moves *b* and *c* into the correct order, shown in the bottom right of the figure. Now all the vertex labels are in the correct order, but because there are two *a* labels it is still possible to achieve a lower label. The two symbols to the right of the *a* vertices are $-$ and y, and here y is considered to be less than $-$. Swapping these two rows will not affect the relative ordering of the preceding part of the label, as the two *a* elements can appear in either order. After this swap the matrix label in now in its canonical form.

The canonical label can be determined by using a brute-force row swapping algorithm, by using row swaps to examine every permutation of the matrix, and comparing the label of the new matrix against the best matrix after each swap. This will eventually find the correct label, but will take $n!$ swap-compares, so even for relatively small matrices this becomes impractical.

A technique known as vertex invariants can be used to limit the number of combinations that are needed. A vertex invariant is a property that can be assigned to a vertex no matter what order the graph is in. A good example of a vertex invariant is the vertex degree (the number of edges that are incident on the vertex). This information can be used to partition the column of vertices into a number of sorted sub-ranges. Then for any sub-range where all the vertex invariants are equal the brute-force computation can be used.

Another type of vertex invariant is the label of the vertex, which was used implicitly in the example of Fig. 2. The two invariants can be combined using some deterministic function to create a single hybrid vertex invariant: the more distinct invariant labels that a vertex can be assigned, the more likely it is that no other vertex will have that invariant label. Ideally all the vertex invariants will be different, reducing the canonical labelling process to that of sorting the list of invariants. If the corresponding set of swaps is applied to the edge matrix then the resulting matrix will be in canonical form.

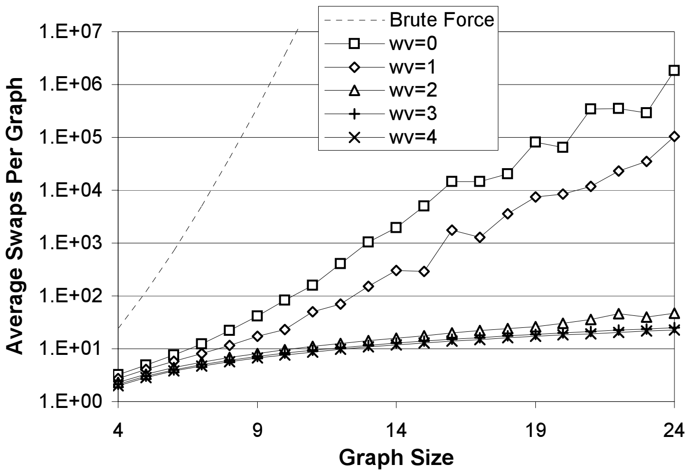


Fig. 3. Average number of swaps required to label random graphs. Brute force is the number of swaps when vertex invariants are not used, $w_v = 0$ uses vertex degrees only, and $w_v = 1, 2, 3$ use vertex degrees and a vertex label randomly chosen from amongst 2, 4 or 8 labels respectively.

Figure 3 examines the computational cost of canonical labelling when applied to random graphs. The random graphs are constructed by creating a set of n nodes, then adding edges between randomly selected nodes until the graph is connected (i.e. there is a path from each node to every other node). Each vertex is also assigned a random w_v bit vertex label, i.e. $2^{w_v} = |L_V^+|$.

The dashed line shows the number of swap-compares for the brute force case, and due to the factorial growth this quickly becomes infeasible to apply. When $w_v = 0$ the only thing that distinguishes between vertices is their degree, and there are likely to many nodes with the same number of degree. This means many ranges must be brute-forced , so the number of swap-compares is high, growing approximately exponentially and on average requiring 10^6 swap-compares for a 24 vertex matrix. In the case of $w_v = 1$ the situation is improved, as some of the ranges of similar degree are now split by the random 1 bit vertex label. However, the growth is still exponential. For larger widths the number of sub-ranges that need to be brute-forced is low on average, so the number of swap compares reduces to an $n \log n$ trend, dominated by the process of sorting the list of vertex invariants.

4 Hardware Implementation

The matrix based canonical labelling algorithm described in the previous section is very simple, but requires irregular memory reads and writes when implemented in software. In this section an architecture for implementing the algorithm in hardware is presented, taking maximum advantage of the parallelism available in the algorithm.

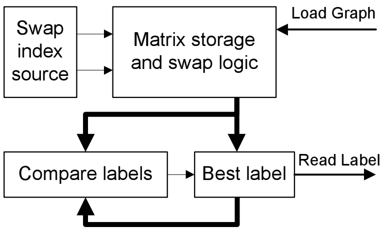


Fig. 4. High level architecture of a canonical labelling unit

Figure 4 shows a high level view of a hardware canonical labelling unit. At the centre is the matrix storage and swap logic, which is responsible both for maintaining the local copy of the graph matrix, using either registers or RAMs, and for swapping rows. Graphs are first loaded into the matrix storage, then the swap index source produces pairs of rows to be swapped. After each swap the label of the current graph is extracted and compared with that of the best known graph. If the current graph's label is lower then the old label is replaced, and once the search space has been explored the minimum label is retrieved.

Within this architecture there is considerable freedom in terms of implementation. For example, the matrix storage could be implemented using registers, distributed RAMs, or block RAMs, and the swap process might take one or many cycles. The number of cycles per swap-compare might be different from turn to turn, for example if the comparison unit searched sequentially for the first element of the label that matched. In such cases the swap and compare units might be decoupled using a FIFO.

In this paper the swap-compare unit is designed to be a single cycle process, so on every cycle a full symmetric matrix swap occurs, and the minimum of the old and new label is captured. There are two clear bottlenecks in this approach: routing data from one position within the matrix to another, and performing the label comparison within a single cycle. It is possible to buffer the label with registers between the swap-compare unit and the comparison, so the two can be considered independently.

Considering the comparison unit first, the length of each label can be determined from the widths of the vertex and edge labels, and the size of the graph. Let $n = |V|$, the size of the graph, and $w_v = \lceil log_2(|L_E^+|)\rceil$, the number of bits per vertex label (allowing that w_v might be zero). The number of bits per edge label is $w_e = \lceil log_2(1 + |L_E^+|)\rceil$, which allows enough bits to hold all edge labels, plus one more state to represent the lack of an edge. At a minimum $w_e = 1$, which allows one label for all edges that exist, and another for edges that don't exist.

The size of each label is $w_v n + w_e n(n + 1)/2$, and in current FPGA architectures the comparison performance will be limited by the linear carry chain propagation. Thus the cycle time of the comparison unit will vary quadratically with the size of the graph. Increasing the size of w_e will also add a large amount of delay. However, w_v will have less affect, which is important if large vertex invariants are to be used.

The matrix storage unit must have at least $w_v n + w_e n^2$ bits of storage to hold the current state of the graph. Because swap-compares occur in a single cycle, no other storage is needed. However, implementing the parallel element routing to allow the swaps will require significant logic resources. Figures 6 and 5 show one way in which this can be achieved. The array consists of two types of cells: diagonal cells, which provide a special case for the behaviour of diagonal elements, and standard cells, which implement the general row and column swapping logic.

Each data bus within the array is composed of an upper and lower bus, which provide the two data-paths required for swapping to take place within a cycle. Figure 5 shows these data buses as thick lines, except for in the top row of the row bus, where the two halves are shown explicitly. During a cycle where indexes a and b (where $a < b$) are to be swapped, the standard cells with index $(i = a, 1 \leq j \leq n)$ will drive the lower half of row bus i and read from the upper half, while the standard cells with index $(i = b, 1 \leq j \leq n)$ will drive the upper half of row bus i and read from the lower half. Similarly the cells in columns a and b drive the lower and upper halves of the column bus.

Diagonal elements require special handling. First, because the matrix is symmetric, element $(a, b) = (b, a)$, so it is not necessary to reflect these elements across the matrix diagonal. It is sufficient to simply disable the two standard cells corresponding to these positions when a cell is in the low row and the high column, or the high row and the low column. Second, in symmetric row swaps the elements running along the matrix diagonal (representing self-connected vertices) are completely independent of the rest of the matrix. Thus the diagonal elements use an independent diagonal bus, with a pair of upper and lower buses as before to allow swaps in a single cycle.

Figure 6 shows the internal arrangement of the standard and diagonal cells. Standard cells are controlled using inputs *colEnHigh*, *colEnLow*, *rowEnHigh*,

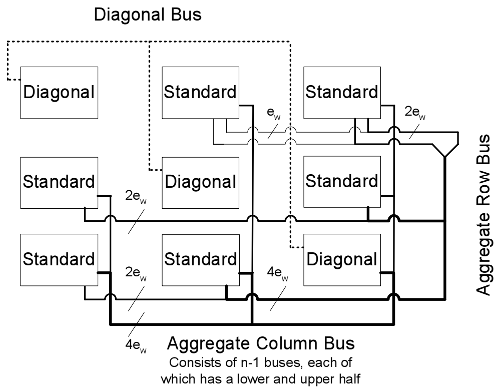


Fig. 5. Data buses between the cells in a swap-compare unit. Note that all data-paths actually contain two buses, as shown for the first row of the row bus, corresponding to the two indices to be swapped.

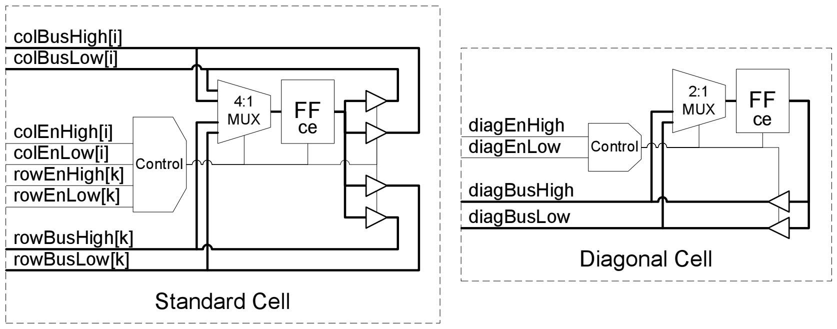


Fig. 6. Edge storage cell used in a swap-compare unit. The four control signals colEn-High..rowEnLow determine whether each standard cell is in one of the rows or columns to be swapped, and if so which of the upper and lower buses to read and write to. The diagonal cell only needs to know whether it is the upper or lower edge of the swap, and reads and writes to the corresponding upper and lower buses.

and *rowEnLow*, which identify whether the cell is in the upper or lower row or column. It is impossible for either $colEnHigh \wedge rowEnHigh$ or $rowEnHigh \wedge rowEnLow$ to occur in a standard cell, as this could only occur in a diagonal location. If only one of the control inputs is true then this identifies that the cell lies only within a row or column. The register's clock-enable is asserted, and the asserted control input determines which of the input buses to select, and which of the output buses to drive. If none of the control inputs are true, or if more than one is true (indicating the cell is on the anti-diagonal), then the register's clock-enable is disabled. The diagonal cell is very similar, but only has one bus.

This architecture was implemented in Handel-C, targeting the Virtex-4 architecture (although the description is platform independent), synthesized using Xilinx ISE 8.1 with default effort and optimisation settings. The code directly reflects the structure outlined above, using two modules containing the swap and comparison modules, with the swap module implemented as a grid of modules representing standard and diagonal cells. The buses were implemented using the *signal* language feature of Handel-C, which act as tri-state buses, but are implemented in hardware as multiplexors. The implementation also contains a graph load facility, allowing a new graph state to be loaded column by column in n cycles, and a label extraction facility, also taking n cycles. The code is also parametrised to allow different widths of w_e and w_v (including $w_v = 0$).

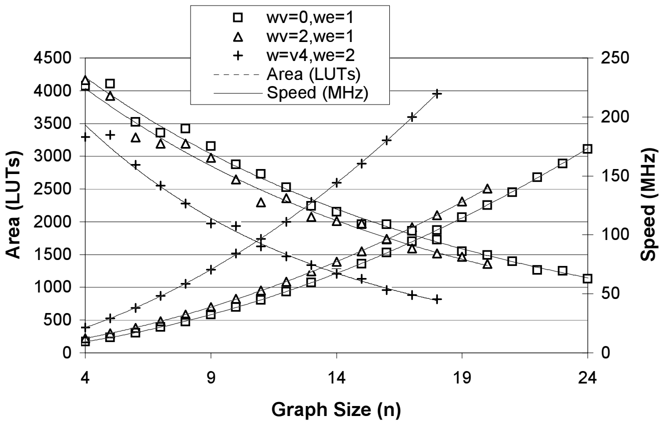


Fig. 7. LUT usage and speed for different combinations of vertex and edge label width

Figure 7 shows the resulting resource usage in LUTs, and speed in MHz for three different combinations of vertex and edge label width. The results are shown for increasing graph size, varying between 4 and 24. The results for the larger label widths are truncated due to tool-chain limitations. Quadratic curves are fitted through the resource usage, showing that as expected LUT usage is almost exactly quadratic with increases in graph size. Exponential curves are fitted to the frequency results, and provide a reasonable fit given the level of noise from the tool-chain.

Figure 8 focusses on the case where $s_v = 0$ and $w_e = 1$, i.e. an unlabelled graph. This represents the hardest case to label, as there are few vertex invariants, and also occurs in many applications, so it is of particular interest. Resource usage is now broken down into registers, LUTs and slices, and again the quadratic fit is near perfect. The quality of the quadratic and exponential model as predictors of resource usage and speed make it possible to get a good estimate for the performance of a hardware implementation without necessarily synthesising it.

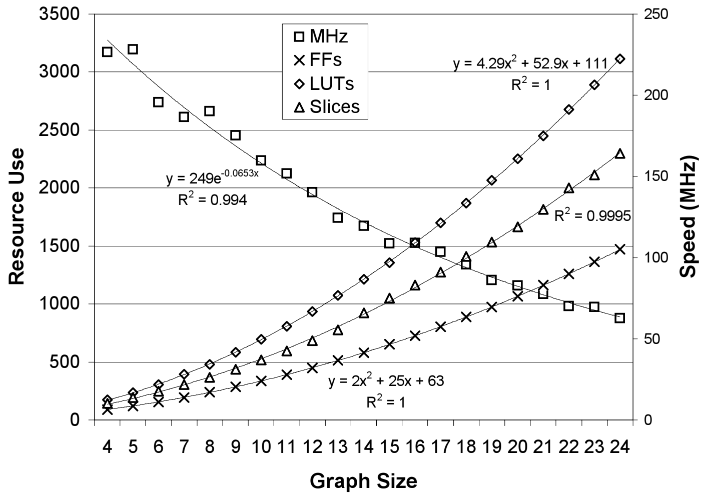


Fig. 8. Resource usage and frequency for an unlabelled graph swap unit (i.e. wv=0,el=1). Quadratic curves are fitted to the resources used, and an exponential curve to the clock frequency, showing an extremely good fit in all cases.

5 Evaluation

In this section we compare the performance of the hardware graph labelling architecture with a software implementation. The software was implemented in C++, and is templatised on the size of the graph, allowing as much constant propagation as possible. The code was compiled using g++ 3.4.5 with -O3, then executed on all four processors of a quad Opteron 2.2GHz machine. Each data-point reflects measurements over a run of at least 10 seconds.

Figure 9 compares the performance of software and hardware purely in terms of the raw number of swap-compare steps per second. The software provides around 27MSteps/sec for the smallest graph size, decreasing down to 3.5MSteps/sec for 24 vertex graphs. By comparison a single graph labelling hardware instance starts at 226MSteps/sec, decreasing down to 61MSteps/sec, providing about a 10x speed-up over the quad Opteron in general.

The graph also estimates the performance if an entire xc4vlx60 device is devoted to canonical labelling. This figure was obtained by estimating the number of replicated instances that could be supported using slice counts, then scaling the speed by the number of instances. Although this ignores many practical problems, such as the amount of hardware dedicated to IO, and the drop in clock rate when targeting a congested chip, it does give some idea of the maximum speed-up possible. For small graphs, where the degree of replication is high, the estimated xc4vlx60 raw performance is around 1000 times that of software, and remains over 100 times faster over all the graph sizes tested.

Figure 10 provides a more practical measurement of performance, as it was derived from cycle accurate simulation of the algorithm using random graphs.

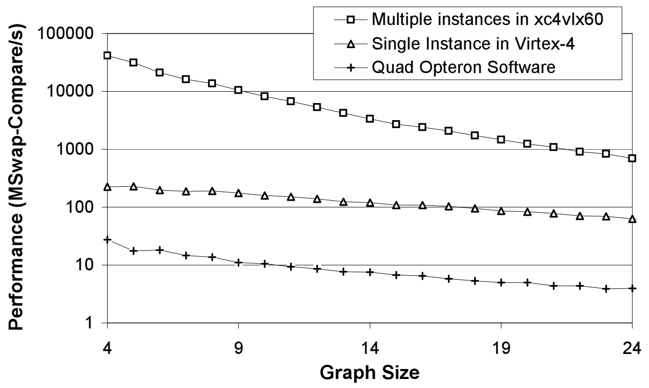


Fig. 9. A comparison of the raw swap-compare performance between the Virtex-4 hardware and a Quad Opteron software implementation. Hardware performance is shown both for a single graph labelling unit, and for an xc4vlx60 device filled with the maximum number of units that will fit.

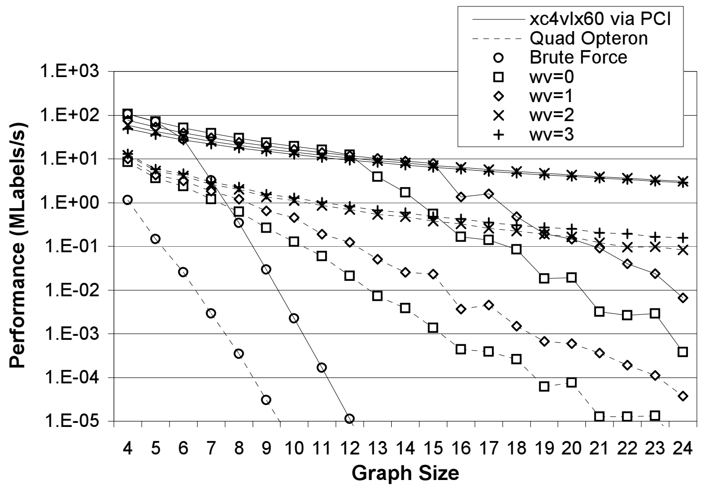


Fig. 10. Practical performance when labelling random graphs for a Virtex-4 xc4vlx60 device compared with that of a Quad Opteron. The FPGA performance includes cycles lost due to loading the graph and extracting the label, and is also bandwidth limited to 133MBytes/s to simulate a connection over a PCI bus. For all graphs of size less than 12 the FPGA is bandwidth limited (except in the Brute Force case), and for $wv = 2$ and $wv = 3$ the computational load stays so low relative to IO, due to the small number of swaps per graph, that the FPGA is bandwidth limited for all graph sizes.

The hardware costs also include the time taken to load input graphs and read back labels between labelling operations. The hardware is also assumed to be connected to a PC via a 133MByte/sec PCI bus, so hardware performance is limited by the bandwidth required to transfer the graphs. The small number of swap-compares required when $w_v \geq 2$ mean that the hardware becomes completely IO limited, and for these graphs can only achieve about a ten times speed-up over the quad processor software. However, for $w_v \leq 1$ the computational load quickly increases to the point where bandwidth is not the limiting factor, and for graph sizes between 12 and 15 the hardware becomes computationally limited. Even before this point is reached the software speed degrades exponentially, so practical speed-ups over 100 times are possible.

6 Conclusion

This paper has presented an architecture, implementation and evaluation of matrix based canonical labelling in hardware. The raw processing rate of a single hardware labelling unit is approximately 10 times that of a quad processor software implementation, and by utilising multiple labelling units within an FPGA, the raw performance is over 100 times that of software.

In a simulation of practical graph labelling, a minimum speed-up of around 10 times is predicted. However, for graphs with few vertex invariants, such as unlabelled graphs, a speed-up of around 100 times is expected. This is important, as it means that the largest speed-up is expected in those cases that are the most computationally expensive in software.

Future work will focus on benchmarking the real-world performance once the hardware labelling unit is integrated into a software application. This will explore the effect that software to hardware bandwidth has on performance, and how to store and transmit graphs to allow efficient processing in both software and hardware.

References

1. Bu, D., Zhao, Y., Cai, L., Xue, H., Zhu, X., Lu, H., Zhang, J., Sun, S., Ling, L., Zhang, N., Li, G., Chen, R.: Topological structure analysis of the proteinprotein interaction network in budding yeast. Nucleic Acids Research **31**(9) (2003) 2443–2450
2. Huan, J., Wang, W., Washington, A., Prins, J., Shah, R., Tropsha, A.: Accurate classification of protein structural families based on coherent subgraph analysis. In: Proc. Pacific Symposium on Biocomputing (PSB). (2004) 411–422
3. Huan, J., Wang, W., , Prins, J.: Efficient mining of frequent subgraph in the presence of isomorphism. In: Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM). (2003) 549–552
4. Kuramochi, M., Karypis, G.: Frequent subgraph dscovery. Technical Report TR-01-028, Department of Computer Science, University of Minnesota (2001)
5. Fortin, S.: The graph isomorphism problem. Technical Report TR-96-20, University of Alberta (1996)
6. McKay, B.: Practical graph isomorphism. Congressus Numerantium (1981) 45–87

Reconfigurable Computing for Accelerating Protein Folding Simulations*

Nilton B. Armstrong Jr.[1,2], Heitor S. Lopes[1], and Carlos R. Erig Lima[1]

[1] Bioinformatics Laboratory, Federal University of Technology – Paraná (UTFPR), Av. 7 de setembro, 3165 80230-901, Curitiba (PR), Brazil
[2] Artificial Intelligence Division, Technology Institute of Paraná, Curitiba (PR), Brazil
narmstrong@tecpar.br, hslopes@pesquisador.cnpq.br, erig@utfpr.edu.br

Abstract. This paper presents a methodology for the design of a reconfigurable computing system applied to a complex problem in molecular Biology: the protein folding problem. An efficient hardware-based approach was devised to achieve a significant reduction of the search space of possible foldings. Several simulations were done to evaluate the performance of the system as well as the demand for FPGA's resources. Also a complete desktop-FPGA system was developed. This work is the base for future hardware implementations aimed at finding the optimal solution for protein folding problems using the 2D-HP model.

1 Introduction

Proteins are complex macromolecules that perform vital functions in all living beings. They are composed of a chain of amino acids, and their function is determined by the way they are folded into their specific tri-dimensional structure. This structure is called its native conformation. Understanding how proteins fold is of great significance for Biology and Biochemistry.

The structure of a protein is defined by its amino acid sequences. If we use a complete analytic model of a protein, the exhaustive search of the possible conformational space to find its native conformation is not possible, even for small proteins. To reduce the computational complexity of the analytic model, several simple lattice models have been proposed [4]. Even so, the problem is still very hard and intractable for most real-world instances [1]. The solution is either using heuristic methods that do not guarantee the optimal solution [7] or some scalable strategy capable of intelligently sweep the search space and find the optimal folding (that corresponds to the native conformation).

Reconfigurable computation is a methodology that has been sparsely explored in molecular Biology applications. For instance, [10] presented a new approach to compute multiple sequence alignments in far shorter time using FPGAs. In the same way, [11] describe the use of FPGA-based systems for the analysis of

* This work was partially supported by the Brazilian National Research Council – CNPq, under research grants no. 305720/2004-0 and 506479/2004-8.

DNA chains. A reconfigurable systolic architecture that implements a dynamic programming algorithm and can be used for sequence alignment was presented by [5]. Sequence alignment was also focused by [9], where a pipeline architecture was implemented using reconfigurable hardware. In addition, [8] presents a parallel hardware generator for the design and prototyping of dedicated systems to the analysis of biological sequences. However, there are still few research exploring the use of FPGAs in Bioinformatics-related problems.

On the other hand, recently, we have witnessed a pronounced growth of the hardware and software technologies for embedded systems, with many technological options arising every year. The use of open and reconfigurable structures is becoming more and more attractive, especially due to its robustness and flexibility, facilitating the adaptation to different project requirements.

The possibility of massive parallel processing makes reconfigurable computing (that is, systems based on reconfigurable hardware) an attractive technology to be applied to the protein folding prediction problem addressed here. Hence, the need for powerful processing of biological sequences, on one hand, and the appealing flexibility and performance of reconfigurable logic, on the other hand, are the primary motivations of this work.

The main goal of this project is to develop a methodology for sweeping all possible folding combinations of a protein, using the 2D-HP model [3], in order to find the conformation in which the number of hydrophobic contacts is maximized (as explained in section 2).

2 The 2D-HP Model for Protein Folding

The Hydrophobic-Polar (HP) model is the simplest and most studied discrete model for protein tertiary structure prediction. This model was proposed by Dill [3], who demonstrated that some behavioral properties of real-world proteins could be inferred by using a simple lattice model. The model is based on the concept that the major contribution to the free energy of a native conformation of a protein is due to interactions among hydrophobic (aversion to water) amino acids. Such amino acids tend to form a core in the protein structure while being surrounded by the polar or hydrophilic (affinity to water) amino acids, in such a way that the core is less susceptible to the influence of the solvent [6].

The HP model classifies the 20 standard amino acids in two types: either hydrophobic (H) or hydrophilic (P, for polar). Therefore, a protein is a string of characters defined over a binary alphabet {H,P}. Each amino acid in the chain is called a residue. In this model, the amino acids chain is embedded in a 2-dimensional square lattice. At each point of the lattice, the chain can turn 90^o left or right, or else, continue ahead. For a given conformation to be valid the adjacent residues in the sequence must be also adjacent in the lattice and each lattice point can be occupied by at most one residue. A very simple example is shown in Fig. 4.

If two hydrophobic residues occupy adjacent grid points in the lattice, not considering the diagonals, but are not consecutive in the sequence, it is said that a non-local bond (or H-H contact) occurs. The free energy of a conformation is inversely proportional to the number of H-H contacts. This yields two basic characteristics of real proteins: the protein fold must be compact and the hydrophobic residues are buried inside to form low-energy conformations [6]. The protein folding problem may be considered as the maximization of the hydrophobic non-local bonds, since this is the same as the minimization of the free energy of a conformation in this model.

