Proceedings of

4th HiPEAC Workshop on

Reconfigurable Computing

January 23, 2010

Pisa, Italy

Organized in the framework of the
2010 International Conference on
High Performance Embedded Architectures & Compilers
HiPEAC 2010

MICROSOFT 1007

Editors: Roberto Giorgi and Stephan Wong
Publisher: TU Delft / EWI Computer Engineering Laboratory
ISBN: 978-90-72298-05-8

Acknowledgment: This work is partially funded by the European Commission through the project ERA (GA 249059).

Preface

This proceedings contain the papers presented at the 4th HiPEAC Workshop on Reconfigurable Computing held on January 23, 2010, in Pisa, Italy.

The HiPEAC Workshop on Reconfigurable Computing provides a forum for researchers active in domains within the reconfigurable computing area. Its main focus is on reconfigurable architectures, tools that facilitate such architectures, and applications tailored for reconfigurable platforms. The workshop brings together both hardware designers and software developers that make extensive use of reconfigurable computing. Moreover, it aims at enabling scientific discussions regarding future challenging issues.

Additionally, an invited contribution from industry offers another opportunity for interacting on compiler tools for FPGA.

In this fourth year of the workshop we attracted 19 submissions.

We would like to thank all of the authors for submitting their papers. The program committee selected 9 papers to be presented at the workshop; that is acceptance rate 47%. We also gratefully acknowledge the program committee and the additional reviewers who contributed their time and expertise for their dedication and diligence. They provided a total of 60 review with an average of 3 reviews per submitted paper. The investment of their time and insight is very much appreciated.

The members of the Organizing and Program committee, as well as the additional reviewers are listed in the following page.

We hope that the attendees enjoyed the HiPEAC Workshop on Reconfigurable Computing 2010 in all aspects.

January 2010

Roberto Giorgi and Stephan Wong
Program Co-Chairs
4th HiPEAC Workshop on
Reconfigurable Computing 2010

Organization

Organizing Commitee

General co-Chairs:	Leonel Sousa	(Technical University of Lisbon, Portugal)
	Pedro Trancoso	(University of Cyprus, Cyprus)
Program co-Chairs:	Roberto Giorgi	(University of Siena, Italy)
	Stephan Wong	(TU Delft, The Netherlands)
Publicity Chair:	Ioannis Sourdis	(TU Delft, The Netherlands)
Web Chair:	Fakhar Anjam	(TU Delft, The Netherlands)

Program Committee

Sandro Bartolini	University of Siena, Italy
Mladen Berekovic	Technische Universität Braunschweig, Germany
Luigi Carro	UFRGS, Brazil
Marcelo Cintra	University of Edinburgh, UK
Giuseppe Desoli	ST Microelectronics, France
Skevos Evripidou	Universty of Cyprus, Cyprus
Paolo Gai	Evidence Srl., Italy
Georgi N. Gaydadjiev	TU Delft, The Netherlands
Michael Hübner	University of Karlsruhe, Germany
Wolfgang Karl	University of Karlsruhe, Germany
Krishna Kavi	University of North Texas, TX, USA
Volodymyr Kindratenko	University of Illinois at Urbana-Champaign, USA
Costas Kyriacou	Frederick University, Cyprus
Stavrou Kyriakos	Intel Labs Barcelona, Spain
Bilha Mendelson	IBM, Israel
Aleksandar Milenkovic	University of Alabama in Huntsville, AL, USA
Nacho Navaro	UPC Barcelona, Spain
Dionisios Pnevmatikatos	Technical University of Crete, Greece
Marco D. Santambrogio	Politecnico di Milano, Italy
Cristina Silvano	Politecnico di Milano, Italy
Dimitrios Soudris	NTUA, Greece
Ioannis Sourdis	TU Delft, The Netherlands
Leonel Sousa	Technical University of Lisbon, Portugal
Dirk Stroobandt	Ghent University, Belgium
David Thomas	Imperial College, UK
Pedro Trancoso	University of Cyprus, Cyprus
Theo Ungerer	University of Augsburg, Germany
Tom VanCourt	Altera, USA
Ayal Zaks	IBM, Israel

Additional Reviewers

Faruk Bagci
Isaac Gelado
M. Gomathisankaran
Matthias Hanke
Sebastian Schlingmann
Thomas Schuster
Muhammad Shafiq
Yana Yankova

Workshop Program

09:15 - 09:30 Opening

09:30 - 10:30 Keynote

Reconfigurable Computing becoming main stream? 3
Walid Najjar

10:30 - 11:00 Coffee break

11:00 - 12:00 SESSION 1: FAULT-TOLERANCE

A Dynamic Reconfigurable Super-VLIW Architecture for a Faul Tolerant Nanoscale Design
Ricardo Ferreira, Cristoferson Bueno, Marcone Laure, Monica Pereira, and Luigi Carro 7

Fault-Free: A Framework for Supporting Fault Tolerance in FPGAs
Kostas Siozios, Dimitrios Soudris, and Dionisios Pnevmatikatos 17

12:00 - 12:30 SESSION 2A: ARCHITECTURAL ISSUES

Decreasing the Impact of the Context Memory aon Reconfigurable Architectures
Thiago Berticelli Lo, Antonio Carlos S. Beck, Mateus Beck Rutzig, and Luigi Carro 29

12:30 - 13:45 Lunch

13:45 - 14:00 INDUSTRY SESSION

Short presentation: Synthesizing Efficient Architectures from ANSI-C Specifications: The HCE Compiler 41
P. Palazzari

14:00 - 14:30 SESSION 2B: ARCHITECTURAL ISSUES

Register File Design in Automatically Generated ASIPs
Roza Ghamari and Arda Yurdakul 45

14:30 - 15:30 SESSION 3: PROGRAMMING AND SCHEDULING

Reconfiguration Aware Task Scheduling for Multi-FPGA Systems
Francesco Redaelli, Marco D. Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik 57

Specifying Run-time Reconfiguration in Processor Arrays using High-level Language
Zain-ul-Abdin and Bertil Svensson 67

15:30 - 16:00 Coffee break

16:00 - 17:30 SESSION 4: APPLICATIONS

Speaker Identification Classification on FPGA
Phak Len Eh Kan, Tim Allen, and Steven F. Quiqley 79
High Efficiency FPGA Regular Expression Pattern Matching
Atta Badii and Adedayo O. Adetoye 89
Row-interleaved Streaming Data Flow Implementaiton of Sparse Matrix Vector Multiplication in FPGA
Branimir Dickov, Miquel Pericas, Nacho Navarro, and Eduard Ayguade 99

VI

KEYNOTE

2

Reconfigurable Computing becoming main stream?

Walid Najjar

University of California Riverside, CA, USA

Abstract. Reconfigurable computing is not anew concept but can potentially become a new phenomenon. Technological obstacles are limiting the performance of single CPUs. It is becoming apparent that hardware code acceleration will soon become the norm rather than the exception. While FPGA-based hardware accelerators have repeatedly been demonstrated as a viable option for faster computing with very large speed-ups, their programmability remains a major barrier to their wider acceptance by application code developers. These platforms are typically programmed in a low level hardware description language, a skill not common among application developers and a process that is often tedious and error-prone. Programming FPGAs from high-level languages would provide easier integration with software systems as well as open up hardware accelerators to a wider spectrum of application developers. This presentation addresses the challenges and potentials of high-level language programming of FPGAs as hardware accelerators.

4

SESSION 1:
FAULT-TOLERANCE

6

A Dynamic Reconfigurable Super-VLIW Architecture for a Fault Tolerant Nanoscale Design

Ricardo Ferreira[1], Cristoferson Bueno[1], Marcone Laure[1], Monica Pereira[2], and Luigi Carro[2].

[1] DPI, Universidade Federal de Viçosa, 36570 000 Viçosa, Brazil

[2] Instituto de Informática – UFRGS, 91501-970, Porto Alegre, Brazil

ricardo@ufv.br, carro@inf.ufrgs.br

Abstract. A new scenario emerges due to nanotechnologies that will enable very high integration at the limits or even beyond silicon. However, the fault rate, which is predicted to range from 1% up to 20% of all devices, could compromise the future of nanotechnologies. This work proposes a fault tolerant reconfigurable architecture of future technologies, named Super-VLIW. The architecture consists of a reconfigurable unit tightly coupled to a MIPS processor. The reconfigurable unit is composed of a binary translation unit, a configuration cache, a reconfigurable coarse-grained array of heterogeneous functional units and an interconnection network. Reconfiguration is done at run-time, by translating the binary code, and no recompilation is needed. The interconnection network is based on a set of multistage networks. These networks provide a fault-tolerant communication between any pair of functional unit and from/to the MIPS register file. This work proposes a mechanism to dynamically allocate the available units to ensure parallel execution of basic operations, performing the placement and routing on a single step, which allows the correct interconnection of units even at huge fault rates. Moreover, the proposed architecture could scale to the future nanotechnologies even under a fault rate of 15%.

1 Introduction

The scaling of CMOS technology brings a new scenario concerning reliability of devices. At nanoscale basis the wires and connections become more fragile and consequently more susceptible to break. Furthermore, due to the inherent variability and the imprecision of fabrication processes at this scale, a large number of manufacturing defects is predicted [1]. While the fault rates are well below 0.1% on current technologies, this number could increase to 20% at nanoscale basis [2]. At these fault rates, traditional static approaches such as triple modular redundancy (TMR) or even N-modular redundancy (N-MR) can be compromised, due to the high probability that the added redundancy also fails.

In addition, there is a need for flexibility after fabrication to achieve high performance at low power levels. Coarse-Grained Reconfigurable Arrays (CGRA)

could be an alternative, and several reconfigurable architectures have been proposed in the past 20 years [3-7]. However, a common characteristic shared among these architectures that make their usage prohibitive is the need of special compilers and tools to select the part of the applications and modify the source code or binary code to be executed on the reconfigurable array. This totally breaks the software compatibility principle that users have become used to.

The interconnection model is another important issue that must be addressed in CGRA designs. The architectures found in the literature are organized in three main topologies: the unidimensional model [4], the stripe model [5,6] and the mesh model [7]. The models have in common the need of design time tools and compilers to perform the placement and routing.

In this context, this work proposes a reconfigurable architecture called Super-VLIW. In this architecture several parallel/sequential computations are dynamically allocated over a large set of functional units, even in presence of faults. This work differs from a traditional VLIW processor where all long instructions are built at compile time. To dynamically configure the architecture, a binary translation mechanism is used, so the binary software compatibility is ensured. The binary translation has been widely used by companies to encapsulate RISC instructions inside the x86's processors in the past 20 years, and an unquestionable advantage is the software compatibility.

This work also proposes the use of a set of Multistage Interconnection Networks (MINs) to send/receive values between the units and the processor register file. The model is logically similar to the unidimensional model, but could be physically implemented on two dimensional organizations as the stripe and the mesh models. In addition, the placement and routing is done at runtime. Therefore, no special tools or compilers are needed.

This work also provides further contributions on fault tolerant homogeneous MIN without extra resources due to a dynamic placement and routing step. In addition, we will show that a MIN could support a 15% fault rate allowing the scaling to reduced feature sizes such as nanoscale. Moreover, as it will be shown, a MIN can be flexible to offer a fault tolerant interconnection for a CGRA to dynamically speedup a MIPS processor.

This paper is organized as follows. Section 2 presents some related work. Section 3 gives some background concerning the multistage networks. Section 4 presents the proposed work with details about how the Super-VLIW architecture works. Section 5 demonstrates the experimental results. Finally, Section 6 draws conclusions and future works.

2 Related Work

Interconnections are one of the main issues on a parallel computer system. A MIN can have a compact VLSI layout, as shown for a butterfly MIN [8]. Recently, a multilevel FPGA architecture (MFPGA) at circuit-switching mode has been proposed

in [9], using a butterfly fat-tree multistage and tree interconnections. Experimental results show that the MFPGA has better area efficiency when compared to the mesh FPGA architectures. However, this approach is implemented at bit level (LUT-level), and it depends on a placement and routing developed at compile time. Our approach differs from the previous one in three aspects. First, we use coarse-grained units instead of LUTs, reducing the time overhead and the memory space required to configure the architecture. Second, interconnections are configured at runtime, thus no extra tools or special compilers are required. Finally in the interconnection model proposed in this work, the faults are taken into account and we will show that the proposed MINs provides fault tolerance allowing execution even at a 15% fault rate.

Although the MIN fault tolerance capability has been widely studied [10] during the 80's, this subject is still an important topic of research [11,12]. One approach consists in adding extra stages [10,11]. One extra stage provides fault tolerance for one switch failure in a MIN with Log N stages [10], and at least K extra stages will be needed to provide multiple fault tolerance to K switches failures [11]. Recently, a fault-tolerant routing for Fat Trees has been presented in [12]. The proposed routing has been implemented inside the switches and it is based on exclusion intervals over the destination address to forward the packets through the network.

In addition to an efficient interconnection network, the reconfigurable capability is another important issue that can ensure flexibility, low power, high performance and fault tolerance. In most approaches the interconnections are simplified by using regular and local topologies like meshes. However the price to pay is the complexity of the placement and routing steps, which should be done at compile time like in ADRES [9]. Recently, an approach has been presented [6], which is based on a Benes MIN that allows connections among any unit from one row to any unit in the previous/next rows. However, the applications have been manually mapped to evaluate this architecture.

Our approach differs from the previous ones in several aspects. First, the placement and routing is done at runtime in a single step, even in presence of faults. No compilation is needed and the software compatibility is sustained. Each unit has a global ID, independent of row and column. A heterogeneous set of units is taken into account, and the architecture is flexible enough to allow a dynamic placement and routing. Moreover, all units can send/receive data to/from all units by using a global low cost interconnection model based on a set of MINs. In addition, a 2D layout of our architecture is feasible, as shown by the previous work on MIN layouts [8,9].

3 Multistage

A multistage interconnection network (MIN) consists of a set of switch columns or stages, where each stage is connected to the previous and to the next one. The MIN shown in Fig 1(a) is an Omega Network with Log N stages [23], where the interstage connection is a perfect-shuffle. There are three main classes of MINs: blocking, rearrangeable and non-blocking. A MIN is blocking when at least one input/output

permutation assignment cannot be performed. These networks can be unique-path blocking or multiple-path blocking. An unique-path blocking MIN has only one path to connect a given input to a given output. A multiple-path blocking has more than one path to connect a given input to a given output, such as an Omega MIN plus extra stages shown in Fig. 1(b).

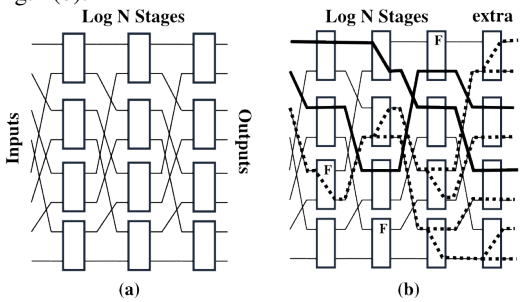


Fig. 1. Multistage Interconnection Networks: (a) Omega Network; (b) One Extra Stage Omega

A rearrangeable MIN can perform any input/output permutation. However, the exiting paths may have to be rearranged by reprogramming the internal switches. Finally, a MIN is nonblocking if it can perform any input/output permutation, independent of the order in which the switches are programmed. However, even for rearrangeable MIN, most routing algorithms are static, and suppose to know a priori all input and output connection pairs [14]. Since in a dynamic context, the connection pairs should be processing in order, the previous algorithms cannot be applied.

Most fault tolerance approaches aim to increase the stage number, and/or the switch radix, and/or the interstage links. As an example, let us consider the fault tolerant MINs shown in Fig. 2. Let us suppose that one switch fault can occur in the extra stage cube network [10] shown in Fig. 2(a). In this case, three stages are added: one extra stage is added to duplicate the paths and two mux stages are added to bypass the first or the last stage. Observing the paths between the input/output pair 0 to 1, one can notice that there are two routing paths. If there is a fault at a single switch at the stage 2 or 1, as each path passes through two switches, the "healthy" switch can be used. However in case of a single fault at the first or the last stage, there is only one switch for both paths, therefore either the first stage or the last stage should be bypassed by using the mux stage. In case of multiple switch faults, some input/output pairs will be disconnected. This fault tolerant MIN costs the double of the single Omega MIN from Fig. 1(a), and it only provides a single fault tolerance.

Other approaches are based on high radix switches. Fig. 2(b) displays a 2-dilated baseline MIN which has been proposed in [15]. There are n multiple paths for each input/output pair, as shown in Fig. 2(b) for the input 0 and the output 0. However, the multiple paths use a set of non-distinct switches. Considering n=8, even for 8 multiple paths, only 3 faults would be enough to disconnect an input/output pair, as shown in Fig. 2(b), due to the multiple path share switches.

The focus of our work is to provide fault tolerance by using a runtime placement and routing. We propose the use of a multipath blocking network, where the number

of stages ranges from log N to 2 log N. Our approach differs from previous work regarding fault tolerance and routing. First, most works in fault-tolerant MIN [18] ensures that any input can reach any output even in the presence of faults. On the other hand, in our approach one can send the input to a different output performing an alternative path without affecting the correct operation of the architecture. Moreover, while in previous approaches the placement is done before the routing, and therefore the MIN should be fault tolerant on any input/output pair, we propose to compute the placement and the routing at the same time and during runtime, avoiding the need of pre-determined alternative paths.

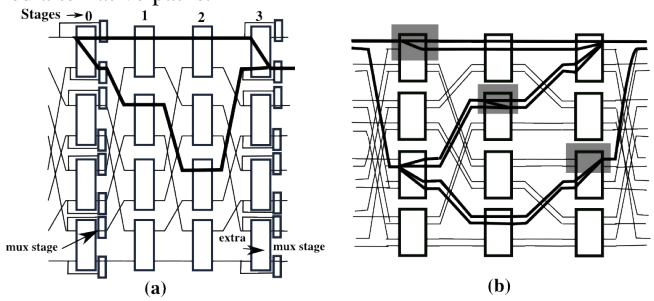


Fig. 2. Fault Tolerant MIN: (a) One Extra stage Omega (b) Dilated-Baseline MIN

In our approach, for a given input we will find which outputs are routable by doing a broadcast as shown in Fig 1(b). The faults are indicated by the letter F, the allocated paths are shown in bold line, and the broadcast from input 2 is shown as a dotted line, and it can reach all outputs. Then, the first free output will be selected by using a priority decoder. In this approach, even under a high fault rate, the MIN will be able to connect some input/output pairs.

4 Super-VLIW Architecture

The reconfigurable architecture proposed in this work consists of a MIPS processor, a reconfigurable array (RA) of functional units (FU), a context cache and a binary translation unit (BT), as shown in Fig. 3(a). A dynamically reconfigurable architecture tightly coupled to a MIPS processor was previously proposed in [16], however the interconnection model presented in these works is based in buses and multiplexers. The architecture proposed here provides an area reduction by replacing the interconnection model with multistage networks. Furthermore, the fault tolerance approach proposed in this work also allows the use of the architecture in future technologies with high fault rates.

The MIPS processor has a five stage pipeline and a register file. An instruction will be firstly executed on the MIPS, and in parallel the BT mechanism will build an instruction block. This step is named detection phase. A block consists of a super very long instruction word (super VLIW), and each block is stored on the context cache.

The cache is indexed by the PC of the first instruction of each block. When the instruction is executed again, if its PC is found in the context cache, the reconfigurable unit array will be configured and the entire block will be executed. This step is named execution phase.

The reconfiguration includes data copied from the MIPS register file to the input context, the interconnection setting and the unit configuration. Only the register values used by the current block will be copied. When the execution is finished, the MIPS register file is updated by the output context, and then either the next instruction block will be executed or a new instruction block will be detected.

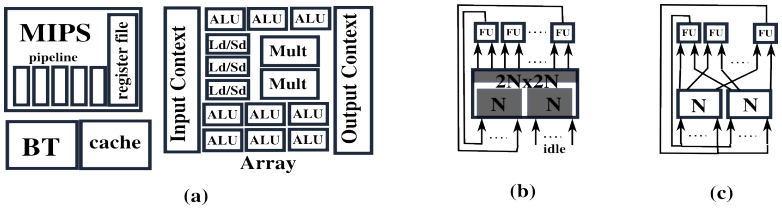


Fig. 3. Super-Vliw Architecture (a) MIPS plus Reconfigurable Unit. (b) Using one MIN to connect to all FUs (c) Using one MIN per FU.

To provide an efficient interconnection a MIN must have the same number of inputs and outputs. However, since each FU has two inputs and only one output, the Super-VLIW interconnection network will be unbalanced. To solve this problem, instead of using only one MIN to interconnect the FUs (as shown in Fig. 3(b)), where half of the inputs will be unused, we propose to use a separated network for each FU input without increasing the network cost, as shown in Fig. 3(c).

At first glance, four NxN MINs seems to be expensive. However, considering N the number of FUs, the number of FU inputs is 2N, and then one needs at least one 2N x 2N MIN. Although four NxN MINs are twice bigger than just one 2Nx2N MIN, the advantage of the proposed solution is the fault tolerance capability. Moreover, previous approaches [10,11,14] on fault tolerant MIN increase the MIN cost by at least a factor of 2. For instance the MIN shown in Fig. 2(a) doubles the cost and it offers only one single switch fault. The dilated MIN in Fig.2(b) is six times bigger than a single MIN, due to the 4x4 switch instead a 2x2.

During the detection phase, the binary translation algorithm will verify the instruction operands. Considering a three operand instructions Op3 = Op1 op Op2, where Op1 and Op2 are the input operands, Op3 is the destination operand, and op is the operation performed by the FU. When the binary translation selects an instruction to be executed on the reconfigurable array, the first step consists in sending a broadcast in parallel through MIN Op1 and MIN Op2. At the same time, it sends a broadcast at the output context MIN from the Op3 output. Each broadcast will find the reachable FUs of each operand. Finally, a bitwise AND of the three broadcast vectors is done, and a priority decoder will select the first reachable FU.

To show how the faults in the MIN's elements can be tolerated, first let us consider a switch to be internal if there are no direct connections to an input or an output. When a single fault affects a link, there are three fault cases. First, a link fault on an internal switch will affect multiple paths. However, if the MIN has extra stages, other

paths will be available to connect some input/outputs. In the second case, a link fault occurs on an output switch. Therefore, this output will be unreachable. However, as the units are allocated dynamically, these affected units will behave as unavailable ones, and the allocation algorithm will avoid these units and search for the first routable unit available. Finally, when a link at first stage switch fails, only one of two inputs will be able to be used, and if both links fail, the two associate inputs will be disconnected. In this case, a register renaming approach is used to overcome the input faults.

It is important to highlight that this work considers only permanent faults generated during the fabrication process. Thus, the information about the faulty units is generated before the binary translator starts, using classical testing techniques. In addition, no modification on the binary translation algorithm is needed, and the solution to provide fault tolerance is transparent.

5 EXPERIMENTAL RESULTS

To evaluate the performance degradation presented by the proposed architecture under different fault rates we consider as a reference value the performance of the MIPS 5 stage processor, and all speedup results are relative to MIPS. Therefore, if an architecture generates a speedup factor of 2 this means that the evaluating architecture is twice faster than standalone MIPS processor.

As case study we analyzed three VLIW processors: simple 8 units VLIW, a dynamic 8 units VLIW (called VLIW8) and a dynamic fault tolerant Super-VLIW. Both dynamic architectures are tightly coupled to a MIPS processor by using a binary translation mechanism. They differ in number of functional units and in the interconnection networks. The MiBench benchmark was used to evaluate the speedup of all approaches.

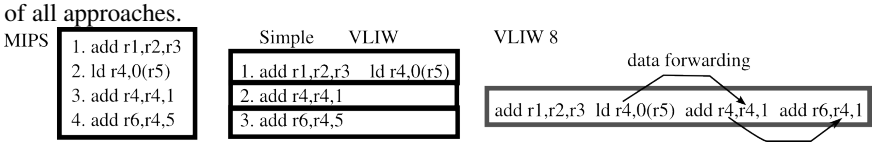


Fig. 4. A four instruction sequence mapped on simple VLIW and on VLIW8

The simple VLIW can only execute up to 8 operations (4 ALU, 3 Load/Store, 1 Multiplication) without any data dependence and it works as the traditional VLIW machine. The dynamic VLIW8 processor has also eight FUs, each FU has two full multiplexers and can connect to any input context or FU using a non-blocking network. The instruction block is built dynamically, and each block can have up to 4 ALUs, 3 Load/Store units and 1 multiplier. In the best case, all FUs will work in parallel, if there are no data dependences and in the worst case they will work sequentially. For instance, Fig. 4 shows a MIPS sequence which will be mapped on three simple VLIW word due to data dependence, and where only one dynamic VLIW8 word is enough. The multiplexers are used to forward data. Finally, the Super-VLIW has 128 FUs (90 ALU, 34 Load/Store and 4 Multipliers) that are

connected through the MIN networks as demonstrated in Fig. 3(c). Fig. 5 presents the speedup achieved by each solution when executing the applications from MiBench suite.

On an ideal scenario without faults, as shown in Fig. 5, the Super-VLIW can accelerate on average 2.84 times the MiBench benchmarks in comparison to VLIW8 and the simple VLIW, whose average speedups are 1.7 and 1.17 respectively. Furthermore, if we consider only seven applications that show the higher individual speedups, the Super-VLIW speedup increases to 3.4 times against 1.64 times for the VLIW8. Therefore, if the applications are more dataflow, the Super-VLIW could be twice faster than VLIW8. The large number of units is needed to capture large block on the dataflow application.

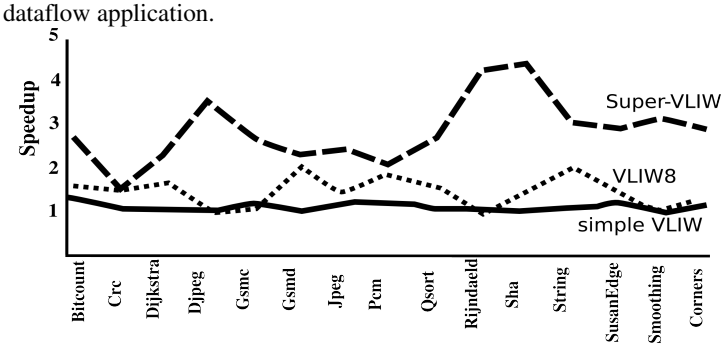


Fig. 5. Speedups of simple VLIW, VLIW8 and VLIW128 on the execution of MiBench benchmarks.

Although the VLIW8 presented a mean speedup of 1.7 times in the execution of MiBench Benchmarks, the main problem of this solution is that it does not support faults. If a fault happens in any part of the multiplexer or the functional unit, it will invalidate the whole element. Therefore, in a more realistic scenario where faults must be considered the best solution is the Super-VLIW, since it ensures software execution even with several faulty FUs and interconnection units.

To evaluate performance degradation of Super-VLIW when faults affect the resources, the same benchmarks were executed considering different cases. Fig. 6 presents an analysis of the speedup as a function of the fault rate. The graph presents the most significant speedups and five different fault rates were used.

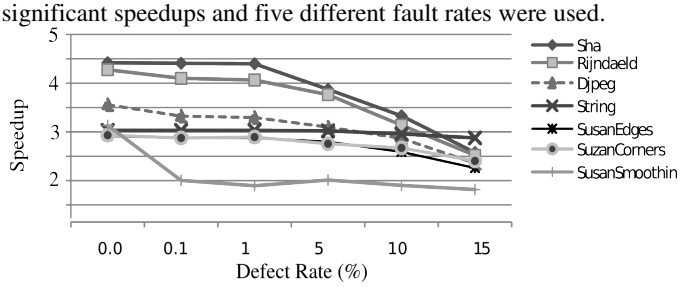


Fig. 6. The Speedup degradation in presence of faults for the most significant MiBench benchmarks on the Super-VLIW.

As can be observed from Fig. 6, the Super-VLIW not only continues working even under a 15% fault rate, but also accelerates execution comparing to the standalone processor. According to the results, in the worst case, the average speedup is around 2.24 for 15% when executing all benchmarks from MiBench suite. Therefore, the Super-VLIW can be scaled to new technologies even in presence of faults.

Table 1 presents an analysis of the area as a function of technology scaling. Since the MIPS and the binary translation unit are critical to the correct function of the system, we have chosen not to scale them, as using larger feature sizes one could increase the reliability of the system. Hence, the size of both units at 90nm technology will be 0.4 mm². Furthermore, the area of VLIW8 will be 0.6 mm², which is 3 times bigger than a single MIPS and the performance is only 1.7 times better.

On the other hand, the Super-VLIW can scale to new technologies that present higher fault rates. Since the functional units are less tolerant than the network, we can manufacture them in a larger feature size than the network. For example, the FU array can scale to 0.16mm² at 32nm, and the MIN size can scale to 0.03mm² at 11nm. The MIN will degrade the performance at 11nm due to the high fault rate. However as demonstrated in Fig. 5, under 15% fault rate the average speedup is 2.24, and the area is around 2.4 times bigger than the single MIPS (0.53 for the Super-VLIW and 0.22 for the MIPS standalone). Therefore, the speedup is proportional to the extra area. If the fault rate is smaller, the speedup can be improved (up to 4.5 for some benchmarks as shown in Fig. 5 and Fig. 6). In these performance results we did not take into account the fact that as technology scales, the cycle time might shrink, and this means that one could expect further speedups by scaling the Super-VLIW.

