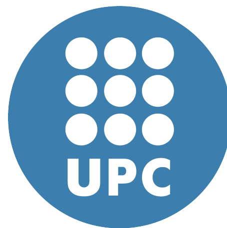
Multicore Architecture Prototyping on Reconfigurable Devices

Oriol Arcas Abella

Advisors:
Dr. Adrián Cristal Kestelman
Dr. Osman S. Ünsal
Dr. Nehir Sönmez

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Barcelona - March 2016

Multicore Architecture Prototyping on Reconfigurable Devices



Oriol Arcas Abella

Department of Computer Architecture

Universitat Politècnica de Catalunya

A dissertation submitted in fulfillment of
the requirements for the degree of

Doctor of Philosophy / Doctor per la UPC

February 2016

A la meva família, per fer-ho possible;
al Nehir, amic i company, l'abella obrera;
i a la Carlota, per recolzar-me fins al final.

Acknowledgements

This dissertation[1] condenses the research and efforts I made in the past years. However, what I value most is not the knowledge I accumulated, but the people I met and the experiences I lived during this journey. In the first place, Nehir Sönmez, friend and neighbour, the other worker bee, who introduced me into the FPGA world. I also want to thank my advisors, Dr. Osman S. Ünsal and Dr. Adrián Cristal, for always looking out for my best interest and for their endless support; and Eduard Aiguadé, my former advisor, for his wise advices. This thesis would not be possible without the direct collaboration from Gökhan Sayilar, Philipp Kirchhofer and Abhinav Agarwal. In addition, during these years I had the luck to meet and learn from many excellent researchers: Prof. Mateo Valero, Dr. Satnam Sing, Prof. Roberto Hexsel, Prof. Arvind and Prof. Mikel Luján.

My colleagues at the BSC helped me constantly and made our daily routine enjoyable, expecting the passionate lunch-time debates. At the risk of forgetting somebody, I want to thank Vasilis Karakostas, Adrià Armejach, Saša Tomić, Srđan Stipić, Ferad Zyulkyarov, Vesna Smiljković, Gülay Yalçın, Oscar Palomar, Javier Arias, Nikola Marković, Daniel Nemirovsky, Otto Pflucker, Azam Seyedi, Vladimir Gajinov, Vladimir Subotić, Behzad Salami, Omer Subasi, Tassadaq Hussain, Cristian Perfumo, Ege Akpinar, Pablo Prieto, Rubén Titos and Paul Carpenter.

One of the periods that I will remember from this thesis was my internship at the CSG of the MIT, kindly hosted by Prof. Arvind. He and his group made me feel like home, especially Myron King, Abhinav Agarwal, Asif Khan, Muralidaran Vijayaraghavan and Sang Woo Jun. I also want to thank Prof. Mikel Luján and his team from University of Manchester, Mohsen Ghasempour, Geoffrey Ndu, John Mawer and Javier Navaridas, for being excellent hosts.

Finally, I want to thank my parents and my brother for always supporting my career, and Carlota, for always enduring my career.

[1] The cover image "Rubik's Cube" by Wikipedia user Booyabazooka is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license:
https://commons.wikimedia.org/wiki/File:Rubik's_cube.svg

Abstract

In the last decades several performance walls were hit. The memory wall and the power wall are limiting the performance scaling of digital microprocessors. Homogeneous multicores rely on thread-level parallelism, which is challenging to exploit. New heterogeneous architectures promise higher performance per watt rates, but software simulators have limited capacity to research them.

In this thesis we investigate the advantages of Field-Programmable Gate Array devices (FPGA) for multicore research. We developed three prototypes, implementing up to 24 cores in a single FPGA, showing their superior performance and precision compared to software simulators. Moreover, our prototypes perform full-system emulation and are totally modifiable.

We use our prototypes to implement novel architectural extensions such as Transactional Memory (TM). This use case allowed us to research different needs that computer architects may have, and how to implement them on FPGAs. We developed several techniques to offer profiling, debugging and verification techniques in each stage of the design process. These solutions may bridge the gap between FPGA-based hardware design and computer architects. In particular, we place a special stress on non-obtrusive techniques, so that the precision of the emulation is not affected.