Although simple, the folding process with the 2D-HP model has behavioral similarities with the real process of folding [3]. Notwithstanding, from the computational point of view, the problem of finding the native structure using the 2D-HP model was proved to be NP-complete [1,2]. Thus, many heuristic algorithms have been proposed to solve this problem [7].

3 Methodology

The structure of the developed system is composed of several hardware and software blocks, described in the next subsections. The system was designed to be capable of analyzing the folding of proteins using a FPGA as a sort of co-processor of a desktop computer. This approach takes the advantage of the FPGA's flexibility and processing power and is integrated with the desktop computer by a user-friendly interface. Also, intelligent strategies are proposed, leading to a dramatic reduction of the search space.

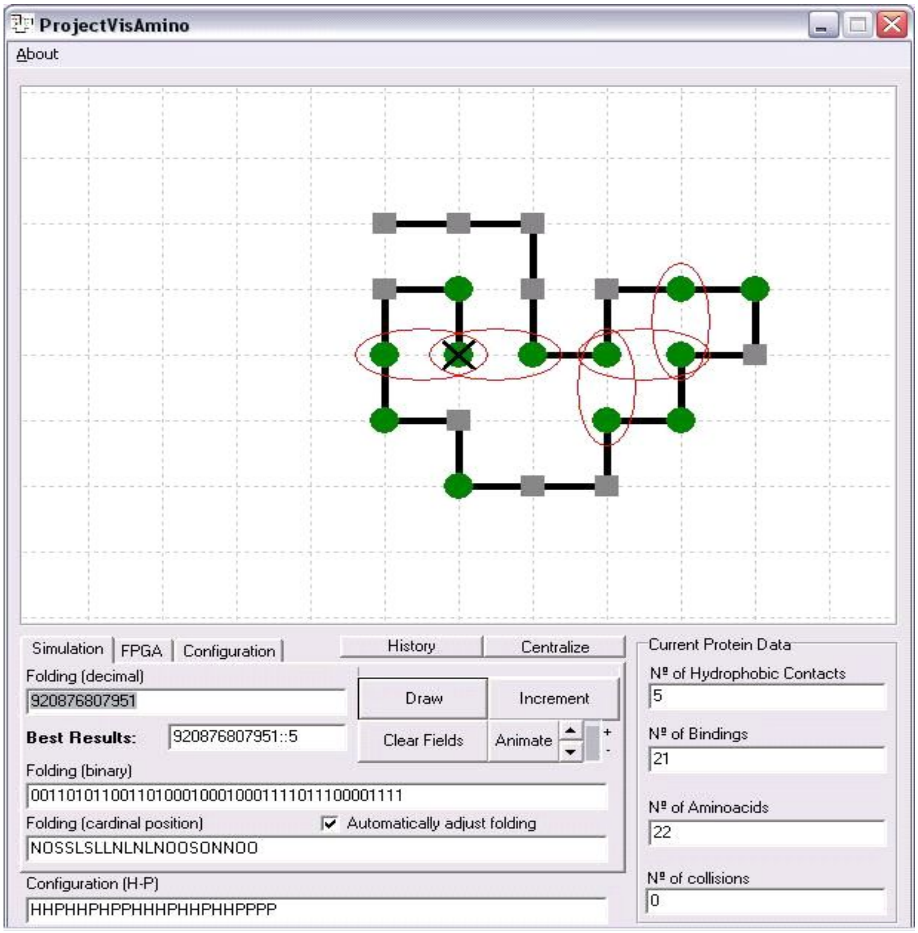
3.1 User Interface

The software developed to run in the desktop computer implements a user-friendly visual interface that enables anyone to easily understand the 2D-HP model. This software was developed in C++ language and it is composed by three basic modules.

The first module is aimed at enabling the user to have a fast visual reference of the protein being analyzed. It automatically displays, on the screen, important information to the researcher: the number of contacts (both graphically and numerically), the collision between amino acids (with graphical marks), and three ways of representing the folding itself: a decimal notation (that is easy to handle), an absolute positional notation (that is easy to understand), and the binary notation (that is the way the FPGA sees the folding). Fig. 1 presents a screen shot of the graphical user interface.

The second module of the software is aimed at providing an intuitive way of exploring the conformational search space. The exhaustive exploration of search space is strongly reduced by using intelligent strategies.

The third module is the communication interface with the FPGA device. As a matter of fact this module does not really communicate directly with the FPGA. It was designed to do the required handshake with a Motorola MCU (Microcontroller Unit) MC68HC908JB8. This MCU is especially suitable for USB (Universal Serial Bus) applications. Considering the small amount of data to be transferred between the desktop computer and the dedicated hardware in our specific



Folding (decimal)
920876807951

Best Results: 920876807951::5

Folding (binary)
00110101100110100010001000111101110000 1111

Folding (cardinal position)
NOSSLSLLNLNLNOOSONNOO

☑ Automatically adjust folding

Configuration (H-P)
HHPHHPHPPHHHPHHPHHPPPP

Current Protein Data

Nº of Hydrophobic Contacts
5

Nº of Bindings
21

Nº of Aminoacids
22

Nº of collisions
0

Fig. 1. Screen shot of the graphical user interface

application, the implemented USB interface is adequate. The implementation of an USB interface external to the FPGA saves internal resources of the device and gives more flexibility to the project. Based on the MCU's architecture a small kit was built to support its operation and allow it to communicate, simultaneously, with the desktop computer and the FPGA. Therefore, a communication protocol has was developed to allow the desktop computer to use the MCU just as a data gateway to reach FPGA. The FPGA works as a slave of the MCU and so it does to the desktop computer. Therefore, the computer communicates to the MCU through the USB and the MCU uses its multi-purpose pins to communicate with the FPGA. When an analysis is to be done, the desktop computer sends the hydrophobicity data related to the protein to the MCU through the USB interface. Then, the MCU serializes this data and shift it into the FPGA input shift register. When the analysis is complete, the MCU acknowledges this completion

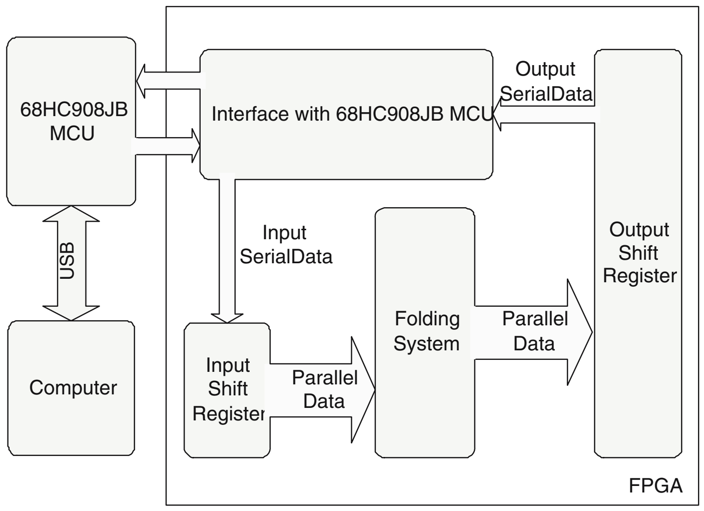


Fig. 2. FPGA solution block diagram

and sends a message to the desktop computer. Next, the computer requests the MCU to retrieve the results and it assembles the data clocked out from the FPGA's output shift register. This FPGA's internal structure is shown in Fig. 2, which we called of FPGA Solution. This architecture based on shift registers to enable a serial communication between the FPGA and the MCU. This serial protocol was used in order to avoid the problems related to the variable width of a parallel bus. With this serial approach, the only thing that changes with the change of the number of amino acids is the number of clock pulses that have to be issued to send the input data and receive back the results.

3.2 Topology of the Folding System

Fig. 3 shows a functional block diagram of a hardware-based system for finding the optimum conformation (folding) of a protein. This system uses the primary structure of a protein and is based on the 2D-HP model. Basically, a counter will swap all possible conformations, according to a given encoding (section 3.3). Conformations have to be converted to Cartesian coordinates (section 3.5) and then checked for validity (sections 3.4 and 3.6). This validity checker reduces the search space analyzed by the next block. After, the number of H-H contacts is counted for the valid conformations found. The conformation with the largest number of contacts is kept and this is one of the solutions for the problem.

3.3 Representation

In order to efficiently sweep the conformational space, the central issues to be addressed is how to represent a protein chain folded in the 2D-HP model using reconfigurable logic, and how to browse the search space.

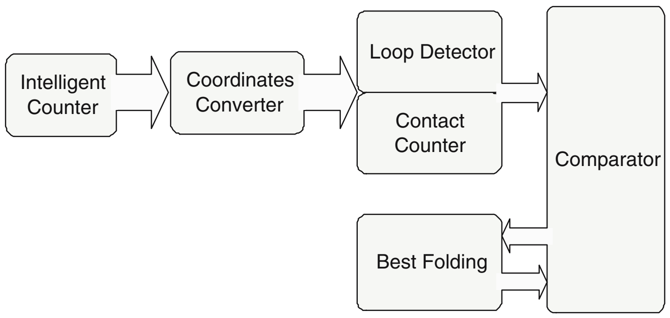


Fig. 3. Functional blocks of the proposed folding system

To solve the representation problem, an absolute positional convention and hydrophobicity information was defined. Basically, only the relative positional information was stored, saving the system from storing the set of Cartesian coordinates of the amino acids in the lattice. This convention is simple and comprises the four possible relative folding directions: North (N), South (S), East (E) and West (W), encoded with two bits, respectively, 00, 01, 10 and 11, and stands for the bindings between the amino acids. Therefore, a complete fold of N amino acids has $(N - 1)$ bindings and is represented by a $2(N - 1)$ long binary number. It is important to note that this representation, by itself, does not avoid any collisions among the amino acids, it needs a validity check, as will be explained later in section 3.6.

Another relevant information is the hydrophobicity data (HD), that is, a single binary number representing which amino acids are Hydrophobic (bit 1) and which are Polar (bit 0). Therefore, an entire protein can be represented by two binary numbers: its positional information and its HD configuration. According to this convention, Fig. 4 shows an example of a hypothetical 6 amino acids-long protein fragment, its representation and how this specific folding would be represented in the lattice. Black dots represent hydrophobic amino acids, and white dots, the polar ones. The square dot indicates the first amino acid of the chain. However, as our model uses an absolute coordinate system, it is necessary to define an initial point as it would result in different folding, yet belonging to the same set of possible foldings for that protein. It is important to notice that both information are read from the left to the right, meaning that the leftmost amino acid is represented by the leftmost letter in the HD info and the leftmost pair of bits in the absolute positional code.

3.4 Intelligent Counter

The straightforward advantage of the binary representation mentioned before, from the folding perspective, is that it enables the creation of a single step binary counter to generate every possible folding (described by a binary number) for a given amino acids chain. In other words, each count would represent a different folding. However, there is a serious drawback. For a N amino acids-long protein

HD info: H H P P H P
HD info code: 1 1 0 0 1 0
Absolute position: E N E S E
Absolute position code: 1000100110
Decimal notation: 550
Cartesian coordinates:
(0,0);(1,0);(1,1);(2,1);(2,0);(3,0)

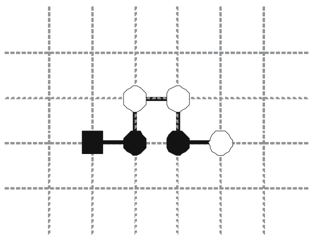


Fig. 4. Example of a folded protein fragment and its 2D-HP representation

it is necessary a number of bits that is almost twice the number of bindings $(2(N-1))$, according to the proposed representation, it would result in $2^{2(N-1)}$ possible foldings. For instance, to analyze a 50 amino acids protein there would be 2^{98} (or $\approx 3.16913 \times 10^{29}$) possible combinations. Such a combinational explosion could render the counter unlikely to sweep through all these combinations in an useful time, even considering a typical maximum clock of 500 MHz of modern FPGA devices.

If it was possible to analyze a valid folding in a single clock time, with a 500MHz clock, it would result in a analysis time of 6.34×10^{20} seconds or 2.01×10^{13} years for 50 amino acids proteins. Besides, such a high clock speed is very difficult to achieve in physical implementations, thus increasing even more the processing time.

However, checking closely the physical behavior of the folding representation, it can be noticed that the folding must follow a self-avoiding path in the lattice. That is, if the previous fold was to the North direction, the next fold cannot be to the South. The same applies to the West-East directions. According to the HP model, these foldings are invalid. In a valid protein conformation a point in the lattice can have at most a single amino acid. Thus, there is no reason to consider any folding that violate this rule, leading to the need of preventing the system of analyzing them, as they are previously known to be invalid. These violations were named of Rule2 violations, for being related to consecutive and adjacent invalid foldings. This counter approach attempts to foresee an invalid folding, according to the Rule2, and skips all binary numbers that could contain that invalid folding at that position.

Consequently, we created an intelligent counter that generates only Rule2 compliant foldings. It can be proved mathematically (not shown here) that using this type of counter a significant reduction of the combinations in the search space is obtained.

Notice that Rule2 does not prevent violations caused by the overlapping of distant amino acids in the chain, as a consequence of a loop in the folding. Although these loop violations are desirable to be eliminated from the analysis, they were not removed from the counting as they are very difficult to be previewed, as will be explained later. As a matter of fact, we already developed a single-clock parallel method to solve this issue. This new approach not only detects any collision in the folding but also counts the number of H-H contacts. However, it is being tested but already represents a significant enhancement to this algorithm.

Another important feature addressed in this counter are two other search space reductions which, even not being as huge as the Rule2 elimination, they also contribute to enhance the processing time. The first one is related to the elimination of repetitive foldings. As shown in Fig. 5, for a protein fragment with 3 amino acids, if all of the possible foldings of this fragment were drawn, it can be seen that there is a pattern (shown in light gray) that repeats itself, rotated in the plane. Since each of the four occurrences of this pattern contains exactly the same set of foldings, 3/4 of the possible foldings can be discarded saving processing time. This strategy adds no complexity to the design of the system. That is, only the most significant pair of bits, which encodes the binding between the first and the second amino acid, would not be part of the counter having its value fixed in "00" or North (see Fig. 5). In other words, the binary count is simply limited to 1/4 of its whole range. Applying this feature to the Rule2, the search space is divided by 4, reducing yet further the processing time.

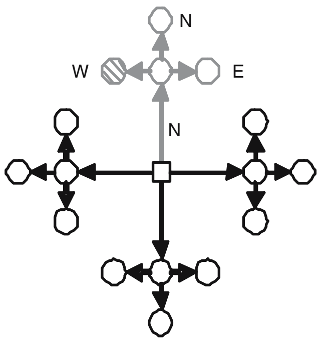


Fig. 5. Sketch of all possible foldings for a hypothetic protein fragment with 3 amino acids. The square dot is the initial amino acid.

The second reduction to the search space can be also seen in Fig. 5 (in light gray). One more duplicate folding can removed, which is related to the mirroring of the foldings from the right to the left of the central vertical axis. This reduction is achieved by making the counter skip every number composed of a folding to the West preceded purely by North folding. In Fig. 5, the eliminated folding would be the NW folding (which is hatched), leaving only two foldings left to be analyzed out of the original search space of 16 foldings. Notice that such elimination of repeated foldings still leaves a set of foldings that represents all the important folding information. Considering the use of Rule2, overall the search space is exponentially reduced. It can be demonstrated that effective search space behaves as: $y = 0.2223.e^{-0.2877.x}$, where x is the size of the amino acid chain and y the effective percent of the search space to be swept by using Rule2.

3.5 Coordinates Converter

The output of the counter, representing a given conformation, has to be converted into Cartesian coordinates (see Fig. 4) so as to effectively embed the amino acid chain in the lattice. Notice that the absolute position encodes only the bindings between the amino acids. However, the Cartesian coordinates represents the amino acids themselves.

Using the first amino acid as reference, the coordinates are generated by a combinational circuit, in real-time, for the whole protein. When a new count, or folding, is generated, based on the absolute reference of the bindings of the protein, the position occupied by each amino acid is promptly computed. This process is done in such a way that the system generates for the current folding being analyzed all the Cartesian Coordinates of its amino acids, all in a parallel circuit. These coordinates need not to be stored as they will be stable for as long as the output of the counter is stable. Therefore, in order for an analysis to be carried out, the counter must be frozen until the Loop Detector and Contact Counter enables it to generate the next folding.

3.6 Loop Detector and Contact Counter

The detection of loops is done at the same time the number of contacts of a valid folding is computed. This block checks for valid conformations, in which there are no overlapped aminoacids. For this purpose, we used the absolute positional information generated by the intelligent counter (section 3.4), based on the Cartesian coordinates explained in section 3.5.

The model intends to validate the current folding as a collision-free chain (i.e. a self-avoiding path). This module is composed basically by a finite state machine (FSM) that has the purpose of accomplishing this validation by building a circuit capable of detecting any pair of identical Cartesian coordinates. The FSM does sequential comparisons, starting at coordinates $(0, 0)$, until the last pair. For each new coordinate pair read, a comparison is carried out with all the pairs yet to be analyzed, to check for collisions of any distant amino acids with the current one. Simultaneously, if the amino acid is hydrophobic, its neighborhood is checked for non adjacent hydrophobic amino acids. As new H-H contacts are found, a contacts counter is incremented. Therefore, this block performs two functions at the same time: detects loops (invalid foldings) and counts H-H contacts (for the valid foldings). If a loop is found, invalidating the folding, the process is aborted and the intelligent counter is requested to generate the next folding.

Currently, this approach stores the absolute position code of the first occurrence of the highest contact count and also its hydrophobic count. Any subsequent occurrence of a folding with the same number of contacts is discarded. To date, there is no known method for predicting how many occurrences of the optimum folding will appear for a given HD configuration.

The main drawback of this approach is that each coordinate pair has to be compared with almost all of the pairs yet to be analyzed. This yields a number of comparisons (nc) computable by the expression: $nc = \Sigma_{i=1}^{na-1} i$, where na is the number of amino acids of the chain. This expression represents the sum of a linear progression. Therefore, the number of comparisons grows quadractically as the number of amino acids increases.

4 Results

In order to evaluate the efficiency of the proposed approach, two sets of experiments were done: simulations in software and hardware implementation. The results of these simulations are grouped in a single table (Table 1).

Several simulations were done using the Quartus II environment from Altera (*http://www.altera.com*). The motivations for these simulations are as follows:

– Check if the system can really identify the first occurrence of the optimum folding, compared to the known value of a benchmark.
– Determine the required processing time for foldings with a given number of amino acids and, further estimate the time required to process larger proteins.
– Estimate the FPGA's resources usage growth with the increment of the size of the amino acids chain.

Table 1. (Left)Comparison of software simulation and hardware. (Right) Resources usage and maximum clock.

na	t_{opt}	t_{sim}	t_{hard}	t_{pc}
6	2.08E-6	1.21E-5	-	-
7	2.93E-6	4.84E-5	-	-
8	1.32E-5	2.27E-4	-	2.00E-2
9	1.52E-4	9.63E-4	-	5.00E-2
10	6.98E-5	3.68E-3	4.00E-3	1.30E-1
11	7.26E-4	1.37E-2	1.50E-2	4.01E-1
12	-	-	6.20E-2	6.00E+1
13	-	-	1.87E-1	3.00E+1
14	-	-	9.90E-1	1.20E+2
15	-	-	2.03E+0	3.00E+2

na	$ALUT's$	Clk_{max}
4	106	274.88
5	173	228.26
6	243	217.78
7	268	235.18
8	442	205.47
9	520	186.39
10	604	184.37
11	687	177.12
30	3241	106.40
50	7611	73.42

For the hardware experiments we used an Altera Stratix II EP2S60F672C5ES device. Each on-board simulation was done considering that the system will supply results to a desktop computer, by reading directly the FPGA's internal memory. Every experiment respected the clock restrictions of the whole system, which is known to decrease as the internal logic is increased. The hardware experiments were carried out using the complete software solution, described earlier.

Table 1(left) shows the processing time needed to find the optimum folding (t_{opt}) and the total processing time necessary to sweep the search space of possible foldings for the simulation, the software (t_{sim}) and the hardware (t_{hard}) implementations. It is also presented the processing time using a desktop computer

(t_{pc}) with Pentium 4 processor at 2.8 GHz. These results are merely illustrative, since the timer resolution of a PC is 1 millisecond, and the algorithm (implemented in C language) is not exactly the same as the one simulated in hardware.

Table 1(right) shows the resources usage of the FPGA device for a growing number of amino acids chains. It is important to note that these values are specific to the FPGA device chosen and may be different for other chips. Column Clk_{max} is the speed the system is able to run in MHz, according to each amino acid chain simulated. The term $ALUT's$ (Adaptive Look-Up Table) is an Altera specific naming used to represent the amount of internal functional logic blocks within the chip. Actually, the chosen FPGA has 48,352 ALUTs (corresponding to 60,440 logic elements).

5 Conclusions

Results of the simulations showed that the proposed algorithm is efficient for finding the optimal folding. The number of H-H contacts found by the system for each simulation did match the expected value of the benchmark. Therefore, the proposed methodology for solving the protein folding problem gives correct answers and is reliable to run such analysis.

The integration between the software and hardware indeed performed well, as expected. Certainly the graphical user interface contributed to the development and the validation of the proposed algorithm. Moreover, the USB interface, using the Motorola MCU, also added flexibility to the project, allowing a transparent communication between the desktop and the FPGA. This USB-based communication protocol allows further development of the systems, especially with other improved protein-related analysis. Future development will focus on a USB 2.0 interface, allowing faster data transfers.

Regarding the growth of resources usage, the implementation behaved within satisfactory limits. Despite this growth is not linear with the increase of the amino acids chain, it does not increase exponentially. The maximum allowed clock cycle decreased slower than expected, and it still can be run in a fair speed even with 30 or 50 amino acids.

The main focus of this work was to devise a methodology for finding the optimal folding of a given protein sequence, in a feasible processing time. This objective was achieved thanks to Rule2 that allowed a dramatic reduction the search space. In some cases, less than 0.001% of the original search space has to be analyzed. These achievements were possible only due to the features of reconfigurable computing, especially the parallelism.

Since one of the main purposes was to find the first occurrence of the optimum folding, sweeping the search space as fast as possible, the system achieved this goal. Recall that the optimum folding is the one with the highest number of H-H contacts. The Table 1(left) shows that the time necessary to run the complete analysis is indeed small, when compared to the time needed by the software implementation. It also shows that the hardware implementation in fact yields a significant improvement on the total analysis time. However, the time required to accomplish the analysis still grows too fast, and can be a serious limitation for sequences with an increased number of amino acids.

A few points should yet be addressed by future versions of this system. Since the search space grows faster, the main priority would be devising strategies for reducing it even more, perhaps by predicting and avoiding any folding that could contain a collision not discarded by Rule2. The idea would be to build a "RuleN" algorithm, in which any collision could be previously detected, not only the adjacent collisions, as does Rule2, yet keeping the capability of counting the H-H contacts in a single step.

On the theoretical ground, further research is necessary to find a method for predicting the maximum number of contacts in a given HP configuration by using only de amino acids sequence. Such upper-bound would allow the system to stop when the first occurrence of the best folding was found. Since such upper-bound value is not known, a full sweep of the search space is required.

Overall, the use of reconfigurable computing for the folding problem using the 2D-HP model is very promising and, for short sequences, it allows obtaining the optimal folding in reasonable processing time.

References

1. Berger, B., Leight, T.: Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete, *J. Comput. Biol.* **5** (1998) 27–40.
2. Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A., Yannakakis, M.: On the complexity of protein folding, *J. Comput. Biol.* **5** (1998) 423–466.
3. Dill, K.A., Bromberg, S., Yue, K., Fiebig, K.M., Yee, D.P., Thomas, P.D., Chan, H.S.: Principles of protein folding - a perspective from simple exact models, *Protein Sci.* **4** (1995) 561–602.
4. Dill, K.A.: Theory for the folding and stability of globular proteins, *Biochemistry* **24** (1985) 1501–1509.
5. Jacobi, R.P., Ayala-Rincón, M., Carvalho, L.G., Quintero, C.H.L., Hartenstein, R.W.: Reconfigurable systems for sequence alignment and for general dynamic programming. *Genetics and Molecular Research* **4** (2005) 543–552.
6. Lehninger, A.L., Nelson, D.L., Cox, M.M.: *Principles of Biochemistry*, 2nd ed. Worth Publishers, New York (1998).
7. Lopes, H.S., Scapin, M.P.: An enhanced genetic algorithm for protein structure prediction using the 2D hydrophobic-polar model. in *Proc. Artificial Evolution, LNCS* **3871** (2005) 238–246.
8. Marongiu, A., Palazzari, P., Rosato, V.: Designing hardware for protein sequence analysis. *Bioinformatics* **19** (2003) 1739–1740.
9. Moritz, G.L., Jory, C., Lopes, H.S., Erig Lima, C.R.: Implementation of a parallel algorithm for pairwise alignment using reconfigurable computing. *Proc. IEEE Int. Conf. on Reconfigurable Computing and FPGAs*, (2006) pp. 99-105.
10. Oliver, T., Schmidt, B., Nathan, D., Clemens, R., Maskell, D.: Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics* **21** (2005) 3431–3432.
11. Yamaguchi, Y., Maruyama, T., Konagaya, A.: High speed homology search with FPGAs. in *Proc. Pacific Symposium on Biocomputing* (2002) 271–282.