Table 1. Estimated chip size according to the technology scaling (mm²).

Technology	90nm	32nm	22nm	18nm	11nm
MIPS	**0.22**	0.22	0.22	0.22	0.22
Binary Translation	0.12	0.12	0.12	0.12	0.12
128 FU	1.3	0.16	0.16	0.16	0.16
MIN	1.8	0.23	0.11	0.07	0.03
Total Area: SuperVLIW	3.5	0.73	0.61	0.57	**0.53**

6 CONCLUSIONS

This paper presented three main contributions. First it presented a reconfigurable Super-VLIW architecture with an online reconfiguration mechanism that also ensures software compatibility. The Super-VLIW uses a set of multistage interconnection network as interconnection model that was used to reduce interconnection costs. The second contribution corresponds to a dynamic placement and routing for the multistage interconnection network. Using the placement and routing as an atomic operation, one can dynamically find a path to a free functional unit even in the presence of a high number of faults. Furthermore, most works support at most a number of faults equal to the number of network stages O(log N). Our approach supports up to O(N) switch faults. Finally, a fault tolerance approach to both

reconfigurable architecture and interconnection model has been presented. The approach consists in avoiding the faulty functional units and interconnections.

We evaluated the system performance by using the MiBench suite. The analysis showed that our approach is still working even when the number of fault is 30 times bigger than the number of stages for a 128x128 MIN. The MIN network still works at 15% of fault rates. Therefore, we conclude that it is possible to scale the system to new technologies that present high fault rates, consequently the area overhead is significantly reduced by scaling the MIN. The entire system can be estimated to be 2.4 times bigger than a single MIPS and its performance is also improved by a factor of 2.4.

References

1. Eshaghian-Wilner,M. BioInspired and Nanoscale Integrated Computing. John Wiley & Sons (2009)
2. DeHon, A. and Naeimi, H. Seven Strategies for Tolerating Highly Defective Fabrication. IEEE Design and Test 22(4), 306-315(2005)
3. Hartenstein, R. . A decade of reconfigurable computing: a visionary retrospective. In Proceedings of the Conference on Design, Automation and Test in Europe, 642-649 (2001)
4. Compton, K. and Hauck, S. 2008. Automatic design of reconfigurable domain-specific flexible cores. IEEE Trans. Very Large Scale Integr. Syst. 16(5), 493-503 (2008)
5. Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R.. PipeRench: A Reconfigurable Architecture and Compiler. IEEE Computer 33 (4) , (2000).
6. Kazuya Tanigawa, et al.: Exploring Compact Design on High Throughput Coarse Grained Reconfigurable Architectures. Field Programmable Logic and Applications 543-546, (2008)
7. Mei, B., Vernalde, S., Verkest, D., De Man, H., and Lauwereins, R. 2003. Exploiting Loop-Level Parallelism on Coarse-Grained Reconfigurable Architectures Using Modulo Scheduling. In Proceedings of the Conf. on Design, Automation and Test in Europe - (2003)
8. DeHon, A. 2000. Compact, Multilayer layout for butterfly fat-tree. In Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures (2000).
9. Zied, M., Hayder, M., Emna, A., and Habib, M. 2008. Efficient tree topology for FPGA interconnect network. In 18th ACM Great Lakes Symposium on VLSI . pp. 321-326 (2008).
10. Adams, G. B., Agrawal, D. P., and Siegel, H. J. . A survey and comparison of fault-tolerant multistage interconnection networks. In IEEE interconnection Networks For High-Performance Parallel Computers, pp. 654-667 (1994)
11. Fan, C. C. and Bruck, J. . Tolerating Multiple Faults in Multistage Interconnection Networks with Minimal Extra Stages. IEEE Trans. Comput. 49,(9) pp. 998-1004.(2000),
12. FT2EI: A Dynamic Fault-Tolerant Routing Methodology for Fat Trees with Exclusion Intervals / Gómez Requena, C / Gómez Requena, M E / López Rodríguez, P J / Duato Marín, J F, IEEE Transactions on Parallel and Distributed System, 2009; 20 (6)
13. D. H. Lawrie, "Access and alignment of data in an array processor,"IEEE Trans. Comput., vol. C-24, pp. 1145-1155, (1975)
14. T.-Y. Feng, S.-W. Seo, "A New Routing Algorithm for a Class of Rearrangeable Networks," IEEE Trans. Computers, vol. 43, no. 11, pp. 1270-1280 (1992)
15. N. Kamiura, T. Kodera, N. Matsui, "Fault tolerant multistage interconnection networks with widely dispersed paths," Asian Test Symposium, pp. 423 (2000)
16. Beck A., Rutzig M., Gaydadjiev G., Carro, L.: Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In IEEE/ACM DATE: pp.1208-1213 (2008)

Fault-Free: A Framework for Supporting Fault Tolerance in FPGAs

Kostas Siozios[1], Dimitrios Soudris[1] and Dionisios Pnevmatikatos[2]

[1] School of Electrical & Computer Engineering, National Technical University of Athens, Greece
{ksiop, dsoudris}@microlab.ntua.gr
[2] Electronic and Computer Engineering Department, Technical University of Crete, Greece
pnevmati@mhl.tuc.gr

Abstract. In this paper we propose a novel methodology for supporting application mapping onto FPGAs with fault tolerance even if this feature is not supported by the target platform. For the purposes of this paper we incorporate three techniques for error correction. The introduced fault tolerance can be implemented either as a hardware modification, or through annotating the application's HDL. Also, we show that the existing approaches for fault tolerance result to hardware wastage, since there is no demand for applied them uniformly over the whole FPGA. Experimental results show the efficiency of the proposed framework in terms of error correction, with acceptable penalties in device area and Energy×Delay Product (EDP) due to the redundant hardware resources.

Keywords: FPGA; fault tolerant; architecture; exploration; CAD tool.

1 Introduction

SRAM-based Field-Programmable Gate Arrays (FPGAs) are two-dimensional arrays of Configurable Logic Blocks (CLBs) and programmable interconnect resources, surrounded by programmable input/output pads on the periphery. Even though programmability feature of these devices makes them suitable for widely application implementation, there are a number of design issues that have to be taken into consideration during application mapping. Among others, reliability issues have become worse as devices have evolved. For instance, as the transistor geometry and core voltages decreased, while the numbers of transistors per chip and the switching frequency increase, the target architectures become more susceptible to incurring faults (i.e., flipped bit or a transient within a combinatorial logic path). Nowadays, the reliability problem becomes even more important due to the industry trend for increasing the logic density [1]. Consequently, fault tolerance need to be though as a critical design issue, even for real-life applications.

The last ten years many discussions were done about the design of reliable architectures able to overcome from faults occurred either during the fabrication process or the execution time. More specifically, the term fault tolerant corresponds to a design which is able to continue operation, possibly at a reduced level, rather than failing completely, when some part of the system fails [1, 2, 3, 4, 5, 6, 8, 9, 12, 13, 14, 15]. These solutions include fabrication process-based techniques (i.e. epitaxial CMOS processes) [11], design-based techniques (i.e. hardware replicas, time redundancy, error detection coding, self-checker techniques) [4], mitigation techniques (i.e. multiple redundancy with voting, error detection and correction coding) [5], and recovery techniques (reconfiguration scrubbing, partial configuration, rerouting design) [14].

Even though fault tolerance is a well known technique, up to now it was mostly studied for ASIC designs. However FPGAs poses new constraints (i.e. higher power density, more logic and

This work was partially supported by the MOSART project (Mapping Optimization for Scalable multi-core ARchiTecture) funded by the EU (IST-215244).

interconnection resources, etc), while the existing fault models are not necessarily applicable. To make matters worse, faults in FPGAs can alter the design, not just user data. In addition to that, the designs mapped onto FPGAs utilize only a subset of the fabricated resources, and hence only a subset of the occurred faults may result to faulty operation. Consequently, FPGA-specific mitigation techniques are required, that can provide a reasonable balance among the desired fault prevention, the performance degradation, the power/energy consumption and the area overhead due to the additional hardware resources.

Up to now there are two approaches for preventing faults occurring on FPGAs. The first of them deals with the design of new hardware elements which are fault tolerant enabled [2, 4, 12, 15]. These resources can either replace existing hardware blocks in FPGAs, or new architectures can be designed to improve robustness. On the other hand, it is possible to use an existing FPGA device and provide the fault tolerance at higher level with CAD tools [2, 3, 6, 8, 13, 14].

Both of these approaches have advantages and disadvantages, which need to be carefully concerned. More specifically, the first approach results to a more complex architecture design, while the derived FPGA provides a static (i.e. defined at design time) fault tolerant mechanism. On the other hand, the second approach potentially is able to combine the required dependability level, offered by fault tolerant architectures, with the low cost of commodity devices. However, this scenario imposes that the designer is responsible for protecting his/her own design.

In [12] a fault tolerant interconnection structure is discussed. The faults in interconnection network are corrected by spare routing channels which are not used during place and route (P&R). A similar work is discussed in [13], where a defect map is taken as input to P&R tool and then application's functionalities are not placed in the faulty blocks. In another approach [14], EDA tools take as input a generic defect map (which may be different from the real defect map of the chip) and generate a P&R according to this. A work that deals with a yield enhancement scheme based on the usage of spare interconnect resources in each routing channel in order to tolerate functional faults, is discussed in [15]. The only known commercial approach for supporting fault tolerance in FPGAs can be found in [8]. This implementation replicates each of the logic blocks that form application. The initial, as well as the two replica blocks work in parallel, while the output is derived by comparing their outputs with a majority voting.

In this work we focus on providing a framework for exploring the efficiency of alternative fault tolerance techniques in terms of numerous design parameters. The introduced fault tolerance can be implemented either on hardware (i.e. block redesign) or software (i.e. HDL annotation) level. In addition to that, the proposed methodology enables conventional FPGA devices to have fault tolerant features, making it suitable for existing commercial architectures. Also, our methodology can provide the desired tradeoff between the required reliability level and the area overhead. Based on experimental results, we achieve significant error correction (similar to existing fault tolerance approaches), but we employ significant fewer hardware resources, leading among others to area, delay and energy savings.

More specifically, the main contributions of this work are summarized, as follows:

- We introduce a novel methodology for supporting fault detection and correction on FPGA devices.
- We identify sensitive sub-circuits (where faults are most possible to occur) and we apply the proposed fault tolerant technique onlt at these points rather than inserting redundancy in the whole device.
- We develop a new tool that automates the introduction of redundancy into certain portions of an HDL design.
- We validate the results with a new platform simulator based on MEANDER framework [10].

The rest paper is formed as follows: In section 2 we discuss the paper motivation, while section 3 describes the fault tolerant FPGA architectures. The proposed fault tolerance methodology is explained in section 4. Experimental results that show the efficiency of the derived fault tolerance architectures are shown in section 5, while conclusions are summarized in section 6.

2 Motivation

The first step in order to build a reliable system is to identify possible regions with increased failure probability. These regions mostly include hardware resources that implement application's functionalities with increased switching activity [17]. Since switching activity is an application property which does not depend either to the target platform or the employed mapping tools, it is possible to identify critical functionalities during the profiling task (step 1 of the proposed methodology – we will describe it in section 4). However, the employed toolset introduce some constraints regarding the spatial distribution of regions with excessive high (or low) values of switching activity, and consequently with increased (or decreased) failure probability [18].

In order to show a case study about the variation of switching activity, Figure 1 plots the sensitive FPGA regions, by calculating the failure probability of hardware resources for an array composed by 64×64 slices, which implements the *frisc* application [7]. More specifically, Figure 1(a) depicts the variation of switching activity for the application mapping without fault tolerance. In this figure, different colors denote different failure probabilities, while as closer to red color a slice is the higher probability to occur a fault.

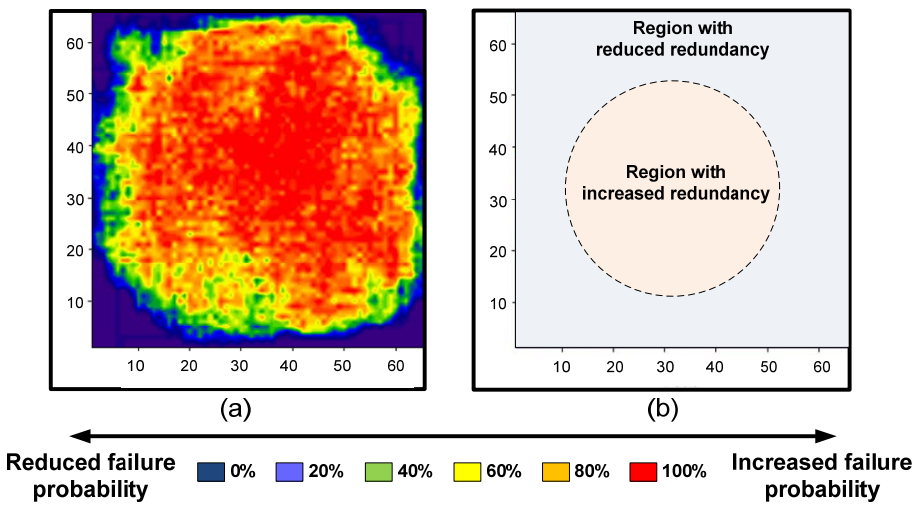


Figure 1. Spatial distribution of failure probability for *frisc* benchmark: (a) without fault tolerance, and (b) with TMR redundancy

Based on this map (Figure 1(a)), it is evident that the switching activity (and hence the failure probability) is not constant across the FPGA, since it varies between two arbitrary points (x_1,y_1) and (x_2,y_2) of device. From this distribution it is feasible to determine regions on the device with excessive high values of switching activity (regions of importance), where we have to pay effort in order to increase the fault tolerance. Consequently, the challenge, with which a designer is faced up, is to choose only the actually needed redundancy level, considering the associated spatial information from the distribution graph. Figure 1(b) depicts a candidate classification of FPGA's hardware resources into two regions, with and without increased demands for redundancy.

A second important conclusion is drawn from Figure 1: although the majority of existing fault tolerant techniques exhibits a homogeneous and regular structure, the actually critical for failure resources provide a *non-homogeneous* and *irregular* picture. Consequently, careful analysis of the points of failure must be performed, while the target system implementation needs to combine regions with different density of fault tolerance.

Based on exhaustive exploration [18], we found that the region with increased failure probability and hence increased demand for redundancy is placed almost in the middle of the device, while it occupies about 40% of the FPGA's area. On the other hand, the rest FPGA (i.e. 60%) exhibits limited failure probability, and hence it is not upmost important to apply redundancy.

3 Proposed Fault Tolerant FPGA Architectures

Our target is a generic recent FPGA device (shown in Figure 2) similar to the Xilinx Virtex architecture, consisting of an array of CLBs, memories and DSP cores. Communication among hardware blocks is provided by a hierarchical interconnection network of fast and versatile routing resources. We assume that CLBs are formed by a number of Basic Logic Elements (BLEs), each of which is composed of a set of programmable Look-Up Tables (LUT), multiplexers, and flip-flops. The output of BLE can connect either to an input pin of any other BLE, or to the logic block output. Apart from this architecture, our proposed methodology can also support almost any other existing FPGA device.

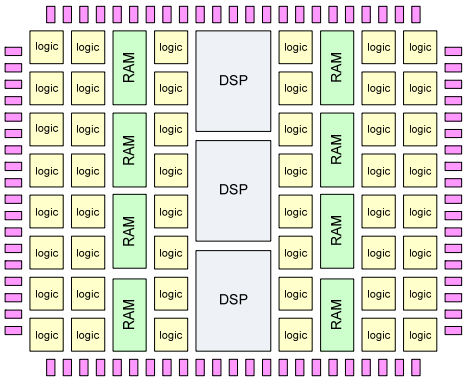


Figure 2. The target FPGA architecture

In order to support the fault tolerance we use a replication technique based on voting, which can be implemented either in hardware (as a dedicated circuit) or in glue logic (through HDL annotation). In contrast to the hardware implementation, the software-based fault repair technique is particular important in the field of FPGAs, as it provides a trade-off between the desired accuracy of detecting and correcting faults with the extra hardware overhead.

The reconfigurable platform is encoded as an M-of-N system, consisting of N hardware blocks where at least M ($M \leq N$) of them are required for proper operation (the platform fails if less than M of the blocks are functional). In our study we assume that different blocks fail with statistically independent order and if a block fails then it remains non-functional (the faults are not temporal). If $R(t)$ is the probability of an individual block to be still operational at time t, then the reliability of a M-of-N architecture corresponds to the probability that at least M blocks are functional at time t. By supposing that f_p denotes the probability that entire architecture suffers by a common failure, the system reliability can be calculated as follows:

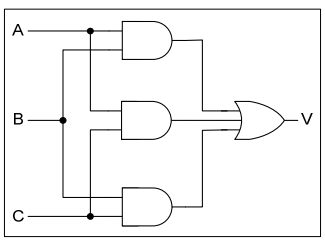
$$R_{M_of_N}(t) = \left[(1 - f_p) \sum_{k=M}^{N} \left\{ \left(\binom{N}{k} R^i(t) \right) [1 - R(t)]^{N-k} \right\} \right] \quad (1)$$

where $\binom{N}{k} = \frac{N!}{(N-k)!k!}$. In case we assume that a fault affects the whole architecture (i.e., f_p=0), then the FPGA's reliability is calculated based on Equation 2. Additionally, whenever $R(t)$<0.5 the hardware redundancy actually become a disadvantage, as compared to a platform without redundancy.

$$R_{M_of_N}(t) = \sum_{k=M}^{N} \left\{ \left(\binom{N}{k} R^i(t) \right) [1 - R(t)]^{N-k} \right\} \quad (2)$$

The functionality of voter is to receive a number of $N = \{i_1, i_2, \ldots, i_N\}$ inputs from an M-of-N architecture and to generate a representative output. For instance, a typical voter does a bit-by-bit comparison, and then outputs the majority of the N inputs. Figure 3 shows a majority voter instantiation with three inputs. Such a voter can block an upset effect through the logic at the final output. Consequently, they can be placed in the end of the combinational and sequential logic blocks, creating barriers for the upset effects.

During the redundancy approach, a critical design issue is to determine the optimal partition size for the replicated logic that have to be voted, in order to reduce the probability upset faults in the routing to affect two distinct redundant parts that are voted by the same voter. A small size block partition requires a large number of majority voters that may be too costly in terms of area and performance. On the other hand, placing only voters at the last output increases the probability of an upset in the routing infrastructure to affect two (or more) distinct redundant logic blocks overcoming the fault correction mechanism.



A	B	C	V
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 3. Majority voter schematic and the truth table

Next paragraph describes some of the supported voting schemes from our proposed framework. We have to mention that apart from these, any other voting scheme can also be employed, if it is appropriately described in Boolean form. Note that, our approach can diagnose with accuracy errors in glue logic. In order to correct faults in wire segments, it is possible to add spare routing resources similar to [12].

3.1 R-fold modular redundancy

The R-fold modular redundancy (RMR), shown in Figure 4, entails $R = \{3, 5, 7, \ldots, C\}$ replica logic blocks working in parallel, while the output results through a majority vote. Such a voter discards errors coming from the faulty replica(s) and selects as block's output one of the partial outputs from the rest (faulty-free) replicas. This technique can effectively mask faults if: (*i*) only less than $((R+1)/2)$ replicas are faulty (either on combinational and sequential logic), but the fault presents in different register locations, and (*ii*) the voter if fault free.

A typical instantiation of this approach is Triple Modular Redundancy (TMR), which guarantees effective functionality as far as up to one replica blocks exhibits a single fault. This instantiation is also used at the only known commercial tool for supporting fault tolerance in FPGAs [8].

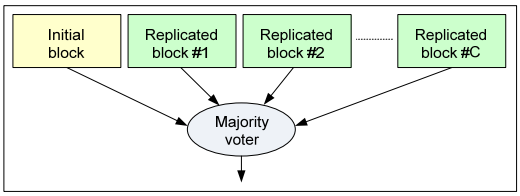


Figure 4. Supported R-fold modular redundancy (RMR) [2]

3.2 Cascaded R-fold modular redundancy

Despite the simplicity of the RMR method, it is ineffective when an upset occurs in the voter or there is an accumulation of upsets. In these cases, an additional mechanism is necessary to correct the faults, before the next upset happens. Cascaded *R*-fold modular redundancy (CMR) is similar to the previous studied architecture, but along with the *R* replicas of the logic blocks, the majority voter is also replicated. Multiple instantiations of this technique are feasible, which mainly differ in the number of voters, as well as the connectivity among them. Figure 5 depicts the *R*-folded modular instantiation employed in this paper.

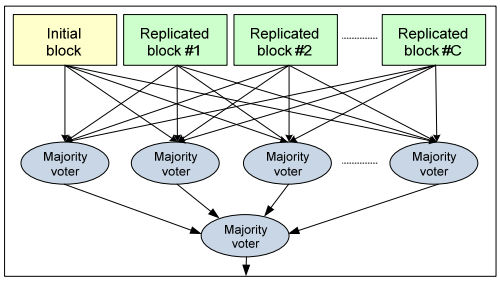


Figure 5. Supported cascaded redundancy (CMR) [3]

3.1 Time redundancy (TR)

Figure 6 depicts another fault tolerant implementation supported by the proposed framework. More specifically, by sampling the signal inserted to the voter more than once with shifted clocks, it is possible to eliminate errors with a pulse width less than the clock cycle. As compared to the rest implementations discussed previously, this one exhibits lower area overheads, but it imposes performance degradation, as well as the requirement for different clock phases.

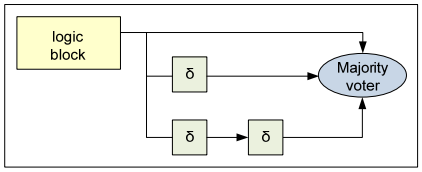


Figure 6. Supported time redundancy (TR) [4]

4 The Proposed Fault-Free Framework

In this section we describe the proposed framework for exploring the efficiency of different fault tolerance techniques. The goal of this methodology is to ensure fault masking at the same time with acceptable area, delay and energy overheads. The methodology consists of three steps (S1, S2 and S3), as they are depicted in Figure 7.

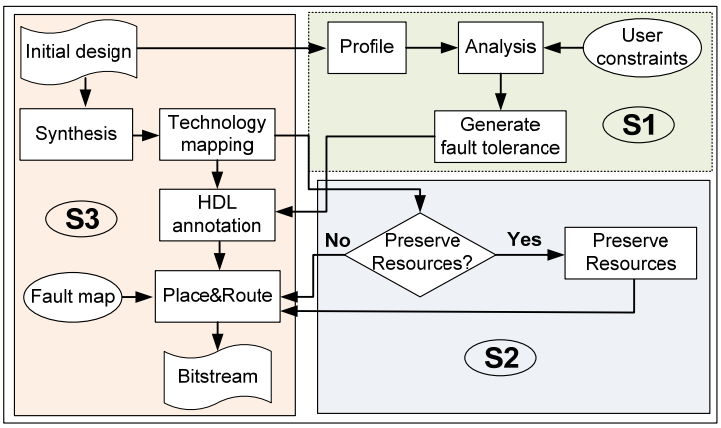


Figure 7. Static fault management

Starting from an HDL system description, during the first step (S1) we perform application profiling and analysis in order to determine sensitive functionalities with increased failure probability. Typical parameter that results at higher failure probability is the increased switching activity, and hence the on-chip temperature, due to the electromigration effect [17]. In section 2 we have already studied the spatial variation of switching activity. Consequently, application's functionalities with increased switching activity are most candidates for applying redundancy, since they have higher failure probabilities. This information, in conjunction to the user constraints (i.e. desirable reliability, area overhead, etc), defines the employed fault tolerance model. Such a model describes among others the selected fault tolerant technique (i.e. RMR, CMR, TR, etc), as well as the amount of redundancy applied on the target platform.

The second step (S2) of the methodology provides a precautionary mechanism for minimizing (or eliminating) reliability issues from defects occurred during execution time due to permanent faults. In order to apply it, a number of hardware resources need to be reserved. More specifically, this step can provide solutions to the following reliability problems: (*i*) to discourage functionalities to be placed or routed to already known faulty resources, and (*ii*) to reserve resources on critical device regions and to invoke them whenever a fault occurs, in order to find a new P&R which occupies only functional components. More specifically, the hardware resources of target device are categorized in three groups, namely: (*i*) unutilized (they are not employed for application mapping), (*ii*) utilized (used for application mapping and are fault free), and (*iii*) faulty (found not to operate properly). Due to the increased complexity of the second task, a co-processor attached to the FPGA is required, in order to find the P&R and re-program faulty resources.

The third step (S3) of the proposed methodology deals with application mapping. The application's HDL description is synthesized and then technology mapped. The retrieved netlist is appropriately annotated based on the desired fault tolerance, as it was retrieved from S1. This task is involved only whenever the redundancy is applied in software (i.e. HDL) level. Otherwise, i.e. when the FPGA incorporate a hardwired fault tolerant mechanism, we bypass this tasks and proceed to application P&R. Constraints posed during the P&R comes from the reservation of hardware resources (step S2), as well as the faulty map (similar to the Figure 1(a)).

In order to provide the necessary software support for our proposed methodology, we employ an open source tool flow [10], while the fault tolerance is applied as a new CAD tool, named *Fault-Free*. However, the methodology is completely transparent to the employed CAD tool chain and can be applied as an intermediate step between synthesis and place and route (P&R) tools at existing design flows.

5 Experimental Results

The first step in calculating reliability is the selection of fault models. There are two major sources of logic faults in FPGAs: cosmic radiation and manufacturing/operating imperfections. Since the size of radiation particles is small as compared to the bitstream size of logic blocks, we employ a fault model that follows uniform distribution of non-correlated failures. In addition to that, the device scaling makes them more sensitive to soft faults [9]. The second class of faults is related to manufacturing and operating imperfections. Even though these defects are not important during the testing after fabrication, they become exposed after a long period of operation. In order to model this type of faults, we use the Stapper fault model [19], which is based on gamma distribution.

The efficiency for the alternative fault tolerant implementations is shown in Figure 8. More specifically, here we study the percentage of repaired over the injected faults for different redundant techniques. In order to make this experiment, for each of the curves we calculate the bitstream file for *frisc* [7] application. This size ranges from *B*=1107Kbits (TR approach) up to *B*=1560 Kbits (CMR approach), while each of these files remains constant along the corresponding curve. Then, we inject randomly a number of faulty bits in the configuration data. The amount of faulty injected bits, mention with *F*, ranges from 5% up to 25% of the configuration's file size. We assume that the faults are randomly distributed between logic and interconnection resources.

The horizontal axis of this figure corresponds to the percentage of injected faulty bits over the size of configuration data, while the vertical one shows the percentage of faulty bits over the total injected bits that remain faulty after applying the recovery mechanism. In other words, the vertical axis plots the efficiency to repair faults for the alternative redundant implementations.

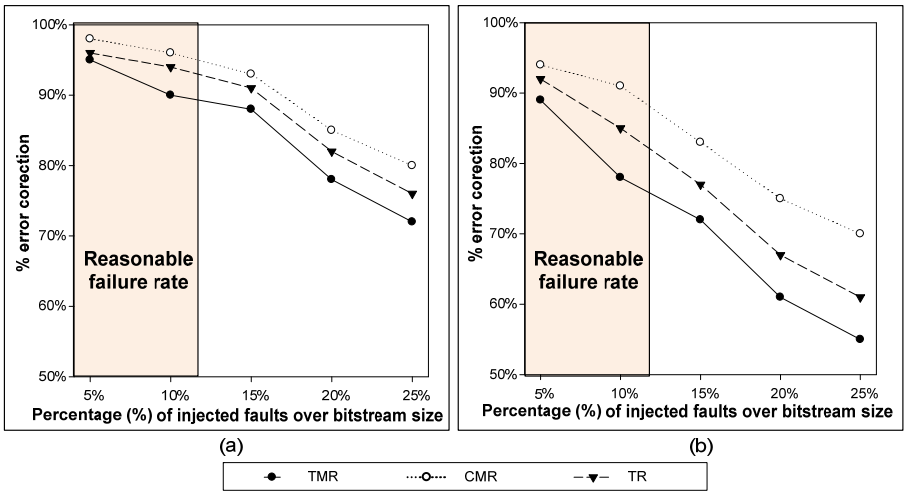


Figure 8. Efficiency of alternative fault tolerant techniques: (a) when they are applied unifomly over FPGA, (b) when applied based on the proposed region-based approach

Based on Figure 8 we can conclude that the proposed methodology results to significant error prevention for all the incorporated mechanisms. More specifically, regarding the case where *F*=5%, our region-based redundancy results up to 7% smaller efficiency in correcting faults, as compared to an implementation with full redundancy. However, we have to mention that our approach results to significant area savings, since 60% of the device area does not incorporate redundant blocks. In addition to that, in case the fault tolerance is applied uniformly over the FPGA, there is still a failure probability, since the employed redundancy scheme cannot prevent faults in routing infrastructure.