Based on the current trends and the sustained growth in the high-level synthesis community, we expect FPGAs to become an integral part of computer architecture design in the next years.

Contents

Contents ix

List of Figures xiii

List of Tables xv

1 Introduction 1
1.1 Field-Programmable Gate Arrays 2
1.2 FPGA-based Computer Architecture Prototyping 4
1.3 Simulation and Accuracy 5
1.4 Thesis Motivation 8
1.5 Thesis Contributions 9
1.5.1 Implementing an Open-Source Multicore: Beefarm and TMbox 10
1.5.2 High-level Methodologies, Debugging and Verification: Bluebox 12
1.6 Acknowledgements 15
1.7 Thesis Organization 16

2 Background 17
2.1 The Rise of Multicore Architectures 17
2.2 The Limitations of Software Architectural Simulators 20
2.3 FPGA-based Simulators and Emulators 21
2.4 The Limitations of FPGA-based Prototyping 22

3 An Open-Source FPGA-based Multicore 25
3.1 Introduction 25

3.2 Transactional Memory 28
3.3 The Beefarm System 30
3.3.1 The Plasma Core 30
3.3.2 The BEE3 Platform 31
3.3.3 The Honeycomb core: Extending Plasma 31
3.3.4 The Beefarm System Architecture 35
3.3.5 FPGA resource usage 37
3.3.6 The Beefarm Software 39
3.3.7 Investigating TM on the Beefarm 40
3.4 Comparison with SW Simulators 41
3.4.1 Methodology 41
3.4.2 Beefarm Multicore Performance with STM Benchmarks 43
3.5 Efficient Implementations for Floating-point Support 44
3.6 The Experience and Trade-offs in Hardware Emulation 47
3.7 Conclusions 49

4 Hybrid Transactional Memory on FPGA **51**
4.1 Introduction 51
4.2 The TMbox Architecture 53
4.2.1 Interconnection 53
4.3 Hybrid TM Support for TMbox 54
4.3.1 Instruction Set Architecture Extensions 56
4.3.2 Bus Extensions 57
4.3.3 Cache Extensions 57
4.4 Experimental Evaluation 58
4.4.1 Architectural Benefits and Drawbacks 58
4.4.2 Experimental Results 61
4.5 Conclusions 62

5 Profiling and Visualization on FPGA **63**
5.1 Introduction 63
5.2 Design Objectives 65
5.2.1 TMbox Architecture 66

5.2.2 Network reuse . . . 67
5.3 The Profiling and Visualization Infrastructure . . . 68
5.3.1 Event specification and generation . . . 68
5.3.2 Event propagation on the bus . . . 70
5.3.3 Transfer of events to the host . . . 71
5.3.4 Post-processing and execution regeneration . . . 71
5.4 Experimental Evaluation . . . 71
5.4.1 Runtime and area overhead . . . 73
5.4.2 Improvement Opportunities . . . 75
5.5 Conclusions . . . 78

6 High-level Debugging and Verification 79
6.1 Introduction . . . 79
6.2 The Bluebox Synthesizable Multicore . . . 81
6.3 Software Extensions For High-Level Debugging . . . 83
6.3.1 Externalizing The State . . . 84
6.3.2 Debugging And Profiling . . . 85
6.3.3 Checkpointing . . . 87
6.3.4 Verification . . . 88
6.4 FPGA-based, On-Chip Debugging And Verification . . . 89
6.4.1 The Debugging Mode . . . 91
6.4.2 The Verification Mode . . . 91
6.5 Experimental Evaluation . . . 92
6.5.1 The Suitability Of On-Chip Verification . . . 93
6.5.2 Use Cases . . . 95
6.6 Conclusions . . . 95

7 Characterizing the High-level Descriptions 97
7.1 Methodology . . . 98
7.2 Results . . . 100
7.2.1 Reed-Solomon decoder . . . 103
7.2.2 Bluebox . . . 103
7.3 Automatic Architectural Optimization . . . 105