Reconfigurable Parallel Architecture for Genetic Algorithms: Application to the Synthesis of Digital Circuits

Edson P. Ferlin[1,2], Heitor S. Lopes[2], Carlos R. Erig Lima[2], and Ederson Cichaczewski[1]

[1] Computer Engineering Department, Positivo University Center (UnicenP), R. Pedro V. P. Souza, 5300, 81280-230 Curitiba (PR), Brazil
ferlin@unicenp.edu.br
[2] Bioinformatics Laboratory, Federal University of Technology Paraná (UTFPR), Av. 7 de setembro, 3165, 80230-901 Curitiba (PR), Brazil
hslopes@pesquisador.cnpq.br,erig@utfpr.edu.br

Abstract. This work presents a proposal and implementation of a reconfigurable parallel architecture, using Genetic Algorithms and applied to synthesis of combinational digital circuits. This reconfigurable parallel architecture uses concepts of computer architecture and parallel processing to obtain a scalable performance. It is developed in VHDL and implemented totally in hardware using FPGA devices. The concept of reconfigurable and parallel architecture enables an easy hardware adaptation to different project requirements. This approach allows applies with flexibility different strategies to synthesis of combinational digital circuits problem.

1 Introduction

Recently, we have witnessed a pronounced growth the hardware and software technologies for embedded systems, with many technological options coming up every year. The use of reconfigurable structures is becoming attractive, especially due to its robustness and flexibility. The possibility of massive parallel processing makes reconfigurable computing a suitable technology to be applied to the growing computational demand of scientific computation.

Reconfigurable architectures or reconfigurable computational systems architecture are those where the logic blocks can be reconfigured, in their logical functions and internal functionality. The interconnection between these logic blocks, that usually performs computational tasks such as processing, storing, communication or data in/out, can be reconfigured too [6].

Because these logic blocks are implemented directly in hardware, it is possible to run algorithms exploiting the inherent parallelism of a hardware solution, achieving a throughput, much higher than if they were run in a sequential processor, subjected to the Von Neumann model [8].

For several complex problems, the solution using traditional software approach, where the hardware executes sequential algorithms, does not satisfy the timing and performance requirements. This justifies the search for new approaches to minimize these bottlenecks. For example, reconfigurable logic allows a dramatic minimization of the processing time, when compared with a traditional software approach. Such performance is possible thanks to the parallel processing and reduced computation time, inherent to a FPGA (Field-Programmable Gate Array) implementation [9].

The motivation for using parallel processing is increasing the computational power of a limited processor. This is accomplished by exploiting the simultaneous events in a software execution [3].

The Genetic Algorithm (GA) was proposed in the 60's by Holland [7], with the initial aim of studying the phenomenon related to species adaptation and natural selection that occurs in the nature. Later, GAs becomes a powerful computational method to solve optimization and machine learning problems in the areas of engineering and computer science [4]. GAs are intrinsically parallel algorithms and particularly attractive for parallel implementations [1].

This work proposes the implementation of a reconfigurable parallel architecture, applied to the synthesis of a combinational digital circuit by using a Genetic Algorithm. This system uses concepts of parallel processing and reconfigurable computing to achieve a scalable performance. It is developed in VHDL (VHSIC Hardware Description Language) [13] and fully implemented in hardware using a FPGA device.

2 Problem Description

The problem treated in this work consists in obtaining a minimal Boolean equation for a combinational digital circuit that implements a determined function specified by a truth table.

In the project of logic circuits we can use many criteria to define the minimal cost expression. The complexity of a logic circuit is a function of the number and complexity of circuit gates. The complexity of a gate is, in general, a function of the number of gate inputs. Since logic circuits will implement a Boolean function in hardware, reducing the number of function literals, will reduce the number of inputs for each gate and the number of gates in the circuit. Consequently, the overall complexity of the circuit will be reduced.

The implementation described here can be used for circuits with up to four input bits and one output, such as the one shown in the example presented in table 1. In the case of problems that have two or more outputs, each output is treated independently.

Table 1. Example of truth table: A, B, C and D are inputs and S is the output

A	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
B	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
C	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
D	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
S	1 0 0 0 1 1 1 1 1 1 1 0 0 1 0 1

This work presents a multiobjective GA for the synthesis of combinational digital circuits. The basic idea is to evaluate fitness in two stages. Firstly, to obtain a circuit that matches the truth table, that is, an equivalent circuit. When this objective is achieved other fitness function is used. This second function aims at minimizing the amount of logic gates in the circuit.

3 A Reconfigurable Parallel Architecture for GAs

One of the first parallel architectures for GAs was SPGA (Splash 2 Parallel Genetic Algorithm) [5]. This architecture was applied to the traveling salesman problem and composed by 16 genetic processors that, in turn, included the following modules: selection, crossover, fitness & mutation and statistics. Such modules were grouped using an insular parallel model. The main difference of SPGA architecture with the one proposed in this work is that the fitness function is replicated to be processed in parallel, keeping a single population.

Other architecture FPGA-based is described for Tang and Yip [15]. This architecture is a general GA using FPGAs with the ability to reconfigure the hardware-based fitness function. In this project, various configurations of parallelization are available with a PCI board based design.

The proposed architecture uses the computational model known as Cartesian Genetic Programming (CGP) [11]. This model, shown in figure 1, is based on a geometric set of $n \times m$ Logic Cells (LC), with n inputs and m outputs. This model was used because it is suited to the hardware parallelism given by the simultaneous configuration of the LCs by the chromosome, and the way each logical function is mapped in a cell. The number of inputs (n) depends on the problem, while the number of outputs (m) depends exclusively on the number of LCs lines of the geometric set (see figure 1).

The logical functions and the inputs for such logical functions can be selected individually in each LC. The connectivity of the LCs is arbitrary, since they can use either the regular inputs of the system or the outputs of other LCs.

This same model was used in other projects such as [2], [14], [17], thus suggesting its usefulness to this work.

This architecture is directed to the execution in the Farm model of parallelism, also known as Global Parallel GA [16]. In this model, the Master-Slave, in which the Master (Control Unit - CU) is responsible for the generation of the initial population, and the selection, crossover and mutation of chromosomes, while other Slaves (here called Processing Elements - PEs) calculate the fitness of each chromosome. This model exploits the parallelism inherent to the evaluation of fitness of individuals in the population. Each chromosome of population is treated individually and, therefore, many chromosomes can be processed simultaneously, reducing the processing time.

The proposed architecture uses a single Control Unit, and several PEs. Figure 2 presents a block diagram showing the CU and several PEs implemented

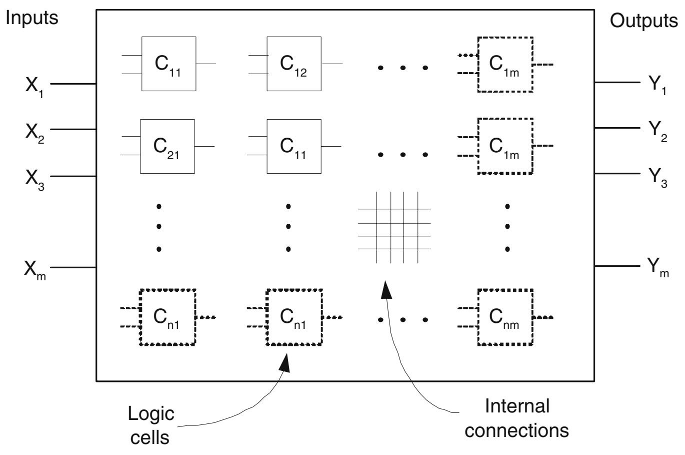


Fig. 1. Geometric mapping of LCs in the CGP model

using reconfigurable logic (FPGA). The host is responsible for sending to the parallel machine the configuration parameters, such as probability of mutation, crossover, number of generations, and the truth table. After this configuration, the CU sends the chromosomes to PEs. The PEs evaluates the chromosomes and computes the fitness. This result is send back to the CU.

The main contribution of this architecture is that it can be applied to several different problems, providing the architecture (in special, the PEs) is reconfigured for a particular problem. Besides being reconfigurable, the architecture is parallel, thus exploring an important feature of implementations in FPGAs.

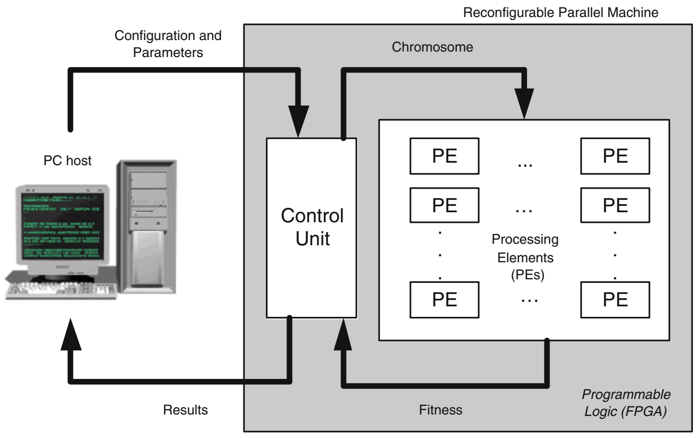

Reconfigurable Parallel Machine
Configuration and Parameters
Chromosome
PC host
Control Unit
PE
Processing Elements (PEs)
Results
Fitness
Programmable Logic (FPGA)

Fig. 2. General vision of the reconfigurable parallel architecture for GA

4 Implementation

The chromosome was encoded with 25 genes (one gene for each LC) and each gene has 7 bits. Therefore, the chromosome was 175 bits-long. Each gene is responsible for the configuration of a LC, and it is composed by three fields: address A, address B and the function. Four bits of the gene are used for the selection of the LC inputs, and 3 bits are used for selecting the function (8 possible functions).

The fitness function for this problem is multiobjective. Initially, it is evaluated how many lines of truth table matches ("matching"). Next, the amount of null LCs is counted ("nulls"). The fitness value is composed by the concatenation of three information: matching (5 bits), nulls (5 bits) and output (3 bits) - number of outputs of the LCs matrix that produced these results.

The proposed parallel architecture is shown in figure 3. It receives from the host four parameters: P_c (crossover probability), P_m (mutation probability), number of generations and reference truth table. Crossover and mutation probabilities are encoded with 10 bits, representing values from 0 to 0.999. Cyclically, the system sends back to the host the best chromosome and respective fitness at each generation.

The initial population is randomly generated with 100 individuals and stored in chromosome memory. After, the individuals are sent to the PEs for computing the fitness. When individuals are under evaluation, they are kept in the scratch memory. At the same time, the best individual is updated to be sent back to the host at the end of the generation. When all individuals are processed, the CU performs the selection procedure and applies crossover and mutation operators (according to P_c and P_m). When this step is concluded, the new individuals of the next generation are stored again in the chromosome memory, and the whole process is repeated until reaching the predefined maximum number of generations.

4.1 Control Unit

The Control Unit is responsible for managing input and output data as well as the other operations previously mentioned. The width of the parallel buses connecting the CU to the PEs is proportional to the number of PEs. This makes possible the simultaneous communication between the memories (in the CU) and the PEs, eliminating a possible communication bottleneck. The CU is composed by the following components:

Random Number Generator: generates random numbers that will be used for many functions in the architecture, for instance, generation of the initial population. In this case, each individual (chromosome) is obtained by concatenation of many binary sequences (numbers). The random number generator is very important to the performance of a GA system. In this case, it was implemented in VHDL based on the Mersenne-Twister algorithm described by [10]. The implemented generator uses 16-bits numbers, allowing

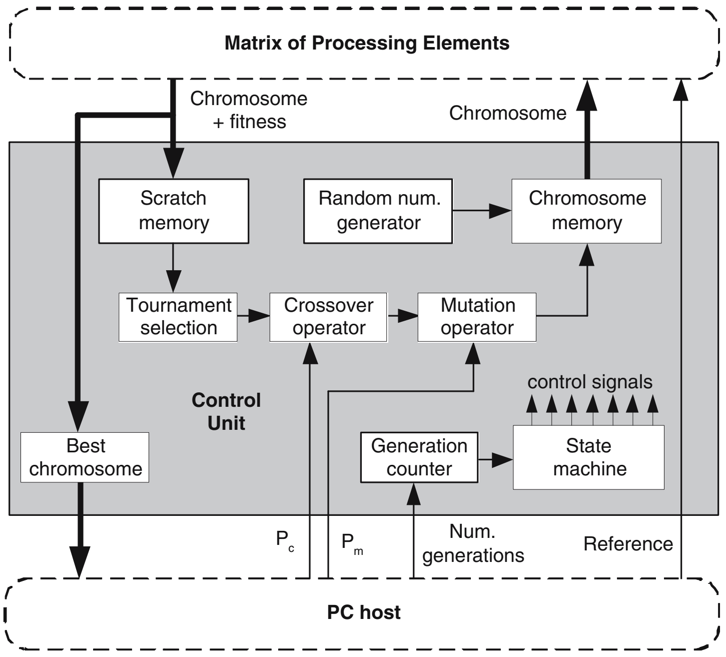


Fig. 3. Block diagram of Control Unit (CU)

the economy of logic elements. This representation results in a repetition cycle similar to the conventional ”rand” function of C language programming.

Chromosome Memory: stores the individuals (chromosomes) to be processed. It is composed by 100 words of 175 bits. It is a double-access memory controlled by the state machine. In a first step, this memory is used for storing the individuals of the initial population. Later, it stores the individuals of the forthcoming generations.

Scratch Memory: stores the already processed individuals and their fitness value. It is capable of storing up to 100 individuals, each one composed by the chromosome (175 bits) and the corresponding fitness (13 bits). This is also a double-access memory controlled by the state machine. Selection, crossover and mutation blocks use the information stored in this memory.

Best Chromosome: finds the best individual (chromosome) of a generation, based in the fitness value. This individual is sent out to the host at the end of the generation. The best individual is obtained by comparison. For each new individual processed, this block compares its fitness with the best fitness stored.

Tournament Selection: implements the well-known stochastic tournament selection using two randomly chosen individuals of the scratch memory. The individual with highest fitness of the two is selected to be submitted to the genetic operators.

Crossover Operator: executes a classical one-point crossover operation with probability P_c. The crossover point is static and predefined to be after the

87th bit. Bits 1 to 87 of the first chromosome are concatenated with bits 88 to 175 of the second chromosome to form the first offspring. The second offspring is formed in a similar way using the complimentary parts of the original chromosomes.

Mutation Operator: executes a point-mutation operation: a randomly chosen bit in the chromosome is complemented, with probability P_m.

Generation Counter: counts the number of generations. It is responsible for controlling the number of generations. It signals to other blocks of the architecture the end of processing when the maximum number of generations is reached. This counter has 10 bits and thus allows the GA to run for up to 1023 generations.

State Machine: generates all the activation signals to the other components of the architecture. The whole operational control of the system is accomplished with 20 states.

4.2 Processing Element

The internal complexity of the PEs depends on the specific application. In our case, each PE is composed by a 5 × 5 LCs matrix (25 LCs), the truth table and a counter to produce the input binary sequences (0 to Fh), besides the circuit for computing the fitness (see figure 4). These elements are detailed below.

It is important to notice that the LCs matrix is dynamically reconfigured at running time. That is, for each new chromosome to be evaluated, the matrix is reconfigured, both the specific function of the LCs and the connectivity between them.

Therefore, each PE has to be as simple as possible to allow the implementation of a large number of PEs running in parallel in a FPGA device.

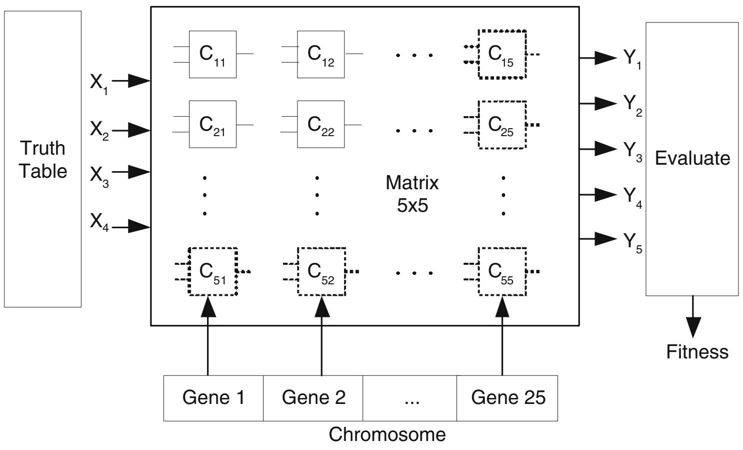


Fig. 4. Block diagram of Processing Element (PE)

The Processing Element is composed by following components:

Logic Cell: performs the Boolean operations shown in table 2.

Truth Table: a 4-bits counter generates all possible input combinations (0000 to 1111) to the truth table. Based on these values, the LCs matrix (after being configured) generates five simultaneous outputs, one for each logic expression, later used to compose the fitness value of the individual.

Evaluate: first, this block evaluates how many lines of the truth table was satisfied (matching). After, it counts the number of "null" LCs. Finally, it finds the number of outputs of the LCs matrix that produced these results. All these information compose the fitness value.

Table 2. Logic functions of Logic Cells (LCs)

#	Operation	Function
0	AND	$x.y$
1	OR	$x + y$
2	XOR	$x \otimes y$
3	NOT(x)	$\bar{x}$
4	NOT(y)	$\bar{y}$
5	Wire x	x
6	Wire y	y
7	Null	null

5 Results

The hardware platform used in the development is Altera^{TM} EP1S10F484C5 Stratix device (*http://www.altera.com*). For compiling the circuit described in VHDL was used the Quartus II Altera^{TM} design toll, version 5.0. The demands of logic elements and internal memory up as function of the number of PEs. One PE demands 5,174 logic elements and 36,300 bits of memory, occupying 48% and 3% of available resources, respectively. Table 3 shows the main components of the implemented architecture with one PE.

The architecture was run at 50MHz clock. The first generation processing time is approximately 465μs, with the generation of initial population spending 53μs to be processed. Thus, considering that the mean time of each generation is approximately 412μs, this architecture can process around 2,426 generations (with 100 individuals) in one second. The fitness processing time for a chromosome in the PE is approximately 3.84μs

The evaluation of the chromosome demands 384μs. This time corresponds to 93% of the processing time. Thus, just 28μs (7% of the time). This step is executed sequentially and divided as follow: 2% for Selection, 1% for Crossover and 4% for Mutation. Therefore, the initial assumption used to justify the adopted model (Master-Slave), where the fitness evaluation of the chromosome is done by PE is correct. Thus, the parallelization of this stage, with the utilization of

Table 3. Main components of the GA architecture and the respective cost of Logic Elements (LE)

Component	Number of LE	%
CU - RNG	631	12
CU - Selection	567	11
CU - Crossover	626	12
CU - Mutation	864	17
CU - Other	1,861	36
PE - Fitness	625	12
Total	5,174	100

several parallel PEs, can reduce the total processing time for obtain a generation, what is the primary objective of this architecture.

An experiment with two PEs working in parallel reveals that, for the processing of a generation with 100 chromosomes, a time of 223μs was demanded, achieving a time reduction of approximately 54%. This two PEs implementation, using the same device, demands 6,966 logic elements and 36,300 bits of memory, occupying 65% and 3% of available resources, respectively.

Other important issue is the Speedup [12], that is the ratio between the execution time with one PE and the parallel execution time with two or more PEs. It is fundamental to analyze the performance of parallel machines. The tests done using this architecture with two PEs achieve around of 85% of Speedup, close to ideal value of 100%.

6 Conclusions and Future Work

The main contribution of this work is the proposal and implementation of a reconfigurable parallel architecture, using Genetic Algorithms and applied to the synthesis (Boolean minimization) of combinational digital circuits.

With the processing in parallel way is to be possible a significant execution time reduction.

In this implementation is observed that approximately 89% of a generation processing time is spent with the chromosome fitness processing. This justifies the PE parallelization approach, where several PEs process the individual fitness in parallel way, regarding to CU other sequential operations processing, such as the selection function operation, the crossover operation and the mutation operation.

In the experiments using two PEs, we observe a performance improvement of 54%, or a speedup of 1.85, reaching 92% of the ideal value. However, we will only be able to affirm the evolution of performance profit when the results will be more consistent.

With the addition of one PE in the architecture, we observe a demand of 17% more logic elements. These additional device logic elements are used to implement the additional PE, the interconnection of many components for the parallel buses, and mainly, the memories division in two separated modules. This division avoids a possible bottleneck in input/output data transfer in the architecture with several PEs.

This architecture explores many levels of parallelism. The first level can be observed in several PEs operating in parallel way. The second level can be observed in geometric set of 5 × 5 LCs internally of each PE. Since each gene configures just one individual LC, the complete configuration is made in parallel. By other hand, since five logic expressions are simultaneously generated to each input element, a parallel processing is also observed. The last level can be observed in the operation proceeding of the "best chromosome" unit. This operation determines the best chromosome, while the others units still processing its operations simultaneously. Finally, the VHDL programming generates a digital circuit implementation that enables a parallel execution of an algorithm. The similar software solution cannot be executed in same way.

Although several improvements can be made in the implemented architecture, the partial obtained results demonstrate the correction of the approach adopted.

Some aspects deserve special attention for the future improvements:

- Increase the number of PEs, to get greater performance;
- Adapt the parallel architecture to be possible working with more complex problems;
- Optimize the circuits to consume less resources of FPGA device.

Finally, this work demonstrates that several positive factors like the processing power of the Parallel Computation, the versatility of Reconfigurable Computation and the robustness of Genetic Algorithm, can be used to created powerful tools to solution of many complex problems. Especially in Engineering and Science problems where the processing time has critical restriction.

Acknowledgment

This work was partially supported by the Brazilian National Research Council – CNPq, under research grant no. 305720/04-0 to H.S. Lopes.

References

1. Cantú-Paz, E.: *Efficient and Accurate Parallel Genetic Algorithms*, vol. 1, Kluwer Academic, Norwel (2001).
2. Coello-Coello, C.A., Aguirre, A.H.: On the use of a population-based particle swarm optimizer to design combinational logic circuits. In: *Proc. NASA/DoD Conference on Evolvable Hardware* (2004) 183–190.
3. Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., White, A.: *Sourcebook of parallel computing*. Morgan Kaufmann, San Francisco (2003).
4. Goldberg, D.: *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, Reading (1989).

5. Graham, P., Nelson, B.: A Hardware genetic algorithm for the traveling salesman problem on Splash 2. In: *Field-Programmable Logic and Applications*, Springer-Verlag, Berlin (1995) 352–361.
6. Hartenstein, R.: A Decade of reconfigurable computing: a visionary retrospective. In: *Proc. IEEE Conf. on Design, Automation and Test in Europe* (2001) 642–649.
7. Holland, J.H.: *Adaptation in Natural and Artificial Systems*. University of Michigan Press, East Lansing (1975).
8. Lysaght, P., Rosenstiel, W.: *New algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, New York (2005).
9. Gokhale, M., Graham, P.S.: *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, Berlin (2005).
10. M. Matsumoto, Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations* (1998) 3–30.
11. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: *Proc. 3rd European Conference on Genetic Programming, LNCS* **1802** (2000) 121–132.
12. Murdocca, M.J., Heuring, V.P.: *Principles of Computer Architecture*. Prentice Hall, New Jersey (2001).
13. Pedroni, V.A.: *Circuit Design with VHDL*. MIT Press, Cambridge (2004).
14. Sekanina, L., Ruzicka, R.: Easily testable image operators: the class of circuits where evolution beats engineers. In: *Proc. NASA/DoD Conference on Evolvable Hardware* (2003) 135–144.
15. Tang, W., Yip, L.: Hardware implementation of genetic algoritms using FPGA. In: *Proc. 47th Midwest Symposium on Circuits and Systems, vol 1* (2004) 549–552.
16. Yue, K.K., Lilja, D.J.: Designing multiprocessor scheduling algorithms using a distributed genetic algorithm system. *Evolutionary Algorithms in Engineering Applications* **33** (1997) 39–40.
17. Zhang, Y., Smith, S.L., Tyrrell, A.M.: Digital circuit design using intrinsic evolvable hardware. In: *Proc. NASA/DoD Conference on Evolvable Hardware* (2004) 55–62.

A Space Variant Mapping Architecture for Reliable Car Segmentation

S. Mota[1], E. Ros[2], J. Díaz[2], R. Rodriguez[2], and R. Carrillo[2]

[1] Department of Computer Sciences and Numerical Analysis, University of Córdoba, Campus de Rabanales s/n, Edificio Albert Einstein, 14071, Córdoba, Spain
[2] Department of Computer Architecture and Technology, University of Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain
smota@uco.es, {eros, jdiaz, rrodriguez, rcarrillo}@atc.ugr.es

Abstract. Side-impact crashes have now become more important than head-on crashes, probably reflecting improvements in protecting occupants. Overtaking scenarios are one of the most dangerous situations in driving. This paper is concerned with a vision-based system on the rear-view mirror for safety in overtaking scenarios. A bio-inspired algorithm segments overtaking vehicles using motion information and rigid-body-motion criterion. The overtaking scene in the rear-view mirror is distorted due to perspective. Therefore we need to devise a way of reducing the distortion effect in order to enhance the segmentation capabilities of a vision system based on motion detection. We adopt a space variant mapping strategy. In this paper we describe a computing architecture that finely pipelines all the processing stages to achieve reliable high frame-rate processing.

1 Introduction

Although the performance of biological visual systems is multimodal, in many applications, visual motion detection is the most important information source. Biological systems have inspired artificial systems the way they extract the motion information. However, even biological systems need to project 3D scene onto a 2D surface before extracting data. Due to the 2D projection the scene is distorted by the perspective, and a moving object, although moving at a constant speed, seems to accelerate and its size increases as it approaches the camera, which adds an expanding motion to the translational one.

Biological systems use low level stereo information or other visual modalities in higher level processing stages to deal with the perspective distortion. But uni-modal motion-based artificial systems need other ways to compensate this effect.

In this work, we used motion detection information to segment moving objects. The segmentation strategy follows a bio-inspired bottom-up structure [1]:

- **Edges extraction:** the edges capture most of the scene structure.
- **Correlation process:** Reichardt has described a motion detection model that emulates the behaviour of early visual stages in insects [2].
- **Rigid body motion detection:** in order to effectively segment independent moving objects we use a clustering method based on the "*rigid body criterion*". Local motion information is integrated into global, that is, a population of pixels in a

window of the image which share the same velocity -rigid body- are likely to represent the independent moving object (the overtaking car) [3, 4].