One of the disadvantages of applying fault tolerant techniques is the performance degradation due to additional hardware blocks. Table 1 shows the result in terms of Energy×Delay Product (EDP) for the original (i.e., non fault tolerant) and the three alternative techniques discussed in this paper. For each of these implementations we evaluate two instantiations: (*i*) all the functionalities are replicated (existing design approach), and (*ii*) only functionalities that are placed on critical device regions are replicated (region-based design approach). The derived modified fault tolerance schemes are denoted as *m_RMR*, *m_CMR* and *m_TR*.

Table 1. Characteristics of applications implementation

Benchmark	**Existing design approaches**				**Proposed (region-based) approaches**		
	Initial	**RMR [8]**	**CMR [3]**	**TR [4]**	**m_RMR**	**m_CMR**	**m_TR**
alu4	4.77	7.85	8.89	6.84	5.81	5.51	5.4
apex2	8.30	9.42	10.5	9.74	7.44	7.66	7.60
apex4	4.13	4.99	5.53	5.05	3.19	3.48	3.99
bigkey	1.41	1.66	1.81	1.69	1.19	1.03	1.00
clma	133	146	163	152	89.1	102.7	104.9
des	16.5	17.7	19.4	18.4	11.0	10.9	14.4
diffeq	2.21	3.14	3.59	2.96	2.29	2.05	1.81
dsip	6.29	7.81	8.63	7.85	4.76	6.21	6.44
elliptic	18.2	23.6	26.1	23.2	16.0	15.7	18.8
ex1010	37.1	41.2	43.4	42.1	26.4	32.6	33.3
ex5p	3.86	4.85	5.33	4.78	2.96	3.41	3.06
Frisc	22.0	26.5	28.9	27.1	17.5	17.6	21.7
misex3	4.35	5.72	6.41	5.62	4.06	3.65	4.44
Pdc	58.0	66.0	71.9	67.7	45.5	53.9	57.5
s298	12.5	21.1	24.3	18.1	12.7	15.1	10.5
s38417	24.2	32.7	37.9	32.1	23.5	25.4	19.3
s38514	25.1	27.0	29.3	28.0	17.3	22.0	19.9
Seq	5.47	6.40	7.15	6.55	4.03	4.72	4.85
Spla	25.8	39.2	43.0	36.0	28.6	26.2	20.5
Tseng	0.69	1.00	1.12	0.93	0.75	0.73	0.57
Average:	**16.29**	**19.9**	**22.1**	**19.9**	16.2	18.0	17.9
Ratio:	**1.00**	**1.22**	**1.36**	**1.22**	**0.99**	**1.11**	**1.10**

From the results provided in Table 1, the existing way of applying the three fault tolerant techniques exhibit increased EDP values, as compared to initial mapping, ranging from 1.22 up to 1.36 times. On the other hand, when we apply redundancy only on the application's functionalities mapped onto the 40% of device hardware resources placed on the middle of FPGA array, then the EDP overhead range from 0.99 up to 1.1 times. These values are almost similar to those retrieved when the device is designed without redundancy.

5 Conclusions

A novel framework for exploring and supporting fault tolerance at FPGAs was introduced. This framework can support alternative fault tolerance mechanisms based on voting, but rather than annotating the whole application's description with redundancy (as existing approaches do), we replicate only application's functionalities implemented onto regions with increased failure

probability. Since our framework does not require dedicated hardware blocks for supporting fault tolerance, it leads to increased flexibility, while it can be also applied to commercial devices as an additional step between synthesis and P&R. Moreover, our approach can provide a tradeoff between the desired reliability level and the extra overhead due to extra hardware resources. Based on experimental results, we shown that the proposed fault tolerant technique is not so much expensive in term of area, delay and energy dissipation, as compared to existing redundant solutions, while it leads to almost similar error correction.

References

[1] J. Lach, W. Mangione-Smith, and M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs", 6th Int. Symposium on Field Programmable Gate Arrays, pp. 105-115, 1998.

[2] K. Nikolic, A, Sadek, and M. Forshaw, "Fault-tolerant techniques for nanocomputers", Nanotechnology 13, pp. 357–362, 2002.

[3] D. Bhaduri, and S. Shukla, "NANOPRISM: A Tool for Evaluating Granularity vs. Reliability Trade-offs in Nano Architectures", Great Lakes Symposium on VLSI, pp. 109-112, 2004.

[4] F. Kastensmidt, et.al., "Fault-Tolerance Techniques for SRAM-Based FPGAs", Springer, 2006.

[5] B. Pratt, M. Caffrey, P.Graham, K. Morgan, and M. Wirthlin, "Improving FPGA Design Robustness with Partial TMR", 44th Int. Reliability Physics Symposium, pp. 226-232, 2006.

[6] L. Sterpone, et.al., "On the design of tunable fault tolerant circuits on SRAM-based FPGAs for safety critical applications", Design, Automation and Test in Europe, pp. 336-341, 2008.

[7] S.Y ang, "Logic Synthesis and Optimization Benchmarks", Version 3.0, Tech. Report, Microelectronics Centre of North Carolina, 1991.

[8] Xilinx TMR Tool (available at http://www.xilinx.com/quickstart/tmrq.htm)

[9] J. Ziegler, et.al., "IBM Experiments in Soft Fails in Computer Electronics (1978-1994)", IBM Journal of Research and Development, vol. 40, no.1, pp. 3-18, 1996.

[10] 2D MEANDER Framework (available at http://proteas.microlab.ntua.gr)

[11] J. Colinge, "Silicon-on-Insulator Technology: Overview and Device Physics", In IEEE Nuclear Space Radiation Effects Conference (NSREC), 2001.

[12] J. Yu and G. Lemieux, "Defect Tolerant FPGA Switch Block and Connection Block with Fine Grain Redundancy for Yield Enhancement", International Conference on Field Programmable Logic and Applications, pp. 255-262, 2005.

[13] R. Jain, A. Mukherjee, K. Paul, "Defect Aware Design Paradigm for Reconfigurable Architectures", IEEE Computer Society Annual Symposium on VLSI, pp. 91, 2006.

[14] A. Doumar and H. Ito, "Defect and Fault tolerance FPGAs by shifting the configuration data", in IEEE Symposium on Defect and Fault-tolerance, pp. 377-385, 1999.

[15] N. Camregher, et.al., "Analysis of Yield Loss due to Random Photolithographic Defects in the Interconnect Structure of FPGAs", Int. Symp. on FPGAs, pp. 138-148, 2005.

[16] J. Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components", Automata Studies, pp. 43-98, Princeton University Press, New Jersey, 1956.

[17] J. Black, "Electromigration - A Brief Survey and Some Recent Results", IEEE Transaction on Electron Devices, Vol. 16, No. 4, pp. 338, 1969.

[18] K. Siozios, et.al., "Designing a General-Purpose Heterogeneous Interconnection Architecture for FPGAs", Journal of Low-Power Electronics, Vol.4, No.1, pp. 34-47, April 2008.

[19] C. Stapper, "A New Statistical Approach For Fault-Tolerant VLSI Systems", 22nd International Symposium on Fault-Tolerant Computing, pp. 356-365, 1992.

SESSION 2A:
ARCHITECTURAL ISSUES

28

Decreasing the Impact of the Context Memory on Reconfigurable Architectures

Thiago Berticelli Ló[1], Antonio Carlos S. Beck[2], Mateus Beck Rutzig[1], Luigi Carro[1]

[1] Universidade Federal do Rio Grande do Sul, Instituto de Informática
Av. Bento Gonçalves, 9500 – Campus do Vale – Porto Alegre, Brasil
[2] Universidade Federal de Santa Maria, Departamento de Eletrônica e Computação
Avenida Roraima, nº 1000 – Bairro Camobi – Santa Maria, Brasil
{tblo, caco, mbrutzig, carro}@inf.ufrgs.br

Abstract. Reconfigurable architectures have already shown to be a potential solution to cope with the increasing complexity found in modern embedded computing systems, where high performance and low energy consumption are mandatory. In most of the works concerning reconfigurable computing, the main objective is to optimize the system by taking into account the known requirements of a project, such as speedup, energy or area. However, the impact of the context memory is very often ignored. In this article we show its significance in terms of power, and discuss a strategy to determine the most efficient way to handle the accesses to the context memory. Using as case study a coarse-grain reconfigurable array tightly coupled to the MIPS R3000 processor, we show that the energy of the context memory can represent up to 89% of the total system energy, and by applying the proposed strategy it is possible to save 59% of this amount.

Keywords: Adaptable Architectures, Reconfigurable Systems, Low-power Design, Memory Access

1 Introduction

The advance of integrated circuits technology over the years has allowed greater transistor integration. Modern mobile phones are an example: one can find several features in a single electronic device. Although these devices are still classified as embedded systems, they have to execute several heterogeneous applications that present a mix of control and dataflow behaviors. Nevertheless, as embedded systems are increasingly complex and need high performance computing, they are also tied to a number of design restrictions such as area occupation, energy consumption, memory footprint and time-to-market constraints, making embedded systems design a greater challenge than before.

An approach that emerged as an alternative to the previously mentioned problems is the employment of reconfigurable architectures. Reconfigurable systems provide flexibility of implementation, thanks to its ability to adapt to the behavior of the target applications even after manufacturing. They have already proven to be able to satisfy the constraints imposed by embedded systems [1].

Reconfigurable architectures are composed of basic configurable logic units, which can implement arithmetic and logical operations, or even more complex functions. The interconnection fabric between these units is also configurable. The set of configuration bits that indicate both configurable logic functions and their interconnection is called a context. Each context is stored in a special memory, the context memory.

In many studies using reconfigurable architectures, the main objective has always been to optimize the reconfigurable unit. However, the impact of the context memory in the overall power consumed in the system has often been neglected. Therefore, besides showing how significant is the power dissipated in the context memory, we also show an approach that can alleviate the problem. By correctly balancing the amount of memory accesses with the right number of memory output bits one can obtain minimum system energy consumption. For that, we studied how to find the ideal granularity to access the context memory, varying the amount of configuration bits fetched per access, which also involves changing the width of the memory port. As a case study, we use a coarse-grain architecture tightly coupled to the MIPS R3000 processor [2].

This paper is organized as follows. Section 2 characterizes the problem of the context of memory. Section 3 analyzes the impact of the context memory in different reconfigurable architectures. Section 4 presents the architecture of the reconfigurable data path. Section 5 presents the optimization strategy. Section 6 presents and discusses the results. Finally, section 7 draws conclusions and future works.

2 Identifying the Problem

If the reconfiguration process has to take place very often in a given reconfigurable system, so that there is a great number of context switches in a short period of time, the mean power of the context memory might be as large as or even larger than the power dissipated for performing the computation. For reconfigurable architectures that use a small number of configurations, acting at very specific kernels, the power dissipated by the context memory may be amortized by its continuous use through the time. However, recent works have shown that dynamic reconfiguration is a clear trend for both coarse and fine grain architectures, with several benefits that cannot be reached by static reconfiguration systems [2][4][5], reinforcing the necessity of the context memory optimization.

One important aspect of any memory is that the power it dissipates is a function of its size and the number of input/output bits. Actually, the number input/output bits play a major role, since for each output bit a sense amplifier and other buffering circuitry is required, even when considering memories embedded inside the chip. Figure 1 shows the impact of the number of input/output bits in the total memory energy for three different memory sizes. As it can be seen, the power overhead of increasing the memory width port by twice is larger than enhance the storage capacity by the same factor. This data has been obtained with Cacti 6.0 [3]. Figure 1 shows that there is a design space that must be explored, related to the number of accesses needed to load an entire configuration and the size of the configuration word.

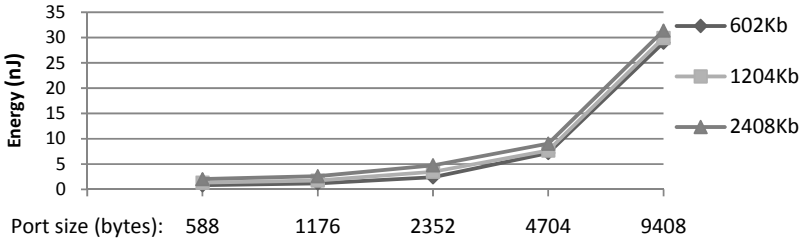


Figure 1. Energy spent per access with different port and memory sizes.

3 Related Work

Several studies on memory techniques have already been done concerning superscalar processors, VLIW and multicores to reduce energy consumption or increase the performance of the program memory. The usage of programmable associativity, size, and line size of an embedded system's cache architecture [6], encoding tag [7], scratchpad memory [8] and instruction compression [9] are some examples.

However, none of the above is related or can be directly applied to reconfigurable systems. As already discussed, different from the regular instruction/data memory, the context memory of Reconfigurable Systems stores configurations. Although both require high speed access times, the context memory presents an important difference: the size of its word and hence the number of output bits is orders of magnitude larger than the regular memory size, considerably increasing energy consumption.

In a reconfigurable architecture the number of basic reconfigurable components is one of the main factors that determine performance: the more functional units the reconfigurable unit have, the more speedup potential it presents. However, the amount of logic available directly impacts the size of the configuration word, increasing the number of bits necessary to store a configuration.

Fine grained architectures that work at the bit level, such as those based on FPGAs, require more configuration bits when compared to coarse grain architectures, which usually work at the word level. However, even in coarse grained architectures the number of bits a configuration takes is still very significant. For instance, let us consider PipeRench [10]. Piperench is composed of a set of stripes, which are physical pipeline stages. Each stripe has an interconnection network and a set of Processing Elements (PEs). The functionality of each PE is specified by 42 configuration bits, which means that each stripe needs 672 configuration bits to be configured. As this architecture has 16 stripes implemented in hardware, 1344 bytes are necessary to store an entire configuration. Configuration data is stored in 22 SRAMs, each with 256 32-bit words. In the CHESS architecture [11], each ALU needs 100 configuration bits. This architecture contains 512 ALUs, so the estimated number of bytes for a single configuration is 6400, including the routing mechanism. Other examples of coarse grain architectures with similar context memory requirements are MorphoSys [12], RaPiD [13] and the CCU [4].

However, none of the aforementioned works takes into account the power dissipated in the context memory on the overall system energy consumption. The question to be answered is whether there is best context memory partitioning, which can program several FUs to sustain high performance, while at the same time minimizing the energy of the whole system. We try to answer this question, by studying the best possible compromise among several design decisions.

4 Description of the Reconfigurable Architecture

Figure 2 shows the structure of the reconfigurable architecture employed as case study for this work. The reconfigurable unit is organized as a two-dimensional array of functional units, interconnected using multiplexers. It can also execute instructions in parallel, according to their data dependencies. As can be observed in Fig. 2a, the functional units are divided into groups (e.g. ALU, Load/Store, Multiplier). According to the delay of each group, more than one operation can be executed within one equivalent processor cycle. In this case study, according to the MIPS R3000 critical path, a reconfigurable architecture level corresponds to three rows of ALU in sequence, one Load or one Multiplication operation. One level can be observed in more details in Fig. 2b.

The reconfiguration and execution processes work as follows: initially, values of the input context are fetched from the register file while the configuration bits are fetched from the context memory. The configuration bits are responsible for routing data between the context bus and the functional units, and for selecting the operation of each functional unit. Then, that configuration is executed, taking a given number of equivalent processor cycles. Finally, results are written back to the register bank.

This architecture has a binary translation mechanism (BT), which is implemented in hardware and operates in parallel to the processor. At run time, the BT unit detects sequences of instructions that can be executed in the reconfigurable architecture. This sequence is translated into a data path configuration through the BT, and saved in the context memory. These sequences are indexed by the Program Counter (PC) register, so they can be used next time they are found. The context memory is similar to a fully associative cache. Each entry of the context memory is responsible for storing a particular configuration that is indexed by that PC. The number of context memory entries determines the number of configurations that can be stored at once. The size of each entry depends on the number of functional units that compose the reconfigurable array. These parameters are determined at design time.

5 Proposed Strategy

According to the experiments shown in Fig. 1, a linear increase in the memory port width reflects a non-linear increase in the energy consumption, even while maintaining the size of the context memory constant. Therefore, one way to reduce the energy during an access is to reduce the memory port bit-width. However, one needs to increase the number of memory accesses to load the very same information.

Memory area is also affected by this change, because a different number of output drivers, size of decoders and sense amplifiers will be required. It is important to notice that the response time necessary to the context memory to deliver data after a request is also affected. In our experiments, we have maintained this smaller than the critical path of the system, in order to avoid compromising the acceleration provided by the reconfigurable unit.

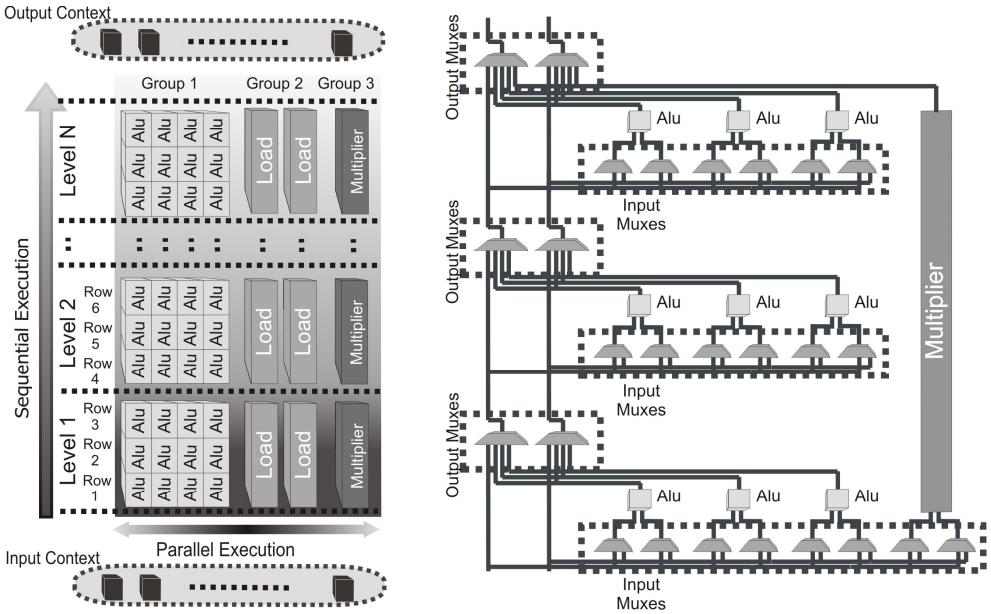


Figure 2: General overview of the reconfigurable array.

Considering a reconfigurable array composed of 4 levels, we divided the memory port bit-width by half (Fig. 3(b)) and by fourth (Fig. 3(c)) to evaluate the total energy consumption to load one configuration. As the total memory size is kept constant, the number of memory entries scales in accordance with the reduction of the memory port width, as shown in Fig. 3. Hence, we aim to find the best way to access each configuration in the context memory, in order to minimize energy consumption.

Applying the strategy described above, two extremes can be observed. The first case is when the configuration is loaded in just one cycle, so all configuration bits are accessed in parallel (Fig. 3(a)). Consequently, a large memory port is required and great amount of energy is consumed per access. In another extreme, only one level of the reconfigurable array is reconfigured per cycle (Fig. 3(c)).

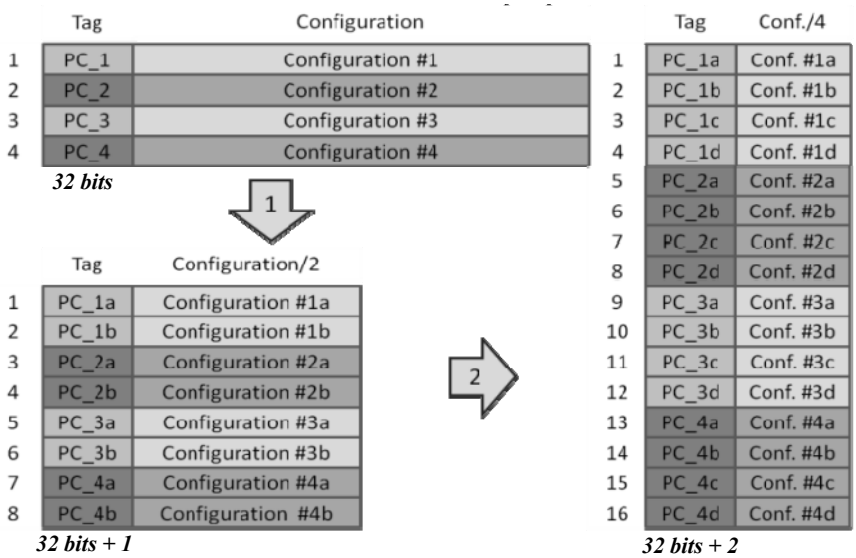


Figure 3. Context memory exploration by dividing the configuration in parts.

6 Results

In our study we have used SystemC to simulate the reconfigurable system coupled to a MIPS R3000 processor executing the Mibench Benchmark Suite [14]. From this model we extracted performance results and the number of memory requests to the context memory. Cacti 6.0 tool has been used [3] to model the different context memory arrangements and extract their power consumption, access energy, area and access time, using a CMOS 90 nm technology.

As presented in Table 1, four different array setups were considered to evaluate the energy consumption and performance of the system. Each setup is composed of a different number of functional units, being the Setup 4 the largest one. This table also shows the required number of bytes necessary to store one configuration considering the number of functional units of each setup. To give an idea on the speedup presented by the reconfigurable system used as case study, Figure 4 presents the average speedup of the four different setups for each application varying the number of memory context entries. More than 64 entries in the context memory do not provide relevant performance gains.

Table 2 presents the energy spent per access, in nJoules, when one varies the number of levels fetched per access and, consequently, the number of bytes. The first row shows the energy needed to fetch only one level per access and so on. As the number of fetched levels dictates the memory port width, the more levels fetched per access, the more energy is necessary.

Table 1 - Different Setups for the Array.

	Setup #1	Setup #2	Setup #3	Setup #4
#Rows	24	48	96	192
#ALU / level	8	8	12	12
#Multipliers / level	1	2	2	2
#Ld |St /level	2	6	6	6
#Levels	8	16	32	64
#Configuration bytes / level	107	124	147	147
Total #Configuration bytes	856	1984	4704	9408

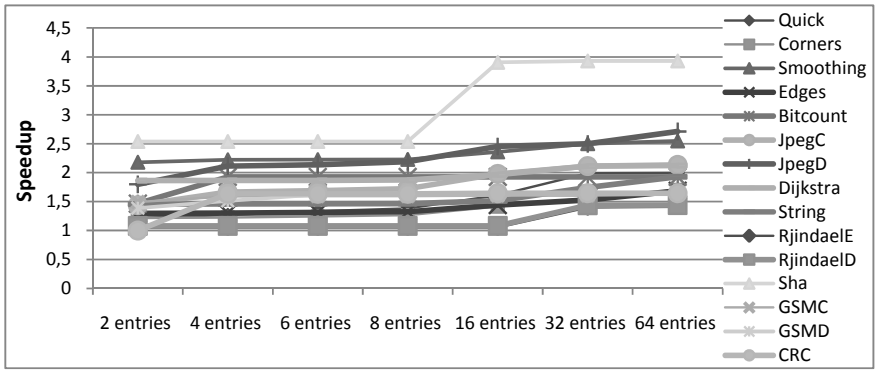


Figure 4. Average speedup of the four setups for each Mibench benchmark application.

Table 2 - Energy (nJ) necessary for different number of accesses

	Setup 1	Setup 2	Setup 3	Setup 4
1 Level/ access	0.147	0.308	0.535	1.141
2 Levels/ access	0.126	0.251	0.414	0.779
4 Levels/ access	0.164	0.311	0.514	0.796
8 Levels/ access	0.305	0.507	0.943	1.175
16 Levels/ access	-	1.376	2.023	2.379
32 Levels/ access	-	-	6.751	7.188
64 Levels/ access	-	-	-	28.987

Figure 5 shows the total energy necessary to load an entire configuration varying the number of levels fetched per access. The number of necessary levels to configure the whole unit depends on the used setup. According to Table 1, for setups #1, #2, #3 and #4; 8, 16, 32 and 64 levels are necessary to compose an entire configuration, respectively. The results demonstrate that, for each array setup, there is an optimal number of levels to be fetched per access that minimizes the energy consumption to load an entire configuration.

For setup #1, which is the smallest one, the best way to configure the array is to load the whole configuration (the 8 levels that compose that configuration) at once. On the other hand, for setups #2, #3 and #4, the best option is to fetch 8 levels per access. In the case of setup #2, for example, two accesses of 8 levels each would be necessary to load an entire configuration, showing the best tradeoff regarding energy consumption. In setup #4, although there is a minimum difference between accessing 8 and 16 levels per cycle, the former still can be considered the best choice in terms of energy, although it will take more cycles to fetch the entire configuration. Considering the best case for each setup, there are 26%, 44% and 67% of energy savings over the original strategy for setups #2, #3 and #4, respectively.

As shown in Fig. 5, in most cases, except for Setup 1, the results suggest that one should make more accesses to the context memory with smaller bit-widths. However, reconfigurable systems that need the complete configuration to be ready before execution could show performance losses when using the proposed approach, since more accesses to the memory must be performed to load an entire configuration. That is not the case with reconfigurable unit used in these experiments. In this unit, while one level is being executed, the next one is fetched. This way, at each clock cycle at last one level is being executed, while the rest of the reconfigurable unit is either processing or waiting for a new level. Therefore, no additional delay is inserted and performance is maintained. This approach is very similar to the virtualization process used in PipeRench[10], and can be applied to several other reconfigurable systems.

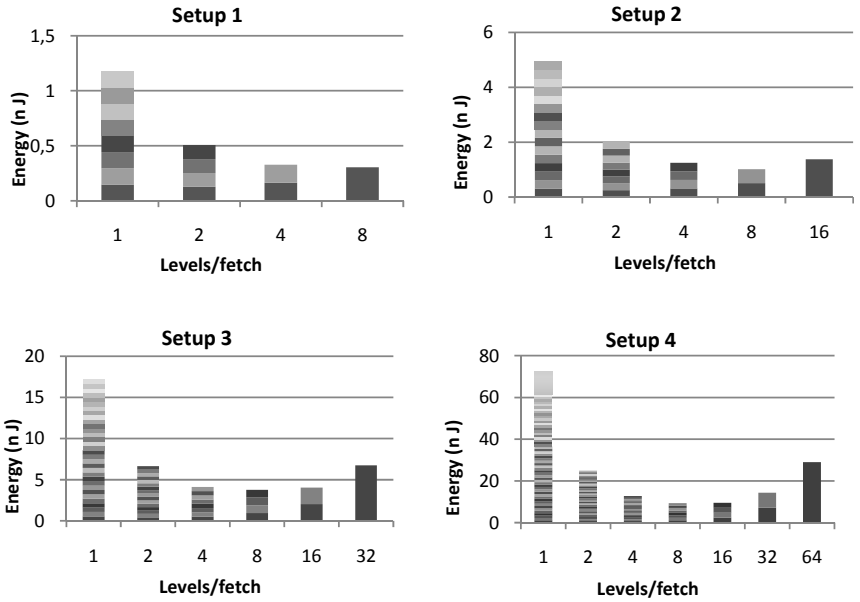


Figure 5. Total energy consumption to load a whole configuration, varying the number of levels fetched per cycle

To better visualize the results in Figure 5, Table 3 shows the energy consumed to fetch an entire configuration considering three different scenarios: fetching only one level per access; fetching the entire configuration in one access; and using the context memory port width that provides best tradeoff possible considering each setup separately.

Table 3 – Energy spent (nJ) to fetch the Configuration

Setup	One level per access	Entire configuration per access	Best tradeoff possible
#1	0,59	0,30	0,30
#2	4,93	1,37	1,01
#3	17,1	6,75	3,77
#4	73,03	28,98	9,40

Figure 6 shows the gains obtained by the use of the technique considering the overall system energy consumption. It compares the original access method to the optimal, which is the best tradeoff possible presented before. Each bar shows the energy saved when using the optimal method, the energy spent by the context memory, and the energy spent by the functional units of the reconfigurable unit. For setups #1, #2, #3 and #4, the context memory is responsible for 13%, 40%, 68% and 89% of the whole system energy, respectively. When applying the proposed technique, 10%, 29% and 59% of system energy was saved, for setups #2, #3 and #4. No gains are shown for Setup #1, since the best policy is exactly the same as before: fetch the entire configuration at once. It is important to note that, despite the gains obtained by the use of the proposed technique, the context memory remains responsible for a significant portion of the total energy consumption. Hence, there is still a great design space to be exploited in this subject.

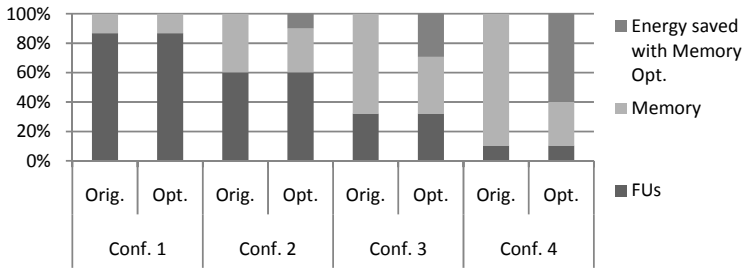


Figure 6. Percentage of energy consumption of memory for the system energy and energy save by optimization

7 Conclusions

This work demonstrated that the context memory is responsible for a large part of the energy consumed in reconfigurable systems. By finding the ideal memory port

width one can minimize energy at the system level. Results show that the reconfiguration memory can represent up 89% of the total system energy, and by applying the strategy here discussed it is possible to save 59% of the total energy. The approach is general in the sense it can be applied to any reconfigurable system, though those that support partial reconfiguration are the best candidates to use it without suffering of significant performance penalties when dealing with pipelined reconfigurable units.