7.4 Conclusions . . . 106

8 Related Work **107**

8.1 FPGA-based Multicore Models . . . 107

8.2 Transactional Memory on FPGA . . . 112

8.3 Profiling . . . 112

8.4 Comparing and Analyzing High-Level Languages and Tools . . . 113

8.5 On-chip Verification and Debugging . . . 115

9 Conclusions **117**

9.1 Research Opportunities . . . 120

10 Publications **121**

A High-level Hardware Design Languages **123**

A.1 Evaluation of HLS Languages and Tools . . . 123

A.1.1 Studied Database Algorithms . . . 125

A.1.2 Programming Experience Evaluation . . . 129

A.2 Empirical Evaluation and Comparison . . . 131

A.2.1 Analytical Analysis . . . 133

A.2.2 Experimental Results . . . 133

A.2.3 Discussion . . . 135

References **137**

List of Figures

1.1 Example FPGA architecture 2
1.2 CPU trends vs. FPGA trends 3

2.1 Trends in computation 19

3.1 Lock-based vs. TM-based atomic blocks 29
3.2 Honeycomb CPU 32
3.3 Beefarm cache state diagram 34
3.4 The Beefarm multiprocessor system. 36
3.5 Beefarm implementation alternatives 38
3.6 Beefarm resource usage 39
3.7 Compare and Swap using LL/SC in MIPS assembly. 41
3.8 Beefarm vs. M5: ScalParC, SSC2 and Intruder 42
3.9 Beefarm vs. M5: ScalParC simulation time. 43
3.10 FPU implementation alternatives 47

4.1 TMbox architecture 54
4.2 MIPS assembly for atomic increment 57
4.3 TMbox cache state diagram 59
4.4 SSCA2 and Intruder benchmarks. 61

5.1 TMbox with profiling extensions 67
5.2 Event format for the profiling packets. 69
5.3 STAMP-Eigenbench Benchmark Overheads 74

6.1 Bluebox core 82
6.2 Execution trace in Paraver 86
6.3 Verification of `timesim` using `funcsim` 88
6.4 Debugging and verification in `fpgasim` 89
6.5 Filter Unit and Verification Unit 90
6.6 Performance in verification mode 93
6.7 Instruciton verification breakdown 93
6.8 Verification data processed by `fpgasim` (GB/s). 94

7.1 Example annotation of GCD 99
7.2 Block delay calculation algorithm 100
7.3 Bluebox and Reed-Solomon decoder analysis 101
7.4 Area, delay and power metrics on three FPGAs 102

A.1 The Bitonic sort, spatial sort and median operator algorithms 126
A.2 Bitonic sorter code snippets for each HLS framework 128