The presented scheme is based on a sparse map of pixels (less than 15% of the pixels in each frame have information different from zero).

We apply this approach to overtaking scenarios, which is one of the most dangerous situations in driving because the rear-view mirror is sometimes not watched by the driver or is momentarily useless because of the blind spot. We have used some sequences taken with a camera fixed onto the driver's rear-view mirror. This view is complex, not only because everything is moving in scene (due to the ego-motion of the camera), besides the scene is distorted by the perspective (the scene converges to a vanishing point at the left-hand side of the image), producing the effect of non-uniformity in the speed of moving objects and the apparently enlargement of the overtaking car. In addition, the front and rear of the overtaking car seem to move at slightly different speeds (the backside of the vehicle is far and it seems to move slowly, while the front of the vehicle is closer and it seems to move faster), so it is difficult to apply a rigid body motion criterion.

The motion-based segmentation algorithm is implemented on a FPGA device running in real-time [5]. A camera placed onto the rear-view mirror sends the image of the overtaking scenario to the FPGA that processes the image and finally segments the overtaking vehicle. The visual segmentation architecture is based on independent processing units (modular architecture), each of them working in parallel at one pixel per clock cycle.

In this contribution we correct the effects of the perspective distortion to reliably segment the overtaking vehicle. It is possible to compensate for the effect of perspective by remapping the image. This new processing units could be connected to the whole system to pre- process the sequences.

2 Space Variant Mapping Method

We introduce here the space-variant-mapping (SVM) method as strategy for dealing with the perspective distortion. SVM approach resamples the original image. The Space Variant Mapping [6, 7] is an affine coordinate transformation that aims at reversing the process of projection of a 3-D scene onto a 2-D surface. It is possible to compensate the effect of perspective by remapping the original image. In the special case where the world plane is orthogonal to the camera's axis, equal distances in the real world are projected onto equal distance in the image. In the general case, distances closer to the image plane are projected onto larger segments. So the parallel lines and equal distances in a real scene are remapped to parallel lines and equal distances in the processed image. Practically, in our application (see Fig. 1), this means expanding the left-hand side of the image (corresponding to the part of the scene furthest away from the camera) and collapsing the right-hand side (corresponding to the part of the scene closest to the camera). The interpolation method used here is the truncated Taylor expansion, known as *local Jet* [7]. At the remapped output the mean speed of the car is more constant along the sequence and each point that belongs to the rigid body moves approximately at the same speed (Fig. 2).

The advantage of SVM is that the effect of perspective is compensated through the remapping scheme and the acceleration problem is solved, furthermore we synchronize the front and rear of a overtaking car speed. Finally, the remapped image is easier to segment and interpret.

Figure 1 shows the original and the remapped image from an overtaking sequence and Figure 2 shows the compensation effect on the speed of the centre of the overtaking car.

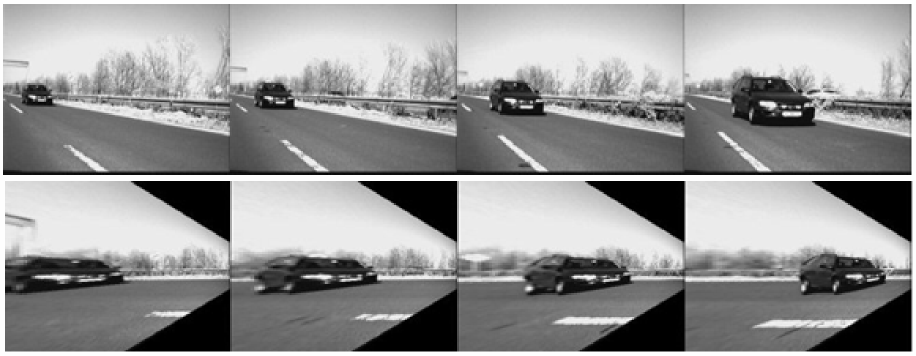


Fig. 1. Original and remapped image of an overtaking sequence

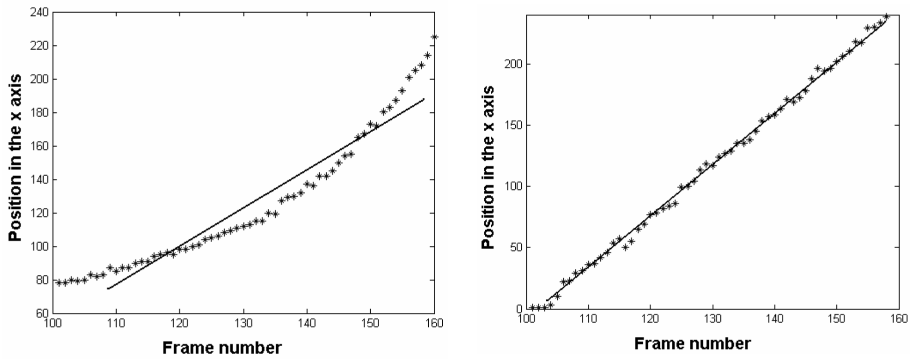


Fig. 2. Space Variant Mapping makes stable the speed along the sequence. On the left plot we represent the x position of the centre of the car along the overtaking sequence. We see that although the overtaking sequence speed is constant the curve is deformed (constant speed is represented by a line) due to the perspective. On the other hand, the right plot shows the same result on a remapped sequence. In this case the obtained overtaking speed is constant (well approximated by a line with a slope that encodes the speed).

3 Hardware Implementation

We use a High Dynamic Range (HDR) camera that provides 30 frames per second of 640 x 480 pixels and 256 gray levels and a prototyping platform with a Xilinx Virtex-II FPGA (XC2V3000 device) [8] and 2 SRAM banks. This device allocates 3 million system gates, distributed in 28,672 logic cells, and 96 embedded memory blocks of a total of 1,728 Kbits.

The motion extraction system follows a fine grain pipeline structure that consumes 1 cycle per stage, therefore using intensively the full potential parallelism available at FPGA devices. The limitation of the system is in the external memory banks access. In spite of this, the system is able to process images of 640x480 pixels at the speed of 250 frames per second, when the global clock is 31.5 MHz [5].

SVM architecture must be implemented as a fine grain pipeline structure to ensure a successful connection to the motion extraction module at 1 pixel per clock cycle. SVM uses several multiscalar units.

SVM needs to do different operations such as sums, multiplications, sine, tangent and divisions.

To compute sine and tangent we use lookup tables. To compute the division we use an optimized core that computes one division and consumes one cycle. Due to this, the whole system frequency is limited by the remapping stage.

Note that the maximum clock frequency advised by the ISE environment is limited to 23.4 MHz, due to the SVM stage. This is because we use a specific core for the division that limits the global frequency of the whole pipeline structure. Nevertheless the system has been tested at 31.5 MHz obtaining fully accurate results. The estimation given by the ISE environment is very conservative, this arises because the analyzer looks at the static logic path rather than the dynamic one, and therefore this performance improvement is correct [9].

Table 1. Hardware cost of the different stages of the described system. The global clock of the design is running at 31.5 MHz, although the tables include the maximum frequency allowed by each stage. The data of the table has been extracted using the ISE environment.

Pipeline stage	Logic Cells	% occupation	% on-chip memory	Max.Fclk (MHz)
Frame-Grabber	1,808	17	3	75.9
Space Variant Mapping	7,644	18	0	23.4
Motion detection	8,168	79	5	41.4
Total system	24,672	88	22	23.4

The described stage is processed in a fine pipeline computing architecture in order to optimize the performance. SVM module takes 12 pipeline stages (Fig. 3).

The outstanding processing speed allows the use of more advanced cameras with higher spatial and temporal resolution. Since our frame-grabber is embedded on the FPGA and can be easily customized, we can process images of 1280x960 at 60 frames per second. High rate cameras significantly reduce the temporal aliasing of high speed motions. This is of critical importance in the last stage of the overtaking sequence. For instance, a car overtaking at a relative overtaking speed of 20 m/s produces displacements of 36 pixels from frame to frame using 30 fps with 640x480 of image resolution. This makes difficult the reliable local motion detection, and this is highly reduced with high frame rate cameras.

Table 1 summarizes the main performance and hardware cost of the different parts of the system implemented. The hardware costs in the table are estimations extracted from sub-designs compiled with the ISE environment.

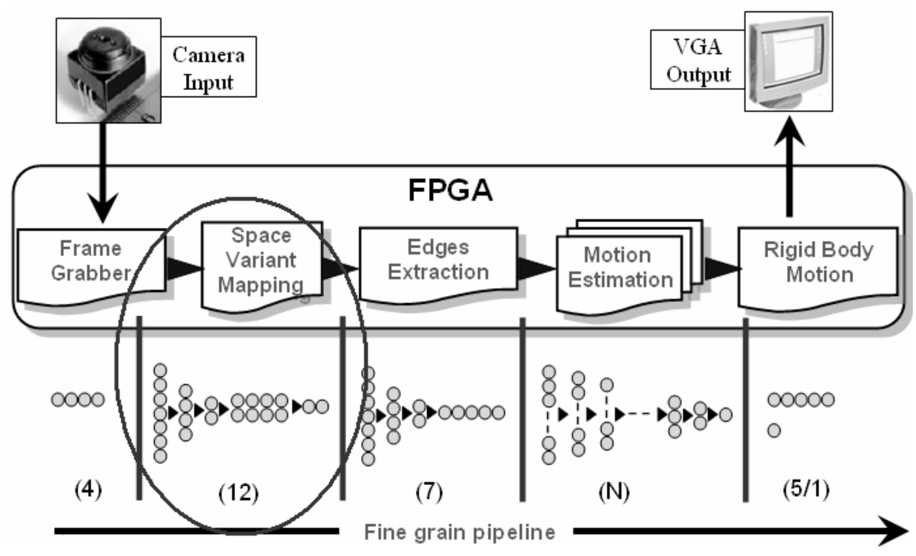


Fig. 3. Data flow of the complete system. Numbers under parenthesis represents the pipeline stages that take each module. Circles represent multiscalar units. The circles on a column are working in parallel. Circles on a row represent different pipeline stages.

4 Conclusions

We have presented a perspective distortion correction for a vision-based car segmentation system. The whole system is composed by different stages: perspective correction, motion detection, robust filtering and segmentation.

Using a real-life sequence of a car moving at constant speed we showed that the SVM considerably reduces the spurious acceleration effect due to perspective projection.

We have designed a pipeline computing architecture that takes full advantage of the inherent parallelism of FPGA technology. In this way we achieve computing speeds of 31,5 mega pixels (for instance 60 frames per second with 1280x960 image resolution). This computational power enables the use of high rate cameras that reduce the temporal aliasing present when computing local motion to extract reliable features.

This contribution is a good case of study that illustrates how very diverse processing stages can be finely pipelined in order to achieve high performance. Finally the hardware resources of the system are not very high. Therefore, the presented approach can be considered a moderate cost module to approve the real world application of overtaking car monitor.

Acknowledgment. This work has been supported by the European Project DRIVSCO (IST-2001-35271) and the National Project DEPROVI (DPI2004-07032).

References

1. Mota, S., Ros, E., Ortigosa, E.M., Pelayo, F. J.: Bio-Inspired motion detection for blind spot overtaking monitor. International Journal of Robotics and Automation, 19 (4) (2004)
2. Reichardt, W.: Autocorrelation, a principle for the evaluation of sensory information by the central nervous system. In: W. A. Rosenblith (ed.): Sensory Communication. MIT Press, Cambridge, MA, (1961) 303-317
3. Díaz, J., Mota, S., Ros, E., Botella, G. Neural Competitive Structures for Segmentation based on Motion Features. Lecture Notes in Computer Science, 2686 (2003) 710-717.
4. Mota, S., Ros, E., Díaz, J., Ortigosa, E.M and Prieto, A. Motion-driven segmentation by competitive neural processing. Neural Processing Letters, 22(2) (2005): 125-177.
5. Mota S., Ros E., Díaz J., de Toro F. General purpose real-time image segmentation system. Lecture Notes in Computer Science 3985 (2006) 164-169
6. Mallot, H. A., Bulthoff, H. H., Little, J. J., Bohrer, S. Inverse perspective mapping simplifies optical flow computation and obstacle detection. Biol. Cybern., 64 (1991) 177-185.
7. Tan, S., Dale, J., Johnston, A, Effects of Inverse Perspective Mapping on Optic Flow. ECOVISION Workshop 2004 (Isle of Skye, Scotland, UK) (2004)
8. http://www.xilinx.com/
9. Celoxica application note AN 68 v1.1, "Timing Analysis. Timing Analysis and Optimisation of Handel-C Designs for Xilinx Chips".

A Hardware SAT Solver Using Non-chronological Backtracking and Clause Recording Without Overheads

Shinya Hiramoto, Masaki Nakanishi, Shigeru Yamashita, and Yasuhiko Nakashima

Nara Institute of Science and Technology
{shinya-h m-naka, ger, nakashim}@is.naist.jp

Abstract. Boolean satisability (SAT), known as an NP-complete problem, has many important applications. The current generation of software SAT solvers implement non-chronological backtracking and clause recording to improve their performance. However, most hardware SAT solvers do not implement these methods since the methods require a complex process to work. To hasten the solving process compared with a contemporary software SAT solver, hardware SAT solvers need to implement these methods. In this paper, we show a method for implementing these methods without a complex process and design a hardware SAT solver using this method. The experimental results indicate that it is possible to estimate a 24-80x speed increase compared with a contemporary software SAT solver in EDA problems based on a reasonable assumption.

1 Introduction

Boolean satisability (SAT) is to decide whether there exists an assignment of truth values to variables such that a given Boolean formula F becomes satisfied. SAT is an NP-complete problem with many applications in a variety of fields.

The current generation of software SAT solvers implement *non-chronological backtracking* and *clause recording* to improve their performance[1]. Recently, these methods have played a key role in software SAT solvers. The *implication* process accounts for more than 90% of software solvers' running time, but hardware SAT solvers can speed up the solving process by parallelizing the implication process[2]. However, most hardware SAT solvers do not implement non-chronological backtracking or clause recording since these methods comprise complex processes. A software SAT solver that implements these methods achieves a two-orders-of-magnitude increase in speed compared with a software SAT solver that does not implement these methods[1]. Therefore, it is difficult for hardware SAT solvers to speed up the solving process compared with a modern software SAT solver only by parallelizing.

In this paper, we propose a way to implement non-chronological backtracking and clause recording implicitly in a hardware SAT solver by partitioning a given SAT formula. Our idea can improve one of the weaknesses of hardware

SAT solvers mentioned above. We demonstrate the effectiveness of our approach through preliminary experiments.

2 Previous Work

2.1 DPLL Algorithm

Most software SAT solvers and hardware SAT solvers are based on the DPLL algorithm[5]. This algorithm consists of the following iterative processes.

1. Decision: We choose an unassigned variable and assign a value (0 or 1) to it.
2. Implication: We find necessary variable assignments such that the given SAT formula is satisfied based on the current variable assignments. Conflict occurs when it is necessary to assign 0 and 1 to one variable in the above process.
3. Backtrack: if a conflict is detected, we return to a decision of the previous variable assignment, and try another assignment.

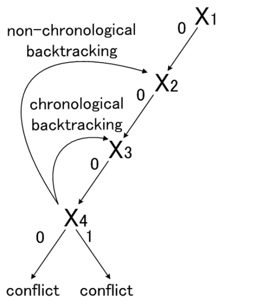
2.2 Non-chronological Backtracking and Clause Recording

Modern software SAT solvers use non-chronological backtracking and clause recording to speed up the solving process[1]. Suppose that the partial formula is $(x_1 \vee x_2 \vee x_4 \vee x_5)(x_1 \vee x_2 \vee x_4 \vee \bar{x}_5)(x_1 \vee x_4 \vee \bar{x}_4 \vee x_5)(x_1 \vee x_2 \vee \bar{x}_4 \vee \bar{x}_5)$. Throughout this paper, we use the above SAT formula, which we call **Example 1**. Also, suppose that we previously assigned 0 to x_1, x_2 and x_3. If we assign 0 to x_4, a conflict occurs on x_5. After changing the assignment of x_4 to 1, another conflict occurs on x_5 again. In *chronological backtracking*, a SAT solver backtracks to the most recently assigned variable, x_3; however, x_3 is not related to these conflicts since it does not appear in the clauses causing these conflicts. Therefore, although a SAT solver backtracks to x_3, these conflicts occur again when we assign a value to x_4. In *non-chronological backtracking*, a SAT solver finds variables that cause conflicts in a process called *conflict analysis*. In this example, a SAT solver finds that x_1 and x_2 cause these conflicts, Thus, a SAT solver can directly backtrack to the variable causing these conflicts, i.e., x_2, which is shown in Fig. 1.

By the above conflict analysis, a SAT solver also finds that if x_1 and x_2 are assigned to 0, the formula cannot be satisfied. Therefore, a SAT solver can add a new clause $(x_1 \vee x_2)$ to prevent the same conflicts in a later search, which is called *clause recording* [1].

Software SAT solvers drastically speed up the solving process by using these methods, but conflict analysis is a complex process, so only a few SAT solvers have implemented them[3].

Consequently, there is a need for a way to implement these methods without adopting conflict analysis at all. The purpose of our work is to investigate the possibility of a hardware SAT solver that implements non-chronological backtracking and clause recording without complex conflict analysis.



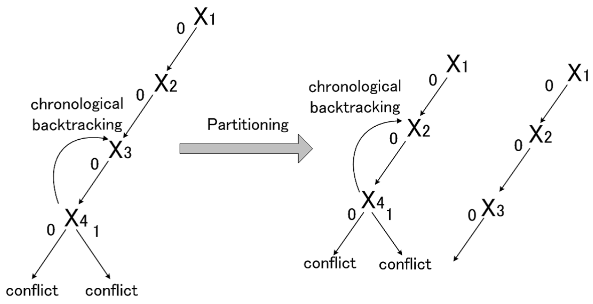


Fig. 1. Chronological and non-chronological backtracking

Fig. 2. Chronological backtracking that has the same effect as non-chronological backtracking

3 Partitioning a SAT Formula

3.1 Key Idea

Our key idea is that by partitioning a SAT formula, a SAT solver may backtrack non-chronologically in the same way as chronological backtracking. Suppose that a SAT formula is partitioned into two parts and one partial formula is the same as Example 1, while the other partial formula has x_1, x_2 and x_3. Also, suppose that we assign 0 to x_1, x_2 and x_3 in this order. This SAT solver memorizes a decision order for each partial formula in order to backtrack non-chronologically. In the case of Example 1, the order is $x_1 \rightarrow x_2$ and that of the other partial formula is $x_1 \rightarrow x_2 \rightarrow x_3$. When conflicts occur for both assignments of x_4, as mentioned above, a conventional SAT solver needs conflict analysis to backtrack to x_2, thereby causing these conflicts. Although a conventional SAT solver should backtrack to the most recently assigned variable, x_3 if no complex conflict analysis is performed, by partitioning the formula, the SAT solver can implicitly backtrack to x_2 without conflict analysis.

Next we explain how to backtrack to x_2 in the above case. The SAT solver finds the partial formula that has the clauses causing conflicts, after which it backtracks to the most recently assigned variable of the partial formula. In the above case, when the partial formula of Example 1 causes the conflicts, its most recently assigned variable is x_2. Therefore, the SAT solver backtracks to x_2 without conflict analysis, which is shown in Fig. 2. At this point, the SAT solver does not backtrack to the variable that does not belong to a partial formula causing conflicts and so is not related to any conflicts. If a partial formula causing conflicts has only a few variables that are unrelated to the conflicts, the backtracks produced by the above method can work well.

A SAT solver can implement clause recording without conflict analysis. When conflicts occur, as in Example 1, the SAT solver can recognize that the formula cannot be satisfied under the previous assignments of $x_1 = 0, x_2 = 0$ and $x_3 = 0$ without conflict analysis. The SAT solver can add a new clause $(x_1 \vee x_2 \vee x_3)$. A hardware SAT solver can implement clause recording in this way. However, the added clause includes unnecessary variable, x_3, which is not actually related to the conflicts. This is because the SAT solver does not, in fact, find the variables causing the conflicts. Therefore, clause recording without conflict analysis does not work well.

Our key idea is that by partitioning a SAT formula, the effect of clause recording by the above method can also be improved. When conflicts occur as in the above example, the previously assigned variables of the partial formula causing the conflicts are $x_1 = 0$ and $x_2 = 0$, which means a SAT solver can implicitly add a new clause $(x_1 \vee x_2)$. The variable x_3 automatically disappears from the clause since it does not belong to the partial formula. The number of previously assigned variables becomes small by partitioning a formula, and clause recording can work well when there are no unnecessary variables in previously assigned variables of partial formula causing conflicts.

Two partial formulas share only x_1 and x_2. There are no unassigned shared variables among two partial formulas after we assign a value to x_1 and x_2. Then, two partial formulas can be solved in parallel, which is ideal for hardware solutions.

3.2 Partition Algorithm

If a formula is partitioned such that the number of shared variables is small, our approach mentioned in the previous section works well. We employ hMETIS[4], which is a multi-level hypergraph partition algorithm, as the partition algorithm. We consider a variable for a hyperedge and a clause for a vertex in order to apply this algorithm to a SAT formula. In most cases, we can partition a formula into an arbitrary number of parts and minimize the number of shared variables by using hMETIS.

4 The Proposed Architecture

Figure 3 shows the proposed SAT solver architecture, where a *Decision unit* (DU) stores *shared variables* that are shared by more than two partial formulas and assigns a value to the shared variables in statically determined order. Whenever a DU assigns a value to a shared variable, the DU memorizes the decision order for each *Partial SAT Solver* (PSS). Each PSS corresponds to a partial SAT formula, which is obtained by hMETIS. A PSS receives assignments of the shared variable from a DU, and individually applies the DPLL algorithm to a partial formula when all shared variables among the other PSSs are assigned.

Consider PSS 1 in Fig. 3 to see the behavior of the proposed SAT solver. The PSS corresponds to a partial formula whose variables are x_1, x_3, x_5, x_6 and x_7, and shares x_1 and x_3 among the other PSSs.

The DU stores shared variables, x_1, x_2, x_3 and x_4 and assigns a value to each of them. Suppose that the decision order is $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$. The PSS receives assignments of shared variables, x_1 and x_3, and can assign a value to each local variable, x_5, x_6 and x_7. At this point, the DU can assign a value to the other shared

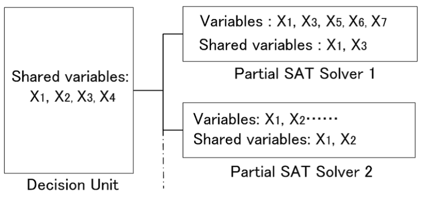


Fig. 3. Proposed SAT Solver Architecture

variable, x_4, in parallel. If conflicts occur for both assignments of x_6, the PSS backtracks to the most recently assigned variable, x_5, and if a conflict occurs again for the other assignment of x_5, the DU backtracks to the most recently assigned variable instead of the PSS. This is because the most recently assigned variable is a shared one. In the above case, the most recently assigned variable of the PSS is x_3. Therefore the DU backtracks to x_3 and stops PSSs that share x_3. The DU changes the assignment of x_3 and sends it to the PSSs. PSSs that share x_3 receive it and restart the DPLL algorithm. In addition, the DU adds a new clause under previous assignments of the PSS's shared variable. In the above case, if x_1 and x_3 were assigned to 0, the DU adds a new clause $(x_1 \vee x_3)$ to the PSS. During this period, the proposed hardware SAT solver does not backtrack to the variable that does not belong to the PSS causing the conflicts, indicating that these backtrackings may have the same effect as software non-chronological backtracking.

In addition, clause recording can work well since the SAT solver adds a new clause when the number of previously assigned variables is small. If a conflict occurs again for the other assignment of x_3, the DU backtracks to the most recently variable, x_2 in conventional way. Therefore, when the DU backtracks to one shared variable from another, the backtracking does not function as non-chronological backtracking.

We are now developing a reconfigurable hardware platform, in which the SAT solver needs to be capable of partial reconfiguration in order for a new clauses to be added for quick run-time operation.

5 Experimental Results

We compared the proposed hardware SAT solver with a contemporary software SAT solver, zCaff[6], which implements non-chronological backtracking and clause recording. To measure the hardware SAT solver's running time, we built a software simulator and assumed our hardware to work at 100 MHz. Note that the proposed hardware SAT solver processes decisions, implications and backtrackings in the same way as conventional hardware SAT solvers. Therefore, the assumption may be appropriate for the state-of-the-art technology. zChaff was executed on a Pentium IV PC with a 2.8-GHz processor with and 1-GB RAM. The problem instances were obtained from a DIMACS suite[7] and SAT 2002 competition benchmarks[8]. Table. 1 shows the compared results.

Table 1. The results compared with those by zCaff

Instances		#Parts	#Shared Vars	#Parallel PSSs	Estimated Speedup Ratio
Name(#vars)	Class				
x1_24.shuffled(70)	EDA	3	15	1.18	80.15
ssa0432-003(435)	EDA	12	49	1.36	24.08
61.shuffled(378)	EDA	16	25	3.82	29.30
dubois50(150)	random	8	14	1.47	13.98
pret150_25(150)	graph coloring	12	19	1.47	84.33
rope_0003.shuffled(108)	graph coloring	4	36	1.05	0.73
dp05u04.shuffled(1571)	dining philosophers	64	263	1.00	4.50

#Parts is the number of partial SAT formulas. #Parallel PSSs is the average number of running PSSs during the solving process.