References

1. P. Ienne and R. Leupers. Customizable Embedded Processors: Design Technologies and Applications. San Mateo : Morgan Kaufmann, 2006.
2. C. Beck, M. B. Rutzig, G. Gaydadjiev, L. Carro. Transparent Reconfigurable Acceleration for Heterogeneous Embedded Applications. In: Design, Automation and Test in Europe, 2008.
3. N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi, ‘‘Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0,’’ Proc. 40th Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 07), IEEE, 2007,pp. 3-14.
4. Clark, N., et al.: An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors. In: ISCA, pp. 272–283 (2005)
5. R. Lysecky, F. Vahid. A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning. Design Automation and Test in Europe Conference, 2004.
6. Zhang, C., Vahid, F., and Najjar, W. 2003. Energy Benefits of a Configurable Line Size Cache for Embedded Systems. In Proceedings of the IEEE Computer Society Annual Symposium on VLSI (Isvlsi'03) (February 20 - 21, 2003). ISVLSI. IEEE Computer Society, Washington, DC, 87.
7. Zhang, M., Chang, X., and Zhang, G. 2007. Reducing cache energy consumption by tag encoding in embedded processors. In Proceedings of the 2007 international Symposium on Low Power Electronics and Design (Portland, OR, USA, August 27 - 29, 2007). ISLPED '07. ACM, New York, NY, 367-370.
8. M. Verma, L. Wehmeyer and P. Marwedel. Dynamic overlay of scratch-pad memory for energy minimization. In International Conference on Hardware/Software Codesign and System Synthesis(CODES+ISSS) (Stockholm, Sweden). ACM Press, New York, 2004.
9. Benini, L., Macii, A., Macii, E., and Poncino, M. 1999. Selective instruction compression for memory energy reduction in embedded systems. In Proceedings of the 1999 international Symposium on Low Power Electronics and Design (San Diego, California, United States, August 16 - 17, 1999). ISLPED '99. ACM, New York, NY, 206-211.
10. S. Goldstien and H. Schmit, M. Moe, M. Budiuy, S. Cadambi, R. Taylora and R. Laufer. "PipeRench: A Coprocessor for Streaming Multimedia Acceleration," in Proc. International Symposium on Computer Architecture (ISCA), Atlanta, GA, 1999.
11. Marshall et al.: A Reconfigurable Arithmetic Array for Multimedia Applications; Proc. ACM/SIGDA FPGA‘99, Monterey, Feb. 21-23, 1999.
12. H. Singh, et al.: MorphoSys: An Integrated Re-configurable Architecture; Proceedings of the NATO RTO Symp. on System Concepts and Integration, Monterey, pp. 20-22, 1998.
13. Ebeling et al. “RaPiD: Reconfigurable Pipelined Datapath“, in Proc. FPL’96, Darmstadt, Germany,Sept. 23-25, 1996, LNCS 1142, Springer Verlag 1996.
14. M. R. Guthaus, J. S. Ringenberg, D. Ernst and T. M. Austin. "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in 4th Workshop on Workload Characterization, 2001.

INDUSTRY SESSION

40

Short presentation: Synthesizing Efficient Architectures from ANSI-C Specifications: The HCE Compiler

P. Palazzari

Ylichron, an ENEA spin-off, Rome, Italy

42

SESSION 2B:
ARCHITECTURAL ISSUES

44

Register File Design in Automatically Generated ASIPs

Roza Ghamari and Arda Yurdakul

CASLAB, Computer Engineering Department,
Bogaziçi University, Istanbul, Turkey
{roza.ghamari, yurdakul}@boun.edu.tr

Abstract. Instruction set identification problem has been one of the major research topics in the last decade. Most of the solution proposals in the literature assume a fixed size register file with pre-specified input and output ports. On the other hand, reconfigurable hardware such as an FPGA has a variety of on-chip resources, which can be configured according to the requirements of the application. Hence, in this work, we propose a register file design methodology for ASIPs on the FPGAs. Our tool uses the instruction set and the execution thread to generate a register file with reduced number of inter-register transfers and maximum number of I/O ports required by the application. Moreover, the file can be partitioned into different size of registers during run-time according to the instruction that is being executed. The experimental results show that for implementations with concurrent instructions, this algorithm can reduce length of register file by 40% in average.

Keywords: Application Specific Instruction-set Processors, Configurable Register file, Reconfigurable Architecture, Register File Management.

1 Introduction

Multifarious needs of industry and science led embedded system manufacturers to design customizable processors, which offer design flexibility, high performance and short time-to-market simultaneously. Thanks to FPGAs, it is possible to reconfigure hardware fabric after production and implement a function-specific processor [1].

Depending on the specifications of the target hardware, we can talk about two different types of application specific processors. If the target hardware is a tightly-coupled processor and reconfigurable fabric pair, then the extension of the instruction set is considered [2][3]. If the target hardware is completely reconfigurable like an FPGA, then the application can be either directly synthesized and mapped on the

This work has been fully supported by the Turkish Foundation of Science and Technology, under Research Project Program (pr. no:104E038)

hardware [4] or implemented as an ASIP whose instruction set is completely determined by the application [5][6]. Hence, the common point in both approaches is to extract the new instruction set according to the application.

In the literature, application specific instruction (ASI) identification is based on a set of constraints due to the register file or other forms of memory where the new instructions have to be connected to [7]. These constraints are the number of input and output ports and the size of the register file, which is fixed in custom or general-purpose processors. However, for ASIPs on the FPGAs, the register file neither has to be fixed nor has to have a fixed number of read/write (R/W) ports, because an FPGA serves a vast amount of on-chip resources. Therefore, in this study, we propose reversing the processor customization process: The instructions must be selected without register file constraints[1]. Once they are selected, the register file must be tailored according to the ASIs. Based on this proposal, we have developed a simple algorithm to generate a run-time flexible register file for ASIPs generated by the RH(+) environment [5]. The size and number of R/W ports of the register file are determined according to the application at design time. However, the size of each register can change during run-time without changing the complete size of the register file. As a result, register reuse is utilized. Additional optimizations are also made to reduce the inter-register transfers. The algorithm operates in the RH(+) environment, which is developed for designing application specific embedded processors on FPGAs. Experimental results show that this algorithm efficiently minimizes the length of register file for different applications.

It should be noted that the study presented here is a preliminary demonstration of an idea. It does not claim to be the best solution. Based on this self-realization, the next section presents a brief overview on previous studies in the literature in the domain of ASI extraction. The algorithm for register file design is described in detail in Section 4. Experiments are evaluated in Section 5 and the final section concludes the work.

2 Literature Survey

Instruction set extension has been extensively studied in the last decade. The basic objective is to improve the computation time by including application specific instructions to the data path of a custom-designed or a general purpose processor [7]. This process usually reduces power consumption in the meantime. However, there is also a study, whose objective is to reduce energy consumption and to increase battery lifetime [8].

The instruction identification process is straightforward: The control and data flow graph (CDFG) of the application is studied so as to extract the most time consuming patterns that will be identified as instructions. In the mean time, a set of constraints needs to be satisfied. The first constraint is the convexity, i.e. each selected instruction has to operate without intervention of the processor. The remaining constraints are imposed by the target hardware. The new instructions have to interface with the

[1] The designer might still impose constraints like overall execution time, data bus width, area, etc.

register file or the main memory. Hence, the size, the number and width of I/O ports constrain the shape of the patterns which will be identified as new instructions.

The existing hardware architecture of the processor imposes a performance bottleneck. To reduce this effect, there have been proposals like shadow registers [9] [10], multiport registers [11], custom registers for multiple inputs and multiple outputs [12][13][14], dedicated data paths [15] and incorporation of an additional register file for the new instructions so as to reduce memory access times [16].

In all these approaches, there exists a custom register file due to the fixed architecture of the custom-designed or general-purpose processor. However, for reconfigurable processor architectures [6][17][5], there are only primitives for generating application specific register files. In [6][17], two level register files are used. The global register files is used as an interface for data transfer and a scratchpad while the local ones are exploited by functional units in the reconfigurable array. The application-specific register file is distributed and has to be formed by combining the necessary global and local register files. There are parameters to determine the size and ports of the register file.

RH(+) [5] is a development environment for generating application specific microprocessors on a reconfigurable hardware, basically an FPGA. In this toolset, the register file is generated automatically by inspecting the application. Therefore, the best register file is extracted according to the application-specific instruction set. This is a quite new approach in automatic ASIP design because the register file is not a constraint for the instructions of the application. In a Xilinx FPGA, each LUT can be configured as a one-bit distributed memory with one read and one write port. One-bit dual port memory can also be done by combining two LUTs. Moreover, each LUT is also coupled with one bit register/latch module. Therefore, in RH(+), a distributed register file is automatically generated by combining one-bit distributed memory units and if necessary, registers. The following sections describe the register file generation in RH(+).

3 Register File Design Algorithm (RFDA)

The required input files for the RFDA are developed by RH(+) toolset. Most of the detailed information is gathered from a XML based Control Flow Data Graph (CDFG) file, which is generated by a front-end compiler running over the application that the user develops in the RH(+) environment. The CDFG includes bit-length and type information (array or scalar) of each variable, and inputs and outputs of the operators. In Fig.1, a sample graph is presented for a simple program.

Based on the generated CDFG, RH(+) performs instruction selection. Next, a template file, which includes the assembly templates for the generated instructions, is provided for the back-end compiler. In the back-end compiler, a primitive assembly code is generated by processing the CDFG and the template file. The primitive assembly code for the CDFG of Fig. 1 is shown in Fig. 2.

The RFDA runs over the CDFG and the primitive assembly code. Its first goal is to find the optimum size for the register file for the user application. Furthermore, it introduces RH(+) how to allocate registers during run-time while preserving the register file's total length. Indeed, this algorithm designs the register file as an array

of single bit registers. In order to access the registers, multiplexers are utilized inside the control and the data paths of the processor. A pseudo code for the RFDA is shown in Fig. 3. This pseudo code can be explained as follows:

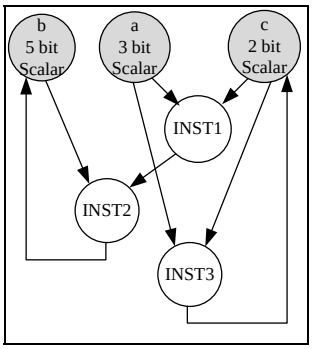


Fig. 1. Example of a Control Data Flow Graph generated by front-end compiler

Line #	Instruction
1	MOV R0 a
2	MOV R1 c
3	INST1 R2 R0 R1
4	MOV R0 b
5	INST2 R1 R0 R2
6	MOV b R1
7	MOV R0 c
8	MOV R1 a
9	INST3 R2 R0 R1
10	MOV c R2

Fig. 2. Resulted primitive assembly code by The back-end compiler

- *Manage_SubRegisters (Primitive_Assembly , CDFG).* The compiler of RH(+) makes register allocation without considering the size of the variables. The size of each register may change according to the size of the variable. A register in the register file is active when a value is written to it. It is inactive after the last instruction, which reads that value. We denote each active period for a register as the sub-register of that register. *Manage_SubRegisters* function generates all sub-registers for each register inside the assembly code. Table 1 shows the sub-register table generated for the primitive assembly code in Fig. 2. For instance, consider register R1; it first appears in line 2 where it gets the value of variable 'c', which is 2 bits long. R1 holds this value until it is accessed in line 3. Therefore, we write R1_2_3_2 inside the table as a sub-register of R1. In Table 1, there are three sub-registers for R1, namely R1_2_3_2, R1_5_6_5 and R1_8_9_3. Note that sizes of sub-registers can be different.

```
Register_File_Generation (CDFG, Primitive_Assembly)
{
    Sub-Register_Table = Manage_SubRegisters (Primitive_Assembly, CDFG);
    Time-Table = Create_Time-Table (Sub-Register_Table);
    OffsetGroup_Table = Set_OffseGroupst (ref Sub-Register_Table,Time-Table);
    Set_Offset_and_Rename (ref Sub-Register_Table, OffsetGroup_Table);
    New_assembly = Rewrite_Assembly(Sub-Register_Table, Primitive_Assembly);
}
```

Fig. 3. Pseudo code for the RFDA

- *Create_Time-Table (Sub-Register_Table).* The time-table keeps track of sub-registers during the program execution time. In this table, columns represent time slots and rows stand for sub-registers. All of the sub-registers that exist in a specified instruction cycle are marked in the related column. Next, sum of bit-lengths for all

registers used in that time slot is calculated. The time-table of Fig. 2 is shown in Table 2. For instance, sub-register R1_5_6_5 starts in line 5. It also appears in line 6. Hence, fifth and sixth columns are marked for the R1_5_6_5 sub-register.

Table 1. Sub-Register Table for example

Register Name	Bit Length	BI	DI	Offset	NewName
R0_1_3_3	3	1	3	--	--
R0_4_5_5	5	4	5	--	--
R0_7_9_2	2	7	9	--	--
R1_2_3_2	2	2	3	--	--
R1_5_6_5	5	5	6	--	--
R1_8_9_3	3	8	9	--	--
R2_3_5_3	3	3	5	--	--
R2_9_10_5	5	9	10	--	--

Table 2. Time-Table of example

Time	1	2	3	4	5	6	7	8	9	10
R0_1_3_3	x	x	x							
R0_4_5_5				x	x					
R0_7_9_2							x	x	x	
R1_2_3_2		x	x							
R1_5_6_5					x	x				
R1_8_9_3								x	x	
R2_3_5_3			x	x	x					
R2_9_10_5									x	x
Length	3	5	8	8	13	5	2	5	10	5

- *Set_OffsetGroups (ref Sub-Register_Table, Time-Table).* An offset is the start address of a sub-register in the register file. To set offset values, we run an algorithm, which finds the best offset group for each sub-register so that the size of the register file will be optimized. Here, sub-registers that have equal or similar sizes (considering a defined threshold) are categorized on offset groups. It is assumed that sub-registers in the same offset group are not accessed simultaneously during runtime. Threshold is an experimentally obtained value and it limits the differences between the sizes of the sub-registers in an offset group. Although applying a threshold may lead a larger register file, it reduces the complexity of addressing. The offset-group table for our example is shown in Table 3.
- *Set_Offset_and_Rename (ref RegisterFileTable, OffsetGroup_Table).* In this step, the same offset value is set for each sub-register inside an offset group. The offset value is also calculated based on the algorithm in Fig. 4. Then, each sub-register is renamed and these new names are used in the next function. The final version of sub-register table is demonstrated in Table 4.

Table 3. OffsetGroup table for example

	Register Group	Maximum Length
1	R0_1_3_3, R0_4_5_5, R1_8_9_3	5
2	R0_7_9_2, R1_2_3_2	2
3	R1_5_6_5, R2_9_10_5	7
4	R2_3_5_3	3

```
Offset = 0;
for each group in OffsetGroup table {
    for each sub-register inside group
        offset of sub-register = Offset;
    Offset = Offset + maximum size of all sub-registers inside group;
}
```

Fig. 4. Offset calculation algorithm

Table 4. Offset calculation and register naming

Register Name	Bit Length	BI	DI	Offset	NewName
R0_1_3_3	3	1	3	0	R0_3_0
R0_4_5_5	5	4	5	0	R0_5_0
R0_7_9_2	2	7	9	5	R0_2_5
R1_2_3_2	2	2	3	5	R1_2_5
R1_5_6_5	5	5	6	7	R1_5_7
R1_8_9_3	3	8	9	0	R1_3_0
R2_3_5_3	3	3	5	12	R2_3_12
R2_9_10_5	5	9	10	7	R2_5_7

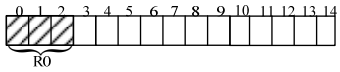
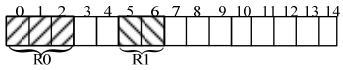
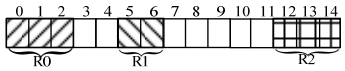
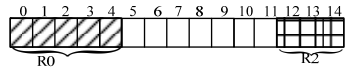
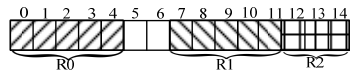
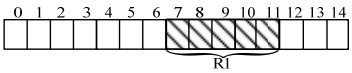
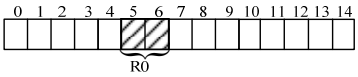
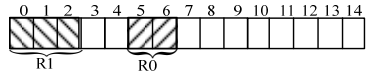
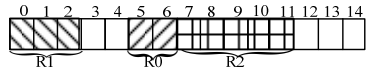
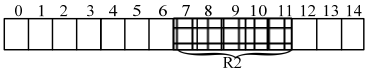
- *Rewrite_Assembly (Primitive_Assembly).* This function replaces the names of the registers in the primitive assembly code with their new names. The final version of the assembly code in Fig. 2 is shown as the last column in Table 5.

After the execution of the algorithm, RH(+) generates a register file whose length is equal to the sum of the offset of the sub-registers in the last offset group and the length of the largest sub-register inside that group. RH(+) also generates the necessary combinational circuit to read and write to the specified sub-registers. Table 5 demonstrates organization of the register array according to the final assembly code. It should be noted that the size of each register changes dynamically. For example the initial size of R0 is 3, and then it becomes 5.

4 Experiments

The proposed algorithm is executed for three different algorithms implemented in RH(+) environment. Since RH(+) provides us with the opportunity of defining both traditional and application specific operators, each of these algorithms is implemented using different sets of instructions. Moreover, RH(+) processor architecture supports VLIW instructions. Hence, we are provided with concurrent operations, which results in a high access rate to the register file for each instruction. Therefore, results of the RFDA tightly depend on the automatically selected instruction set in addition to the defined operators. In Table 6, specifications of each experiment are listed. Each specification shows the defined operator types and the maximum amount of concurrency in case of VLIW utilization. In Table 7, we show the resulting register file sizes for each experiment considering whether RFDA is applied or not. Moreover, we take into account the impact of the instruction selection unit. Thus, for each implementation, we run algorithm for both of the cases: the instruction set consists of 1) concurrent operations, 2) no concurrent operations, i.e. sequential implementation.

Table 5. Resulted assembly code and the organization of registers in each instruction cycle

Inst. Step	Dynamically Adaptive Register File	Assembly Code
1		MOV R0_3_0 a
2		MOV R1_2_5 c
3		INST1 R2_3_12 R0_3_0 R1_2_5
4		MOV R0_5_0 b
5		INST2 R1_5_7 R0_5_0 R2_3_12
6		MOV b R1_5_7
7		MOV R0_2_5 c
8		MOV R1_3_0 a
9		INST3 R2_5_7 R0_2_5 R1_3_0
10		MOV c R2_5_7

In the first experiment a 15-tap Finite Impulse Response (FIR) filter is implemented. The coefficient set consists of 8-bit scalars and is stored in an array. The input is also 8-bit long and input at each tap is stored in an array. “Double loop” version of the algorithm is implemented by defining four traditional operators listed in Table 6. In the concurrent version of this implementation, at least one VLIW instruction including three concurrent instructions exists. In the “Single loop” approach, we defined array adder and array multiplier operators. These array operators actually take two arrays as inputs, perform the operations on their elements, and then store each result in a new array. Actually, by using these operators, we do not need a loop for the calculation of the multiplication and addition operations one by one for all elements of arrays. As a result, we only need one loop for shifting the input array.

Table 6. Implementation specifications of experiments

Experiment		Defined Operators	Maximum concurrency
FIR	Double loop	Scalar add/sub - Scalar Mul –Indexing - Shift	3
	Single loop	Array Add - Array Mul – Shift - Increment	3
FFT	Classic	Scalar add/sub - Scalar Mul – Indexing – Cosin - Sine	8
	Unfolded by two	Scalar add/sub – Indexing	4
MIC	Double loop	Scalar add - Scalar Div - Scalar root -Scalar sqr - Indexing	3
	Hybrid	Scalar add- Scalar Div - Scalar root – Indexing - Array Add - Array Sqr	3

The FFT implementation in RH(+) is radix-2 decimation-in-time algorithm. The real and imaginary parts of each number are stored in separate arrays and output values are written over input values, which make this implementation an in-place one. The algorithm calculates the twiddle factors by using sine and cosine operators, so sine and cosine hardware modules have to be available. The "classic" implementation consists of trivial three nested for loops that go through the FFT-stages, the butterflies with the same twiddle factors and each individual butterfly. This algorithm is implemented with a 4-tap array whose elements are 8 bits long. In the "unfolded by two" implementation, there are no loops and temporary storages except individual butterfly nodes. This algorithm is implemented with a 2-tap array with 8-bit long elements. In Table 6, the defined operators and the maximum number of concurrent instructions are given.

The Motion Intensity Calculator (MIC) is implemented based on the algorithm defined in [18]. This algorithm is also implemented in two different ways. In the "double loop" implementation, we defined purely scalar operators. Hence, we need two loops, one for the calculation of the average value of the input arrays and the other one for the calculation of motion intensity values. The second approach is the "hybrid" one in which we use a combination of scalar and array operators. We define square and add operators for array variables. In addition, there are some other operators like division and root, which are implemented with scalar variables. For both implementations, we consider arrays with twelve 8-bit motion vectors as inputs.

The experimental results of Table 7 can be studied in two ways:

- *Improvement in register file sizes*: Register files for sequential and concurrent implementations have to be considered separately. For the sequential case where we have only ASIs, the average amount of register file size reduction after applying the RFDA is equal to 36%. For the concurrent case where we have both ASIs and VLIW, the reduction in the size of the register file after RFDA is 41% in average.
- *Improvement in performance*: Application specific instruction selection is done to improve the performance. The performance of the concurrent implementation is 21% better than that of the sequential implementation. As stated above, this improvement is due to the selected VLIW instructions in the concurrent implementation. Since these instructions are selected without taking the register file into account, the performance measures for these implementations are the

optimum values. Moreover, RFDA provides an optimized register file for the optimum number of execution steps.

Table 7. Experimental results of algorithm

		Concurrent Instructions				Sequential Instructions			
		Register File Length			Total Exec. Steps	Register File Length			Total Exec. Steps
		RFDA	Normal	%Imp		RFDA	Normal	%Imp	
FIR	Double loop	290	840	65%	166	290	480	65%	178
	Single loop	390	840	54%	81	265	360	26%	93
FFT	Folded	247	384	36%	1054	122	128	5%	1925
	Unfolded by two	112	128	12%	24	48	64	25%	42
MIC	Double loop	228	576	60%	696	121	288	77%	816
	Hybrid	429	552	22%	572	321	384	16%	634
Average Imp in Register File Size				**41%**				**36%**	
Average Imp in Performance		**%21**							

5 Conclusion

In this work, we propose an efficient algorithm for designing a register file for Application Specific Instruction-set Processors. Our aim in this paper is to reverse the methodology of ASI selection for ASIPs, which are designed for FPGAs where a vast amount of resources are available: We generate ASIs without considering register file constraints. Hence we aim to maximize the performance. Once the instructions are selected, then the register file can be tailored to meet the architectural constraints set by the ASIs.

This work is done in the RH(+) design automation tool which is a system level design tool for embedded systems on reconfigurable fabrics. The experimental results show that register file design algorithm, RFDA, makes efficient use of memory and area. After RFDA, the size of the register file is reduced up to 35% in average for the experiments implemented by ASIs only. However, when we combine ASIs in VLIW instructions, the number of execution steps decreases by 20% in the average. For this case, the application of RFDA results in 41% reduction in the register file size.

Acknowledgment

We would like to thank anonymous reviewers for their comments to improve the quality of this paper. We also thank Alp Malazgirt and Caglar Yali from CASLAB for their help on providing tests for the experimental work.

References

1. A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," IEEE/ACM Design Automation Conference, 1999.
2. R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60–70, 2000.
3. Target Compiler Tech., "CHESS/CHECKERS: A retargetable tool-suite for embedded processors," Belgium, Tech. Rep., June 2003.
4. Synfora Inc., "www.synfora.com," 2009.
5. B. Kurumahmut, G. Kabukcu, R. Ghamari, A. Yurdakul, "Design Automation Model for Application-Specific Processors on Reconfigurable Fabric", *Proc. of FDL'09*, 2009.
6. B. Mei and et.al, "Architecture exploration for a reconfigurable architecture template," *IEEE Des. Test*, vol. 22, pp. 90–101, 2005.
7. P. Ienne,R. Leupers, "Customizable embedded processors: design technologies and applications", *Morgan Kaufmann*, 2006.
8. N. Cheung, S. Parameswaran, J. Henkel, "Battery-Aware Instruction Generation for Embedded Processors," *Proceedings of ASP-DAC'05*, 2005.
9. J. Cong, et al, "Instruction Set Extension with Shadow Registers for Configurable Processors." *13th ACM Inter. Symposium on Field-Programmable Gate Arrays*, Feb 2005.
10. J. Cong, G. Han, and Z. Zhang. "Architecture and compilation for data bandwidth improvement in configurable embedded processors." In *Proc. Int. Conf. Computer-Aided Design*, pages 263–270, Nov. 2005.
11. S. Rixner, et al, "Register Organization for Media Processing," in *Proc. Sixth Int. Symp. on High-Performance Computer Architecture*, pp. 375-386, Jan. 2000.
12. F. Sun, S. Ravi, A. Raghunathan, N. K. Jha. "A Scalable Synthesis Methodology for Application-Specific Processors." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol: 14, pp 1175-1188, Nov. 2006.
13. R. E. Gonzalez. "Xtensa: A configurable and extensible processor." *IEEE Micro*, 20(2):60–70, Mar./Apr. 2000.
14. C Galuzzi, K Bertels and S Vassiliadis, "A linear complexity algorithm for the automatic generation of convex multiple input multiple output instructions," Int. Journal of Electronics, Vol. 95, No. 7, pp 603–619, July 2008.
15. Xilinx, Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel, http://www.xilinx.com/
16. H. Lin, Y. Fei. "Utilizing custom registers in application-specific instruction set processors for register spills elimination." *Proc. of 17th Great lakes symp. on VLSI*, pp 323-328, 2007.
17. Z. Kwok, S. J. E. Wilton, "Register File Architecture Optimization in a Coarse-Grained Reconfigurable Architecture," *Proc. of 13th Field-Programmable Custom Computing Machine (FCCM'05)*, pp 35- 44, 2005.
18. X. Sun, D. Ajay, and B. S. Manjunath, "A Motion Activity Descriptor and Its Extraction in Compressed Domain,"*IEEE Pacific-Rim Conf. Multimedia (PCM),* pp. 450-453, October 2001.

SESSION 3: PROGRAMMING AND SCHEDULING

56

Reconfiguration Aware Task Scheduling for Multi-FPGA Systems

Francesco Redaelli[2], Marco D. Santambrogio[1,2], Donatella Sciuto[2], and Seda Ogrenci Memik[3]

[1] Massachusetts Institute of Technology
[2] Politecnico di Milano
[3] Northwestern University
santambr@mit.edu
{fredaelli, santambr, sciuto}@elet.polimi.it
seda@eecs.northwestern.edu

Abstract. The benefits of partially and dynamically reconfigurable systems have been traditionally defined as the ability of such systems to adapt for unknown run-time changes in resource allocation and resource types in the system. However, partially and dynamically reconfigurable systems can also be effective when they serve in a multi-user environment for application domains that are relatively well defined (e.g., a high performance computing node for scientific computing). Particularly for data intensive high performance computing applications, the system needs to be flexible for varying amount of data parallelism. In this paper, we present a design framework to help guide dynamic decisions about sharing the computational environment and task placement in partially and dynamically reconfigurable FPGA clusters deployed for data intensive high performance computing applications. In order to exploit the system features, the proposed baseline scheduling solution, as shown by the results, is able to suggest the number of FPGAs that an online scheduler should later use for a known situation or for a situation similar to a known one.

1 Introduction

There is an ever increasing need for computing systems that are cost effective, efficient, and flexible. Multi-FPGA systems are becoming attractive for meeting all three expectations. It is possible to exploit the superior performance per Watt efficiency of FPGAs and the speed of the hardware execution, maintaining also the flexibility of changing the job performed during the system lifetime. Exploiting the full potential of multi-FPGA platforms is a very difficult problem, especially because there are no well established systematic design frameworks for the designer to rely on. The most significant advantage of these systems arises from partial dynamic reconfiguration. They can potentially be configured as necessary at any time: it is possible to configure the entire system, to reconfigure each FPGA separately or to reconfigure only a portion of a single FPGA. These

various types of adaptations are generally exploited to best match the hardware parallelism in the system with the current workload for optimal performance.

The aim of this work is to develop a design framework, which can help evaluate the performance impact of system-level design choices in a cluster of partially and dynamically reconfigurable devices, such as number of devices and the communication infrastructure deployed in the system. The proposed framework performs three main tasks: (a) Evaluate the expected performance for a given workload by varying the amount of FPGA devices in the cluster as well as considering different communication structures, (b) Determine the best initial configuration in terms of number of FPGA devices for a given workload, (c) Derive a suggested task placement and scheduling policy, which an online scheduler can further refine during run-time with various optimization criteria. The proposed tool can also provide useful feedback to determine transformations that can be applied to the task graphs, which represent target applications, in order to achieve the best performance. Furthermore, the proposed framework generates a static task placement and *baseline schedule* for run-time reconfiguration of programmable devices that can be used as an initial solution by an online scheduler to address runtime constraints. Our solution considers communication delay between FPGAs and all the features exploitable on reconfigurable devices such as *module reuse*, *configuration prefetching* and *anti-fragmentation techniques*. It can explore the system design space in different ways:

- given a different amount of FPGAs and a given number of task graphs to schedule, it tries to minimize the total execution time by parallelizing the tasks. It can help identify when it is necessary to parallelize tasks and when the communication delay will dominate the execution time. Therefore, it can help find the best architecture for a given set of task graphs;
- for a given architecture the tool can determine the baseline policy that the online scheduler should follow to increase the probability of reducing the total completion time;
- the tool can provide a policy for the online scheduler based on the number of tasks in the task graphs, the ratio between the reconfiguration time[4] of a task and its execution time, the ratio between the execution time of a task and its own data set size and in the end based on the number of different task types in the set of task graphs.

Section 2 presents the related work. Section 3 provides a description of the problem, including the architectural model on which the proposed tool has been based. Section 3.2 describes the proposed heuristic scheduler, which is embedded into the proposed framework. Section 4 presents our experimental results. Finally, the conclusions regarding the proposed approach will be summarized in Section 5.

[4] Computed as a linear function of the bitstream dimension

2 Related Work

For a single FPGA case, several approaches have been proposed to accomplish the online scheduling problem for dynamic reconfiguration. Some of these solutions also involve a *design-time component*. The approach proposed in [1] considers the scheduling problem of multiple applications on a runtime partial reconfigurable architecture as constituted by two phases: one at design-time and one at run-time. The design-time scheduler explores the design space for each task and generates a small set of schedules with different energy-performance trade-offs. A complete methodology for scheduling and placing tasks at runtime onto partially dynamically reconfigurable FPGA-based systems was explored also in [2]. The scheduling problem was presented for both the 1D and 2D reconfiguration models, proposing two heuristics, the horizon and the stuffing techniques, [3], to tackle them.

Most of the works proposed for multi-FPGA systems focus on static schedulers. These schedulers take as input a task graph of an application and they partition it temporally and spatially, [4] [5]. In [6] the authors first divide the application in time partitions and place each one of them onto the multi-FPGA architecture by leveraging a RTL partitioning algorithm. This work does not consider runtime reconfiguration, but just a static system with just one application executed. Other approaches focus on online schedulers directly, [7] [8], without taking into account any static baseline scheduler. *Module reuse* and *configuration prefetching* are considered in a very few cases.

3 Proposed Approach

The reconfiguration process for FPGA devices requires a specific configuration file called *bitstream*. If the bitstream is designed to change the whole FPGA area behavior, it is called total bitstream. Otherwise, if it configures only a portion, it is called partial bitstream. In this work partial dynamic reconfiguration is taken into account, thus, partial bitstreams are used.

The main characteristic of bitstreams is that they have a correlation with the operation they implement: once the bitstream is defined, the operation is defined too. On the other hand, given an operation, there could exist more than one bitstream implementing it.

It is possible to assign to each bitstream an attribute called *type* used to identify the operation implemented, the area occupied on the target architecture, and the time needed for configuration. Let us now define a set of reconfiguration features that have to be taken into account to define the scheduler. *Module reuse* means that two tasks of the same type have the possibility to be executed exactly on the same module on board, with a single configuration at the beginning. The *deconfiguration policy* is a set of rules used to decide when and how to remove a module from the FPGA. *Anti-fragmentation techniques* avoid the fragmentation of the available space trying to maximize the dimension of free connected areas. *Configuration prefetching* means that a module is loaded onto the FPGA as soon as possible in order to hide its reconfiguration time as much as possible.

3.1 Architectural Setup

Figure 1 (a) presents the logic view of the FPGA architecture on which our scheduling approach has been validated. This logic model has been implemented on a real architecture, see Figure 1 (b), where it is possible to distinguish between two distinct areas: a static and a reconfigurable one. The static side includes a

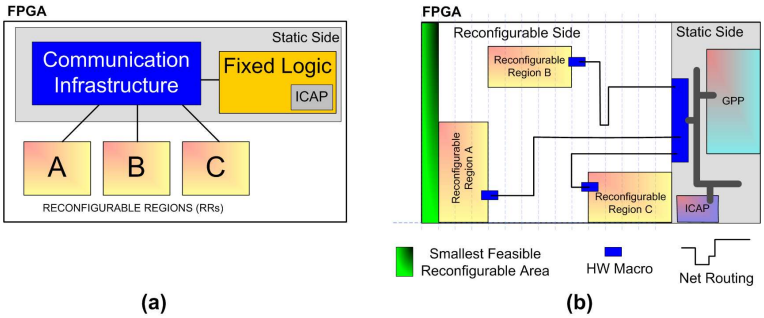


Fig. 1. Self dynamic reconfigurable architecture views: (a) Logical view (b) Physical view.

General Purpose Processors, GPP, used to execute the reconfiguration management. On the other hand, the reconfigurable area can be seen as a set of reconfigurable slots used to map the desired modules. The target architecture exploits a mono-dimensional reconfiguration approach and it is characterized by a communication infrastructure able to support the intra-module communication without introducing new constraints during the reconfiguration phase. The delay of this infrastructure has not been taken into account in this work because it highly depends on runtime conditions: a way to model this delay has been to augment the execution time of each task by a value depending on the amount of data it needs as input and it provides as output. At the same time, the software running on the static side is able to perform reconfigurations: when it is needed it sends to the ICAP [9] driver the order for reconfiguring a selected bitstream in a particular position and this, using the ICAP on-chip device, will perform the reconfiguration. Since our framework only performs a static scheduling of the task graphs, the ICAP device is considered to reconfigure at the fastest possible speed. The architectural model described above is used in each FPGA of our target system. The communication infrastructures between FPGAs considered in this work are of three types:

1. task to task communication: there is a hard-wired connection between each pair of hardware modules that need to communicate. This is an ideal configuration for point to point communication, however, it is infeasible because too many pins would be needed for each FPGA. Furthermore, this communication infrastructure is not scalable with respect to the number of FPGAs. We have only used it as a baseline comparison for the other communication structures.

2. FPGA to FPGA communication: there is a bus between each pair of FPGAs. This is a feasible communication infrastructure, but it is not scalable: the number of pins required in each FPGAdepends linearly on the number of FPGAs.
3. bus based: there is a single bus on which all the FPGAs are connected. This is a feasible and scalable solution.

All the connections are expected to operate at $100Mb/sec$. The communication is controlled by the processors located on the static sides.

3.2 Exploring the Design Space and Generating a Baseline Reconfiguration Schedule

In this section, we present the details of our methodology. The pseudocode of our algorithm is shown in Algorithm 1. The proposed solution takes into account *con-*

Algorithm 1 Scheduling algorithm Pseudocode

```
sLength ← 0
t ← 1
g ← readGraph()
setALAP(g)
RNs ← getRootNodes(g)
while ∃ not scheduled tasks do
  Control possibility of reuse for available tasks in RNs
  if ∃ not scheduled tasks then
    avTask ← getFirstALAPAvailableNode(RNs)
    endT ← findEndTime(avTask,t)
    while not all the available nodes in RNs have been observed do
      if ∃ a position on the FPGAs for avTask then
        schedule avTask
        if sLength ≤ endT then
          sLength ← endT
        end if
        for all avTask child nodes chTask do
          if All chTask parents have been scheduled then
            RNs ← RNs + chTask
          end if
        end for
        Control possibility of reuse for available tasks in RNs
        avTask ← getNextALAPAvailableNode(RNs)
      end if
    end while
  end if
  t ← nextControlStep(t)
end while
```

figuration prefetching, *module reuse*, and *anti-fragmentation* techniques. It tries

to take advantage of all the features of the multi-FPGA architecture and it tries to parallelize the execution as much as possible and evaluate the performance for a given system configuration.

The backbone of the framework is the static scheduler, which is built on top of a list based approach. First of all it performs an infinite-resource scheduling in order to sort the task set $\mathcal{S}$ by increasing ALAP[5] values. This phase is performed for all the task graphs considered in a particular instance of the problem. Then, the scheduler builds a subset RN with all tasks having no predecessor. In the scheduling algorithm, RN will be updated at each step so as to include all tasks whose predecessors have all been already scheduled (*available tasks*). As long as there is at least one unscheduled task, the algorithm performs the following operations. First it scans the available tasks in increasing ALAP order to determine those that can reuse the modules currently placed on the FPGAs. Each time this occurs, a task S is placed in the position k which hosts a compatible module and is the farthest from the center of the considered FPGA. The *farthest placement* criterion is an anti-fragmentation technique, that aims at favoring future placements, as it is usually easier to place large modules in the center of the FPGA [10]. We adopted this technique for both first module configuration and module reuse. A task will exploit module reuse in the FPGA already containing the input pins of the task graph containing it only if its final termination time will be the lowest one. If this does not happen, the scheduler will place the task in another FPGA trying to reach a better termination time.

In this way our baseline scheduler finds a suitable number of FPGAs for a particular input set. For the same reason it would be possible to ignore the reuse of a module if a completely new module leads to a better local solution. All these considerations are addressed by the scheduler, taking into account also the latency needed to transf data between FPGAs. Unused modules can be present on the FPGAs because the scheduler adopts *limited deconfiguration*. Modules are left intact on the FPGA until other tasks require their space, in order to increase the probability of reuse. The scheduler selects tasks based on increasing ALAP time, but it allows out of order scheduling. First of all it looks for module reuse opportunities and places all the tasks that can be reused. After this phase, it searches, again in increasing ALAP time, for tasks to be placed. In this last step, if a task cannot fit in the system at the current time it will be delayed and subsequent tasks will be considered to be scheduled. Thus, when no further reuse is possible, the scheduler scans the *available tasks* in increasing ALAP order to determine those which can be placed on the FPGAs at the current time step. The placement is feasible when sufficient space is currently free or it can be freed by removing an unused module. Another constraint taken into account by the scheduler is the possibility of configuring each time at most one module on each FPGA. No more than one reconfiguration (no more that one new module) per time step can be performed on a single FPGA. The position for a task S is chosen once again by the *farthest placement* criterion. There might be an interval between the end of the reconfiguration and the beginning of the execution of a task: this

[5] ALAP stands for As Late As Possible

is done to exploit *configuration prefetching*. This allows *invisible* data transfer and increase the possibility of parallel execution. When all possible tasks have been scheduled, the set of available tasks RN is updated. Finally, the current time step is updated by replacing it with the minimum between the first time step in which a reconfigurator device is available and the minimum time in which a task terminates its execution.

4 Results

Partial dynamic reconfiguration is one of the key features that make FPGAs unique devices, offering degrees of freedom not available in other akin technologies and in some cases pushing FPGA-based solutions towards the standard application platform e.g., *network controller* capable of handling the TCP and UDP protocols by exploiting partial reconfiguration [11], data mining applications [12], cryptographic system [13]. In all these kinds of applications *configuration prefetching* [14], *module reuse* [15], and *anti-fragmentation* techniques, combined with a multi-FPGA architecture can provide interesting and promising improvements trying to parallelize the execution as much as possible and evaluate the performance for a given system configuration, as proposed in this work. The proposed framework has been tested on a large set of task graphs to evaluate its behavior and the policies it provides for the online scheduler. The task graphs represent data mining applications from the *NU-MineBench* suite [12]:

1. distance application: it receives as inputs two sets of data of equal size and calculates the distance between them;
2. variance application: it receives as input a single set of data and calculates the mean and the variance among the whole data set;
3. variance1 application: it receives as input a single set of data and calculates the mean and the variance among the whole data set. The tasks graph is different than the former variance application, because it involves different task types.

Task graphs for these applications have been extracted from VHDL descriptions for each benchmark kernel. The common feature of the tasks in this application set is that they generally have very short execution time while their reconfiguration time is very high. Furthermore, the communication delay between tasks is 8 or 16 times larger than the execution time. The task graphs of these applications increase in size according to the data size they process. *distance* and *variance1* have some task types in common, furthermore, also *variance* and *variance1* have some task types in common.

The results of our baseline schedule are shown in Table 1. We observe that if the number of task graphs is close to but not more than the number of FPGAs in the system, the schedule length for the feasible communication style *C3* is longer for no more then the 26% with respect to *C1*. If too few task graphs or too many task graphs are scheduled, the scheduler tries to schedule each task

Table 1. Schedule length for the applications. The first column reports the #FPGAs, 1, 2, 3, 6 and 16, used to define the underline architecture. Different workloads, #TGs, containing several task graphs varying between 1, 2, 5 and 10 have been evaluated. Furthermore three different types of communication infrastructures has been take into consideration.*C1* denotes *Task to Task communication*, *C2* denotes FPGA *to* FPGA *communication*, and *C3* denotes *Bus based communication*.

	# TGs	2			5			10		
# FPGAs	Com. Type	C1	C2	C3	C1	C2	C3	C1	C2	C3
1	VP7	981	981	981	1438	1438	1438	2129	2129	2129
	VP30	1704	1704	1704	2244	2244	2244	3018	3018	3018
2	VP7	990	1475	1475	1495	1731	1731	1829	5763	5763
	VP30	1713	1743	1743	2301	3205	3205	2955	5189	5189
3	VP7	990	1066	1507	1493	2115	1701	1893	3333	2755
	VP30	1713	1744	2051	2299	3267	2765	2462	5541	3467
6	VP7	986	1015	993	1492	1492	1827	1645	3429	6903
	VP30	1709	1738	1214	2298	2298	2595	2451	4263	7843
16	VP7	548	722	3081	1016	1016	16935	1094	1610	12287
	VP30	1150	1150	3081	1194	1217	16935	1236	2408	55431

graph on a different FPGA in order to avoid the communication delay. If a local optimization would allow the schedule of tasks from the same task graph on different FPGAs, our scheduler finds out that the communication delay needed by the results to go to one child is too high, worsening the global scheduling time. Therefore, it will try to suggest local optimizations only if the communication delay does not become too high. From the results it is possible to notice that using just one FPGA the communication delay is equal to zero and the results are very good (even 70% better than the best results computed by the other). This observation is very interesting because it leads to the conclusion that best results is to execute 10 graphs on each FPGA . In such a context, the parallelization will be very high and the power consumtion, on the contrary, will be very low. The high reconfiguration time helps to hide the communication time.

On the other hand, the reconfiguration process is a very power consuming operation. For this reason the system that will exploit our baseline scheduler will work very well, in terms of both execution time and power consumption, when the workload will reach 10 graphs for each FPGA. When the amount of data necessary to execute those graphs becomes too high and there will be the necessity of having more memory, the online schedule will follow the advices of the baseline one increasing the number of FPGAs involved in the execution process. This helps us to determine the exact amount of FPGAs needed for a particular situation. When an online scheduler further operates on the configured system, it will be very likely to encounter an input situation similar to the one predicted by the baseline scheduler and it can schedule those task graphs in an effective way. In presence of dynamically reconfigurable multi-FPGA systems, the online scheduler may change the shape of the system itself. A suitable communication infrastructure for a system using our baseline scheduler starts at the beginning

with a single bus connected to all the FPGAs. For every N FPGAs there is a bridge that allows the uniform connection of the bus. Each bridge allows the bus to be separated into two completely independent ones. The number N has to be chosen, analysing different input benchmarks. From the results of this analysis it is possible to select the minimum number of FPGAs needed for a general input set. Exploiting this kind of hardware it is possible to increase the parallelism of the system, processing different input sets in different locations of the system.

A more advanced way of dealing with this separation concept can be implemented with reconfigurable connections. The connections are provided by FPGAs that can be reconfigured at runtime. Once the online scheduler has selected the amount of FPGAs needed by a particular input set, and as a result partitioned the system, the communication infrastructure between those FPGAs will be modified. Following the suggested policy by the baseline scheduler, the online scheduler can provide FPGA *to* FPGA *communication* for those FPGAs. If this communication structure still ends up to be too expensive, it is possible to design in advance, at compile time, a mesh or crossbar communication infrastructure for a general situation. Clearly this infrastructure needs to be easily extendable and placeable onto the FPGAs providing the communication.

Our baseline solution suggests the number of FPGAs that an online scheduler should later use for a known situation or for a situation similar to a known one. In order to choose which FPGAs will be really selected at execution time, the online scheduler may rely on communication overload, temperature of the devices, and many other criteria. Furthermore, by using our baseline scheduler it is possible to save power in situations of a low workload: the FPGAs that are not needed can be turned off. When they are needed again they will be turned on. This same principle can be applied when certain devices have reached high operating temperatures. In this case a reconfigurable communication infrastructure can help the system and the scheduler: the system shape will change but the execution will not be stopped or degraded significantly.

5 Conclusions

The goal of this work is to develop a reconfiguration aware design framework to support an online reconfiguration scenario. The proposed solution evaluates different system configurations and generates suggested scheduling policies for the online scheduler based on the specific instance of the considered problem. Section 4 shows the results obtained and it is clear how different choices can affect an online scenario. Being able to evaluate them with a systematic tool is highly beneficial and improves the efficiency of the entire design cycle. As a future work, we are planning to develop an online scheduler capable of using the suggested policies by the previous phases. Furthermore, it is possible to try to obtain better results: when two or more task graphs can be created starting from the same application, each one with different task types. In that case, it is possible to utilize the generated baseline schedule to find out which representation of the same application is the best choice for a given architecture at run-time.

References

1. J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A reconfigurable manager for dynamically reconfigurable hardware. *Design & Test of Computers, IEEE*, 22(5):452–460, Sept.-Oct. 2005.
2. C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. *IEEE Transactions on Computers*, 53(11):1393–1407, Nov. 2004.
3. Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for online scheduling real-time tasks to partially reconfigurable devices. In *Proceedings of the 13rd International Conference on Field Programmable Logic and Application (FPL03)*, pages 575–584. Springer, 2003.
4. Vinoo Srinivasan and Ranga Vemuri. Task-level partitioning and rtl design space exploration for multi-fpga architectures. In *FCCM '99: Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 272, Washington, DC, USA, 1999. IEEE Computer Society.
5. Vinoo Srinivasan and Ranga Vemuri. Throughput optimization with design space exploration during partitioning for multi-fpga architectures. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, page 253, New York, NY, USA, 1999. ACM.
6. Vinoo Srinivasan and Ranga Vemuri. Task-level partitioning and RTL design space exploration for multi-FPGA architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 272–273, 1999.
7. P. Benoit, J. Becker, G. Sassatelli, L. Torres, M. Robert, and G. Cambon. Run-time scheduling for random multi-tasking in reconfigurable coprocessors. *fpl*, 2005.
8. K. Danne and M. Platzner. Periodic real time scheduling for fpga computers. In *The Third IEEE International Workshop on Intelligent Solutions in Embedded Systems (WISES'05) at Hamburg University of Technology*, 2005.
9. Xilinx. Opb hwicap product specification. Technical report, March 2004.
10. S. Banerjee, E. Bozorgzadeh, and N. Dutt. Considering run-time reconfiguration overhead in task graph transformation for dynamically reconfigurable architectures. *In FCCM'05*, 2005.
11. Aditya Prakash Chaubal. Design and implementation of an fpga-based partially reconfigurable network controller. Master's thesis, Virginia Polytechnic Institute and State University, 2004.
12. Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Jayaprakash Pisharath, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pages 31–36, 2006.
13. Javier Castillo, Pablo Huerta, Victor López, and José Ignacio Martínez. A secure self-reconfiguring architecture based on open-source hardware. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, 2005.
14. Francesco Redaelli, Marco D. Santambrogio, and Donatella Sciuto. Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 519–522, New York, NY, USA, 2008. ACM.
15. Roberto Cordone, Francesco Redaelli, Massimo Antonio Redaelli, Marco Domenico Santambrogio, and Donatella Sciuto. Partitioning and scheduling of task graphs on partially dynamically reconfigurable fpgas. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(5):662–675, 2009.

Specifying Run-time Reconfiguration in Processor Arrays using High-level language

Zain-ul-Abdin and Bertil Svensson

Centre for Research on Embedded Systems (CERES),
Halmstad University, Halmstad, Sweden.

Abstract. The adoption of run-time reconfigurable parallel architectures for high-performance embedded systems is constrained by the lack of a unified programming model which can express both parallelism and reconfigurability. We propose to program an emerging class of reconfigurable processor arrays by using the programming model of `occam-pi` and describe how the extensions of channel direction specifiers, mobile data, dynamic process invocation, and process placement attributes can be used to express run-time reconfiguration in `occam-pi`. We present implementations of DCT algorithm to demonstrate the applicability of `occam-pi` to express reconfigurability. We concluded that `occam-pi` appears to be a suitable programming model for programming run-time reconfigurable processor arrays.

1 Introduction and Motivation

The design of high-performance embedded systems for signal processing applications is facing the challenges of not only increased computational demands but also increased demands for adaptability to future functional requirements for these applications. Reconfigurable parallel architectures offer the possibility to dynamically allocate the resources during run-time, which allows the user to implement applications which adapt according to changing demands and workloads. The reconfigurable computing devices have evolved over the years from gate-level arrays to a more coarse-grained composition of highly optimized functional blocks or even program controlled processing elements, which are operated in a coordinated manner to improve performance and energy efficiency. These reconfigurable processor arrays are well suited for streaming applications that have highly regular computational patterns.

However, developing applications that employ massively parallel reconfigurable architectures poses several challenges. Traditionally, system developers have either used low-level proprietary languages or relied on programming in C and the use of advanced synthesis tools and automatic parallelization techniques; however, the latter techniques lag in terms of achieved run-time performance. Moreover, existing tools mainly support reconfiguration of the complete device, thus allowing changes in the hardware only at a relatively slow rate. The procedural models of imperative languages, such as C and Pascal, rely on sequential

control flow because these languages were originally designed for sequential computers with unified memory system. Applying them for arrays of reconfigurable processing units result in limited extraction of instruction level parallelism, leading to inefficient use of available hardware and increased power consumption.

We propose to use a concurrent programming model that allows the programmer to express computations in a productive manner by matching it to the target hardware using high-level constructs. Portability across different hardware resources is provided by means of a compiler. `Occam` is a programming language based on the Communicating Sequential Processes (CSP) [1] concurrent model of computation. However, CSP can only represent a static model of the application, where processes synchronize communication over fixed channels. In contrast, the pi-calculus [2] allows modeling of dynamic constructions of channels and processes, which enables the dynamic connectivity of networks of processes. Thus, `occam-pi` [3], combining CSP with pi-calculus, seems to be an interesting approach to programming of run-time reconfigurable systems.

In earlier work, we have demonstrated the effectiveness of generated code from the `occam-pi` language for the Ambric [4] array of processors [5]. In this paper, we will also be focusing on expressing the reconfigurability of the underlying hardware in a programming model by relying on the concepts of mobility introduced in the pi-calculus. The target architecture for our first proof of concept implementations is the Ambric fabric of processors and we believe that the use of `occam-pi` as a unified programming language is suitable for other reconfigurable architectures such as PACT XPP [6] and ElementCXI programmable device [7]. We present the results of streaming DCT algorithm implementation.

2 `Occam-pi` language

The `occam` language [8] is based on the CSP process algebra with well-defined semantics and is a suitable source language because of its simplicity, minimal run-time overhead and power to express parallelism. `Occam` has built in semantics for concurrency and interprocess communication. The communication between the processes is handled via channels using message passing, which helps in avoiding interference problems.

`Occam-pi` [3] can be regarded as an extension of `occam` to include the mobility features of the pi-calculus [2]. The mobility feature is provided by the dynamic asynchronous communication capability of the pi-calculus, which is useful when creating a network of processes that changes its configuration at run-time.

2.1 Basic Constructs

The hierarchical modules in `occam` are composed of procedures and functions. The primitive processes provided by `occam` include assignment, input process (`?`), output process (`!`), skip process (`SKIP`), and stop process (`STOP`). In addition to these there are also structural processes such as sequential processes (`SEQ`), parallel processes (`PAR`), `WHILE`, `IF/ELSE`, `CASE`, and replicated processes [8].

A process in occam contains both the data and the operations it is required to perform on the data. The data in a process is strictly private and can be observed and modified by the owner process only. In contrast, in occam-pi the data can be declared as MOBILE, which means that the ownership of the data can be passed between different processes.

2.2 Language Extensions to Support Reconfigurability

In the following section, we will describe the semantics of the extensions in the occam-pi language such as channel direction specifiers, mobile data, dynamic process invocation, and process placement attributes. These extensions are used to express the reconfiguration of hardware resources in the programming model.

Channel Direction Specifier: The channel type definition has been extended to include the direction specifiers, Input (?) and Output (!). Thus a variable of channel type refers to only one end of channel. A channel direction specifier is added to the type of a channel definition and not to its name. Based on the direction specification, the compiler performs its usage checking both outside and within the body of the process. Channel direction specifiers are also used when referring to channel variables as parameters of a process call.

Mobile Data: The assignment and communication in classical occam follows the copy semantics, i.e., for transferring data from the sender process to the receiver both the sender and the receiver maintain separate copies of the communicated data. The mobility concept of the pi-calculus enables the movement semantics during assignment and communication, which means that the respective data has moved from the source to the target and afterwards the source loses the possession of the data. In case the source and the target reside in the same memory space, then the movement is realized by swapping of pointers, which is secure and no aliasing is introduced.

In order to incorporate mobile semantics into the occam language, the keyword MOBILE has been introduced as a qualifier for data types [9]. The definition of the MOBILE types is consistent with the ordinary types when considered in the context of defining expressions, procedures and functions.

Dynamic Process Invocation: For run-time reconfiguration dynamic invocation of processes is necessary. In occam-pi concurrency can be introduced by not only using the classical PAR construct but also by dynamic parallel process creation using forking. Forking is used whenever there is any requirement of dynamically invoking a new process which can execute concurrently with the dispatching process. In order to implement dynamic process creation in occam-pi, two new keywords FORK and FORKING, are introduced [10]. The scope of the forked process is controlled by the FORKING block in which it is being invoked.

The parameters that are allowed for a forked process are:

- `VAL` data type: whose value is copied to the forked process.
- `MOBILE` data type and channels of `MOBILE` data type: which are moved to the forked process.

The parameters of a forked process follow the communication semantics instead of the renaming semantics adopted by parameters of ordinary processes.

Process Placement Attribute: The placement attribute is essential in order to identify the location of the components that will be reconfigured in the reconfiguration process, and it is inspired by the *placed parallel* concept of `occam`. The qualifier `PLACED` is introduced in the language followed by two integers to identify the location of the hardware resource where the associated process will be mapped. The identifying integers are logical numbers which are translated by the compiler to the physical address of the resource.

3 Compilation Methodology

In this section we will give a brief overview of the Ambric architecture before presenting a method for compiling `occam-pi` programs to reconfigurable processor arrays. The method is based on implementing a compiler backend for generating native code.

3.1 Ambric Architecture and Programming Model

Ambric is an asynchronous array of so called brics, each composed of two pairs of Compute Unit (CU) and RAM Unit (RU) [4]. The CU consists of two 32-bit Streaming RISC (SR) processors, two 32-bit Streaming RISC processors with DSP extensions (SRD), and a 32-bit channel interconnect for interprocessor and inter CU communications. The RU consists of four banks of RAM along with a dynamic channel interconnect to facilitate communication with these memories. The Am2045 device has a total of 336 processors in 45 brics.

The architecture was designed to support a structured object programming model. Using the proprietary tools the individual objects are programmed in a sequential manner in a subset of the java language, called `aJava` or in assembly language [11]. Objects communicate with each other using hardware channels without using any shared memory. Each channel is unidirectional, point-to-point, and has a data path width of a single word. The individual software objects are then linked together using the proprietary language called `aStruct`.

3.2 Compiler for Ambric

When developing a compiler for Ambric, we have made use of the frontend of an existing *Translator from Occam to C from Kent* (Tock) [12]. The compiler is divided into front end, which consists of phases up to machine independent

optimization, and back end, which includes the remaining phases that are dependent upon the target machine architecture. In this case, we have extended the frontend for supporting `occam-pi` and developed a new backend, targeting Ambric, thus generating native code in `aJava` and `aStruct`.

In the following we give a brief description of the modifications that are incorporated in the compiler to support the language extensions of `occam-pi`, introduced to express reconfigurability.

Frontend: The frontend of the compiler, which analyzes the source code in `occam-pi`, consists of several modules for parsing and syntax and semantic analysis. We have extended the parser and the lexical analyzer to take into account the additional constructs for introducing mobile data types, dynamic process invocation and process placement attributes. We have also introduced new grammar rules corresponding to these additional constructs to create Abstract Syntax Trees (AST) from tokens generated by the lexical analysis. Steps for resolving names and type checking are performed at this stage. The frontend also tests the scope of the forking block and whether the data passed to a forked process is of `MOBILE` data type, thus fulfilling the requirement for communication semantics.