Note that even if we have many PSSs, they cannot always work in parallel, so #Parallel PSSs is much smaller than #Parts. The reason is that a PSS often backtracks to a shared variable and stop processes of other PSSs. However, the result reveals that if the number of shared variables among PSSs is small, we estimate a speed increase of 5-84x compared with zChaff. If the number of shared variables is small, the number of previously assigned variables of a PSS is also small. Therefore, chronological backtracking may have the same effect as non-chronological backtracking, and clause recording can work well. We would also like to note that even though the number of parallel running PSSs is small, a PSS parallelizes the implication process internally. Consequently, we are able to estimate speed increase compared with zChaff.

Furthermore, we observed that EDA problems tend toward a small number of shared variables, suggesting that EDA problems may be effectively solved by the proposed hardware SAT solver.

6 Conclusions and Future Work

In this paper we showed that a hardware SAT solver can implicitly implement non-chronological backtracking and clause recording without complex conflict analysis by partitioning a SAT formula. We can estimate that there will be a 5-84x increase in speed compared with a contemporary software SAT solver if a formula is partitioned such that the number of shared variables is small.

We are now developing a SAT solver architecture that is capable of partial reconfigurable functionality.

References

1. L.M. Silva and K.A. Sakallah, gGRASP: A Search Algorithm for Propositional Satisfiability,h IEEE Trans. Computers, vol. 48, no. 5, pp. 506-521, May 1999.
2. A. Dandalis and V.K. Prasanna, gRun-Time Performance Optimization of an FPGA-Based Deduction Engine for SAT Solvers,h ACM Trans. Design Automation of Electronic Systems, vol. 7, no. 4, pp. 547-562, Oct. 2002.

3. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, gSolving Boolean Satisfiability with Dynamic Hardware Configurations,h Field- Programmable Logic: From FPGAs to Computing Paradigm, R.W. Hartenstein and A. Keevallik, eds., pp. 326-235, Springer, 1998.
4. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. gMultilevel hypergraph partitioning: Application in the vlsi domain.h In Proceedings of the Design and Automation Conference, 1997.
5. M. Davis, G. Logemann, and D. Loveland, gA Machine Program for Theorem Proving,h Comm. ACM, no. 5, pp. 394-397, 1962.
6. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. gChaff: Engineering an efficient SAT Solver,h Proceedings of the Design Automation Conference, July 2001.
7. DIMACS satsiability benchmark suite, available at ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/
8. SAT 2002 competition benchmarks, available at http://www.satlib.org/

Searching the Web with an FPGA Based Search Engine

Séamas McGettrick, Dermot Geraghty, and Ciarán McElroy

Dept. of Mechanical and Manufacturing Engineering,
Trinity College Dublin,
Ireland
{mcgettrs, tgerghty, ciaran.mcelroy}@tcd.ie

Abstract. In this paper we describe an FPGA implementation for solving Internet ranking algorithms. Sparse Matrix by Vector multiplication forms a major part of these algorithms. Due to memory bandwidth problems, general purpose processors only achieve a fraction of peak processing power when dealing with sparse matrices. Field-Programmable Gate Arrays (FPGAs) have the potential to significantly improve this performance. In order to support real-life Internet ranking problems a large memory is needed to store the sparse matrix. This memory requirement cannot be fulfilled with the FPGA's on-board block-RAM and SDRAM is expensive and limited in size. Commodity memory such as Double Data Rate (DDR) SDRAM is cheap and fully scalable and would appear to be the best solution. This paper discusses the possibility of a custom architecture on FPGA, for Internet ranking algorithms, using DDR SDRAM. An overview of work to date is also presented as well as a plan for future work.

1 Introduction

Internet Ranking algorithms are computed using Sparse Matrix by Vector Multiplications (SMVM). These calculations have been dubbed "The largest matrix calculations in the world" [8]. However, General-purpose processors are inefficient in calculating SMVM. The cache structure, one instruction completed per cycle data path and fixed precision of the GPP are some of the reasons for this inefficiency. Modern advancement in FPGA technology has made it possible to use FPGAs to calculate real-world numerically intensive calculations. FPGAs have the potential for high memory bandwidth. They can take advantage of parallelism in algorithms to split the algorithm over multiple processing units and can be designed to use any precision needed for a calculation. These characteristics make the FPGA a useful candidate-architecture for a ranking algorithm calculation accelerator by overcoming the limitations of GPP. No work to date has been published on how ranking algorithm calculations perform on FPGA or GPP. In this paper we discuss the possibility of using FPGAs to calculate internet ranking algorithms. We do this by comparing these problems to problems in the well studied field of Finite Element Analysis.

2 Internet Ranking Algorithms Overview

The Internet is a massive and highly dynamic environment. Pages are constantly being added, removed and updated. Google™ Web Search indexes over 8 billion pages and this number is constantly growing [1]. There is a plethora of pages available on almost every subject, some of which are useful and some of which are not. Search Engines use ranking algorithms to gauge the usefulness of a page's content to a submitted query. There are many different ranking algorithms. PageRank, ExpertRank and MSN Rank are names of the ranking algorithms used by Google, Ask and MSN Search engines respectively [2][3][4]. These algorithms all take advantage of the linking structure of the Internet. In this section we will look at some of these algorithms and draw attention to the operations used to compute them.

2.1 HITS (Hypertext Induced Tropic Search)

Jon Kleinberg developed HITS in 1997 [5]. ExpertRank ™ used in the ASK search engine is based on the HITS ranking algorithm. The HITS algorithm uses the linking structure of the Internet to divide web pages into two categories, Hubs and Authorities. The algorithm computes both a Hub and an Authority score. The Hits thesis is that good Authorities are pointed to by good Hubs, and good Hubs point to good Authorities [5]. A high Hub score suggests that the page links to pages containing relevant information and a high Authority score indicates the page contains relevant information for the query. The Hub and Authority score for a page can be given by equation 1 [5]. In this equation, $x^{(k)}$ is a vector of Hub scores, $y^{(k)}$ is a vector of Authority scores, k is the current iteration and L is a adjacency matrix corresponding to the web being searched.

$$\begin{aligned} x^{(k)} &= L^T L x^{(k-1)} \\ y^{(k)} &= L L^T y^{(k-1)} \end{aligned} \tag{1}$$

To achieve a good estimation of Hub and Authority scores for a given search these equations must be iterated multiple times when a user makes a query on a HITS search engine. The calculations are simple sparse matrix by vector multiplication. They are solved at query time [6], so the solve time is critical to the efficient operation of the search engine. To ensure convergence of the HITS algorithm normalization may be applied to the L matrix. This requires each term in the matrix to be divided at the beginning of the algorithm [6].

2.2 PageRank

Sergey Brin and Lawrence Page developed PageRank in 1998 [7]. They went on to found the Internet Search Engine Google™ and PageRank™ remains the heart of Google to this day. Like HITS it uses the linking structure of the Internet to calculate its ranking scheme. PageRank avoids the inherent weakness of HITS. It is query independent. This means that it is calculated prior to query time and the same ranking score can be used for all queries until a new PageRank vector has been calculated [7].

PageRank calculates an importance score for every web page on the Internet and not a relevancy score to a given query [6]. The importance score is calculated by a vote system. Inbound hyperlinks to a page are votes for that page and outbound links are a pages way of casting its votes for other pages. The PageRank vector, R, can be calculated using equation 2 [6], where P is the Google matrix, which can be easily obtained from the adjacency matrix for the entire Internet [7].

$$R_j^T = R_{j-1}^T P \tag{2}$$

The Google's PageRank calculation has been referred to as the world's biggest matrix calculation [8]. The matrix, P, has over 8 billion columns and rows [1]. The calculation in Equation 2 has to be computed numerous times to achieve an accurate value of PageRank and convergence. Like HITS, PageRank is calculated using a simple matrix by vector multiplication.

3 Limitation of Current Systems

In the previous section we saw that Sparse Matrix by Vector Multiplication (SMVM) is an important operation when computing ranking algorithms. The General purpose processors that calculate these operations often perform poorly. McSweeney [9] benchmarked SMVM operations on a 3GHz Intel Pentium 4 processor. He found that the processor only achieved 10.5% of the theoretical peak. This is due to memory bandwidth issues [10]. Figure 1 shows the memory hierarchy for the Pentium 4 processor with small, but fast, memory at the top and large and slow memory at the bottom. As problem sizes grow the General-Purpose processor is forced to use slower memory. If the problem outgrows the system RAM, then the processor is forced to continuously page hard-disk. This slow process stops the GPP from achieving its peak performance. Performance therefore is connected to the size of the problem. This memory hierarchy also stops the CPU taking advantage of parallelism in the operations because it has only one channel to memory [9].

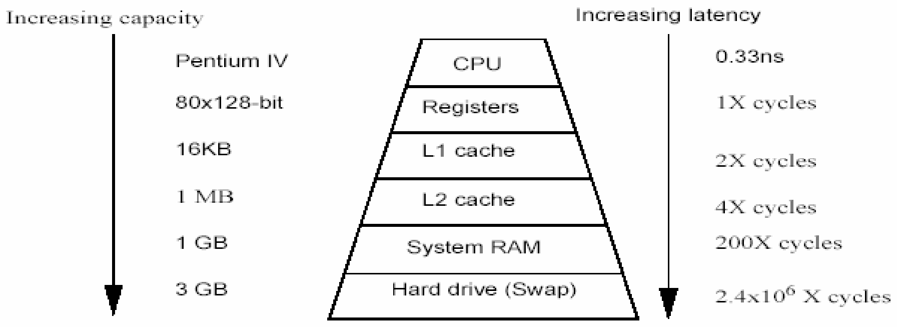


Fig. 1. Memory Hierarch for Pentium 4 with memory latency [9]

McSweeney's calculations were based on general purpose processor calculating Finite Element problems for use in mechanical engineering systems. To the best of the author's knowledge no benchmarking has been published using Internet style matrices to date.

3.1 Internet vs Finite Element Matrices

There are a number of differences between matrices that describe the linking structure of the Internet and matrices used to solve Finite Element (FE) problems. Figure 2 shows a typical example of a FE matrix used in the benchmarks and an example of an Internet matrix taken from a crawl of Trinity College Dublin's website.

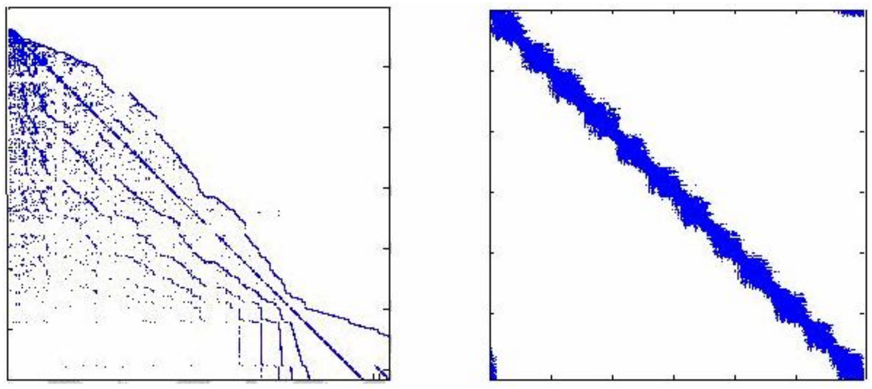


Fig. 2. Web crawl matrix (left) and FE matrix (right)

Table 1 lists the major features of the two types of matrices.

Table 1. FE matrices Vs Internet Matrices

FE Matrices [9]	Internet Matrices (from crawls)
Very Sparse (0.01 – 1.1%)	Very Sparse (0.02%)
Often symmetric	Never symmetric
Double Precision floating-point	Single Precision or less f-point [11]
Millions of rows	Billions of rows
11-72 entries per column (average)	4-12 entries per column (average)

The major difference between FE matrices and Internet is symmetry. FE matrices tend to be clustered around the diagonal and Internet matrices tend to more scattered. However since General purpose processors treat symmetrical matrices the same as unsymmetrical matrices it is still reasonable to use these figures as a comparison for Internet style matrices.

4 Specialized Hardware

Large SMVM performs poorly on General purpose processors. Specialized hardware for large SMVM could greatly increase the performance of ranking algorithms for the Internet. In recent years FPGAs peak performance has become highly competitive with General purpose processors [12]. The introduction of large FPGAs has made it possible to design and implement a wide variety of floating point numerical operations. The large number of user IO pins allows for the possibility of large memory bandwidth and the ability to reconfigure makes the FPGA very beneficial for use in scientific problems. This also allows the hardware to be optimized to exploit parallelisms and differences in algorithms. Multiple parallel banks of memory can be connected to the FPGA allowing multiple data stream coming from and going to memory. A number of papers have been published on accelerators for floating-point operations in engineering applications [14, 15, 16]. These solutions use the FPGAs block RAM and some external SRAM for matrix storage. They cannot handle large data sets like those found in ranking algorithm calculations. In order to do this the FPGA solution would need access to large quantities of commodity memory. The ability to increase this memory capacity as problem sizes grow is essential. The use of commodity memory like DDR 2 SDRAM would allow the FPGA system to scale successfully. Internet ranking algorithms do not need to use double precision floating point arithmetic [11]. General purpose processors always calculate floating point numbers in double precision or more and then mask them to lower precision numbers. An FPGA solution could save time and resources by just computing the required precision. FPGAs are more suited to integer or lower precision floating point calculations than they are to double precision arithmetic. Since Internet ranking arithmetic does not require double precision arithmetic it is a perfect architecture for implementation on FPGA.

4.1 Proposed Hardware

Figure 3 shows an outline of a possible hardware architecture that could be used to solve Internet ranking algorithms. Multiple parallel banks of memory feed parallel SMVM units on the FPGA and results stream into separate shared memory. This external memory allows the solution to scale to solve large problems and allows parallelism in the ranking algorithms. The figure shows 3 SMVM units in parallel. This number of units was included in the initial design as proof of concept; however, more units can be added to utilize the full memory bandwidth and the full FPGA computation capabilities. The DDR SDRAM runs at a higher clock speed than the FPGA so each bank could be used to feed multiple SMVM units. A PCI interface allows the PC to upload problems to the co-processor board and to download answers from the board.

The SMVM units used in the initial investigation, into solving ranking algorithms on FPGA, are based on the SPAR system described by Taylor et al [16]. In SPAR the matrix is compressed into two vectors. The non-zero entries of the matrix are placed

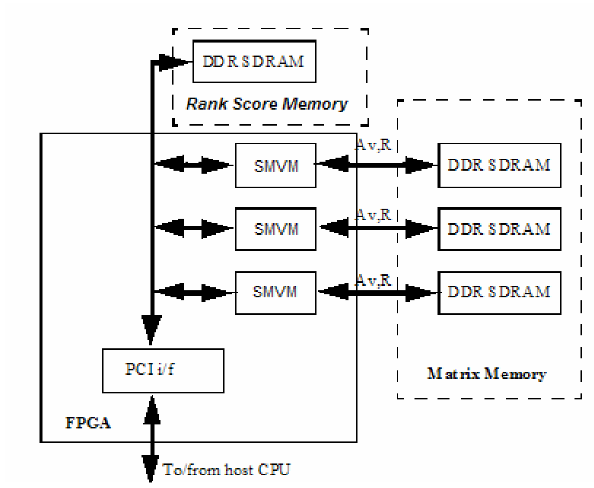


Fig. 3. FPGA solution with external RAM interface

in one vector, with end of columns denoted by floating point zeros. The second vector contains the row information for the non-zero entries. Entries that are aligned with the end of column delimiting zeros contain the next column number. Figure 4 show the block diagram for the SPAR unit used. The compressed matrix vectors are streamed from DDR SDRAM into the spar unit where using a multiplier, adder and result cache the SMVM solution is produced.

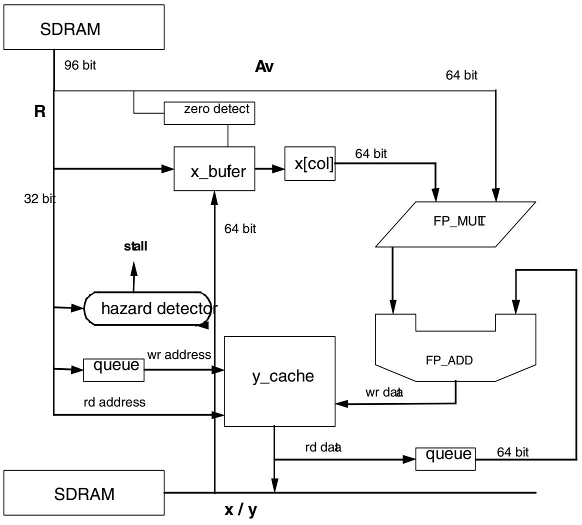


Fig. 4. SPAR hardware architecture

4.2 Custom Hardware Benchmarking

The SMVM units were benchmarked using the same FE matrices used to benchmark the general purpose processor. Figure 5 shows the results of these tests in MFLOPS (**FL**oating point **OP**erations per **S**econd). These results were obtained by using a system with 3 SPAR units which has a peak MFLOPS rating of 570 MFLOPS.

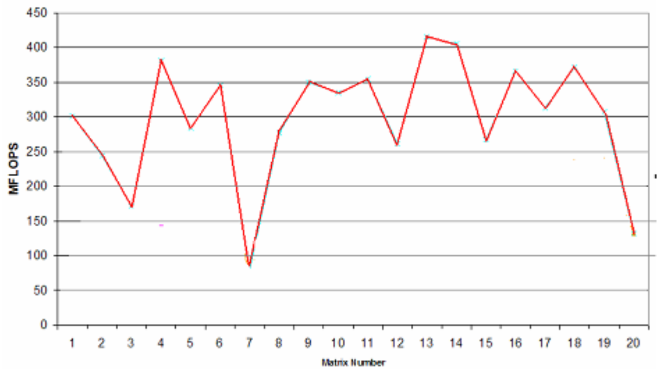


Fig. 5. Custom architecture SMVM performance

The maximum calculation speed for the custom architecture system achieved 420 MFLOPS. This is equivalent to 74% of the theoretical peak. The general purpose processor achieved only 10.5% its theoretical peak. This shows that the FPGA solution performed more efficiently. The general purpose processor did calculate the answer quicker. The FPGA solution achieved about 55% of the reference machines speed this is due to the much slower clock on the FPGA (1/31.5).

5 Conclusions

Internet ranking algorithms are large computationally intensive matrix problems. General purpose processors only achieve a fraction of their theoretical peak when calculating these problems [9]. This is due to memory bandwidth issues. The large number of IO pins on modern FPGAs makes it possible to use them to achieve a high memory bandwidth and exploit parallelisms in algorithms.

A custom architecture for SMVM calculations was implemented on FPGA. It has access to external banks of DDR SDRAM, which made it scalable to any sized problem, and communicated with the host PC via PCI. The architecture managed to achieve up to 74% of its theoretical peak performance, which, due to its lower clock rate, was about 55% the speed of the general purpose processor. A number of things can be done to further improve this result so that the FPGA solution might out perform the general purpose processor. This solution was implemented on Virtex ii FPGAs. The next generation Virtex IV are available and newer FPGAs will soon be available. These have increased clock speeds, which would increase the custom architectures performance. More SMVM units canbe added to the FPGA to allow greater exploitation of parallelism. The architecture used to run these benchmarks calculated all SMVM in the IEEE double precision format. Internet Ranking algorithms do not use double precision floating point numbers they use lower precision. The custom architecture could take advantage of this to decrease adder and multiplier latencies to allow for a faster throughput. This would increase the FLOPS rating significantly.

5.1 Future Work

This project is still in its early stages. The architecture described above does is a generic SMVM accelerator. To achieve an optimum result, an architecture designed with specifically Internet ranking algorithms in mind needs to be implemented. This will be attempted over the next number of months. Benchmarking of Internet style matrices will also be carried out. Work is currently being carried out on precision investigation of Internet ranking algorithms.

References

1. http://www.google.co.uk/corporate/facts.html
2. The Google Search Engine, http://www.google.com
3. The ASK Search Engine, http://www.ask.com
4. The MSN Search Engine, http://www.msn.com
5. Jon Kleinberg, *Authoritative Sources in a hyperlinked environment*. Journal of ACM 46 , 1999
6. Amy N. Langville and Carl D Meyer, *A survey of Eigenvector methods for web Information retrieval*, SIAM Review , 2004
7. Sergey Brin and Lawrence Page, *The Anatomy of a Large-Scale Hypertexual Web Search Engine*, Computer Networks and ISDN systems 33:107 – 117, 1998
8. Cleve Moler, *The world's largest computation. Mathlab news and notes*, pages 12-13, October 2002
9. Colm Mc Sweeney, *An FPGA accelerator for the iterative solution of sparse linear systems,* MSc Thesis, 2006
10. W. D. Gropp, D. K. Kasushik, D. E. Keyes, and B. F. Smith. *Towards realistic bounds for implicit CFD codes*, Proceedings of Parallel Computational Fluid Dynamics, pp. 241-248, 1999.
11. Amy N. Langville and Carl D Meyer, *A survey of Eigenvector methods for web Information retrieval*, SIAM Review , 2004.
12. K. Underwood. *FPGA vs. CPU: Trends in peak floating point performance*. In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays, Monterrey, CA February 2004.
13. William D. Smith, Austars R. Schnore, *Towards an RCC-Based Accelerator for Computational Fluid Dynamics Applications*, The Journal of Supercomputing, 30, pp 239-261, 2004.
14. Fithian, Brown, Singleterry, Storaasli. *Iterative Matrix Equation Solver for a reconfigurable FPAG-Based HyperComputer*, Star Bridge Systems, 2002.
15. K. Ramachandran. Unstructured Finite Element Computations on Configurable Computers, Masters Thesis, Virginia Polytechnic Institute and State University, 1998
16. Taylor, Valerie E, application-specific architectures for large finite element applications, PhD Thesis Berkeley California, 1991

An Acceleration Method for Evolutionary Systems Based on Iterated Prisoner's Dilemma

Yoshiki Yamaguchi, Kenji Kanazawa, Yoshiharu Ohke, and Tsutomu Maruyama

Systems and Information Engineerings, University of Tsukuba
1-1-1 Ten-ou-dai Tsukuba Ibaraki 305-8573 Japan
yoshiki@cs.tsukuba.ac.jp

Abstract. In this paper, we describe an acceleration method for evolutionary systems based on Iterated Prisoner's Dilemma (IPD). In the systems, agents play IPD games with other agents, and strategies which earn more payoffs will gradually increase their ratio in the total population. Most computation time of the systems is occupied by IPD games, which are repeatedly played among the agents. In our method, repetition of moves in IPD games are detected, and payoffs by those moves are calculated at once by multipliers. In order to maximize the number of units for IPD games on one FPGA, multipliers are shared by the agents. We have implemented a system with the method on XC2VP100, and the speedup by the method is about four times compared with a system without the method.

1 Introduction

It was shown that the evolution of a world consisting of a number of agents that play Iterated Prisoner's Dilemma (IPD) games each other is open-ended by Lindgren [KL91][KL94]. In the model, strategies of the agents are evolved by mutations, and the agents that obtained more payoffs will gradually increase their ratios, and those with less payoffs will decrease in the total population. This Lindgren's model is used as a basis of many systems for simulating evolutionary behaviors in real world because this model shows very complex behaviors in despite of its very simple structures. This model is, however, very time consuming. Evolutions of the agents have been observed and discussed under limited number of agents and limited number of generations.

The computation of IPD games is very regular and simple. It can be accelerated by dedicated hardware systems. The others have already proposed acceleration methods using FPGAs [YY99][YY00]. In those methods, $N/2$ units are used to execute IPD games among N agents in parallel (each IPD unit executes a game between two agents). All units run synchronously; they start IPD games at the same time, and finish the games exactly in same clock cycles. With those methods, we could not simulate evolutions of many agents, because the number of agents are limited by the amount of hardware resources.

In this paper, we propose a new acceleration method for simulating large number of agents for long generations. In this method, repetition of moves in

Table 1. Payoff Matrix for Co-evolution Model

		Agent-B	
		Cooperate (C)	Defect (D)
Agent-A	Cooperate(C)	Reward, A:3 / B:3	Sucker, A:0 / B:5
	Defect(D)	Temptation, A:5 / B:0	Punishment, A:1 / B:1

IPD games are detected by IPD units, and payoffs by those moves are calculated at once using multipliers. In order to maximize the number of units for IPD games on one FPGA, multipliers are shared by the agents. With this method, the computation time of each game becomes different. In order to average the total computation time of each IPD unit, many IPD games are assigned to each IPD unit, and each IPD unit start next games independently.

2 Iterated Prisoner's Dilemma (IPD)

The Prisoner's Dilemma, which is classified as two-person non-zero sum game, is an experimental and theoretical model for investigating the problem of co-operation [PW92]. In the game, two agents select one of the following moves; *Cooperate (C)* or *Defect (D)*, and get a payoff according to their moves using a payoff matrix shown in Table 1.

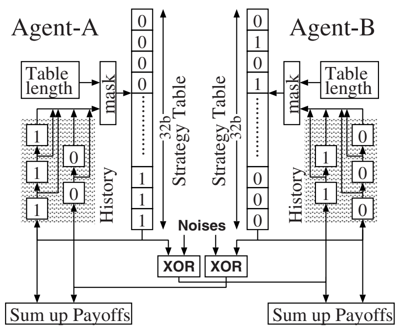
In Iterated Prisoner's Dilemma (IPD), the game is played repeatedly. If the number of iterations is limited and known, rational agents always choose D because the expected payoffs by D in the last game is higher than that of C, and the expected payoffs by D in its previous game is also higher. However, if the number of iterations is unknown or unlimited, cooperative strategies have chances to get higher payoffs. When the IPD games are played among many agents, cooperative agents can get 3 point in average each other, while a defective agent gets 5 point when played with cooperative agents, but only 1 point when played with defective agents. Therefore, the best strategy depends on the environment where the agents belong to.

3 Evolution Model Based on IPD

Lindgren's evolution model is a simple model based on the IPD [KL91][KL94]. In this model, IPD games among agents, population dynamics based on the total payoffs obtained through the games and the mutation of the agents are repeatedly executed. One iteration is called *a generation*.

The main features of the model are as follows.