In order to support the channel end definition, we have extended the definition of channel type to include the direction whenever a channel name is found followed by a direction token, i.e., '`?`' for input and '`!`' for output. In order to implement the channel end definition for a procedure call, we have used the *DirectedVariable* constructor to be passed to the AST whenever a channel end definition is found in the procedure call.

Ambric backend: The Ambric backend is further divided into two main passes. The first pass generates declarations of `aStruct` code including the top-level design, the interface and binding declarations for each of the composite as well as primitive objects corresponding to the different processes specified in the `occam-pi` source code. Before generating the `aStruct` code, the backend traverses the AST to collect a list of all the parameters passed in procedure calls specified for processes to be executed in parallel. This list of parameters, along with the list of names of procedures called is used to generate the structural interface and binding code for each of the parallel objects.

The next pass makes use of the structured composition of the `occam` constructs, such as `SEQ, PAR,` and `CASE`, which allows intermingling processes and declarations and replication of the constructs like (`SEQ, PAR, IF`). The backend uses the genStructured function from the generateC module of the C backend to generate the aJava class code corresponding to processes which do not have the `PAR` construction. In case of the `FORK` construct, the backend generates the background code for managing the loading of the successive configuration from the local storage and communicating it to the concerned processing elements.

4 Implementing the Reconfigurable Framework

Let us explain how the `occam-pi` language can be applied for the realization of dynamic reconfiguration of hardware resources. The reconfiguration process based on its specification in the `occam-pi` language can be performed by taking into account a work farm design approach [13] as shown in Figure 1.

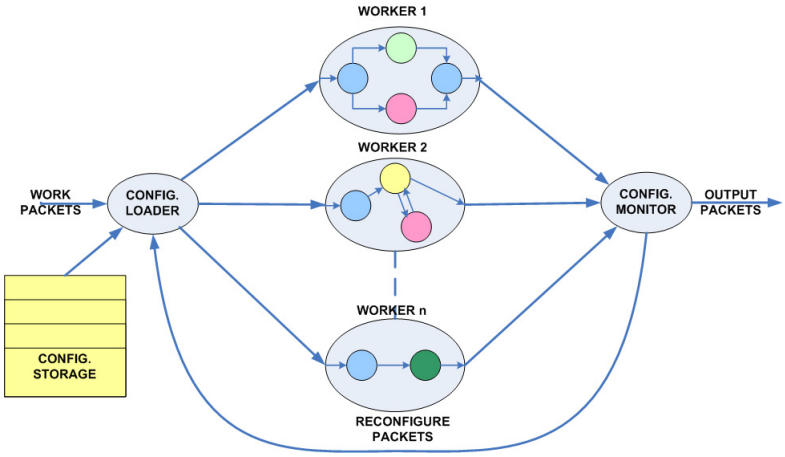


Fig. 1. Framework of Reconfigurable Components.

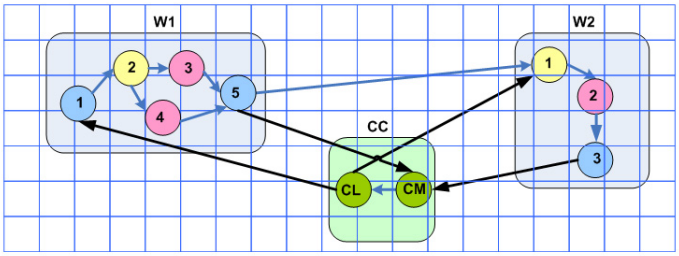


Fig. 2. Reconfigurable Components Mapping.

A worker is a specific area of hardware executing a particular task. The task can either consist of one process, or it can be composed of a number of processes which are interconnected according to their communication requirements. A worker can either occupy one processing element or be mapped to a collection of processing elements. Each worker can have multiple inputs and outputs, but in Figure 1, we show only the connections used during the reconfiguration

process. The reconfiguration process is controlled by a configuration loader and a configuration monitor. In Ambric, both the loader and the monitor processes are mapped to some of the processors in the array, but in other cases the reconfiguration management processes can instead be mapped to dedicated hardware. The configuration loader has a local storage of all the configurations in the form of pre-compiled object codes. Two types of packets are communicated from the loader to the workers: work packets and configuration packets. The former consist of the data to be processed and the latter contain the configuration data. Both types of packets are routed to different workers based on either the worker ID or some other identifier. Each worker has a small kernel to differentiate between the incoming packets based on their header information. Whenever a worker finishes its task, it returns control to its input kernel after sending a reconfiguration request packet indicating that the particular worker has completed its task and is ready to be reconfigured to a new configuration. The configuration monitor observes the reconfiguration request and issues it to the configuration loader, which forks a new worker process to be reconfigured in place of the existing worker. The location of the worker is specified by the placement attribute, which consists of two integers. The first integer relates to the identification of worker and the second integer identifies the individual processing element within the worker, as shown in Figure 2. The configuration data is communicated in the form of a configuration packet that includes the instruction code for the individual processing elements. The configuration packet is passed around all the processing elements within the worker, where each processing element extracts its own configuration data and passes the rest to its adjacent neighbor.

5 1D-DCT Case Study

In this section, we present and discuss the implementation of the One-Dimensional Discrete Cosine Transform (1D-DCT), which is developed in `occam-pi` and then ported to Ambric using our compilation platform. DCT is a lossy compression technique used in video compression encoders to transform an $N \times N$ image block from the spatial domain to the DCT domain [14].

We have used a streaming approach to implement the 1D-DCT algorithm, and the dataflow diagram of an 8-point 1D-DCT algorithm is shown in Figure 3. When computing the forward DCT, an 8×8 samples block is input on the left, and the forward DCT vector is received as output on the right. The implementation is based on a set of filters which operate in four stages, and two of these stages are reconfigured at run-time based on the framework presented in Section 4. The reconfiguration process is applied between these stages in such a way that when the first two stages are completed, the next two stages of the pipeline are configured on the same physical resources, thus reusing the same processors. The function of *'worker1'* is described by a process named *'task1'*, which consists of the first two stages of the DCT algorithm that are mapped to two individual SRD processors of *'compute-unit 1'*, as they are invoked in a parallel block. The implementation of the configuration loader as expressed in the `occam-pi` pro-

gram is shown in Figure 4a, which has one output channel-end *'cnf'* of mobile type because it is used to communicate the configuration data (Note that Figure 4 only shows the code related to configuration management, not the complete code). The implementation of the configuration monitor is shown in Figure 4b. The configuration monitor will wait until it receives a `'RECONFIG'` message from the worker, which indicates that the worker has finished performing its task and is ready to be reconfigured. The monitor will generate a reconfiguration request message along with the logical address of the resource to be reconfigured, to the configuration loader. The configuration loader, upon reception of a reconfiguration request, will issue a `FORK` statement as shown in Figure 4a, which includes the name of the process to be configured in place of *'worker1'*, its corresponding configuration data, and its associated channels. The new forked *'task2'* process has the same placement attributes as those of *'task1'* as shown in Figure 4c, to determine the mapping locations. The newly configured *'task2'* process consists of the last two stages of the DCT algorithm.

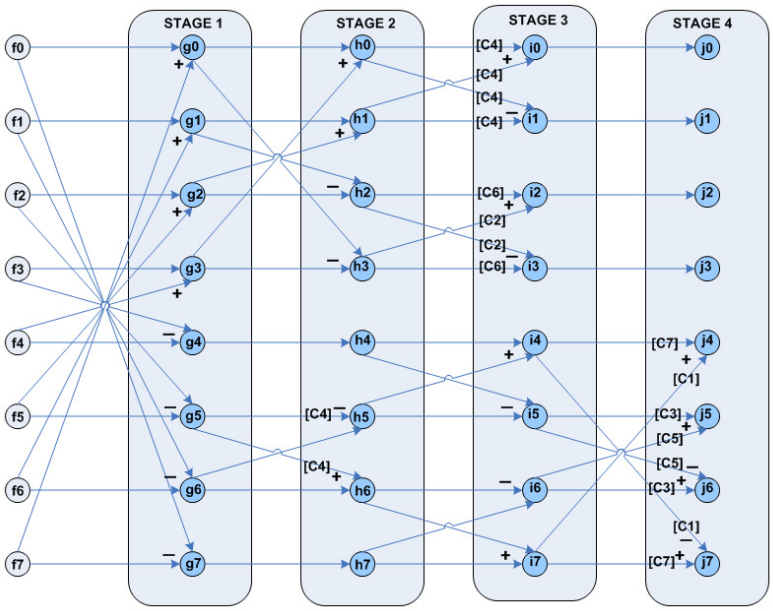


Fig. 3. Dataflow diagram for 1D-DCT.

5.1 Implementation Results and Discussion

We now present the results of the reconfigurable 1D-DCT which is implemented by using the framework presented in Section 4. Our aim here is to demonstrate

```
PROC loader(CHAN INT inp?, CHAN MOBILE INT cnf!,
              CHAN INT ack?)
  INT cstatus, value, id:
  MOBILE [100]INT config:
  CHAN MOBILE INT cnf:
  CHAN INT res:
  VAL RECONFIG IS 255:
  SEQ
    FORKING
      WHILE TRUE
        SEQ
          inp ? value
          cnf ! value
          ack ? cstatus
          IF
            cstatus = RECONFIG
              SEQ
                ack ? id
                IF
                  id = 1
                    FORK task2(config, cnf?, res!)
                  id = 2
                    ...
:
```

(a)

```
PROC monitor(CHAN INT res?, CHAN INT ack!,
                            CHAN INT outp!)
  INT status:
  VAL RECONFIG IS 255:
  WHILE TRUE
    SEQ
      res ? status
      IF
        status = RECONFIG
          ack ! RECONFIG
        status <> RECONFIG
          outp ! status
:
```

(b)

```
PROC task2(MOBILE [100]INT config,
           CHAN MOBILE INT cnf?, CHAN INT res!)
  CHAN INT ch:
  PLACED PAR
    PROCESSOR 1,1
      stage3(config,cnf?,ch!)
    PROCESSOR 1,2
      stage4(config,ch?,res!)
:
```

(c)

Fig. 4. (a) Configuration Loader, (b) Configuration Monitor, (c) Worker Process.

the applicability of the programming model of `occam-pi`, together with the proposed framework for expressing reconfigurability, thus we do not claim to achieve efficient implementations with respect to performance.

The coarse-grained parallelized DCT is implemented in a four stage pipeline and earlier results reveal that the 4-stage DCT implementation that uses four SRD processors, takes 1340 cycles to compute 64 samples of 1D-DCT. This time includes the time consumed during communication stalls between different stages. The computation of the same amount of samples performed by two SRD processors, which are reconfigured to perform the different stages successively takes 2612 cycles, which includes the cycle count for the reconfiguration process, which is 550 cycles. The number of instruction words to be stored in the local memories of individual processors are 97. The SRD processor takes 2 cycles to write one memory word in its local memory, thus the memory writing time is a significant part of the overall reconfiguration time. The reconfiguration process is controlled in such a way that the time taken by the two processors to update their instruction memories is partially overlapped. The above-mentioned stalls can be eliminated in the reconfigurable two-processor implementation. This time is instead used for the reconfiguration management. The results also show that the reconfiguration time is one fifth of the overall time of computation, which depict the feasibility of the approach.

6 Conclusions and Future Work

We have presented our concept about using the mobility features of the `occam-pi` language and the extensions in language constructs to express run-time reconfig-

urablity in processor arrays. The ideas are demonstrated by a working compiler, which compiles `occam-pi` programs to native code for an array of processors, Ambric. A reconfigurable component framework is presented, which is adopted to control the reconfiguration of dynamic processes with minimal disruption of the rest of the system. An application study is also performed and the results show two different ways to implement the 1D-DCT algorithm, which are compared on the basis of performance versus resource requirements.

We believe that the compositional nature of process-oriented parallel programming enhances the programmer's understanding when developing multimedia signal processing systems. The properties of exposing parallelism and separating communication from computation help in the task of parallelization, and the support for expressing reconfigurability enables effective use of resources as demonstrated by the cycle-count results of 1D-DCT algorithm.

In the future we plan to perform more application studies using the compiler platform and demonstrate the usefulness of the approach in implementing runtime reconfiguration of radar signal processing applications.

References

1. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall. (1985).
2. Milner, R., Parrow, J., and Walker, D.: A Calculus of Mobile Processes, Part I. Information and Computation, 100, (1989).
3. Welch, P.H., and Barnes, F.R.M.: Communicating mobile processes: Introducing occam-pi. Lecture Notes in Computer Science, Springer Verlag. 175-210 (2005).
4. Jones, A. M., and Butts, M.: TeraOPS hardware: A new massively-parallel MIMD computing fabric IC. *In Proceedings of IEEE Hot Chips Symposium.* (2006)
5. Zain-ul-Abdin, and Svensson, B.: Using a CSP based Progamming Model for Reconfigurable Processor Arrays. *ReConFig'08.* (2008)
6. XPP Processor Overview. "http://www.pactxpp.com/main/index.php",[Online; accessed 13th March, 2008]
7. ECA-64 Device Architecture Overview. "http://www.elementcxi.com/technology1.html", [Online; accessed 8th November, 2009]
8. Occam® 2.1 Reference Manual, SGS-Thomson Microelectronics Limited. (1995)
9. Welch, P.H., and Barnes, F.R.M.: Prioritised dynamic communicating processes: Part I. Communicating Process Architectures, IOS Press. 321-352 (2002).
10. Welch, P.H., and Barnes, F.R.M.: Prioritised dynamic communicating processes: Part II. Communicating Process Architectures, IOS Press. 353-370 (2002).
11. Butts, M., Jones, A. M., and Wasson, P.: A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. *FCCM '07.* 55-64 (2007).
12. Tock: Translator from Occam to C by Kent. "https://www.cs.kent.ac.uk/research/groups/sys/wiki/Tock", [Online; accessed 8th July, 2008]
13. Butts, M., Budlong, B., Wasson, P., and White, E.: Reconfigurable Work Farms on a Massively Parallel Processor Array. *FCCM '08.* 206-215 (2008)
14. Xilinx: XAPP610: Video compression using DCT. "http://direct.xilinx.com/bvdocs/appnotes/xapp610.pdf",[Online; accessed 16th September, 2006]

SESSION 4: APPLICATIONS

78

Speaker Identification Classification on FPGA

Phak Len Eh Kan[1], Tim Allen[1] and Steven F. Quiqley[1],

[1] School of Electronic, Electrical and Computer Engineering,
University of Birmingham, Edgbaston,
Birmingham B15 2TT,
United Kingdom
{exp692, s.f.quigley}@bham.ac.uk

Abstract. Biometric-based speaker identification is a method of identifying persons from their voice. Speaker-specific characteristics exist in speech signals due to different speakers having different resonances of the vocal tract and these can be exploited by extracting feature vectors such as Mel frequency cepstral coefficients (MFCCs) from the speech signal. A well-known statistical model, Gaussian Mixture Model (GMM) then models the distribution of each speaker's MFCCs in a multidimensional acoustic space. The GMM-based speaker identification system has features that make it promising for hardware acceleration. This paper describes the classification hardware implementation of a text-independent GMM-based speaker identification system. A speed factor of 90 was achieved compared to software-based implementation on a standard PC.

Keywords: Speaker Identification, MFCC, GMM, Field Programmable Gate Array (FPGA).

1 Introduction

Speaker recognition is an important branch of speech processing. It is the process of automatically recognizing who is speaking by using speaker-specific information included in the speech waveform and receiving increasing attention due to its practical value. It has applications ranging from police work to automation of call centres. Speaker recognition can be classified into speaker identification (discovering identity) and speaker verification (authenticating a claim of identity). A closed-set speaker identification system selects the speaker in the training set who best matches the unknown speaker. Open-set speaker identification allows for the possibility that the unknown speaker may not exist in the training set; thus an additional decision alternative is required for the unknown speaker who does not match any of the models in the training set [1].

Most speaker identification systems have been based on software running on a single microprocessor. The problem with software is that its sequential operation means that it can be slow for high throughput real time signal processing applications. Improvements in FPGA technology and design tools have recently introduced a new option for Digital Signal Processing (DSP) applications that require high performance and low development costs. The latest FPGAs have a very high logic capacity and contain embedded Arithmetic Logic Units (ALUs) to optimize signal processing

performance [2]. FPGAs have been used in many areas to accelerate algorithms that can make use of massive parallelism, improving the flexibility and reducing costs as well as time to market. FPGAs are also able to exploit pipelining and parallelism in a much more thorough way that can be done with parallel computers using general-purpose microprocessors.

In this paper we present results for the implementation of speaker identification classification on a platform consisting of an Alpha Data RC2000 PCI card equipped with a single Xilinx Virtex-II XC2V6000 FPGA. The goal was to achieve a system that can process a large number of voice streams simultaneously in real time.

2 Speaker Identification System

A block diagram shown in figure 1 is the top-level system designed to implement text independent speaker identification. The input speech is sampled and converted into digital format. Feature vectors are extracted from the input speech in the form of MFCCs. The system then branches into two separate phases; *training* and *classification*. In the training phase, each registered speaker has to provide samples of their speech so that the system can train reference models for that speaker, whilst in the classification phase the input speech is matched with the stored reference models and identification is made.

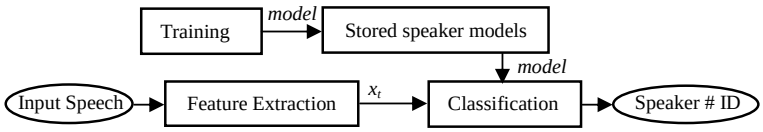


Fig. 1. Top-level structure of speaker identification system

2.1 Feature Extraction

The speech waveform is extracted into a set of features for further analysis. The speech signal is a slowly time varying signal and when it examined over a sufficient short period of time, its characteristics are fairly stationary, whilst over long periods of time the signal characteristics change to reflect the different speech sounds being spoken. In many cases, short time spectral analysis is the most common way to characterize the speech signal. Several possibilities exist for parametrically representing the speech signal for the speaker identification task, such as MFCC, Linear Prediction Coding (LPC), and others. In this work MFCCs are chosen because they are based on the perceptual characteristics of the human auditory system [3], [6].

Figure 2 shows a block diagram of the MFCC feature extraction. The digital speech signal is blocked into frames of N samples, with adjacent frames being separated by M samples. The first frame consists of the first N samples. The second frame begins M samples after the first frame, and overlaps it by N-M samples and so on. Each individual frame is windowed so as to minimize the signal discontinuity at the beginning and end of each frame. The FFT converts each frame of samples from

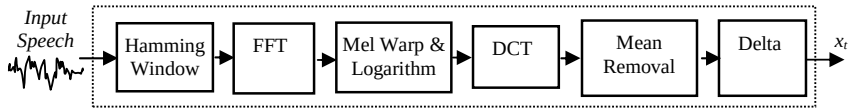


Fig. 2. MFCC feature extraction block diagram.

time domain into the frequency domain. The frequency scale is then converted from the hertz to the mel scale, using filter banks, with frequency spaced linearly at low frequencies and logarithmically at high frequencies, and the logarithm is then taken. This stage is done in order to capture the phonetically important characteristics of speech in a manner that reflects the human perceptual system. The DCT is then applied to the output to produce a cepstrum. The first 17 cepstral coefficients of the series are retained, their means are removed and their first order derivatives are computed. This results in a feature vector of 34 elements, 17 MFCCs and 17 deltas. These vectors (x_t) are then passed on to the training or classification stages.

2.2 Gaussian Mixture Models (GMM)

The GMM forms the basis for both the training and classification processes. This is a statistical method that classifies the speaker based on the probability that the test data could have originated from each speaker in the set [1], [4], [5],[7].

2.2.1 Training

A statistical model for each speaker in the set is developed and denoted by λ. For instance, speaker s in the set of size speaker S can be written as follows

$$\lambda_s = \{w_i, \mu_i, \sigma_i\} \qquad i = 1,\ldots,M \; ; \; s = 1,\ldots,S \tag{1}$$

where, w: weight, μ: mean, and σ: diagonal covariance

A diagonal covariance, σ is used rather than a full covariance matrix, Σ, for the speaker model in order to simplify the hardware design. However, this means that a greater number of mixture components will need to be used to provide adequate classification performance. The training phase consists of two steps, namely *initialisation* and *expectation maximisation (EM)*. The initialisation step provides initial estimates of the means for each Gaussian component in the GMM model. The EM algorithm recomputes the μ, σ and w of each component in the GMM iteratively. The algorithm is monotonically increasing hence each iteration provides increased accuracy in the estimates of all three parameters. The EM algorithm [1], [4], [5] are,

Posterior probability,

$$p(i \mid x_t, \lambda) = \frac{p_i b_i(x_t)}{\sum_{k=1}^{M} p_k b_k(x_t)} \tag{2}$$

New estimates of ith weight,

$$\overline{w_i} = \frac{1}{T}\sum_{t=1}^{T} p(i \mid x_t, \lambda) \tag{3}$$

New estimates of mean,

$$\overline{\mu_i} = \frac{\sum_{t=1}^{T} p(i \mid x_t, \lambda) x_t}{\sum_{t=1}^{T} p(i \mid x_t, \lambda)} \tag{4}$$

New estimates of diagonal elements of ith covariance matrix,

$$\overline{\sigma_i} = \frac{\sum_{t=1}^{T} p(i \mid x_t, \lambda)(x_t \bullet x_t)}{\sum_{t=1}^{T} p(i \mid x_t, \lambda)} - \overline{\mu_i}^2 \tag{5}$$

2.2.2 Classification

In this stage a series of input vectors are compared and a decision is made as to which of the speakers in the set is the most likely to have spoken the test data. The input to the classification system is denoted as $X = \{x_1, x_2, x_3, \ldots\ldots\ldots, x_T\}$. The rule to determine if X has come from speaker s can be stated as,

$$p(\lambda_s \mid X) > p(\lambda_r \mid X) \quad r = 1,2,\ldots\ldots S \ (r \neq s) \tag{6}$$

Therefore, for each speaker s in the speaker set, the classification system needs to compute and find the value of s that maximizes $p(\lambda_s|X)$ according to

$$p(\lambda_s \mid X) = \frac{p(X \mid \lambda_s)\, p(\lambda_s)}{p(X)} \tag{7}$$

The classification is based on a comparison between the probabilities for each speaker. If it can be assumed that the prior probability of each speaker is equal, then the term of $p(\lambda_s)$ can be ignored. The term p(X) can also be ignored as this value is the same for each speaker [1], so we are seeking the value of s that maximizes

$$p(X \mid \lambda_s) = \prod_{t=1}^{T} p(x_t \mid \lambda_s) \tag{8}$$

Practically the individual probabilities, $p(x_t \mid \lambda_s)$, are typically in the range 10^{-3} to 10^{-8}. There are 1000 test vectors with a test input of 10 seconds. When 10^{-8} is multiplied to itself 1000 times a standard computer and certainly any system implemented on an FPGA will underflow and the probability for all speakers will be calculated as zero. Thus $p(X \mid \lambda_s)$ is computed in the log domain in order to avoid this problem. The likelihood of any speaker having spoken the test data is then referred to as the log-likelihood and is represented by the symbol *L* [1].

$$L(\lambda_s) = \sum_{t=1}^{T} \ln(p(x_t \mid \lambda_s)) \tag{9}$$

The speaker of the test data is statistically chosen by the following,

$$\text{speaker} = \max_{s=1}^{S} L(\lambda_s) \tag{10}$$

3 Hardware Implementation of Speaker Identification Classification

The designed of hardware is based on working system in software. The reason for using hardware is to obtain significant speed improvements over software and allow processing of multiple voice streams on an increased population of speakers. The generation of the feature vectors for the classification stage was performed offline. Both the speaker models and feature vectors from the test data were stored in random access memory (RAM) connected to the FPGA. Equations 9, 10, 11 and 12 are used in hardware implementation.

$$p(x_t \mid \lambda_s) = \sum_{i=1}^{M} w_i b_i(x_t) \tag{11}$$

$$b_i(x) = \frac{1}{\sqrt{(2\pi)^D |\Sigma_i|}} \exp\left(-\frac{1}{2}(x-\mu_i)' \Sigma_i^{-1} (x-\mu_i)\right) \tag{12}$$

The classification phase of the speaker identification system was designed using separate datapath and control circuitry. The link between the two is through control signals and flags.

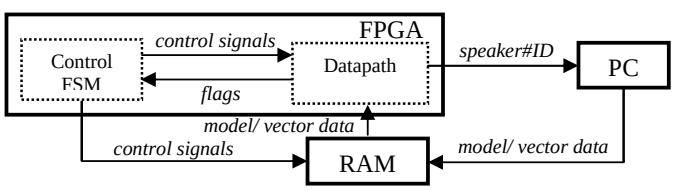


Fig. 3. Top level of speaker identification classification.

Figure 3 forms the top level overview of the speaker identification classification system and shows the link between PC, RAM and FPGA. The datapath section performs all the mathematical operations and the control system is a finite state machine (FSM) which produces control signals based on the current state and current inputs.

Figure 4 shows the datapath broken down further into its individual operations. The stage computing the natural log of the probability of each vectors having come from a particular component of a given speaker model will be repeated as many times as the area of the FPGA allows.

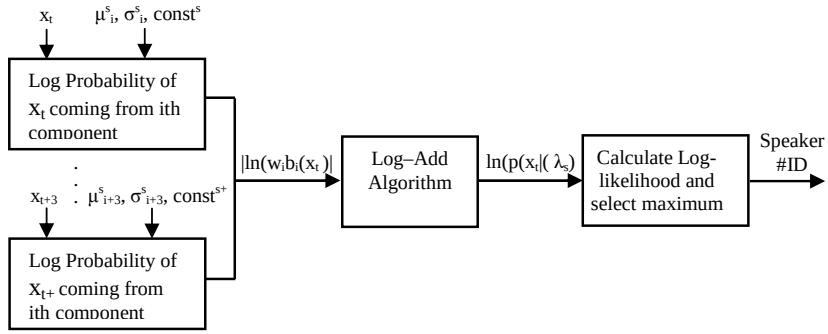


Fig. 4. Datapath represented into three main segments.

3.1 Log-Add Algorithm

The reason for starting with the middle block in figure 4 is that the changes in the GMM formulae are more easily explained with reference to this stage and these changes govern the changes in the preceding and following stages. Equation 9 requires the natural logarithm of the result for each vector from equation 11 to be computed. This is shown in equation 13.

$$L(\lambda_s) = \sum_{t=1}^{T} \ln(p(x_t \mid \lambda_s)) \qquad \textbf{(13)}$$

$$\ln(p(x_t \mid \lambda_s)) = \ln\left(\sum_{i=1}^{M} w_i b_i(x_t)\right)$$

In software implementation equation 13 is computed exactly as stated; the probability of an input vector having come from a speaker model is calculated, the logarithm is taken and then the sum is taken over all input vectors. To mirror software implementation in hardware exactly would require the computation of the exponential term in equation 12. It requires a large LUT to compute an exponential term in hardware if high accuracy is required. A large LUT table is expensive in hardware and should be avoided. Modification needed to be done to avoid calculating the exponential term. At first glance it appears that it is not possible to avoid calculating equation 12 directly because the logarithm term in equation 9 is a summation over all 32 Gaussian components and there is no obvious way to compute the logarithm of a summation without computing the sum first. However, there is an algorithm for computing the logarithm of a summation without computing the sum first [8], [9].

Equation 14 shows the basic theory behind the log-add algorithm.

$$\ln(A + B) = \ln A\left(1 + \frac{B}{A}\right) = \ln A + \ln\left(1 + \frac{B}{A}\right) \qquad \textbf{(14)}$$

where, A > B; if A<B then switch A and B in formula.

For the ln(1+B/A) term the system can calculate,

$$\ln\left(\frac{B}{A}\right) = \ln(B) - \ln(A) \tag{15}$$

A LUT can then be used to map ln(B/A) to ln(1+B/A). The LUT table required here is much smaller than the one required for calculating an exponential term. This is due to the constraint imposed on A and B. The term B/A is limited to the interval [0,1], hence the term 1+B/A is limited to the interval [1,2]. Limiting the range of numbers in this way obviously reduces the size of LUT required. All necessary computations can be performed on the logarithm of the individual elements of the summation. Hence the input to the log-add algorithm is the logarithm of each element of the summation. This is shown in equation 16. This equation is implemented in the first column of blocks shown in figure 4. It is the datapath for calculating this equation that forms the most logic resource intensive section of the datapath.

$$\begin{aligned} \ln(w_i b_i(x_t)) &= \ln(w_i) + \ln(b_i(x_t)) \\ &= \ln(w_i) - \frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln\left(\left|\Sigma_i\right|\right) - \frac{1}{2}(x_t - \mu_i)' \Sigma_i^{-1}(x_t - \mu_i) \end{aligned} \tag{16}$$

3.2 Log probability computation

The first block shown in figure 4 calculates the log of the probability of each input vector having come from the ith component in a speaker model for all components of the GMM. Due to hardware limitations it is not possible to calculate all 32 components at once. Therefore each datapath will compute equation 16 for eight (32/4) components of the GMM. Figure 5 shows the datapath for computing equation 16 for one component of the GMM. The feature vector (x_t), mean (μ), covariance (σ) and constant $\ln(w_i) - \frac{D}{2}\ln(2\pi) - \frac{1}{2}\ln\left(\left|\Sigma_i\right|\right)$ are generated during the offline training of the speaker model and are read from the RAM when required.