1. selection of moves using a strategy table and a history of moves
2. mutation of the values in strategy tables, and length of the tables
3. population dynamics using payoffs obtained through IPD games with all other agents



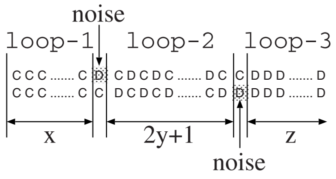


Fig. 1. Lindgren's IPD game model(left) and an example of loops in games(right)

Table 2. The number of moves in loops (generation = 1000000)

loop-type	1	2	3	4	8
moves in loops	0.86	0.91	0.94	0.94	0.94

Each agents has a strategy table (whose size is $1b \times 2$, $1b \times 4$, $1b \times 8$, $1b \times 16$ or $1b \times 32$), and a history of moves (up to five moves). Figure 1 shows the data structures for IPD games, when the size of the strategy table is $1b \times 32$. The values in the strategy table are 0 (C) or 1 (D). The history stores the last five moves (three self moves, and two opponent's moves), and is used as the address to the strategy table. In Figure 1 (left), the history of agent A is 10101 (self-moves are 111, and opponent's moves are 00. Therefore, 10101 is used as the address of the strategy table, and a value in the strategy table is read out as its next move. *Noise* is introduced to cause miscommunication between the two agents. With small probability, *noises* become high, and opposite moves are transferred to opponents. In the mutation, values in the strategy table are flipped at random. Furthermore, the size of the strategy table is doubled (values are copied; $32b$ is the maximum) or reduced to half (the first or second half; $2b$ is the minimum). The length of the strategy table is used to mask the address (the history) to the strategy table.

In this model, all agents play IPD games with all other agents. Agents that earned more payoffs will be duplicated according to its payoffs, and agents with less payoffs may be deleted. The computation order of the naive implementation becomes

$$N \times (N-1)/2 \times L + PopulationDynamics + Mutation$$

where N is the number of the agents, and L is the number of iterations in one IPD game (given constant L, agents play as if they do not know L is a constant). The computation time for the IPD games ($N \times (N-1)/2 \times L$) dominates the total computation time, and other terms can be almost ignored, because other terms are proportional to only N. In software, in order to reduce the computation time, payoffs of IPD games are stored, and reused for games between same strategies.

However, even between the same strategies, its results may be different because noises are generated using random number sequences. This optimization is a trade-off between the computation time and maintaining diversity.

4 An Acceleration Method for IPD

In IPD games, same moves are often repeated. Figure 1 (right) shows an example of repetition of moves in an IPD game between two agents whose strategies are *TFT (Tit-For-Tat)*. In this strategy, the next move of an agent is the previous move of its opponent. The game starts with CC, and it continues till miscommunication happens by noise (loop-1). Then, the next loop which consists of two moves DC and CD (loop-2) continues until the next noise, and the loop by DD continues (loop-3) to the end. Total payoffs of the first agent in this game can be calculated as follows by detecting those loops.

$CC \times x + DC$ (move by noise) +
$(CD + DC) \times y + CD$ (last move of loop-2) +
DD (move by noise)$+DD \times z$

In our experiments, loops can be found in more than 99.999% of IPD games, and moves in the loops occupies 94% of total moves, when the number of agents is 2048, and the number of iteration is 8192. This means that by detecting loops, we can reduce the computation time drastically. Table 2 shows how many moves in loops can be found, when detecting only loops which repeat one move (*loop-type 1*), up to two moves (*loop-type 2*), up to three moves (*loop-type 3*), up to four moves (*loop-type 4*) and up to eight moves (*loop-type 8*). In Figure 1 (right), loop-1 and loop-3 can be detected by finding *loop-type 1*, and all loops can be detected by finding *loop-type 2*. As shown in Table 2, in most loops, up to three moves are repeated.

When the number of agents is N, the total number of IPD games becomes $N \times (N-1)/2$. Figure 2 (left) shows how to assign IPD games to IPD units,

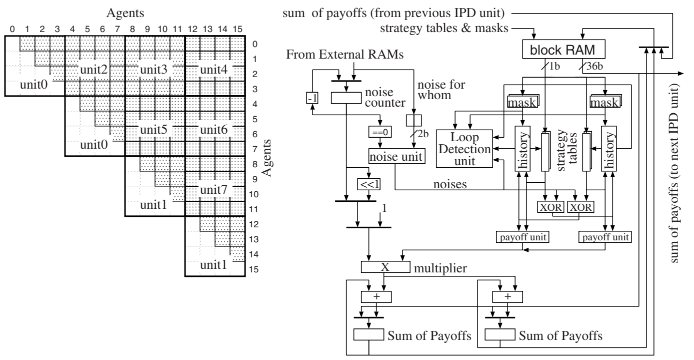


Fig. 2. How to assign games to IPD units (left) and the details of IPD Unit (right)

when the number of agents is 16, and the number of IPD units is 8. In this example, unit0 executes games among agents {0,1,2,3} vs. {0,1,2,3} and {4,5,6,7} vs.{4,5,6,7}. Unit0 needs to hold data for these eight agents. Unit2 executes games among agents {4,5,6,7} vs.{0,1,2,3}, and needs to hold data for these eight agents. Suppose that the number of IPD units is $K^2/2$ (K=4 in Figure 2 (left)). Then, each unit executes $(N/K)^2$ games, and needs to hold data for $N/K \times 2$ agents.

5 Details of the Circuit for the Method

Figure 2 (right) shows the details of IPD unit for the acceleration method. Strategy tables and masks (length of the strategy table) are given and stored in a block RAM. Each IPD unit can hold strategies and masks for up to 256 agents because the size of one block RAM is $36b \times 512$ (the other half of the block RAM is used to store payoffs of each agent). IPD unit executes IPD games among the agents asynchronously with other IPD units using the strategy tables and masks stored in the block RAM. Each unit starts next game as soon as the current game finishes. Strategy tables, masks, and payoffs (initialized to zero in each generation) for two agents are loaded from the block RAM before starting a new game, and new payoffs are written back when the game is finished.

The main features of the IPD unit are

1. generation of noises using *noise counter*,
2. detection of loops by *loop detection unit*,
3. calculation of payoffs using a multiplier, and
4. dual-port memories for strategy tables and masks in order to load those data for the next game while playing current game.

Noises are generated by *noise unit*, and sent to agents designated by 2b register, when the value of *noise counter* becomes zero. This is because we need to know when the current loop will be finished by noises in order to calculate payoffs in the loop using multipliers. Values for those registers are given from external memory banks.

The inputs to *loop detection unit* are two histories of moves, noises and masks. Loops are detected when both agents repeat same moves, and noises are not included in those moves. This unit detects loops by repetition of up to two moves (*loop-type 2*).

When a loop is detected, payoffs by one iteration in the loop (generated by *payoff unit*) are multiplied by the number of iterations (the value of *noise counter* for *loop-type 1*, and half of that for *loop-type 2*). For loop-type 2, payoffs by the last moves in the loop are added after the multiply operation, when the value of *noise counter* is odd. In order to maximize the performance, it is very important to implement more number of IPD units, as well as to improve the performance of each IPD unit. By limiting the detection of loops to those by up to two moves, we can calculate payoffs in the loops for the two agents with one multiplier. Figure 3 shows how to use the multiplier. Payoffs for up to two moves are not

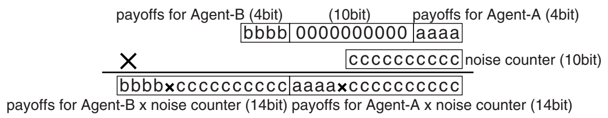

payoffs for Agent-B (4bit)
(10bit)
payoffs for Agent-A (4bit)
bbbb 0000000000 aaaa
×
cccccccccc
noise counter (10bit)
bbbb×cccccccccc aaaa×cccccccccc
payoffs for Agent-B x noise counter (14bit)
payoffs for Agent-A x noise counter (14bit)

Fig. 3. Usage of the Multiplier

larger than 10 (4*b*), because the maximum payoff in Table 1 is 5. If the number of iterations in a loop is less than 1024 (10*b*), payoffs for the two agent can be calculated in one multiply operation by one $18b \times 18b$ multiplier. The number of multiplier in one FPGA is limited, though the multipliers are key units for this acceleration method. With this technique, we can double the number of IPD units on one FPGA. However, this technique introduces a new limitation; the distance between noises must be less than 1024. In order to realize longer distance, 1023 is set to *noise counter*, but 00 is set to 2b register which shows the noises are for whom; namely, noises are not generated for any agent. With this technique, we can deal with noises of arbitrary distance, though the maximum length of the loop is limited to 1023. The practical interval between noises is about 100. Therefore, such cases are very rare, and have no significant effect on total performance.

6 Performance

We have implemented the circuit on Xilinx XC2VP100 (on ADM-XP by Alpha-Data), and the circuit runs at 76.799 MHz. The speedup by our method is 3.7 times compared with a circuit without loop detection.

The speedup by our method is decided by the following three factors.

1. improvement of the computation time of each IPD game
2. the number of IPD units which can be implemented on one FPGA
3. variation of the total computation time of IPD games in each IPD unit

The IPD unit needs $log(StrategyTableLegth) + 1$ clock cycles to detect loops, and one more clock cycle (or two when *loop-type 2* is detected, and the value of *noise counter* is odd) to calculate payoffs in the loop using the multiplier. Therefore, the expected speed up by the unit becomes about 7 times when the noise ratio is 0.01 (average distance between noises is 100), because the average of clock cycles to detect loops is about 4.

On XC2VP100, one IPD unit with loop detection occupies 145 slices, 1 multiplier and 1 block RAM, while one IPD unit without loop detection occupies 74 slices and 1 block RAM. Therefore, we can implement 288 ($= 24 \times 24/2$) IPD units with loop detection, and 421 ($= 29 \times 29/2$) IPD units without loop detection on one XC2VP100. Both circuits runs at same frequency, because units for loop detection are not the bottlenecks.

In our method, the computation of the next generation can not be started till all IPD units finish IPD games which are assigned to them. Therefore, the computation time of one generation is decided by the IPD unit which requires the longest computation time (with IPD units without loop detection, all IPD units finish their computation in the same clock cycles). The circuit can simulate up to 2048 agents, and 6241 games are assigned to each IPD unit in each generation. In this case, the difference of the computation time of each IPD unit is less than 10% in average.

7 Conclusions and Future Works

In this paper, we proposed an acceleration method for evolutionary systems based on Iterated Prisoner's Dilemma (IPD). A circuit with the method has been implemented on Xilinx XC2VP100, and the speedup by the method is about four times compared with a circuit without the method.

We are now improving the control unit of the circuit, and trying to evaluate the behaviors of agents in the system for very long generation.

References

[PW92] W. Poundstone, "Prisoner's Dilemma", Doubleday, 1992.

[KL91] K. Lindgren, "Evolutionary Phenomena in Simple Dynamics", Artificial Life II, 1991, pp.295–312.

[KL94] K. Lindgren and M.G. Nordah, "Cooperation and Community Structure in Artificial Ecosystems", Artificial Life I, 1994, pp.15–37.

[YY99] Y. Yamaguchi, T. Maruyama and T.Hoshino, "High Speed Hardware Computation of Co-evolution Models", 5th European Conference on Artificial Life, 1999, pp.566–574.

[YY00] Y. Yamaguchi, A. Miyashita, T. Maruyama and T.Hoshino, "A Co-processor System with a Virtex FPGA for Evolutionary Computation", 10th International Conferene on Field Programmable Logic and Applications, 2000, pp. 240–249.

Real Time Architectures for Moving-Objects Tracking

Matteo Tomasi, Javier Díaz, and Eduardo Ros

Dep. Arquitectura y Tecnología de Computadores, Universidad de Granada, Spain

Abstract. The problem of object tracking is of considerable interest in the scientific community and it is still an open and active field of research. In this paper we address the comparison of two different specific purpose architectures for object tracking based on motion and colour segmentation. On one hand, we have developed a new multi-object segmentation device based on an existing optical flow estimation system. This architecture allows video tracking of fast moving objects based on high speed acquisition cameras. On the other hand, the second approach consists on real time filtering of chromatic components. Multi-object tracking is performed based on segmentation of pixel neighbourhoods according to a predefined colour. In this contribution we evaluate the two methods, comparing their performance, resource consumption and finally, we discuss which architecture fits better in different working scenarios.

Keywords: Real-time image processing, object tracking, motion estimation, colour segmentation.

1 Introduction

The problem of object tracking is of considerable interest in the scientific community and is still open and active field of research [1], [2]. This is a very useful skill that can be used in many fields including visual serving, surveillance, gesture based human-machine interfaces, video editing, compression, augmented reality, visual effects, motion capture, medical and meteorological imaging, etc… [3], [4]. In general, tracking applications solve the problem in software, most of them computing off-line the tracking trajectories. Nevertheless, since we are interested on embedded system applications, we solve this problem using FPGA devices. This is an alternative which already has been used by other authors [5]. This approach has the advantage of extending the applicability of tracking system and also, as presented here, improves the performance of the tracker making feasible high frame rate video tracking. The goal of a tracking system is to analyse the video frames and estimate the position of part of the input video frame (usually a moving object) which is searched against a reference model describing the appearance of the object. This reference can be based on image patches, thus describing how the tracked region should look like pixel-wise [6], on contours (e.g. active contours or condensation algorithm [7]), thus describing the overall shape, and/or on global descriptors such as colour models [8] or motion information [9]. In this contribution we focus on the two last approaches based on colour and motion, which can be defined as blob tracking techniques because object position is computed as the centroid of a filtered saliency map of the corresponding feature. The first approach uses the optical flow which is a well known research field used to recover 2-D motion from image sequences. There are different methods based on image block-matching, gradient constraints, phase conservation or energy models [10]. In this work, we use the Lucas & Kanade (L&K) gradient based method [10], [11] that is highlighted as a good candidate to be implemented on hardware with affordable resources consumption [12], [13], [14], [9]. For the implementation of motion estimation we use the highly parallelized architecture described in [15]. The second approach uses tracking based colour segmentation. Its architecture consists in a pixel wise filtering pipeline which, after binarization, searches in a local neighbourhood to reject/enhance this binary saliency map. Both tracking systems include a new algorithm of multiple objects segmentation. The final system output represent a vector of coordinates which indicates the number and position of the tracked objects. The whole architecture is optimized adopting a hardware friendly method that allows real time processing with moderate resources consumption.

The paper is structured as follows: in the section 2 we illustrate the segmentation approach based on motion or colour segmentation. We also discuss the applicability of both alternatives. Section 3 presents the hardware implementation, system resources and performance of both architectures. Finally, section 4 present the achieved results and concludes discussing future improvements.

2 Motion or Colour Based Tracking of Multiple Moving Objects

We have developed a flexible scheme witch allows different blob tracking alternatives. The processing stages can be summarized as follows:

1. Motion estimation or colour filtering
2. Object segmentation
3. Erosion or dilatation process based on convolution
4. Multiple object segmentation using block connections

Colour tracking is based on chromatic filtering. This is performed based on RGB components differences (and corresponds to stage 1). For instance, to detect a blue colour object, we have used the following colour filtering equation:

$$(B-R, B-G, B) > (T_1, T_2, T_3) \tag{1}$$

Where T_i stands for three independent thresholds values that we use to filter and binarize the input image. They can be set with a user interface to allow a system customization. This colour component subtraction is a simple technique, hardware friendly and leads to a good trade-off between resources consumption and filtering robustness. Qualitative results can be seen in Fig. 1. After colour component filtering, the binary map is collected over a neighbourhood of 9x9 pixels and thresholded again. This process is achieved applying a local convolution process and it reduces spurious outputs which correspond to isolate and noise areas. The result is presented at the right image of Fig. 1.

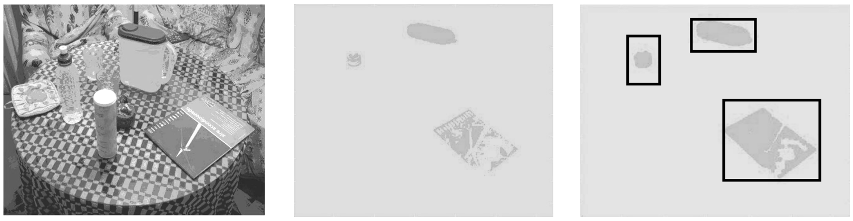


Fig. 1. Different processing stages: original image (left), the pixel segmentation (centre) and the image after pixel dilatation and erosion process (right) are presented. Object detection is marked by a black square indicating the presence of the target object.

Optical flow based video tracking uses approximately the same segmentation block but the colour processing unit is replaced with superpipelined and fully parallelized architecture for optical flow processing described in [15]. The velocity map is binarized using equation (2), where V_x and V_y stand for the pixel velocity components and V_t for a predefined velocity threshold.

$$|V_x| + |V_y| \geq V_t \tag{2}$$

The information of the processing unit is converted into a binary matrix and sent to a convolution filter to reduce the noise and dilate confidence areas. When a pixel is classified as belonging to a target object, information is sent to a circuit module that returns the centre of the object (and corresponds to the stage 4). This unit includes a new multi-object segmentation algorithm, able to work in real time. The algorithm is based on local connection of pixels of the image, which is divided into regions of 8x8 pixels that we call *pixel blocks*. The tracking method can be summarized as follows:

1. Neighbour active pixels are considered to belong to the same object.
2. We can calculate the centre of mass of each pixel block (segmented as an individual object). This estimation becomes the output of the tracking system.

The image is scanned horizontally and vertically block by block looking for "connected" zones (labelled as the same object). This segmentation method only requires storing two horizontal image block lines (each of them with a number of elements equal to the image column resolution divided by 8). At the same time, the system calculates the video output (VGA module). Object data are continuously updated until image's last block, where we obtain the centre after using of a dedicated division unit. Object's centre is displayed in the video output.

2.1 Evaluation Method: Advantages and Drawbacks of the Proposed Methods

After implementation, we have compared and evaluated the two algorithms. We have evaluated velocity and position of the detected objects. We experimentally obtain good optical flow performance for a specific velocity range; for objects with a low level of texture this interval is between 0,5 and 8 pixels per frame. The optical flow drawback is the complete loss of information for stopped objects. In our experiments we analyzed also a pendulum moving example to evaluate different velocities and acceleration driven artefacts. Moving objects with significant speed variations along their trajectories are hardly tracked since motion cues become unreliable. The tracking results for this experiment are presented in Fig. 2.

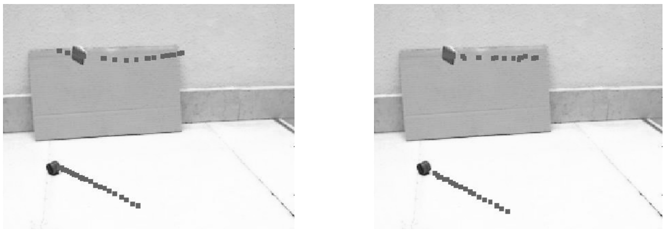


Fig. 2. Tracking of two moving objects; in the left side, computation with colour based segmentation, in the right side motion-based tracking. Tracking position is presented as a grey square which represents the object position along the trajectory.

We have done the tracking experiment in an environment in which colour tracking will be reliable (since the colour component of the object is highly contrasted with its background). We have used this to cross-validate the motion-based tracking approach and analyze under which conditions these visual cues are reliable.

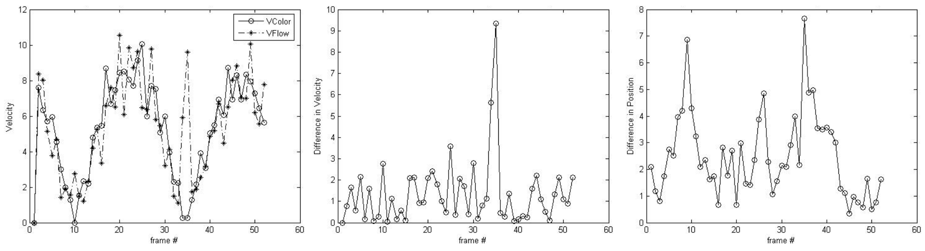


Fig. 3. Evaluation plots of velocity for a video sequence at 30fps. On the left we compare the velocities; below, difference in velocity estimation (centre) and difference in position estimation (right).

Note in Fig. 3 that the differences between the object positions estimated using motion cues and colour cues become significant when the speed of the object is slow (below 2 pixels/second). In these intervals some motion cues are lost but still the velocity of the object is reliably estimated (it is still similar to the colour-based approach). When the speed of the object becomes below 0.5 pixels/second all reliable motion cues vanish and therefore even the speed of the object cannot be reliably recovered. Thus in this range we find great differences between the velocity estimation given by colour and motion cues.

3 Colour and Motion Based Object Tracking Architectures

The whole system has been successfully implemented and tested on a stand-alone board for image processing applications that includes a Xilinx Virtex II FPGA XC2V6000-4. The architecture development consists in a general scheme composed of different units, which are schematically shown in Fig. 4. First of all, we have a frame grabber and a MMU that performs sequence acquisition and visualization. We also have a user interface and a processing block that depends on the goal tracking cues. We design the constructing architecture with a fine pipeline datapath and we store image information into external RAM, to allow a good processing rate, essential for real-time applications. Recourses have been optimized also in memory consumption. In the multiobject tracking unit we store only the previous, the actual block line and a support line for continuously updating the data in the RAM memory.

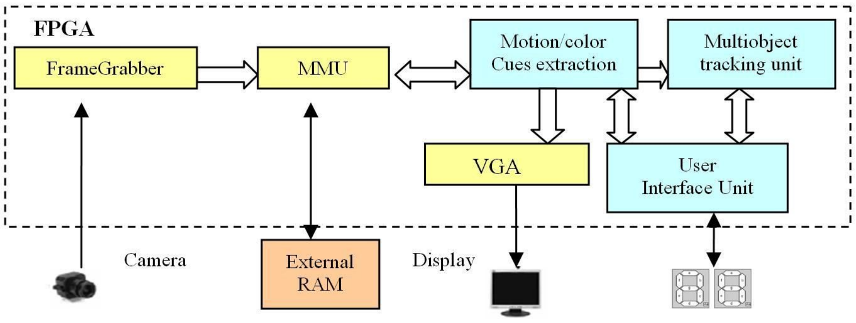


Fig. 4. Architecture scheme of complete system. The processing block can include colour or motion based processing.

The system resources consumption is presented in table 1 for colour based tracking and for motion based tracking in Table 2. Tables divide the resources consumption at three different stages, interfacing circuits, colour filtering or motion detection and finally the multiobject segmentation unit, common to both modules. The last rows show the global systems resources consumption.

Table 1. Recourses consumption for a colour-based tracking system using a Virtex II XC2V1000-4 FPGA. The whole system consumption is presented on the fourth row.

Hardware block	NAND Kgates	FFs	Memory bits	Max clock freq.
FrameGrabber & User interface	168,800	1646	992	44,2 MHz
Color Processing	350,100	1853	157296	52,6 MHz
Multiobject tracking unit	42,559	4005	7920	58,4 MHz
Global Colour based System	561,459	7504	166208	44.2 MHz

Table 2. Recourses consumption for a motion based tracking system. We have used a Virtex II XC2V6000-4 FPGA. Note that the tracking unit is the same that we use on the colour based approach. The whole system consumption is presented on the fourth row.

Hardware block	NAND Kgates	FFs	Memory bits	Max clock freq.
Interface +Hw controllers	65,9	2363	18208	45 MHz
Motion Processing	1145,5	6529	516096	45,5 MHz
Multiobject tracking unit	42,559	4005	7920	58,4 MHz
Global motion based System	1253,959	12897	542224	45 MHz

As we can see on these tables, the most complex system is represented by the optical flow unit that is the more demanding module as presented in Table 2. All the image processing operations are computed at one estimation per clock cycle, and the multiobject tracking is done in just one image reading time. Because of that, the tracking rate is constrained by the maximum clock frequency obtained by the system. For instance, in the case of the colour based tracking, the slower clock rate is 44 MHz which allows a maximum tracking system rate of 143 images per second at VGA resolution.

4 Conclusions

In this contribution we present a specific purpose multi-object tracking architecture that allows the tracking of several objects at a high frame-rate (up to 143 images/s for colour based tracking and 146 images/s for motion based tracking of VGA resolution). The object segmentation is done detecting connected components (visual cues) and labelling them as the same object. Finally the object position is simply estimated calculating the centre of mass.

We briefly compare and discuss the performance, reliability and hardware consumption of two different approaches, one of them based on motion cues and the other one based on colour. We have done the tracking experiments in a controlled scenario in which we know that the colour-based tracking is reliable. Therefore we use this visual modality to evaluate the motion-based approach (which will be more meaningful in a non controlled scenario with diverse colour components in the background). Motion-based tracking becomes unstable for slow velocities (below 0.5 pixels/second) or when an object is static. But although colour-based tracking does not depend on the trajectory characteristics (velocity curves, etc) its reliability is highly dependent on the colour contrast between the object and the background. In the results we highlight that the differences between the two kinds of tracking are significant when the object speed becomes low (below 2 pixels/second). Nevertheless, the advantage of implementing a fine grain pipeline (also for the tracking modules) is that the system can use high performance modules (such as the optic flow circuit described in [15]) to extract more reliable cues. In the case of motion cues, the system is able to process images obtained from a high rate camera and therefore, the object speed range in which the system works reliably gets wider. Reliable motion cues are more expensive in terms of hardware resources. Note that we have implemented a very simple colour component subtraction to extract colour cues.

The conclusion of this work is the necessity of specific purpose architecture capable of using different visual cues for object tracking. Since the whole architecture remains the same independently of the visual cues in which it is based, we can explore different alternatives and even use several of them in a complementary way. Therefore, this work presents an object tracking architecture that can embed different visual processing modules (as cues extraction circuits). Future works will address the integration of multiple cues on this architecture to achieve higher robustness in the tracking process.

Acknowledgment

This work has been supported by the Spanish project National Project DEPROVI (DPI2004-07032) and by the EU grant DRIVSCO (IST-016276-2).