The datapath shown in figure 5 contains some simplifications of equation 16. These simplifications exist because the covariance matrix is diagonal. Firstly, having subtracted the mean from the feature vector each element is squared. Then the square is multiplied by the inverse of the corresponding element in the covariance diagonal. The inverse covariance is stored so that multiplication is required in hardware and not division. Finally all 34 elements are summed together. This section of the datapath just described is equivalent to $(x_t - \mu_i)' \Sigma_i^{-1}(x_t - \mu_i)$ of equation 16. The rest of the datapath is self explanatory except before squaring each element the signal can be positive or negative as the feature vectors and means can be positive or negative. After squaring each element the signal is always positive. The covariances are also always positive and hence the signal is always positive until it is multiplied by -0.5. As the constant is always negative the signal after being multiplied by -0.5 is always negative. Hence after the squaring of the signal its sign is always known and only the magnitude of the signal is stored. Therefore the output of the system is the magnitude of $\ln(w_i b_i(x_t))$ and is known to be always negative. The inputs to both multipliers are 16 bits and the output is also 16 bits. The output from the first multiplier is in fact 32

bits. If full 32 bits were kept the output from the second multiplier would be 48 bits (32 + 16) and it is not practical to pass all 48 bits to the rest of the system. The 32 bits output from the first multiplier is limited to 16 bits. The output from the second multiplier is also limited to 16 bits to reduce the word size in the rest of the system. The inputs to and outputs from both multipliers are therefore 16 bits and the multiplier operations can share the same hardware multiplier block with a multiplexer controlling the inputs.

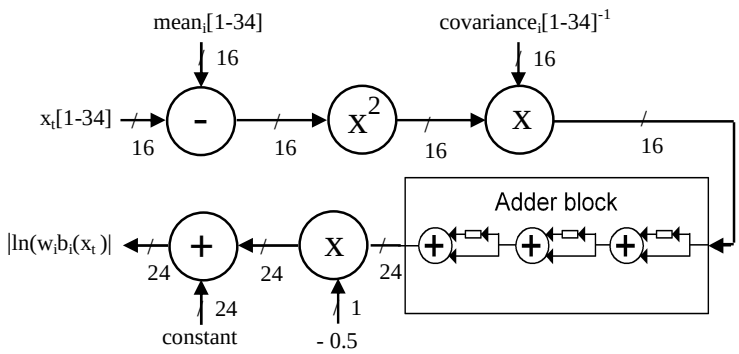


Fig. 5. Dataflow showing the calculation of equation 16.

When reducing the multiplier outputs from 32 to 16 bits if the upper 16 bits of the 32 bit output were kept there would be a significant loss of accuracy when small numbers were involved. In fact the number would often be reduced to zero. To avoid this problem the 16 bits that are kept are allowed to vary based on the first most significant bit that is logic level '1'. The outputs from both multipliers are positive hence the binary representation is unsigned and the MSB that is logic level '1' indicates the start of the number and not the sign.

3.3 Logic Resource Requirements

The most logic resource intensive section of the datapath is to compute $\ln[p(x_t \mid \lambda_s)]$. The reason for this can be seen by recognising that equation 16 must be computed once for each GMM component and by examining the dataflow diagram for computing as shown in figure 5. The limiting logic factor in the datapath is the multipliers. Two 16 bit multipliers are required per element of x_t. These can be multiplexed together meaning 34 multipliers are required (i.e. one per component of x_t). The standard multipliers on the XC2V6000 are 18 bit by 18 bit. If the word size was increased above 18 bits (eg: 24 bit) then three multipliers are required per element of x_t. Hence, a total of 102 multipliers are required in total. When considering that the datapath in figure 5 must be repeated once per component then for a word length of 16 bits with 32 components the required number of multipliers is 1088.

4 Results

4.1 Hardware Resources Utilization

The breakdowns of hardware resources utilization for the speaker identification classification system are presented in table 1.

Table 1. Logic resources for speaker identification classification.

Logic Resources	Numbers
Slices	23251
FFs	35104
LUTs	39452
Block RAMs	24
IOBs	295
MULT18x18	136

4.2 Accuracy

The testing of speech was carried out on the hardware system only uses a period of 5s same as the data used for the software. Table 2 restates these results for both hardware and software. The accuracy is fairly similar with the software system showing a slight improvement over the hardware system. This is to be expected as the software implementation uses full double precision accuracy. A test utterance greater than 5s should be used to achieve higher accuracy. Further accuracy improvements can also be achieved by removing segments of number speech from the speech signal [4], [10].

Table 2. Hardware and software results for testing with 5s of test utterance.

Utterance length	Software – 5 seconds	Hardware – 5 seconds
Test 1	80.77%	78.30%
Test 2	56.40%	55.20%
Test 3	68.54%	64.90%
Test 4	72.75%	69.40%
Test 5	65.62%	64.95%

4.3 Speed

The speed of the software system was measured using a speaker set of size 20. Test data from one of the speakers was used and the experiment was repeated 100 times consecutively with an average being computed over the hundred. Table 3 presents the results from the software testing along with the results from the hardware testing on the XC2V6000 board. The classification part of the speaker identification system implemented on the XC2V6000 platform is about 90 times faster than software. When considering real time implementation of speaker identification with feature vectors from one speech input being provided every 10ms the software system would only be able to perform calculations on five speaker models.

Table 3. Hardware and software timing parameters.

Parameter	System	
	Hardware – XC2V6000	Software - Matlab
Data Transfer	16.8ms for 500 vector transfer	Not applicable
Classification	0.8ms per vector for speaker set of size 20	69ms per vector for speaker set of size 20

5 Conclusion

The analysis of hardware versus software has demonstrated that speaker identification classification is about 90 times faster on hardware. This means that the hardware system is capable of processing 90 times more audio streams in real time than could be done in a PC. The limiting factor for implementation on the XC2V6000 device is the number of multipliers and its maximum clock speed. The future improvement step will be generating a real time implementation of open-set text independent speaker identification with greater length of speech utterance. The design will also be optimized to reduce latency and to optimize use of memory on the platform of Xilinx Virtex-IV XC4LX160-FF1148 grade 11 FPGA.

References

1. Reynolds, D., Rose, R.: Robust Text-independent Speaker Identification using Gaussian mixture speaker models. IEEE Trans. on Speech and Audio Processing vol. 3, no. 1, 72–83 (1995)
2. Thompson, A.: Hardware evolution: Automatic design of electronic circuits in reconfigurable hardware by Artificial Evolution. Springer, London; New York (1998)
3. Vergin, R., O'Shaughnessy, D., Farhat, A.: Generalized Mel Frequency Cepstral Coefficient for Large-Vocabulary Speaker-Independent Continuous-Speech Recognition. IEEE Transaction on Speech and Audio Processing, vol.7, no.5, 525-532 (1999)
4. Auckenthaler, R.: Test-Independent Speaker Identification with Limited Resources. Ph.D thesis, University of Wales (2001)
5. Holmes, J.N., Holmes, W.J.: Speech Synthesis and Recognition. London,U.K. Taylor & Francis, second edition (2001)
6. Hassan, M., Jamil, M., Rabbani, M., Rahman, M.: Speaker Identification using Mel Frequency Cepstral Coefficients. 3rd International Conference on Electrical & Computer Engineering, 565-568 (2004)
7. Reynolds, D., Quateri, T., Dunn, R.: Speaker Verification Using Adapted Gaussian Mixture Models. Digital Signal Processing 10, 19-41 (2000)
8. Melnikoff, S., Quigley, S.F.: Implementing the Log-add Algorithm in Hardware, Electronic Letters (2003)
9. Melnikoff, S.: Speech Recognition in Programmable Logic, University of Birmingham (2003)
10. Auckenthaler, R., Mason, J.S.: Gaussian Selection Applied to Text-independent Speaker Verification. In Proc. Speaker Odyssey 2001, 83–88 (2001)

High Efficiency FPGA Regular Expression Pattern Matching

Atta Badii, Adedayo O. Adetoye

Intelligent Media Systems & Services Research Laboratories
School of Systems Engineering, University of Reading, Whiteknights,
Reading, RG6 6AY, United Kingdom

Abstract. We present an approach to the automatic generation of efficient Field Programmable Gate Arrays (FPGAs) circuits for regular expression-based pattern matching problems. This paper focuses on the optimisation of the character class construct that is widely used in regular expressions. The optimisation technique uses a novel design strategy that can be used to generate circuits that are highly area-and-time-efficient for arbitrary sets of regular expressions. Thus, the technique is suitable for applications that must handle very large sets of patterns at high speed, such as in network security and intrusion detection domains. Empirical results demonstrate the advantages of our approach when applied to regular expressions in the official rulesets of the Snort intrusion detection system.

1 Introduction

Regular expressions are widely used in computing applications to find patterns or strings of interest in a given stream of data or text. The usefulness of regular expressions stems from the fact that they can concisely represent quite complex combinations of different patterns, saving on the storage requirements. A very important application of regular expressions is in Network Intrusion Detection Systems (NIDS), where malicious traffic is identified by matching against previously known set of patterns or strings, represented in the form of regular expressions. However, a relatively high proportion of the computational load in NIDS applications can involve regular expression matching, which thus consumes a high percentage of the computational time as the size of the set of regular expression being processed increases as new attack patterns are discovered. This can turn the NIDS into a bottleneck when it is used for intrusion prevention, especially when network speeds have to keep up with user demands for higher quality of services and rising multimedia volumes of traffic arising from ambient and social computing spaces. The need for more and faster traffic throughput suggests that a hardware solution would be a compelling requirement for NIDS systems that must operate in gigabit-speed networks.

In the NIDS domain, the set of patterns to be examined is expected to continue to grow as well as change over time, which means that the underlying computation needs to be reconfigurable to accommodate the changing patterns. Therefore, Field Programmable Gate Array (FPGA) devices are suitable due to their inherent characteristics of reconfigurability and their parallel processing potential that confers a capability for dynamically reprogrammable hardware-based logic which supports ultra-fast computation in a domain such as intrusion detection and prevention.

Regular expressions with their space compacting properties and FPGA logic with its reconfigurable and efficient computational capabilities can thus be brought together to offer very efficient solutions to pattern matching problems in domains such as NIDS, lexical analysis, virus detection, forensics, DNA analysis, and data mining where frequent updates to the reference patterns

are required. FPGAs and regular expressions have therefore been the technologies of choice in these domains, and there has been a considerable interest in optimising methods for the automatic generation of efficient FPGA circuits for a given set of regular expressions [9, 6, 11, 8, 10].

Our approach builds on existing works such as [7, 9, 10], by using the similar circuitry for regular expression constructs such as concatenation ($r_1 r_2$), alternation ($r_1 | r_2$), Kleene star closure ($r*$), and other metacharacter constructions such as $r?$ and $r+$. However, this paper presents a novel technique for the generation of area and time efficient circuits for the character class construct, $[c_0 c_1 \ldots c_n]$, which is often used heavily in regular expressions[1]. The technique leads to significant reductions in the number of comparators needed to implement character classes in large sets of regular expressions without any overhead in time efficiency. To demonstrate the efficacy of our approach, we applied the technique to the regular expressions in the Snort [1] IDS rulesets, where as much as 90% area reduction was achieved in some rulesets when compared with the traditional singleton character matching approach.

2 Regular Expressions

We shall start with a description of commonly used regular expression constructors, and introduce notational conventions that we shall follow. We adopt the Perl-Compatible Regular Expression (PCRE) regular expression syntax. We assume in general that characters are drawn from a coding scheme where all characters are encoded in a fixed number of bits. In this paper, we assume an 8-bit character encoding scheme. This assumption also accommodates characters which are drawn from a stream with less than 8-bit encoding, such as the 7-bit ASCII character set. However, the technique presented in this paper does not depend on the particular underlying encoding scheme that is used. We assume that $\mathcal{C}$ is the alphabet set of all the possible characters considered, so that in this paper $|\mathcal{C}| = 256$. For any $c \in \mathcal{C}$, we write c^H for the four Most Significant Bits (MSBs) and c^L for the four Least Significant Bits (LSBs). In particular, under the ASCII coding scheme, the character a corresponds to the binary code point 01100001; hence, $\mathsf{a}^H = 0110$ and $\mathsf{a}^L = 0001$. The notation $c_1^H c_2^L$ stands for the character obtained by concatenating the four MSBs of c_1 and the four LSBs of c_2. For example, $\mathsf{a}^L \mathsf{a}^H$ now stands for the character at the code point 00010110.

We use the typeface a,b,c,... for characters used to construct regular expressions and the typeface $c, c_i, \ldots$ for character *variables* and $r, r_i, \ldots$ for general regular expression *variables*. For example, a regular expression r is $[c_0 c_1 \ldots c_5]$, and an instance of r is [abcdef]; where, c_0 is the character a, c_1 is the character b, and so on. Under the PCRE scheme, the semantics of the expression [abcdef] is to match any of the characters a, b, c, d, e, or f. The character class constructor $[\cdots]$, can be "negated" or "inverted" by prefixing the characters with a ˆ, so that $[\hat{}c_0 \ldots c_n]$ now matches any single character, except those in the set $\{c_0, \ldots, c_n\}$.

In addition to the character class constructor, we also refer to other standard constructors such as the alternation |, where $r_1 | r_2$ matches either of r_1 or r_2; and concatenation, where $r_1 r_2$ matches only when pattern r_1 is matched first, immediately followed by a matching of pattern r_2. We also consider quantifier metacharacters such as $*$ (match 0 or more times), $?$ (optionally match, but at most once), and, $+$ (match at least one or more times). The matching semantics are all standard [4].

[1] The optimisation technique is also applicable to equivalent constructions such as $c_0 | c_1 | \cdots | c_n$, where characters or their classes are alternated instead.

2.1 Regular expression matching circuits

The key challenge that we wish to address is the optimal implementation of efficient pattern matching circuits based on a given set of regular expressions, in an environment (such as in network security and intrusion detection and prevention systems) where there might be several thousands of regular expressions to be encoded, and where the regular expressions might change over time. This makes an FPGA-based architecture suitable, firstly because of the amount of functional units available in modern hardware and the ability to re-program the system. We present in the remainder of this section, the general building blocks for FPGA-based regular expression patter matching.

Modelling the matching state Regular expression matchers can be implemented via finite state machines such as Deterministic Finite Automata (DFA) or Nondeterministic Finite Automata (NFA) which accept precisely the same strings as the regular expression. We shall use NFAs to model the regular expression matching state similarly to the technique originally proposed by [9]. The basic idea is that a regular expression may be viewed as a sequence of subexpressions denoting matching points in the pattern to be matched. Each matching point corresponds to a relevant regular expression circuit whose result must be activated when that point is reached in the matching process. Sequential Flip Flops are used to keep track of the current matching point in parallel[2] as it ripples through the Flip Flop sequence whose outputs are combined with *AND* gates to the current matching circuit to "activate" the next matching stage. This setup is illustrated in Fig. 1(a) for the expression $r_1 r_2 \cdots r_n$, where r_i is the circuit for the ith regular expression matching point. The *AND* gate connected to the r_n circuit is active at the nth clock cycle if the Flip Flop input is 1 at the $(n-1)^{th}$ clock cycle.

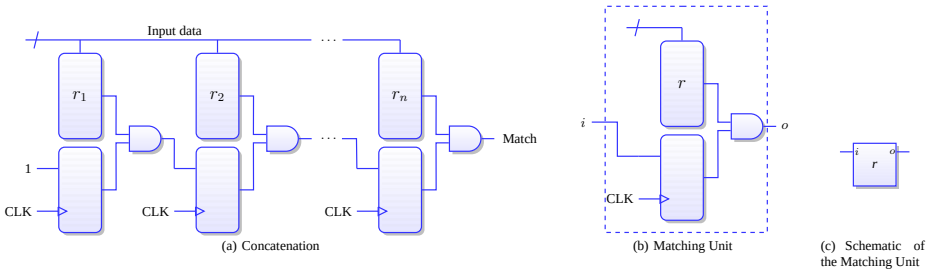
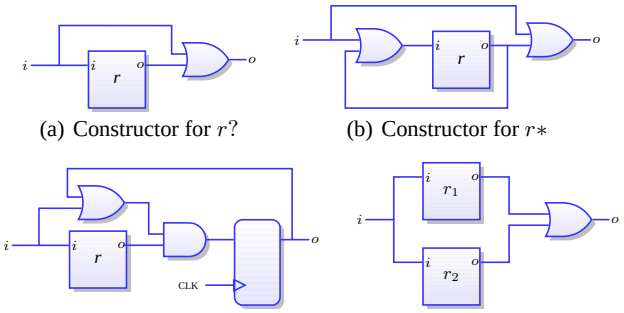


Fig. 1: *Modelling regular expression matching state*

The basic matching unit for the regular expression r is shown in Fig. 1(b). It consists of a matching circuitry for the expression r itself, a Flip Flop that becomes active when r is ready to be matched, and an *AND* gate which becomes active when r is matched and the Flip Flop is active. The schematic diagram of Fig. 1(c) summarises this setup. The circuitry for common regular expression constructors are shown in Fig. 2. In the circuit diagrams r and r_i could be any arbitrary regular expression.

[2] There may be more than one active matching point at a time.



(a) Constructor for $r?$ (b) Constructor for $r*$

(c) Constructor for $r+$ (d) Constructor for $r_1 \mid r_2$

Fig. 2: *Common regular expression constructors*

3 FPGA Design

In this section we present the design strategy for efficient and optimal implementation regular expression-based pattern matching circuits using FPGA technology. This paper focuses on the efficient encoding of the regular expression character class construct $[\cdots]$ and semantically equivalent constructs, such as $c_1|c_2|\cdots|c_n$, where each c_i is a single character. It is standard practice to implement the characters in a regular expression via dedicated matching units such as is shown in Fig. 1(b), where r is a unique character [7, 9, 10]. This means that a construct such as $c_1|c_2|\cdots|c_n$ consisting of n unique characters would need n such basic matching units. Thus, when implementing a large set of regular expressions in hardware, this can result in high area cost and poor timing performances, even when matching units are shared among regular expression implementations.

The main contribution of this paper is the innovative use of the basic matching unit to implement character classes. The approach leads to significant area and timing efficiency when applied to large sets of regular expressions. The key idea is that the semantics of the $[\cdots]$ and equivalent constructors implement a set of characters, whose operational interpretation is to test a given character for membership of that set. In the traditional approach [9], the basic matching unit is a circuit that tests membership of a singleton set - the set consisting of the single character that the matching unit encodes. A generalisation of this to arbitrary sets leads to significant gains in area efficiency without any overhead added to the matching and timing efficiency of the basic matching unit. In the remainder of this section we shall motivate the approach and develop techniques for the automatic generation of circuits, which take advantage of this semantics-based optimisation technique.

3.1 Efficiently encoding a set of characters

The basic logic construction that we use as a *comparator* is the same as that used in [9] as shown in Fig. 3. It consists of two 16-to-1 bit multiplexers, which very efficiently match a single 8-bit character. However, we present a novel technique whereby a single comparator can be used to *encode* a *set* of characters, as opposed to *one* character as proposed in [9]. We say that a comparator *encodes* a set of characters if it matches a character *if and only if* that character is in that set. As mentioned earlier, under this definition, the comparator of [9] is simply a singleton set comparator.

The definition generalises to arbitrarily-sized sets, and the larger the size of a set that is encoded by a comparator the more utilisation of hardware that is achieved by that comparator. In other words, the comparator which encodes a set $S \subseteq \mathcal{C}$ implements the set membership relation $\in$ over S. The technique results in a very significant utilisation of hardware without altering the intrinsic character-matching efficiency of the comparator. In particular, the comparator can efficiently encode up to the theoretical limit[3] of $|\mathcal{C}| = 256$ characters.

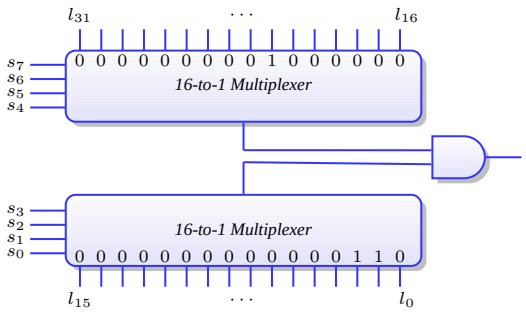


Fig. 3: *An efficient comparator showing input line values* $(l_0, \ldots, l_{31})$ *to match the regular expression [ab].*

For ease of reference, we have assigned names to the comparator inputs as shown in Fig. 3. The comparator implements a look-up table with the two 16-to-1 bit multiplexers, and has altogether 32 *control inputs* labelled $l_0, \ldots, l_{31}$; and an 8-bit *selector* labelled as $s_0, \ldots, s_7$. The control inputs encode the set of characters against which a character may be matched. The incoming 8-bit character [4] to be matched is assigned, in order, to the selector lines with the LSB on line s_0 and the MSB on line s_7. The detail of how to assign values to the control lines, based on the intended set of characters to be matched, is presented in section 3.1.

To illustrate the area efficiency that can be achieved by encoding set of characters in a comparator, suppose that in an e-mail related application we wish to check that a given character is not @ or any of the ASCII characters between A and O inclusive. This requirement may be captured by the regular expression [ˆ@A-O], which matches any character in the set $S = \mathcal{C} \backslash \{@, \mathsf{A}, \mathsf{B}, \ldots, \mathsf{O}\}$. Using the technique presented in section 3.1 we discover that the set S, which has a cardinality of 240, can be implemented on a single comparator with all the input bits, except l_{19}, set to 1. This is a very significant save in area and complexity of circuitry as opposed to; a scenario (a), where the same regular expression is naïvely implemented using a comparator for *each* character in S combined together with *OR* gates; or even in the scenario (b), where the characters of the set $\{@, \mathsf{A}, \mathsf{B}, \ldots, \mathsf{O}\}$ are loaded in a comparator each, and their *OR* combination inverted.

[3] Although in practice this particular configuration is superfluous, because an output line set to 1 achieves the same effect of matching all possible characters.

[4] The reader should note that the techniques presented in this paper can be similarly applied to any n-bit character encoding schemes.

Assigning control line values We now show how to assign the control lines in a comparator in order to encode a set of characters. Let *dec*$(\cdot)$ be a function which converts a binary number to its decimal value. Given a set $S \subseteq \mathcal{C}$ of characters to be encoded, we define a function E (for *Enable*) that returns a set of control line indices that must be enabled in order to match all the characters in the set S as

$$E(S) \triangleq \left\{ l_i, l_j \mid \exists c \in S, i = \textit{dec}(c^L), j = 16 + \textit{dec}(c^H) \right\}. \tag{1}$$

Applying (1) to the set $S = \{\mathsf{a},\mathsf{b}\}$, since the ASCII code for a and b respectively are 01100001 and 01100010, we have *dec*$(\mathsf{a}^H) =$ *dec*$(\mathsf{b}^H) = 6$ and *dec*$(\mathsf{a}^L) = 1$ and *dec*$(\mathsf{b}^L) = 2$. Hence, $E(S) = \{l_1, l_2, l_{22}\}$, as demonstrated by the enabled control lines of Fig. 3. The comparator encoding shown Fig. 3 matches precisely only a or b, however for arbitrary sets $S \subseteq \mathcal{C}$, enabling the lines $E(S)$ in a *single* comparator may lead to matching of characters not in S: a situation that we refer to as *code collision* in the comparator. Thus, in order to encode an arbitrary set S we must encode only collision-free subsets of S in comparators whose results are then combined in an *OR* formation. We start by presenting the property of collision-free subsets of an arbitrary subset of $\mathcal{C}$, and then present an efficient technique for computing one.

Deriving collision-free subsets Code collision results in a comparator matching more than the intended set of characters. To illustrate this problem, consider the regular expression $[\mathsf{aR}]$, whose semantics is to match any character in the set $S = \{\mathsf{a},\mathsf{R}\}$. Using equation (1), if we enabled the lines $E(S) = \{l_1, l_2, l_{21}, l_{22}\}$ in a single comparator, that comparator will match not only characters a and R, but also b and Q. This is because the character code of b can be derived by pairing $\mathsf{a}^H\mathsf{R}^L$; and similarly, Q has the code $\mathsf{R}^H\mathsf{a}^L$. This problem is easily avoided by using only *collision-free* subsets of S in each comparator: which, in this example, is the singleton subsets of $\{\mathsf{a},\mathsf{R}\}$. Let us now present the detail in the more general case.

The general problem is that given a set $S \subseteq \mathcal{C}$, we want to find the minimal number of comparators such that a character c is matched if and only if $c \in S$. Any subset $S' \subseteq S$ is collision-free (with respect to S), if for any pair of characters $c_1, c_2 \in S'$ we also have that $c_1^H c_2^L, c_2^H c_1^L \in S$. This means that by swapping the four LSBs of any pair of characters in S', no new character can be derived (and thus matched) that is not in S. The set S' can thus be loaded into a comparator with the guaranteed property that the comparator matches only characters in S. The next step is to ensure that we *cover* the set S by using enough collision-free subsets so that *all* characters in S are matched.

This problem can be constructed as the standard *set-covering* problem [2] as follows. Define the set of all collision-free subsets of S as

$$\mathcal{F}_S \triangleq \{S' \subseteq S \mid c_1, c_2 \in S' \Rightarrow c_1^H c_2^L, c_2^H c_1^L \in S\}. \tag{2}$$

Find a minimum-size subset $\mathcal{F} \subseteq \mathcal{F}_S$ such that

$$\bigcup_{S' \in \mathcal{F}} S' = S. \tag{3}$$

Finding minimal systems of comparators The optimisation problem posed by (2) and (3) is NP-hard [5]. However, greedy set-covering algorithms [3] can be implemented to run in time polynomial in $|S|$ and $|\mathcal{F}_S|$.

Given an arbitrary set $S \subseteq \mathcal{C}$, the first problem is to compute $\mathcal{F}_S$ efficiently. Secondly, a minimal subset of $\mathcal{F}_S$ must be chosen, which covers S. We propose in this paper a technique which uses partial equivalence relations[5] (PERs) over $\mathcal{C}$, to efficiently compute minimal collision-free subsets of S, which can be efficiently implemented in a simple algorithm which runs in time linear in $|S|$. The algorithm takes advantage of the algebraic properties of PERs to systematically compute collision-free subsets of an arbitrary set $S \subseteq \mathcal{C}$, that minimally cover S.

Define an equivalence relation, R_S, over S such that for any $c_1, c_2 \in \mathcal{C}$, $c_1 \; R_S \; c_2$ iff $c_1, c_2 \in S$ and

$$\{c_3^L \mid c_3 \in S, c_3^H = c_1^H\} = \{c_4^L \mid c_4 \in S, c_4^H = c_2^H\}. \tag{4}$$

The relation R_S is an equivalence relation over S, and in general a partial equivalence relation over $\mathcal{C}$. Intuitively, two elements c_1 and c_2 of S are related by R_S if by replacing the four LSBs of c_1 with those of c_2, or vice versa, no new character can be derived that is not in S. That is, $c_1 \; R_S \; c_2 \implies c_1^H c_2^L, c_2^H c_1^L \in S$. The definition of R_S ensures that each equivalence class of R_S is closed under this property. This property means that if a comparator is encoded with only characters that are all related by R_S, then no character can be incorrectly matched. This guarantees the semantic correctness of the matching unit, regardless of the number of characters that the comparator encodes: only characters in S will be matched.

We now have a strategy for obtaining minimal matching circuits: given an arbitrary set S of characters to be matched, we partition it via R_S and encode each (collision-free) equivalence class in a comparator. When this results in more than one comparator, that is, whenever R_S has more than one equivalence class, the resulting comparators are combined together in an *OR* formation to obtain a matching circuit that covers the set S. Because of the reflexivity property of R_S over S and the algebraic property that each character belongs to exactly one equivalence class of R_S, we know that the union of the equivalence classes of R_S covers S minimally. Since the equivalence classes of R_S are, by definition, collision-free with respect to S, we obtain on this basis the collision-free subsets of S, given by

$$\textit{c-free}(S) \triangleq \{[c]_{R_S} \mid c \in S\}. \tag{5}$$

As usual, the standard notation $[c]_{R_S}$ stands for the *equivalence class* of c under the equivalence relation R_S.

By exploiting the *reflexive and transitive closure* property of equivalence relations, we can efficiently compute the sets *c-free*(S) of the equivalence classes of R_S. In particular, let $I = \{c^H \mid c \in S\}$ be an index set of the H-bits of characters in S; reflexivity of R_S means that for any $i \in I$, all elements of the set $S_i \triangleq \{c \in S \mid c^H = i\}$ belong to the same equivalence class of R_S. Furthermore, if we define $S_i^L \triangleq \{c^L \mid c \in S_i\}$ for any $i \in I$, the transitivity of R_S allows us to traverse the set S once, and means that for any pair $i, j \in I$ such that $S_i^L = S_j^L$ then all elements of the set $S_i \cup S_j$ also belong to the same equivalence class of R_S. This gives us a straightforward algorithm to compute R_S for any set S. This is shown in Algorithm 1, which efficiently computes *c-free*(S).