References

[1] Manohar, V., Soundararajan, P., Raju, H., Goldgof, D., Kasturi, R., Garofolo, J.: Performance Evaluation of Object Detection and Tracking in Video. *LNCS*, vol. 3852, 2 (2006) pp. 151-161.

[2] Hsiao, Y.T., Chuang, C.L., Lu, Y.L., Jiang, J.A.: Robust multiple objects tracking using image segmentation and trajectory estimation scheme in video frames. *Image and Vision Computing,* 24, (2006) pp. 1123–1136.

[3] Ellis, T.J.: Performance metrics and methods for tracking in surveillance. *3rd IEEE Workshop on PETS*, Copenhagen, Denmark (2002) pp. 26-31.

[4] Pérez, P., Hue, C., Vermaak, J., Gangnet., M.: Color-based probabilistic tracking. *Conf. on Computer Vision, LNCS*, vol. 2350 (2002) pp. 661-675.

[5] Hamamoto, T., Nagao, S., Aizawa, K.: Real-time objects tracking by using smart image sensor and FPGA. IEEE *Image Processing Proc. Intern. Conf.*, vol. 3, (2002) pp. 441-444.

[6] Shi, J. and Tomasi, C.: Good Features to Track. *IEEE Conference on Computer Vision and Pattern Recognition*, (1994) pp. 593-600.

[7] Yilmaz, A., Li, X., Shah, M.: Contour-Based Object Tracking with Occlusion Handling in Video Acquired Using Mobile Cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 26, n. 11, (2004) pp. 1531-1536.

[8] Zivkovic, Z., Kröse, B.: An EM-like Algorithm for Color-Histogram-based Tracking. *IEEE Conf. on Comp. Vision and Pattern Recognition*, vol. 1, (June 2004) pp.798-803.

[9] Díaz, J., Ros, E., Pelayo, F., Ortigosa, E.M., Mota, S.: FPGA based real-time optical-flow system. IEEE *Trans. on Circ. and Syst. for Video Tec.*, vol. 16, n. 2 (2006) pp. 274-279.

[10] Barron, J.L., Fleet, D.J., Beauchemin, S.: Performance of optical-flow techniques. *International Journal of Computer Vision*, vol. 12, n. 1, (1994) pp. 43-77.

[11] Lucas, B.D., Kanade T.: An Iterative Image Registration Technique with an Application to Stereo Vision. *Proc. of the DARPA Image Understanding Workshop,* (1984) pp. 121-130.

[12] McCane, B., Novins, K., Crannitch, D., Galvin, B.: On Benchmarking Optical Flow. *Computer Vision and Image Understanding*, vol. 84, (2001) pp.126–143.

[13] Liu, H.C., Hong, T.S., Herman, M., Camus, T., Chellappa, R.: Accuracy *vs.* Efficiency Trade-offs in Optical Flow Algorithms. *Comp. Vis. and Image Understanding*, vol. 72, 3, (1998) pp. 271-286.

[14] Díaz, J., Ros, E., Mota, S., Carrillo, R., Agís, R.: Real time optical flow processing system. *Lecture Notes in Computer Science*, vol. 3203, (2004) pp. 617-626.

[15] Díaz, J., Ros, E., Mota, S., Rodriguez-Gomez, R.: Highly parallelized architecture for image motion estimation. *LNCS*, vol. 3985. Springer-Verlag. (2006) pp. 75-86.

Reconfigurable Hardware Evolution Platform for a Spiking Neural Network Robotics Controller

Patrick Rocke, Brian McGinley, Fearghal Morgan, and John Maher

Bio-Inspired and Reconfigurable Computing (BIRC) Research Group,
Dept. Electronic Engineering, NUI Galway, Ireland
{patrick.rocke, brian.mcginley, fearghal.morgan,
john.maher}@nuigalway.ie

Abstract. This paper describes a platform for the hardware evolution of Spiking Neural Network (SNN) based robotics controllers on multiple Field Programmable Analogue Arrays (FPAAs). The SNN robotics controller, evolved using a GA, performs obstacle avoidance and navigation. A robotics simulator is used to evaluate the performance of the evolved hardware SNN. Simulated sonar data is input to FPAA neurons and the SNN returns motor control data to the simulator. Initial results indicate the emergence of effective navigation behavior.

Keywords: Reconfigurable Hardware, FPAA, Genetic Algorithm, Spiking Neural Networks, Evolutionary Computation.

1 Introduction

This paper describes a platform for the evolution of a fixed architecture hardware SNN ultrasound-based obstacle avoidance controller. The hardware SNNs are implemented on cascaded Anadigm AN221E04 FPAA devices interfaced to a software robotics simulator. FPAA-based evolved hardware ANN/SNN implementations have been reported in [1,4,5,6] while [7,8,9,10,11] have demonstrated emergent robotic intelligence by employing GAs to evolve SNNs in hardware. Also SNN robotics controllers have been evolved on FPGAs through interfacing with a simulated robotics model [2].

This research extends on previous work by implementing a hardware SNN robotics controller on multiple FPAAs, using a simulated robotics model. Figure 4 illustrates the reconfigurable hardware evolution platform for a SNN robotics controller. A MobileSim [12] simulation of an Activmedia Pioneer robot [13] is employed to assess the performance of the FPAA hardware controller. While the evaluation of the robot controllers is performed in simulation, the SNN controller is implemented in FPAA hardware.

This use of a hybrid hardware SNN and simulated environment for controller evolution offers an efficient, versatile and adaptable platform for evolutionary development and investigation of bio-inspired concepts and methodologies. It has been shown that initial evolution using a simulated environment and then transferring evolved configurations to hardware for further modification and tuning through evolution is a viable and time/effort effective approach [14].

2 Spiking Neural Networks

Artificial Neural Networks (ANNs) are data processing structures, taking inspiration from the workings of biological neurons in the brain and nervous system. Efforts to mimic the human brain arise from a desire to replicate its computationally powerful characteristics. The favourable characteristics of learning by training, adaptability and redundancy have caused ANN to be applied in fields as varied as engineering, economics, military planning and neuroscience [15].

SNNs are known as the third generation of Neural Network models [3]. Spiking neurons are closely modelled on their biological counterparts and communicate by transmitting short transient spikes to other neurons, via their axons. As illustrated in Figure 1, a neuron fires when the sum of its weighted input spikes exceeds a certain potential, the membrane potential of the cell. Spike generation is followed by a short refractory period, where no other output pulse can be produced. This also causes the membrane potential to be reset to zero [16].

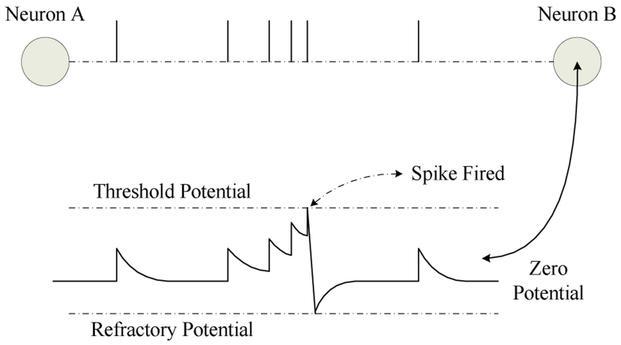


Fig. 1. Neural Potential Variation and Spike Generation Mechanism

Outputs from neurons consist of a series of pulses known as a spike train. These spike sequences may, in turn, be interpreted by other neurons or used as outputs of the network (for output nodes).

In the SNN-based platform reported in this paper, the output spike trains are converted to analogue voltages using two output neurons. These voltages are used within the robot simulation environment to drive the simulated robot's motors.

3 Genetic Algorithms

Genetic Algorithms (GAs) [17] are a type of Evolutionary Algorithm based on the same mechanisms of adaptation observed in nature, including natural selection and genetic diversity.

Evolutionary techniques have been used to perform ANN architecture design, weight training, weight initialisation and learning rule/activation function selection [18]. For this research, evolution of connection weights and thresholds will be investigated. The initial population of SNNs is randomly generated by the GA where the properties of each node are encoded in a data string or genome. The resulting genome is then decoded and configured into its corresponding SNN using Anadigms' API software. The FPAA hardware SNN is then downloaded and its performance evaluated using the robot simulation environment.

4 FPAA Hardware

The reconfigurable hardware evolution platform uses Anadigm Inc. ® AN221E04 [19] FPAAs for SNN implementation. The AN221E04 contains a 2x2 matrix of configurable-analog-blocks (CAB's) which in turn can hold a number of predefined analogue functions provided as Configurable-Analog-Modules (CAM's) with the Anadigm Designer 2 software installation. I/O capabilities are provided through two dedicated analogue output cells and four analogue cells configurable as input or output. A Look up Table (LUT) is provided which is used by certain CAMs.

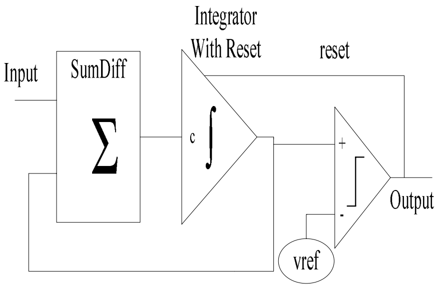


Fig. 2. Spiking Neuron Hardware Model

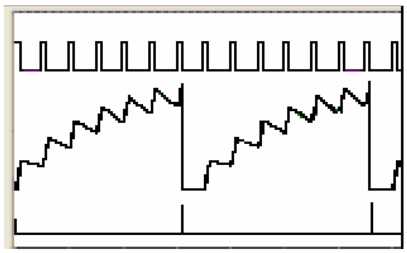


Fig. 3. Sample Simulation of a Spiking Neuron Implemented on the FPAA (top – input spike train, middle – internal potential, bottom – output spike train)

The FPAA spiking neuron model implemented in this research is the leaky integrate-and-fire model [16]. With this model, neurons maintain an internal potential which is accentuated on the reception of weighted incoming spikes. If the reception of an incoming spike causes the internal potential to exceed a predefined threshold, the neuron emits an output spike and its internal potential is reset. During the periods where no spikes arrive at the inputs the internal potential gradually decreases towards a resting potential. A single input spiking neuron model implemented through Anadigm Designer 2 [19] is shown in figure 2. Figure 3 shows a sample simulation of the neurons response.

Due to limited hardware resources on the AN221E04 FPAA, a maximum of two four-input spiking neurons can be implemented on a single FPAA [4]. To realise a network capable of implementing functionality suitable for a robotics controller, multiple AN221E04 are required. Four AN221E04 have been cascaded together to form the SNN. Figure 4 illustrates the multi FPAA SNN architecture along with the rest of the evolutionary system. The neurons are hardwired together to form a fixed architecture neural network. A single FPAA (FPAA4) is used to convert output (robot motor control) spike trains into suitable analogue voltages as inputs to the robotics simulation environment.

5 System Architecture

Figure 4 illustrates the reconfigurable hardware evolution platform for a SNN robotics controller. The simulated robot is equipped with eight forward facing sonar sensors, four of which are used as inputs to the network. These four sonar readings are converted to analogue voltages and sent to the SNN through a National Instruments I/O card. The simulated robot has two motors which are stimulated by the SNN voltage outputs. The analogue readings set the wheel speeds used to drive the simulated robot's motors.

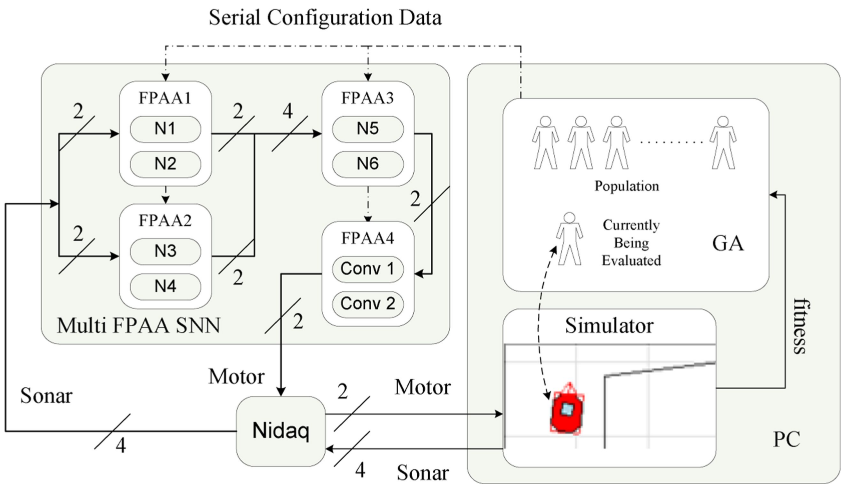


Fig. 4. Reconfigurable Hardware Evolution Platform for Spiking Neural Network Robotics Controller

The SNN is implemented as described in section 4. Individuals in the GA are decoded into their neural network representations and their SNN configuration parameters sent over the serial port to the FPAAs. The sonar inputs propagate their effect through the SNN which provides two output spike trains from N5 and N6.

These output spike trains are converted to appropriate voltage levels using leaky integrators (Conv1 and Conv2) implemented on an FPAA (see Fig 4).

Evaluation of the individuals is accomplished by allowing the robot to roam a simulated environment for an arbitrary length of time. Once this length of time has expired, or the robot has crashed, the GA reads all relevant data from the simulator and assigns a fitness score to the individual. These fitness scores are then used by the GA to determine an individual's probability of progression to later generations.

6 Conclusion and Future Work

This paper describes a platform for the evolution of a fixed architecture hardware SNN ultrasound-based obstacle avoidance controller implemented on cascaded FPAA devices interfaced to a software robotics simulator. The research demonstrates that FPAAs are a viable platform for implementing hardware-based reconfigurable SNN robotics controllers. Their analogue properties coupled with their low power consumption allows for implementation of a scalable SNN hardware platform through cascading of multiple FPAA. Their inherent parallelism addresses the scalability issues present in implementing ANNs on sequential processors.

Initial results indicate the emergence of effective obstacle avoidance and navigation behavior with the successful individuals being evolved within 40 generations.

This project has demonstrated that the hybrid use of FPAA hardware in conjunction with software simulation tools is a practical approach to the development of automated robotics evolution.

The platform offers an efficient, versatile and adaptable tool for evolutionary development and investigation of bio-inspired concepts and methodologies. This approach does away with the need for constant human supervision of the robotic hardware experiment and reduces the initial set-up costs for such a project.

This research focuses on the evolution of weights and thresholds. Also, FPAA device density is currently low. Work on the creation of a large scale FPAA matrix is in progress to accommodate more complex networks. FPAA devices can be cascaded to add many more neurons and include dynamic reconfiguration of connections. Currently a system is under development which will allow dynamic reconfiguration of connections between the neurons implemented on the FPAA.

Future work on the reconfigurable hardware platform will focus on evolving higher-level behaviour in more complex environments. Incorporation of fault tolerance and recovery will also be investigated.

Porting of evolved robotics controllers to real robotics hardware will be performed for further evaluation and an examination of whether possible further evolution will be necessary to achieve effective obstacle avoidance behavior.

Acknowledgments. The authors thank the Irish Research Council for Science, Engineering and Technology (IRCSET) for funding this research project under the Embark Initiative. Thanks also to Anadigm for providing FPAA devices and software.

References

1. D. Berenson, Nicolás Estévez and H. Lipson, Hardware Evolution of Analog Circuits for In-situ Robotic Fault-Recovery, Submitted to 2005 NASA/DoD Conference on Evolvable Hardware
2. S. Bellis, K. M. Razeeb, C. Saha, K. Delaney, C. O'Mathuna, A. Pounds-Cornish, G. de Souza, M. Colley, H. Hagras, G. Clarke, V. Callaghan, C. Argyropoulos, C. Karistianos, G. Nikiforidis, "FPGA Implementation of Spiking Neural Networks - an Initial Step towards Building Tangible Collaborative Autonomous Agents", FPT'04, International Conference on Field-Programmable Technology, The University of Queensland, Brisbane, Australia, 6-8 December, 2004, pp. 449-452.
3. Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. Neural Networks, 10(9):1659--1671, 1997.
4. Patrick Rocke, John Maher, Fearghal Morgan, "Platform for Intrinsic Evolution of Analogue Neural Networks," Reconfig, p. 11, 2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05), 2005.
5. J. Maher, B. McGinley, P. Rocke, F. Morgan, "Intrinsic Hardware Evolution of Neural Networks in Reconfigurable Analogue and Digital Devices", IEEE Symposium on Field Programmable Custom Computing Machines, FCCM06, Napa, California, USA., April 2006
6. Michael A. Terry, Jonathan Marcus, Matthew Farrell, Varun Aggarwal, Una-May O'Reilly, GRACE: Generative Robust Analog Circuit Design, Proceedings of Applications of Evolutionary Computing, EvoWorkshops 2006: (EvoHOT), Lecture Notes in Computer Science 3907, pp 332-343, Springer Verlag.
7. Upegui, C. Andres Pena-Reyes, E. Sanchez. "A methodology for evolving spiking neural network topologies on line using partial dynamic reconfiguration", Swiss Federal Institute of Technology, Lausanne, Switzerland.
8. J. Urzelai, D. Floreano, "Evolution of Adaptive Synapses: Robots with Fast Adaptive Behaviour in New Environments", Swiss Federal Institute of Technology, Lausanne, Switzerland.
9. D. Floreano, N. Schoeni, G. Capriari, J. Blynel, "Evolutionary Bits'n'Spikes" Swiss Federal Institute of Technology, Lausanne, Switzerland.
10. D. Floreano, JC Zufferey, C. Mattiussi, "Evolving Neural Neurons from Wheels to Wings" Swiss Federal Institute of Technology, Lausanne, Switzerland.
11. B. McGinley, F. Morgan, "Evolved Obstacle Avoidance Controller Using a Spiking Neural Network", Workshop on Life-Like Perception Systems 2005, March 2005
12. MobileSim Robot Simulator: http://robots.mobilerobots.com/MobileSim/
13. ActivMedia : http://www.activmedia.com
14. Nolfi, S., Floreano, D., Miglino, O. and Mondada, F. (1994) How to Evolve Autonomous Robots: Different Approaches in Evolutionary Robotics. 4th International Workshop on Artificial Life.
15. S. Haykin, "Neural Networks; A Comprehensive Foundation", Macmillan College Publishing Company, 1994.
16. Gerstner W., Spiking Neuron Models, Cambridge Univ. Press, 2002.
17. Holland J.H., Adaptation in natural and artificial ystem, Ann Arbor, The University of Michigan Press, 1975.
18. XinYao, "Evolving Artificial Networks", Proceedings of the IEEE, Vol. 87, No. 9, September 1999.
19. Anadigm Field Programmable Analog Arrays: www.anadigm.com

Multiple Sequence Alignment Using Reconfigurable Computing*

Carlos R. Erig Lima, Heitor S. Lopes, Maiko R. Moroz, and Ramon M. Menezes

Bioinformatics Laboratory, Federal University of Technology Paraná (UTFPR),
Av. 7 de setembro, 3165 80230-901, Curitiba (PR), Brazil
erig@utfpr.edu.br, hslopes@pesquisador.cnpq.br

Abstract. The alignment of multiple protein (or DNA) sequences is a current problem in Bioinformatics. ClustalW is the most popular heuristic algorithm for multiple sequence alignment. Pairwise alignment has exponential complexity and it is the most time-consuming part of ClustalW. This part of ClustalW was implemented using a reconfigurable logic hardware solution: Hardalign. The system was evaluated using data sets of different dimensionality, and compared with a pure software version running in a embedded processor, as well as running in a desktop computer. Results indicate that such implementation is capable of accelerating significantly part of the algorithm, and this is especially important for processing large protein data sets.

1 Introduction

Sequence alignment is a basic procedure frequently used in Bioinformatics. In the case of proteins, the alignment is done by a systematic comparison of the amino acids throughout the whole extension of the sequences. The main objective is to compute a score that indicates the similarity between proteins. Alignment can be done with a pair of sequences (pairwise alignment) or with several ones (multiple sequence alignment). In general, sequence alignment is the most important method for discovering and representing similarities between sequences.

From the computational point of view, sequence alignment, especially for multiple sequences, is a difficult task. In recent literature, many computational algorithms were proposed for this purpose. The main differences between them is the overall quality of the alignment and the computational effort required. Therefore, many heuristic methods have been proposed for multiple sequence alignment [3,6], since the exact algorithm is computationally unfeasible.

Recently, we have witnessed a pronounced growth of the hardware and software technologies for embedded systems, with many technological options arising every year. In particular, applications based on reconfigurable computing for sequence alignment can found in recent works [1,5].

The objective of this work is to implement a multiple sequence alignment algorithm in reconfigurable hardware, taking advantage of the possible parallelism of operations.

* This work was partially supported by the Brazilian National Research Council – CNPq, under research grants no. 305720/2004-0 and 506479/2004-8.

2 The ClustalW Algorithm

This work is based on ClustalW [6], a progressive alignment algorithm for multiple sequencesThe algorithm is divided into three basic steps, as follows:

The first step is the pairwise alignment, where all pairs of sequences are aligned using a dynamic programming (DP) algorithm for global alignment. It builds a $m \times n$ matrix (m and n are the length of the two sequences) and computes a score walking backwards in the matrix, looking for the minimal cost associated with substitutions, insertions and deletions. This step is repeated iteratively for all $n(n-1)/2$ pairs of sequences to be aligned, thus obtaining a distance matrix. The computed score is meant as the similarity degree between two sequences and is computed as evolutionary distances using the Kimura [2] model.

The second step is the computation of an unrouted guide tree (a kind of phylogenetic tree) based on the constructed distance matrix. This tree shows the evolution of the sequences, grouped in pairs of minimum distances (scores) using a neighbor-joining clustering algorithm. This tree is a profile of the order in which sequences should be aligned for maximal efficiency of the next step.

Finally, the final step is a progressive alignment of the sequences, traversing the distance tree in order of decreasing similarity: sequence-sequence, sequence-profile, and profile-profile alignment. This step is based on an improved DP algorithm [6]. Although it is very time-consuming, the final alignment is not the optimal alignment for the sequences under study.

3 Pairwise Alignment with Hardalign

Hardalign [4] is a dedicated processing system, working as a peripheral of the NIOS II Altera embedded processor, interconnected via the Avalon bus Hardalign contains an arrangement of N Matrix Line Processor Units (MLPUs - see below) and all logic for driving the arrangement. The critical part of the process that requires computational power is the computation of the DP matrix and, for this reason, it is run in parallel by the MLPUs. The software running in the NIOS II processor performs the first two steps and writes data to internal registers of Hardalign. Then, the driving logic is started and NIOS II reads sequentially the results. A SDRAM memory is used for storing vectors generated by the MLPUs. Next, the progressive alignment routine is started by software running in NIOS II. This algorithm uses the vectors previously generated and finds a path from the last cell of the matrix towards the first one.

Each MLPU computes one cell of the DP matrix, in a single clock cycle, and it is responsible for computing a line of the matrix. Once completed, a new line is started until the matrix is fully done. Since N cells are computed by clock cycle, the whole matrix is concluded in $L.C/N$ clock cycles, where L is the number of lines and C the number of columns of the matrix. The substitution matrix is encoded as a combinatorial circuit: every possible combination of the 20 amino acids gives an evolutionary distance value as result.

The MLPU entity can be replicated to compute simultaneously several lines in parallel. All the data for an entity can be read from other entities.The line amino acids are read in the beginning of the process, and the pipeline used for the first line of alignment should receive the column amino acids sequentially. Fig. 1(left) shows the MLPU entity in details, where an amino acid requested by a line is always requested by the pipeline that was above it in the previous clock cycle. The previous computed value is registered to be reused for the computation of the subsequent cells. The values computed in the two subsequent clock cycles are registered. They are necessary for the computation of the line below. The gap penalty is common to all of the entities and registered outside of the MLPU.

The pipeline entity is arranged to compute several lines in parallel. In Fig. 1(right) an arrangement with 3 MLPUs is shown, allowing the simultaneous computation to 3 lines of the DP matrix. The inputs of this entity are the gap penalty, the line amino acids and the column amino acids, in a sequential way.

Each MLPU entity has four outputs: two cell values, an amino acid value and a vector. Only the vector is necessary for the backtracking procedure later performed. The other outputs are important for the computation of the subsequent cells. In fact, only the vectors are defined as outputs, and the other signals are internal values of the pipeline. These vectors are defined like an output bus (vector bus) of $2 \times N$ bits, where N is the pipelines number. Initially, just the first MLPU contains valid values in their internal registers, and only some bits

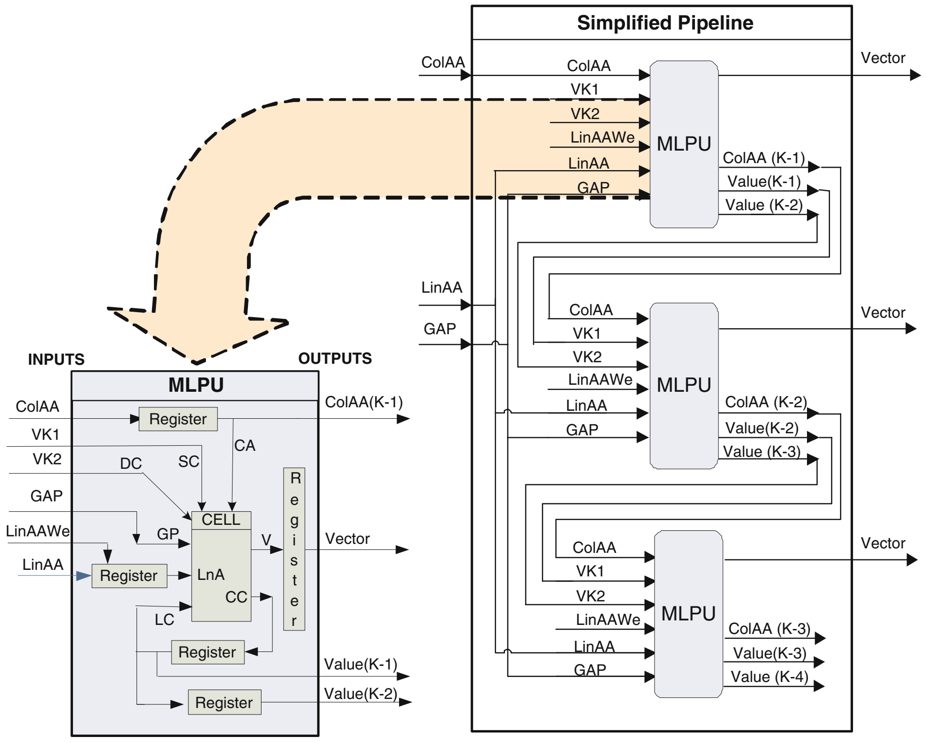


Fig. 1. Left: Block diagram of the MLPU. Right: Block diagram of the Pipeline.

of the vector bus are valid. After some cycles the own MLPU apply valid values to the subsequent MLPU.