[5] Recall that a relation R over a set S is a partial equivalence relation if it is symmetric ($s \, R \, s' \implies s' \, R \, s$), and transitive ($s \, R \, s', s' \, R \, s'' \implies s \, R \, s''$) for all $s, s', s'' \in S$. If, in addition, R is reflexive ($s \, R \, s$), then it is an equivalence relation.

```
Input  : Set S of characters.
Output: Collision-free subsets {S_j | j ∈ J} of S.
/* Partition S to subsets, indexed by the set I of H-bits.        */
I ← ∅
forall c ∈ S do
    if c^H ∉ I then
        S_{c^H} ← {c}
        I ← I ∪ {c^H}
    else
        S_{c^H} ← S_{c^H} ∪ {c}
    end
end
For all i ∈ I, let S_i^L = {c^L | c ∈ S_i}
/* Merge equivalence classes, and create a new index set J        */
J ← ∅
K ← ∅
while i ∈ I\K do
    J ← J ∪ {i}
    K ← K ∪ J
    forall k ∈ I\K do
        if S_i = S_k then
            S_i ← S_i ∪ S_k
            K ← K ∪ {k}
        end
    end
end
```

Algorithm 1: Computing collision-free subsets

4 Empirical Analysis

We demonstrate the area efficiency properties of our approach by applying the analysis technique to the Snort [1] ruleset. Snort is a leading open source NIDS, which supports the use of regular expressions in its pattern matching rules. We have applied the technique to the usage of character class construct in the official Snort rules[6]. For simplicity, we only applied the technique to character classes explicitly created through the $[\cdots]$ construct. The characters in the class are counted, and the number of comparator units required to implement the class is computed. For example the class [@A-O] contains 16 characters, namely, @ and the 15 characters A, B, . . . , O between A and O inclusive in the underlying character encoding system. The character class [@A-O] requires only one comparator unit, when we apply the analysis technique. In the analysis, built-in character classes are expanded into the constituent characters that they represent: for example, \w, which is a shorthand for the character class [a-zA-Z0-9], is made up of 62 characters.

We show the results obtained when we apply the PER-based technique in Table 1. For each rules file in the Snort ruleset, column "# of characters" shows the number of characters in total

[6] This refers to the version released on 08 October 2009.

used in that file. For each character class construct used in a rules file, we compute the number of comparators needed to implement that construct, and the column "# of comparators needed" contains the total of all such comparator needed for the file. The column "% area reduction" shows the reduction in area obtained by applying the technique.

Ruleset	**# of chars**	**# of comparators needed**	**% area reduction**
chat.rules	393	39	90.08
ddos.rules	20	2	90.00
specific-threats.rules	236	35	85.17
exploit.rules	1868	330	82.33
web-client.rules	1488	349	75.55
smtp.rules	2034	529	73.99
deleted.rules	1361	386	72.96
spyware-put.rules	1930	603	68.76
oracle.rules	9774	4648	52.45
web-activex.rules	6925	3546	48.79
⋮	⋮	⋮	⋮

Table 1: *Area reduction in Snort Rulesets*

Since the same character class is often reused in a single file, Table 2 shows the same analysis, but with sharing of matching units that implement a given character class. So, for each unique character class in a rules file, only one matching unit is created. As the table demonstrates, we obtain even more significant reduction in the overall logic block area, although this is usually a trade-off against more complex interconnectivity. The significant gain is due to the fact that the same character class is often heavily reused within a rules file, and even within the same regular expression. For example, the number of characters in the *oracle.rules* file is relatively large due to the repeated use of the classes [\r \n \s], [^\x22] and [^\x27]. When unique character classes are considered, this leads to a reduction from 9774 to 71, which requires the use of only 7 comparators. Similar gains are achieved in the *web-activex.rules* file.

5 Conclusion and Future Work

We have presented a framework for the generation of efficient FPGA circuitry for regular-expression-based pattern matching problems. The current work has focused on the character class construct, which, on the basis of its matching semantics allows multiple characters to be encoded in a single comparator unit. We have developed a PER-based model for generating minimal circuits that implement a given character class construct. The advantage of this approach is that the PER-based model can be implemented in a very efficient and simple algorithm.

To demonstrate the usefulness of our approach, we applied the technique to the rules files of the Snort IDS. We see that significant reductions in area can be obtained by reusing matching circuits for unique character classes in each file. The metric used, which is based on the relative reduction in

Ruleset	# of chars	# of comparators needed	% area reduction
chat.rules	62	7	88.71
ddos.rules	10	1	90.00
specific-threats.rules	68	8	88.24
exploit.rules	256	16	93.75
web-client.rules	106	9	91.51
smtp.rules	86	7	91.86
deleted.rules	78	7	91.03
spyware-put.rules	81	7	91.36
oracle.rules	71	7	90.14
web-activex.rules	11	3	72.73
⋮	⋮	⋮	⋮

Table 2: *Area reduction in Snort Rulesets when Matching Units are shared*

number of matching units under our approach against the number of units in a system that encodes a character per unit, does not consider the interconnectivity implications. In practice the interconnectivity metric is equally as important and will be the subject of future study, where we seek to develop techniques for optimal tradeoffs between matching units and the overall interconnect costs.

References

1. Snort website. http://www.snort.org/.
2. E. Balas and M. Padberg. Set partitioning: A survey. *SIAM Review*, 18(4):710–760, 1976.
3. Thomas H. Cormen, Charles E. Leiserson, and Ronald R. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.
4. Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.
6. Brad L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *FCCM*, pages 111–120. IEEE Computer Society, 2002.
7. A. Mukhopadhyay. Hardware algorithms for nonnumeric computation. *IEEE Trans. Computers*, C-28(6):384–394, June 1979.
8. Jia Ni, Chuang Lin, Zhen Chen, and Peter Ungsunan. A fast multi-pattern matching algorithm for deep packet inspection on a network processor. In *Proc. 2007 International Conference on Parallel Processing (36th ICPP'07)*, page 16, Xi-An, China, September 2007. IEEE Computer Society.
9. Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching using FPGAs. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.
10. Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Signal Processing Systems*, 51(1):99–121, 2008.
11. Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In Laxmi N. Bhuyan, Michel Dubois, and Will Eatherton, editors, *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2006, San Jose, California, USA, December 3-5, 2006*, pages 93–102. ACM, 2006.

Row-interleaved streaming data flow implementation of Sparse Matrix Vector Multiplication in FPGA

Branimir Dickov *†, Miquel Pericàs †,
Nacho Navarro *†, and Eduard Ayguadé *†

† Barcelona Supercomputing Center, Barcelona
{branimir.dickov,miquel.pericas,nacho.navarro,eduard.ayguade}@bsc.es
http://www.bsc.es
* Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya, Barcelona
http://www.ac.upc.edu

Abstract. *Sparse Matrix-Vector Multiplication (SMVM) is the critical computational kernel of many iterative solvers for systems of sparse linear equations. In this paper we propose an FPGA design for SMVM which interleaves CRS (Compressed Row Storage) format so that just a single floating point accumulator is needed, which simplifies control, avoids any idle clock cycles and sustains high throughput. For the evaluation of the proposed design we use a RASC RC 100 blade attached to a SGI Altix multiprocessor architecture. The limited memory bandwidth of this architecture heavily constraints the performance demonstrated. However, the use of FIFO buffers to stream input data makes the design portable to other FPGA-based platforms with higher memory bandwith.*

Key words: Sparse Matrix Vector Multiplication, Floating Point, FPGA, Compressed Row Storage

1 Introduction

As floating-point performance achievable on FPGA has risen beyond that of processors [1] the motivation for the science community to move computationally intensive kernels to FPGA and improve performance of scientific application has grown considerably. When computing SMVM, microprocessors have been shown to achieve very low floating point efficiency [2]. Compared to dense kernels, sparse kernels incur more overhead per non-zero matrix entry due to extra instructions and indirect, irregular memory accesses. Also, the large number of operands required per result and minimal reuse stress load/store units while floating point units are often under-utilized. In FPGA implementation, data structure interpretation is performed by spatial logic. Thus, by using streaming of data to/from memory and fully pipelined functional units, FPGA systems can obtain high levels of performance compared to their clock speeds. Current FPGAs contain many more configurable logic blocks which allows implementing

multiple copies of the same computations.

Given the importance of the SMVM kernel, many early attempts to implement this algorithm on FPGAs exist. Zhuo [6] and Sun [7] both implemented an adder tree based design with an optimized reduction circuit that requires only a few floating point adders. These designs streamed the matrices using the Compressed Row Storage (CRS) format. However, this design creates complications in the accumulator circuitry as a new element to be accumulated is generated every cycle, far too much for a single floating point adder implemented in FPGA.

In this paper we propose to overcome this problem by streaming the matrix in a row-interleaved variant of the CRS format for sparse matrix representation. By packing the matrix in this way independent dot products are interleaved on a single floating point adder, which can then implement the reduction all by itself. By using less logic resources for reduction circuit we achieve both power and area savings. As another important feature, our design doesn't depend on the sparsity structure of the matrix or on the number of non zero elements per row.

To evaluate these ideas we implemented the design using a binary adder tree with 4 leaves on a Virtex4 LX200 device present in a RASC RC100 blade of a SGI Altix machine. Test matrices for our design were created in software with completely random structure. Given the amount of Block RAMs (BRAM) present in the FPGA device, matrices cannot have more than 16,000 columns. Since rows of matrices are streamed one after another, no limitation exists on the number of rows. On the V4LX200 device, the algorithm requires up to 50% less slices than competing approaches. Running at 200MHz, a maximum troughput of 1.6 GFLOPS can be achieved, although this value is limited by the parameters of the platform.

The rest of the paper is organized as follows. In Section 2 we introduce the SMVM algorithm used in this paper. We then elaborate on basic SMVM designs in FPGA and propose our design (Section 3). Implementation details on our target hardware are given in Section 4, while the design is evaluated in Section 5. Some additional related work is presented in Section 6 in the context of our design. Finally, we conclude the paper with some ideas for future work that can further improve the design.

2 SMVM Problem

Sparse matrix vector multiplication in the standard form $y = Ax$ is one of the most time consuming computational kernels for iterative solvers of sparse linear systems such as Conjugate Gradient method (CG) [3]. The extra computations besides SMVM in CG are some vector parallel operations which are small compared to SMVM. Profiling PETSc's [4] CG algorithm for the bi-laplacian stencil we obtain that more than 75 % of all computations are spent on SMVM. In general, the SMVM $y{=}Ax$ is defined as:

$$y_i = \Sigma_{j=0}^{N} a_{i,j} x_j, (0 \leq i < M)$$

Where A is an $M \times N$ matrix, while y and x are $M \times 1$ and $N \times 1$ vectors, respectively. In general some kind of compressed storage that just saves non zero elements, such as Compressed Row Storage (CRS), is used to store sparse matrices.

A square sparse matrix A with dimensions $n \times n$ and with m non zeros, is represented with three arrays in the CRS format:

- *rowID*: Stores the starting index of the first non zero element of each row. Has the same length as the dimension of the matrix plus one *n+1*.
- *Col*: Stores the column indices of non-zero elements. The length is the number of non-zeros m.
- *Val*: Stores all non-zero elements in row major order. The size is the number of non-zeros m.

3 SMVM design in FPGA

In this paper we show an innovative SMVM design for FPGA. In order to do so we will first explain a design for pure CRS that is based on the works of Zhuo and Prasanna [6] and Sun and Peterson [7] which also serve as this work's motivation.

3.1 SMVM design using CRS

To achieve high frequency, floating point adders are usually deeply pipelined, which difficults accumulation of floating point data. To fully exploit computational throughput of the FPGA and avoid any idle cycles we want to pipeline the dot-product accumulation as heavily as possible.

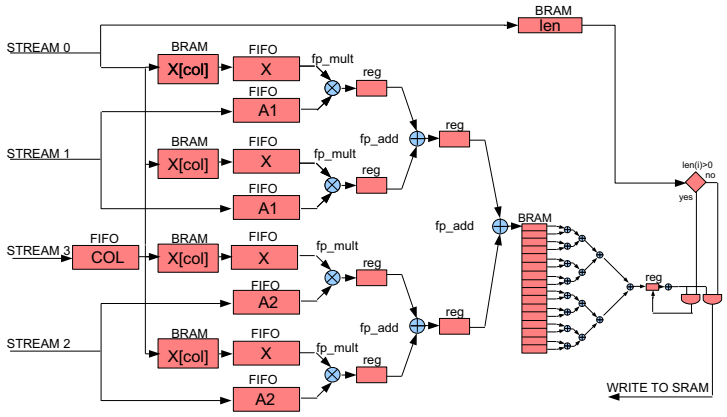


Fig. 1. Complete SMVM design using CRS

On the front end of our design (Fig. 1) the binary tree with k leaves accepts data, every leaf contains one multiplier and BRAM in which the multiplicand

vector X is preloaded which is addressed by the *Col* vector. Obtained products are then accumulated by internal adders in the binary tree and finally one partial sum is produced every clock cycle. On the back end, the latency prevents us from pipelining partial results from adder tree so we need temporary storage (BRAM) with the same size as the latency of the adder, L_{add}. Once the BRAM is full we employ a reduction circuit. The first part of the reduction circuit is an adder tree from which we obtain new partial sum. The partial sum is then passed to the adder which works like a back loop adder that provides us with a final result for the corresponding row.

An improvement can be made in reduction circuit by using instead of the adder tree just a pipeline of adders with buffers [6] [7], (Fig. 2) where all circuit is driven just by data flow and without any control logic. When the first adder gives output, the value is passed through one buffer register in order that next adder start when both inputs are available. Adders at different levels will be used at different cycles producing finally one accumulated result. But in order to work for different matrices independently of number of non zeros per row one BRAM, size of vector X, has to be implemented as a temporary storage.

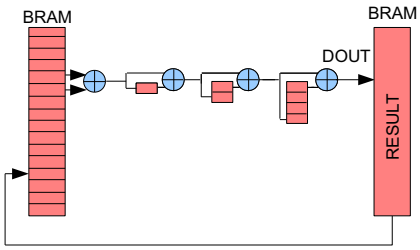


Fig. 2. Reduction Circuit

3.2 SMVM design with new interleaved CRS

In the new design, the front end is not changed but the back end is considerably simplified, (Fig. 3). As we mentioned adder latency L_{add} prevents us from pipelining the partial result obtained from front end but we can interleave the independent partial result on a single floating point adder and achieve maximum throughput. Thus, adder latency L_{add} becomes the interleaving factor. By modifying traditional CRS format on software side we achieve that after every L_{add} cycles we get a new dot product that can be added to the last corresponding partial dot product. So the idea is to use a kind of window whose slots get filling with independent dot products (rows) of the matrix. The size of the window should be equal to L_{add} to achieve paralelization due to pipelining. Every clock cycle k (on Fig. 3 k=*4*) elements of vector *Val* size of 64 bit and k elements of vector *Col* size of 16 bit are streamed respectively to multipliers and BRAMs in this new rescheduled CRS format. If the number of non zeros in a row is not multiple of the number of k we have to pad them with zeros. When some slot of the window is empty (there are no more elements in current row) this

slot is granted to next corresponding row. By doing this the throughput of the pipeline adder is sustained. As we see by using single floating point pipeline we can calculate dot product for every row without almost any idle clock cycles. At the end of the design due to different number of non zero elements in rows some bubbles have to be introduced in order to finish summation correctly. The bubbles are not introduced physically in the sense that they are streamed with non zeros. What happens is that when some element of vector *Len* is zero(there are no more non zero elements for that row and there are no new rows) and still some rows have to finish their accumulation, the reading from input FIFOs is stopped but multipliers and adder tree continue in order to deliver new elements to single adder loop at the correct cycle.

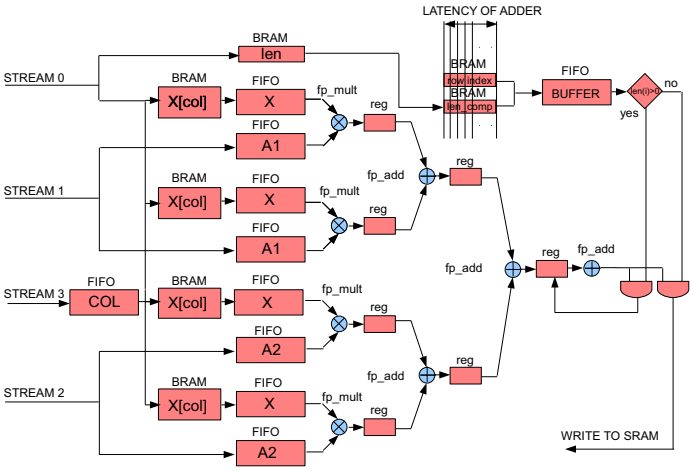


Fig. 3. Complete SMVM design using rescheduled CRS

Because the time for accumulating a full row on the single pipeline adder depends on how many non zeros that row has, results will not get out in row major order. For example if row X has less than k non zero elements and first row has multiple number of non zeros, accumulation for row X will be finished in just one pass through adder pipeline. Thus, to have in every moment control and also to be sure that all non zeros are accumulated for some row, FSM with L_{add} states is implemented. As was mentioned earlier about vector X also vector *Len* of CRS format are preloaded to FPGA BRAM before starting any computation. The FSM controls two arrays *row_index* and *len_comp* which have size of L_{add}. The FSM and computation are started when "empty" signal of entry FIFOs is pulled to zero. In the *row_index* array we keep the indexes of actual rows that are to be computed. *len_comp* contains the corresponding number of elements that need to be computed to obtain final result. After every new k elements are feed to multiplier, vector *len_comp* is updated. When value of vector *len_comp* for corresponding row reaches 0 the row computation is completed and the result

is written to *row_index* location in SRAM memory. To be sure that the flag for the corresponding row that indicates that the last elements of some row are sent to multipliers, a FIFO buffer is introduced so that the flag and result match up at the output of single adder pipeline. If the flag is not pulled up this means that there are still some non zeros in a row that have to be calculated so the output is back-looped. In Table 1. results for the total number of LUTs spent for adders and multipliers are shown for all three designs explained here.

Table 1.Sum of LUTs used by floating point multiplier and adder

Type of storage	Number of LUTs	Design of back end
CRS format	24612	binary adder tree
CRS format	10500	optimized reduction circuit
Interleaved CRS format	6972	single floating point adder

4 Implementation details

4.1 Targeted platform

Altix is Silicon Graphics' line of servers and supercomputers based on Intel Itanium processors. In this work we target the ALTIX 4700 platform. The SGI Altix 4700 platform is comprised of modular blades: interchangeable compute, memory, I/O and special purpose blades for 'plug and solve' configuration flexibility. Besides microprocessor nodes, it also features reconfigurable nodes named SGI RASC RC100 [9], (Fig. 4), which have 2 Virtex 4 XC4VLX200 FPGAs and 5× SRAM local memories for every FPGA. The peak transfer rate between host and RC 100 blade is provided as 3.2 GB/s in each direction.

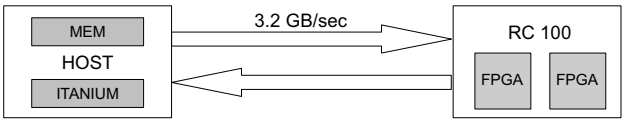


Fig. 4. RC100 blade on Altix 4700

4.2 CLB utilization and IP cores

The majority of logic is used for generation of mathematical operators. We used Xilinx IP cores which follow the IEEE 754 standard that can be customized for generation of mathematical operators and memories. For generation of mathematical operators we used DSP48 slices [10]. Virtex 4 XC4VLX200 has 96 DSP48 slices and for generation of floating point IP multipliers 16 DPS48 slices are used in "full usage" mode, so in the case where k=4 it's possible to generate all multipliers with DSP48, and even to use DSP48 to create the adders. With respect a floating point adders using just logic doesn't result in an important increase of slices while in case of multipliers this drastically change. In Table 2. main characteristics of the design are shown.

Table 2.Main characteristic of 64-bit SMVM design

Total slices	22898(25%)
BRAMs	311 (92%)
Multiplier latency	22
Adder latency	10
Required I/O bandwidth	8 GB/sec
Available I/O bandwidth	3.2 GB/sec
Operation Frequency	200 MHz

4.3 BRAM

In the design we used 4 BRAMs for vector X whose length is n and width is of 64 bits. Also, 1 BRAM is used for vector Len with length of n and width of 32 bits. The Virtex-4 LX200 FPGA contains 336 BRAM units, each with the capacity of 18Kb. This yields an internal high speed memory capacity of 6,048 Kb = 756 KB. For example, for storing vector X of size of 16.000 in BRAM would require 125 KB. For the design to support SMVM with matrices of up to 16.000 rows, 311 BRAM units are needed which represents 92% usage of BRAM memory. Thus, we can see that this can be a limiting factor for the size of matrices that can be calculated using the FPGA implementation. If memory bandwidth increases, it would be even worst, as we would need more copies of vector X, which would lead to further reduction of the size of matrices in order to fit all vectors in BRAM.

5 Evaluation of proposed design

5.1 Performance

In our design we have 8 fully pipelined floating point units, so with operation frequency at 200 MHz for this implementation we get peak floating point performance of 1.6 GFlops/sec. Considering that the available bandwith is 3.2 GB/sec and that design needs 8 GB/sec to reach peak performance we get that peak theoretical efficiency of our design can be:

$$\frac{3.2\ GB/sec}{8\ GB/sec} = 0.4 * 100\% = 40\%$$

Thus, peak floating point performance that can be achieved is:

$$1.6\ GFlops * 0.4 = 640\ MFlops$$

The area efficiency of our design is:

$$\frac{1.6\ GFlops}{22898\ slice} \approx 0.07\ MFlops/slice\ .$$

The idea of this work wasn't to achieve any spectacular performance for ALTIX platform but to show that on another platforms with the reasonable good memory bandwith portability is preserve in order to achieve good performance with low slice usage.

5.2 Scalability of design

The implementation of this design is done with k=4, but to be in position to feed all multipliers with new data every cycle we need 8 GB/sec of memory bandwith. We need 4 non zeros form vector *Val* size of 64 bit and 4 elements from vector *Col* of 16 bit, in total $8B \times 4 + 2B \times 4 = 40B$, which gives the needed memory bandwith of 8 GB/sec at 200MHz clock speed. In ALTIX platform for streaming, independently of how many streams we use, total rate remain 3.2GB/sec. The idea was to show that if memory bandwith increase our design can adopt, but also if it increase more that 8 GB/sec the implementation can be easily augmented by simple instantiations of more multipliers, adders and FIFOs in the front end but the overall design would stay the same.

6 Comparison with other designs for SMVM

We focus on three works that implemented SMVM on FPGA, that we consider the most relevant and novel. Some of them are meant to work iteratively by doing SMVM multiple times on FPGA with some other operations included in iterative solvers others just do SMVM and stream the result out. The great majority of them use CRS as a standard compressed format. Thus, we will try to stand out what could be the disadvantages of their designs. We start with the work of Zhuo and Prasanna [6] who designed an adder tree based SMVM implementation for double precision floating point where multiplicand vector X is preloaded into FPGA. Computation is done row by row where the row is divided into subrows. The length of subrows is equal to number of leaves k of front end binary tree. Zero padding is implemented when the number of non zeros is not multiple of k. They propose technique called merging in order to reduce the overhead that can result from zero padding. To accumulate all subrows they use reduction circuit that contains 7 floating point adders. By using just 7 adders they limit the number of non zeros per row that can be calculated to $2^7 \times k$. This means that matrices that have rows with more than $2^7 \times k$ non zeros can not be calculated and that design needs to be modified. In contrast thanks to interleaving, our design accepts any input matrices with no hardware changes required with just one single pipeline adder as reduction circuit.

Sun and Peterson [7] proposed Row Blocked CRS to represent sparse matrix. They use multiple processing elements PEs along with reduction circuit to perform the SMVM. PEs are basically multipliers with FIFOs for storing intermediate results. By implementing simple PEs instead of an adder tree, zero padding is avoided. For reduction circuit they proposed a similar circuit to [6] with one exception that circuit has one input from resulting BRAM which enables performing accumulation independently of the number of non zeros per row. They use 4 adders along with buffers in reduction circuit whereas the design presented here achieves reduction with just one simple pipeline adder. Because time spent on I/O is greater than time needed by reduction circuit to finish SMVM, reduction circuit is shared between PEs. In the case that larger bandwidth is available this can be bottleneck, in the sense that more reduction circuitry would need to

be implemented which can considerably increase number of slices used.

The design of deLorimier and DeHon [8] computes iteratively $A^i \times X = Y$ and uses exclusively on-chip BRAM to hold all of the SMVM data in order to achieve big memory bandwidth. Here, the matrix is partition across multiples PEs. The number of dot products assigned to PE should be the size of latency of adder, L_{add} to achieve parallelism due to pipelining. By scheduling parts of dot products in correct order parallelism is obtained and any stalls are avoided. To achieve good efficiency all PEs should get approximately same number of non zeros. At the end of computation, some slots can not be fed and some bubbles have to be introduced. This design is very similar to ours in the sense that both reschedule the rows in order to achieve parallelism and avoid any stalls. The difference is that in our design rescheduling is done for all non zero values of matrix in such a way that when there are no more non zeros in a window slot a new row with its non zeros is introduced to that slot. By this we will have just some bubbles at the end and not at every partition part of design. Also in our design all accumulation is done on single floating point adder while they are using one adder for every PEs. Also due to the fact that all SMVM data resides inside the FPGA before computation starts these extra bubbles can affect the size of BRAM used. In contrast, in our case instead of putting all SMVM data on FPGA we just stream vectors *Val* and X inside the FPGA.

7 Future Work

For future implementations we believe it is necessary to further improve the design by overcoming the space limitations due to multiple instances of the X vector. This will be particularly important for higher bandwidths, as in this case even more copies of the vector will need to be stored. To reduce the storage required by X we want to study the possibility of distributing a single copy of the vector across several multi-ported memory banks. Each bank will then hold a subset of the vector and can be accessed independently. In order for this to work, the elements of the rows will need to be scheduled so to avoid conflicts when too many multipliers want to access X vector elements that reside in the same bank. Row elements can be rearranged and reordered for this to happen. Although it might not always be possible to find row elements that avoid access conflicts, and thus performance may slightly degrade, the possibility of storing a single copy of vector X in the FPGA would have obvious advantages both in terms of matrix size and as well as area efficiency. Thus we consider it worth to take a look at this optimization. The area reduction, for instance, would simplify the possibility of integrating the SMVM design together with a complete conjugate gradient implementation.

8 Conclusions

In this paper we haved developed a design for interleaved CRS. We show how using this new arrangement we can reduce the number of slices of the imple-

mentation. As a consequence, both the area and power efficiency of the design are improved. The design does not need to be changed for different matrices and overall performance just depends on the total number of non zeros of the matrix. One of the goals of the design is portability to other FPGA-based platforms with higher I/O bandwidth. Also, the overall scalability of the design is preserved by using a k binary tree structure in the front end. Based on our experience implementing this design we identify the size of the on-chip memory as an important bottleneck as it limits the size of the matrices that can be computed. Thus our further efforts will concentrate on trying to solve this issue.

9 Acknowledgements

This work is supported by the Spanish Ministry of Science and Innovation (contracts no. TIN2007-60625 and CSD2007-00050) and the European Commission in the context of the HiPEAC Network of Excellence (contract no. IST-004408). Branimir Dickov is supported by Catalan Government with Pre-doctoral scholarship FI (reference no. 2009FI-B00077).

References

1. Underwood, K.: FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In Proceedings of the International Symposium on Field-Programmable Gate Arrays, pp 171–180, (2004).
2. Vuduc, R.: Automatic performance tuning of sparse matrix kernels. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, December 2003.
3. Shewchuk, J.: An Introduction to the conjugate Gradient Method Without Agonizing Pain. School of Computer Science, Carnegie Mellon University (1994)
4. Portable, Extensible Toolkit for Scientific Computation, `http://www.mcs.anl.gov/petsc/petsc-2/`, Argonne National Laboratory, Illinois, Chicago, United States
5. Barrett, R.: Templates for the solution of Linear Systems: Building Blocks for Iterative methods, 2nd Edition. SLAM , Philadelphia, PA (1994)
6. Zhou, L., Prassana, V.: Sparse Matrix-Vector Multiplication on FPGAs. Proceedings of the 2005 ACM/SIGDA 13th International Symposium on the Field-Programmable Gate Arrays, pp 63–74, Monterey, California, USA (2005)
7. Sun, J., Peterson, G., Storaasli, O.:Mapping Sparse Matrix-Vector Multiplication on FPGAs.Proceedings of the Third Annual Reconfigurable Systems Summer Institute, NCSA, Urbana IL, USA, 2007
8. De Lorimier, M., De Hon, A.: Floating-point Sparse matrix-vector Multiply for FPGAs. Proceedings of the 2005 ACM/SIGDA 13th International Symposium on the Field-Programmable Gate Arrays, pp 75-85, Monterey, California, USA (2005)
9. Reconfigurable Application-Specific Computing User's Guide, `http://techpubs.sgi.com`, SGI (2008)
10. Xilinx, Inc. `http://www.xilinx.com`