4 Computational Experiments and Results

The hardware was synthesized in an Altera Stratix II EP2S60F672C5ES device (*http://www.altera.com*) running at 40 MHz. We used the SOPC builder (System on a Programmable Chip Builder) software to integrate the several modules with the NIOS II core. The physical synthesis and simulations was done using Quartus II environment.

The first experiment is a performance analysis for different sizes of DP matrix, comparing an implementation in PC and the hardware (with 8 MLPUs). The C language version running in the PC version runs the same algorithm implemented in hardware. We used a PC with Athlon XP 1600+ processor, 512Mb de RAM DDR 266 and Windows XP operational system. Three experiments were done, as follows.

Table 1 shows the computation time for pairs of 20- to 2000-amino acids-long sequences. The resolution of the PC timer is 1 millisecond, precluding to measure the processing time for the 20 x 20 matrix. In this table, we observed a 1:10 ratio, approximately. A further improvement of Hardalign is the use of a DMA (Direct Memory Access) controller module to improve the transfer rate between the SRAM and the Hardalign Pipeline Data was obtained by simulation and the comparison is shown in the same Table. An improvement of about 28 times in performance can be observed comparing the approaches with and without DMA.

The second experiment analyzes the performances of multiple sequence alignment with and without the use of Hardalign. For this experiment, we fixed the number of proteins to 5 and analyzed the performance for sequences of several amino acids lengths, as shown in Table 2. For each comparison (Software approach versus Hardware approach), three columns are presented: the time necessary for the pairwise alignment of all pairs of sequences (Pairwise), not including data transfer time; the time necessary for the whole progressive alignment using the guide tree (Progressive); and the total time for the multiple alignment (Total).

Regarding the time for pairwise alignment only, an improvement of of about 110 times in performance can be observed comparing the hardware approach and the software approach. On the other hand, the total processing time did not presented a significant improvement. In this case, besides the speed-up observed in pairwise computation, the advantage obtained is surpassed by the larger time demanded in the progressive alignment computation step. Another feature observed was the advantage of the hardware approach for larger amino acids sequences. In this case, the efficiency of Hardalign can be better explored.

In the third experiment, performance was tested for several sets of proteins to be aligned using the same protein size. Table 3 shows the computation time for sets of 10- to 100 proteins using a 200-amino acids-long sequence. The meaning of the columns in both comparisons are the same as in Table 2.

The demand for LUTs (Look Up Table) and internal registers grows up linearly as function of the number of MLPUs. For 8 MLPUs, 2189 LUTs and 589 registers

Table 1. Performance comparison of pairwise alignment (Hardalign versus PC)

Matrix size (lines x columns)	Hardalign (ms)	Hardalign with DMA (ms)	Athlon XP (ms)
20x20	0.074	0.001	-
100x100	0.949	0.031	10
200x200	3.548	0.125	30
500x500	22.123	0.781	200
1000x1000	87.618	3.125	831
2000x2000	350.207	12.500	3465

Table 2. Performance comparison between hardware and software for pairwise alignment (different lengths, same set). Processing time is given in seconds.

	NIOS II - Software			NIOS II - Hardalign		
#Amino Acids	Pairwise	Progressive	Total	Pairwise	Progressive	Total
30	0.03	0.56	0.63	0.01	0.56	0.61
90	0.21	3.46	3.73	0.02	3.41	3.50
200	1.04	14.67	15.85	0.05	14.89	15.08
300	2.98	30.80	34.26	0.09	31.24	31.80
901	35.20	263.94	301.31	0.46	269.74	272.29
1703	155.58	1060.01	1224.00	1.38	1009.75	1018.89

Table 3. Performance comparison between hardware and software for pairwise alignment (different sets, same lengths). Processing time is given in seconds.

	NIOS II - Software			NIOS II - Hardalign		
#Amino acids	Pairwise	Progressive	Total	Pairwise	Progressive	Total
10	13.467	35.710	50.006	0.308	35.297	36.307
20	30.121	84.643	116.650	2.097	74.270	78.368
50	215.595	228.133	455.059	27.530	194.432	235.377
100	776.435	494.265	1323.333	205.260	397.789	668.648

are used, taking into account only the pipeline logic, excluding the Avalon bus, NIOS II core, additional memories and pipeline drivers. The maximum frequency operation of pipeline is not affected by the number of MLPUs used.

5 Conclusions and Future Work

Multiple sequence alignment is an important problem in Bionformatics, but still open issue when dealing with large sequences. This work aims at exploring an alternative solution to this problem, using reconfigurable computing to substitute a computationally-intensive part of the ClustaW algorithm. The methodology for parallelizing a pairwise sequence alignment algorithm contributes to identify and circumvent the bottlenecks of a conventional software implementation.

The performance analysis reveals that the improvement by using the hardware approach achieves around 1:10 ratio of speed-up, compared with the conventional PC software processing. Using DMA, the ratio achieves around 1:280.

The performance of NIOS II software processing achieves a speed-up ratio of around 1:140 ratio, when compared with the NIOS II with Hardalign. This huge difference is due to features such as operational system, clock frequency and implementation languages used. Particularly, the comparison between different hardware platforms such as PC and FPGA-based kits is not completely fair, but necessary to emphasize significant performance differences.

Besides the significant minimization of computation time of the DP matrix, a relative low minimization in total computation time of ClustaW algorithm was observed: only 100% speed-up. There are some reasons for this shallow result: important time is spent to manage and transfer data, particularly in the progressive part of the algorithm and the fact that no parallel resources in the progressive part of the algorithm were used.

We observed that the growth in the number of proteins to be aligned decreases the speed-up of the system, and the growth of the length of the sequences increases speed-up. Therefore, the proposed system is more efficient for alignments with few sequences of large number of amino acids. Fortunately, this is the case of many real-world applications.

We believe that the proposed solution is an important contribution to both reconfigurable systems technology and Bioinformatics. Further research will focus on devising new parallelization levels for the multiple sequence alignment problem. ClustalW was developed to run using sequential processor. On the other hand, reconfigurable computing is a paradigm that strongly suggests the conception of new algorithms that explores multiple levels of parallelization.

References

1. Jacobi, R.P., Ayala-Rincon, M., Carvalho, L.G.A., et al.: Reconfigurable systems for sequence alignment and for general dynamic programming, *Genet Mol Res* **4** (2005) 543–552.
2. Kimura, M.: *The Neutral Theory of Molecular Evolution.* Cambridge University Press, New York (1983).
3. Lopes, H.S., Moritz, G.L.: A graph-based genetic algorithm for the multiple sequence alignment problem, *Lect Notes Artif Intel* **4029** (2006) 420–429.
4. Moritz, G.L., Jory, C., Lopes, H.S., Erig Lima, C.R.: Implementation of a parallel algorithm for pairwise alignment using reconfigurable computing. *Proc. IEEE Int. Conf. on Reconfigurable Computing and FPGAs*, (2006) pp. 99-105.
5. Oliver, T., Schmidt, B., Nathan, D., et al.: Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW. *Bioinformatics* **21** (2005) 3431–3432.
6. Thompson, J.D., Higgins, D.G., et al.: CLUSTALW: improving the sensitivity of progres-sive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice, *Nucleic Acids Res* **22** (1994) 4673–4680.

Simulation of the Dynamic Behavior of One-Dimensional Cellular Automata Using Reconfigurable Computing

Wagner R. Weinert, César Benitez, Heitor S. Lopes*, and Carlos R. Erig Lima

Bioinformatics Laboratory, Federal University of Technology Paraná (UTFPR),
Av. 7 de setembro, 3165 80230-901, Curitiba (PR), Brazil
hslopes@pesquisador.cnpq.br, erig@utfpr.edu.br

Abstract. This paper presents the implementation of an environment for the evolution of one-dimensional cellular automata using a reconfigurable logic device. This configware is aimed at evaluating the dynamic behavior of automata rules, generated by a computer system. The performance of the configware system was compared with an equivalent software-based approach running in a desktop computer. Results strongly suggest that such implementation is useful for research purposes and that the reconfigurable logic approach is fast and efficient.

1 Introduction

Cellular automata (CA) are discrete distributed systems formed by simple and identical elements. The dynamic behavior of a CA is represented by its evolution along time and this evolution depends on a transition rule [8]. Finding a transition rule capable of modelling a given behavior is a rather difficult task, since the search space promptly becomes computationally intractable. Many works in the literature proposed the use of evolutionary computation techniques for the task of finding suitable transition rule that leads a CA to display a desired behavior (see, for instance, [3,5,6]). Usually, the basic approach in these cited works is to induce rules. Rules are evaluated according to a fitness function, regarding its utility to simulate the desired behavior. Iteratively, the best performed rules are selected and modified by using genetic operators. The evolutionary process finishes after n generations when a given rule achieves the a satisfactory behavior. Frequently, the computation of the fitness function is the most time-consuming task in an evolutionary computation algorithm. Recently, we have witnessed a pronounced growth of the hardware and software technologies for embedded systems, with many technological options arising every year. In particular, applications based on CA can found in recent works [1,2,7]. In our approach to use evolutionary computation for inducing transition rules for cellular automata, the fitness function is computed by simulating the dynamic behavior of the automaton. This is accomplished evolving many automata using a single

* This work was partially supported by the Brazilian National Research Council – CNPq, under research grant no. 305720/2004-0 to H.S. Lopes.

transition rule. Such automata are randomly generated and evolved during a fixed number of time steps. The computation of the fitness function is based on the comparison of the obtained behavior and the desired one. The discovery of transition rules for CAs requests the computation of a fitness function that is computationally-intensive. To reduce this computational cost, we propose a methodology for evolving CAs using reconfigurable computing. The system was developed in VHDL (VHSIC Hardware Description Language) and implemented in a FPGA (Field Programmable Gate Array) device.

The case-study addressed in this work is the classic problem of a CA to perform a density classification task [1,3,5,6]. A typical evolutionary computation algorithm for finding a transition rule for such CA will need to try some 1500 different rules until finding a good one. For each rule, its appropriateness is evaluated by applying it to a randomly generated automaton and evolving it for a given number of time steps. Due to the stochastic nature of the algorithm and the randomness of the initial condition of the automaton, around 10,000 different automata are evolved for each rule, and results are statistically evaluated.

In this work it is not aimed to propose any evolutionary computation technique for inducing CAs, but evaluating the usefulness of reconfigurable computing as a hardware accelerator for evolving CAs, in comparison with a software-based approach.

2 Cellular Automata and the Density Classification Task

A Cellular Automata is defined by its cellular space and by its transition rule. The cellular space is represented by a lattice of N cells connected according a boundary condition in a d-dimensional space. The transition rule gives the next state for the cell, considering the configuration of its current neighborhood. At each time step, all cells in the lattice update their current state, according to the transition rule (representing the dynamic nature of the system)[8]. A formal definition of CA is given by [4]:

$$\begin{array}{ll}
\Sigma : & \text{set of possible states for each cell;} \\
k : & \text{cardinality of } \Sigma; \\
i : & \text{index of each cell;} \\
S_i^t : & \text{state of a given cell } i \text{ at time } t \ (S_i^t \in \Sigma); \\
\eta_i^t : & \text{neighborhood of cell } i; \\
\Phi(\eta_i) : & \text{transition rule that leads cell } i \text{ to the next state } (S_i^{t+1}) \text{ as function of } \eta_i^t.
\end{array} \tag{1}$$

Two parameters are necessary to deal with the neighborhood concept: (m) that represents the size of the neighborhood, and (r) representing its radius. Usually, parameter m is given as function of r, in the form: $m = 2r + 1$. Since a CA is represented by a linear structure, the leftmost cell (i_0) and the rightmost cell (i_{n-1}) of a one-dimensional automaton do not have left and right neighbors, respectively. Therefore, a bounding condition is necessary, such that the leftmost cell is connected with the rightmost cell, and then the transition rule $\Phi(\eta_i)$ can be applied to the whole lattice. Considering that the number of cells of a given neighborhood is $2r + 1$, the number of different neighborhood that can be generated using a given rule is $k^{(2r+1)}$. Also, the number of possible transition rules that can be generated for a CA is $k^{k^{(2r+1)}}$.

The dynamic behavior of an one-dimensional CA generated by the application of a transition rule over the automaton for n time steps is usually illustrated by a spatiotemporal diagram. In such diagram, the configuration of states in a lattice is plot as a function of time.

The number of possible rules for a given automaton is $k^{k^{(2r+1)}}$. The larger k and r, the larger the set of rules applicable to a given CA. In most cases it is not computationally feasible to evaluate the whole set of rules so as to find a given one that explains the dynamic behavior of a system.

The density classification task, also known as majority problem, is a classical problem in CA theory [8]. The objective is to find a transition rule such that, when applied for M time steps to a random initial configuration, will lead all cells either to state 0 or state 1, depending on the density of the initial configuration. The parameter ρ is defined as a threshold for the density classification task [4], in such a way that ρ_0 represents the density of cells in state 1 in the initial configuration, and ρ_c is the same density in a given configuration (for $\forall t > 0$).

The density classification task can be modelled in several ways. Here we use the approach proposed by [3]. In this model, all cells can have binary states ($k = 2$), the lattice is composed by an odd number of cells ($N = 149$), and the neighborhood radius of $r = 3$. The number of possible transition rules for this CA is 2^{128}.

3 Methodology

To implement the system, we used the Altera Quartus II development system, version 5.1, and a Cyclone EP1C6Q240C8 FPGA device. The software running in the desktop PC was developed in C++ language.

The software running in the desktop PC generates a transition rule (as part of other evolutionary strategy for inducing rules), encodes and sends it to the FPGA through a parallel interface. Inside the FPGA, several hardware blocks (see Figure 1 perform the evaluation of the transition rule, and send back both, the result of such evaluation and the processing time.

Using a serial to parallel converter, a 128-bits long transition rule is assembled (from the parallel interface block) and stored in the register block.

The command decoder block decodes the received commands and executes control actions in the other FPGA blocks: the pseudo-random number generator block, the CA evolver block, the chronometer block and to the accuracy calculator block. In the current version, five different commands can be decoded and executed:

- Internal reset.
- Restart MLS pseudo-random number generator.
- Clear current transition rule in register.

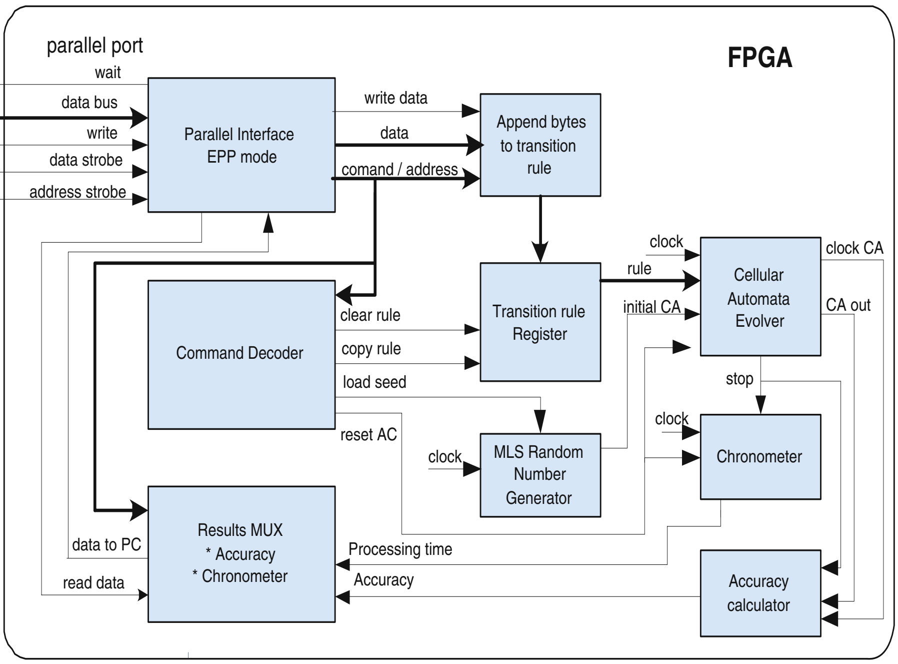


Fig. 1. Block diagram of the proposed system for evolving CAs

– Latch transition rule.
– Start CA evolution.

The CA evolver block receives a previously registered transition rule and an initial configuration for the automaton, generated by the pseudo-random number generator, and evolves the CA for a given number of iterations (in our case, 200). Inside this block there are two Finite State Machines (FSM). The first one controls the number of iterations of the CA. The second FSM controls the execution of 10,000 runs, with a random initial configuration.

The accuracy calculator block verifies if the final configuration is the expected one, considering the density of the initial configuration. If positive, the score of the rule under evaluation is incremented. At the end of all runs, this block retains the total number of hits, which, divided by the number of runs and taken as a percentage, is the accuracy rate.

To generate random initial configurations, a Maximum Length Sequence (MLS) pseudo-random number generator is used [1]. MLS is an n-stage linear shift-register that can generate binary periodical sequences of maximal period length of $L = 2^n - 1$ These sequences are referred to as maximal-length sequences (MLS), and n is said to be the degree of the sequences. In our work, we used $n = 16$, thus generating 16-bits sequences. The random seed is loaded into the

[1] Available in *http://www.ph.ed.ac.uk/˜jonathan/thesis/node83.html*

shift-register by a command. Ten parallel shift-registers were implemented, and out from these 160 bits, 149 are used for the initial configuration of the CA.

The chronometer block counts and register the total elapsed time for generate, evolve and evaluate the behavior of 10,000 CAs. The multiplex block selects output data between the computed accuracy rate and the processing time, to be sent out to the desktop computer.

4 Results

The evaluation of the proposed system consists of five experiments. For each experiment, 10,000 one-dimensional CAs were randomly generated and evolved for 200 iterations. For these experiments, we used a single rule, proposed by Juillé et al. [3], named "Coevolution(1)". To date, this rule is supposed to be the best known rule for the density classification task. Results take into account the average accuracy (that evaluates how good the rule performs) and the average processing time (that represents the computational cost). In order to compare such results with other implementation, a similar software was developed in C++ programming language, compiled without any optimization flag and run in a desktop computer under Microsoft Windows XP operational system. Exactly the same rule and parameters were used for both systems, and Table 1 shows the results obtained. The FPGA was run at 33,33 Mhz and the real clock of the desktop PC was 1800 MHz. The comparison between different hardware platforms such as a desktop PC and FPGA-based system is not completely fair, but necessary to emphasize the dramatic performance difference.

Table 1. Comparison PC versus hardware-based approach

Device	Accuracy	Processing Time
FPGA CYCLONE EP1C6Q24068	82.28	58.660ms
PC AMD Athlon XP 2400+, 512MB	79.33	3h:9m:26.6s

5 Conclusions and Future Work

In this work we described a reconfigurable computing system for evolving cellular automata. This system is to be used in conjunction with a software application, running in a desktop PC, to study the behavior of transition rules. The reconfigurable hardware is able to communicate with any software application, since a standard parallel interface was implemented.

For this work, the average accuracy of the system has small importance, since the main focus is the processing speed. The difference of 2.95% between the accuracy rates of the hardware-based system and the software-based system is due to different random number generators, responsible for the generation of the initial configuration of the CAs.

The processing time needed to evolve the CAs in the PC is extremely high, when compared with the time needed by the hardware approach. This is due to the fact that, in the PC, processing is sequential, and in the FPGA parallelization was largely explored. Even considering that the FPGA was running at 33,33 MHz and the PC was 54 times faster, the processing time of the later was more than 5 orders of magnitude higher than the former. This makes evident the great advantages of reconfigurable computing, strongly suggesting its adequacy for this kind of task.

Future work will focus on the implementation of some evolutionary computation technique to search transition rules for one-dimensional CA, using the reconfigurable computing module as an external accelerator. Also, other levels of parallelization will be sought, thus improving even more the efficiency of the system.

References

1. Corsonello, P., Spezzano, G., Staino, G., et al.: Efficient implementation of cellular algorithms on reconfigurable hardware. In: *Proc. of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing* (2002) 211–218.
2. Halbach, M., Holffmann, R.: Implementing cellular automata in FPGA logic. In: P*roceedings of the 18th International Parallel and Distributed Processing Symposium* (2004) 258–262.
3. Juillé, H., Pollack, J.B.: Coevolving the ideal trainer: application to the discovery of cellular automata rules. In: Koza, J.R et al. (eds.), *Genetic Programming 1996: Proc. 3rd Annual Conference*, (1998) 519–527.
4. Mitchell, M.: Computation in cellular automata: a selected review. In: Gramms, T., ed., *Nonstandard Computation.* VCH Verlagsgesellschaft, Weinheim (1996).
5. Morales, F.J., Crutchfield, J.P., Mitchell, M.: Evolving two-dimensional cellular automata to perform density classification: a report on work in progress. *Parallel Computing* **27** (2001) 571–585.
6. Oliveira, G.M.B., Bortot, J.C., Oliveira, P.P.B.: Multiobjective evolutionary search for one-dimensional cellular automata in the density classification task. In: *Proc. 8th Int. Conf. on Artificial Life*, (2002) 202–206.
7. Shackleford, B., Tanaka, M., Carter, R.J., et al.: FPGA implementation of neighborhood-of-four cellular automata random number generators. In: *Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, (2002) 106–112.
8. Wolfram, S.: *Cellular Automata and Complexity.* Westview Press, Boulder (1994).

Author Index

Amagasaki, Motoki 142
Armstrong, Nilton B. (Jr.) 314

Benitez, César 385
Berekovic, Mladen 1, 26
Bertels, Koen 130
Beuchat, Jean-Luc 214
Bispo, João 179
Bollman, Dorothy 238
Bouwens, Frank 1

Caffarena, Gabriel 201
Cardoso, João M.P. 179
Carreras, Carlos 201
Carrillo, Richard 259, 337
Chikhi, Rayan 247
Cho, Kyoung-Rok 39
Choi, Chang-Seok 283
Cichaczewski, Ederson 326
Cruz-Cortés, Nareli 226

Derrien, Steven 247
Díaz, Javier 259, 337, 365
Diniz, Pedro C. 97
Dong, Yazhuo 110
Dou, Yong 110, 155, 293
Drimer, Saar 73

Ferlin, Edson P. 326
Ferrer, Edgar 238

Galuzzi, Carlo 130
Gauffriau, Adrien 167
Gaydadjiev, Georgi 1
Geraghty, Dermot 350
Girau, Bernard 167

Hervé, Nicolas 191
Hiramoto, Shinya 343
Hur, Jae Young 49, 61
Hwang, Dong-Guk 85
Hwang, Seung-Gon 283

Iida, Masahiro 142

Jevtic, Ruzica 201

Kanazawa, Kenji 358
Kanstein, Andreas 1, 26
Kim, Hyun Dong 283
Kim, Tae Seon 283
Knittel, Guenter 271

Lee, Chong-Ho 283
Lee, Hanho 283
Lee, Jae-Jin 85
Lee, Je-Hoon 39
Lee, Seung-Sook 39
Lee, Yong-Min 283
Lima, Carlos R. Erig 314, 326, 379, 385
Lopes, Heitor S. 314, 326, 379, 385
Luk, Wayne 302

Madsen, Jan 26
Maher, John 373
Mamagkakis, Stelios 91
Maruyama, Tsutomu 358
Matsuyama, Kazunori 142
McElroy, Ciarán 350
McGettrick, Séamas 350
McGinley, Brian 373
Ménard, Daniel 191
Menezes, Ramon M. 379
Min, Chul Hong 283
Miyoshi, Takanori 214
Morales-Luna, Guillermo 226
Moreno, Oscar 238
Morgan, Fearghal 373
Moroz, Maiko R. 379
Mota, Sonia 259, 337
Mou, Shengmei 293

Nakanishi, Masaki 343
Nakashima, Yasuhiko 343
Nakayama, Hideaki 142
Naous, Rawan 14
Noumsi, Auguste 247

Ohke, Yoshiharu 358
Okamoto, Eiji 214
Oyama, Yoshihito 214

Park, Jae-Hyun 283
Park, Joonseok 97

Quinton, Patrice 247

Rocke, Patrick 373
Rodriguez, Rafael 337
Rodríguez-Henríquez, Francisco 226
Ros, Eduardo 259, 337, 365

Saghir, Mazen A.R. 14
Saqib, Nazar A. 226
Scholz, Rainer 122
Sentieys, Olivier 191
Siozios, Kostas 91
Song, Gi-Yong 85
Soudris, Dimitrios 91
Sourdis, Ioannis 179
Stefanov, Todor 61
Stumpf, Michael 302
Sueyoshi, Toshinori 142

Thanailakis, Antonios 91
Thomas, David B. 302
Tomasi, Matteo 365
Torres-Huitzil, Cesar 167

Vassiliadis, Stamatis 49, 61, 130, 179

Weinert, Wagner R. 385
Wong, Stephan 49, 61
Wu, Guiming 155
Wu, Kehuai 26

Xu, Jinhui 155

Yamaguchi, Ryoichi 142
Yamaguchi, Yoshiki 358
Yamashita, Shigeru 343
Yang, Xiaodong 293

Zhou, Jie 110