List of Tables

1.1 Multiprocessor architecture simulator comparison 6
1.2 Multicore prototypes developed during this thesis 10

2.1 Terminology used in this thesis. 21

4.1 LUT occupation of components 53
4.2 HTM instructions for TMbox 55
4.3 TM Benchmarks Used in TMbox 60

5.1 Area overhead and tracked events per profiling option 72

8.1 FPGA-based Designs for Hardware Prototyping 108

A.1 Analytical baseline models 131
A.2 Four algorithms in five environments 132

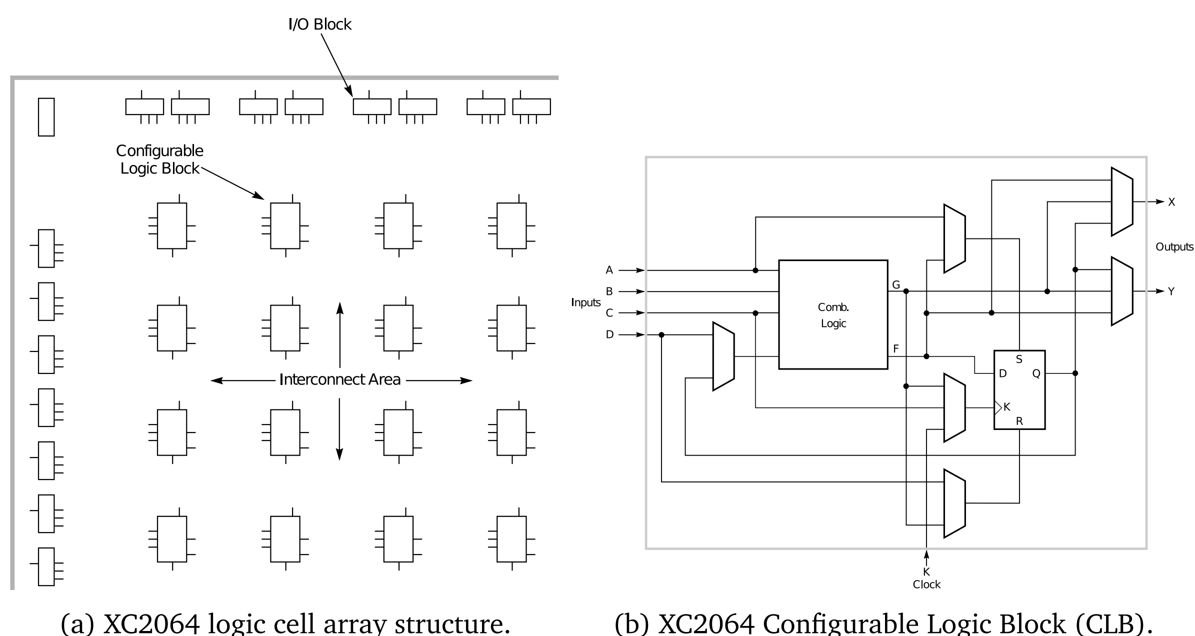
1 Introduction

In the last decade multicore architectures have emerged due to the end of the performance scaling trends of single core computers. However, exploiting the Thread-Level Paralellism (TLP) that these architectures can offer is more challenging than exposing the traditional Instruction-Level Parallelism (ILP). Computer architects need powerful tools to tackle these problems, like architectural simulators to validate their proposals.

Software architectural simulators are widely used for this purpose, but multicore architectures are more demanding in terms of performance and precision than uni-core systems. There are two main causes for this phenomenon. On one hand, the algorithms used by software architectural simulators are difficult to parallelize without losing precision. On the other hand, multicore architectures multiply the amount of cores to be simulated. Thus, researchers have to simulate more cores using quasi-sequential simulators. The answer to this problem has generally been to trade off performance for precision.

1.1 Field-Programmable Gate Arrays

In 1960, Gerald Estrin proposed a “fixed-plus-variable” structure computer to extend “the borders of practicable computability” [63]. He proposed a computer with hardware that could be reconfigured, capable of accelerating some algorithms several orders of magnitude compared to software-based computers. Two decades later, Ross Freeman and Bernard Vonderschmitt presented the first commercial Field-Programmable Gate Array (FPGA), a general purpose computer with programmable logic gates and programmable interconnects. Figure 1.1 depicts the basic architecture of their first device, the XC2064 FPGA [46].



(a) XC2064 logic cell array structure.

(b) XC2064 Configurable Logic Block (CLB).

Figure 1.1: Example FPGA architecture: the Xilinx XC2064 FPGA.

The XC2064 had 8 × 8 Configurable Logic Blocks (CLBs), arranged in an 8-row by 8-column matrix (see Figure 1.1a). Each CLB had a combinatorial logic unit, a storage element, and routing and control circuitry. The combinatorial logic unit was a programmable truth table, or Look-Up Table (LUT), to implement Boolean functions. Each CLB also had a number of inputs and outputs. In Figure 1.1b a 4-input, 2-output CLB is shown. The array of CLBs could be configured to emulate a given digital circuit.

Modern FPGAs are based on the earlier XC2064, and their basic architecture is very similar. But modern devices may contain more than one million LUTs, perform at hundreds

of MHz, and embed hard blocks such as RAMs and Digital Signal Processing units (DSPs). In Figure 1.2 we compare the trends in number of logic elements and maximum frequency for CPUs and FPGAs. FPGA's trends in logic elements and maximum frequency evolve in parallel with CPU's trends. However, in contrast to CPUs, FPGAs can accelerate certain classes of algorithms running at much lower frequencies than traditional computers: on average, the exponential trend of FPGA frequencies is lower than that of CPU frequencies (as shown in Figure 1.2). Thus, for these applications FPGAs can obtain similar or higher performance rates consuming less energy.

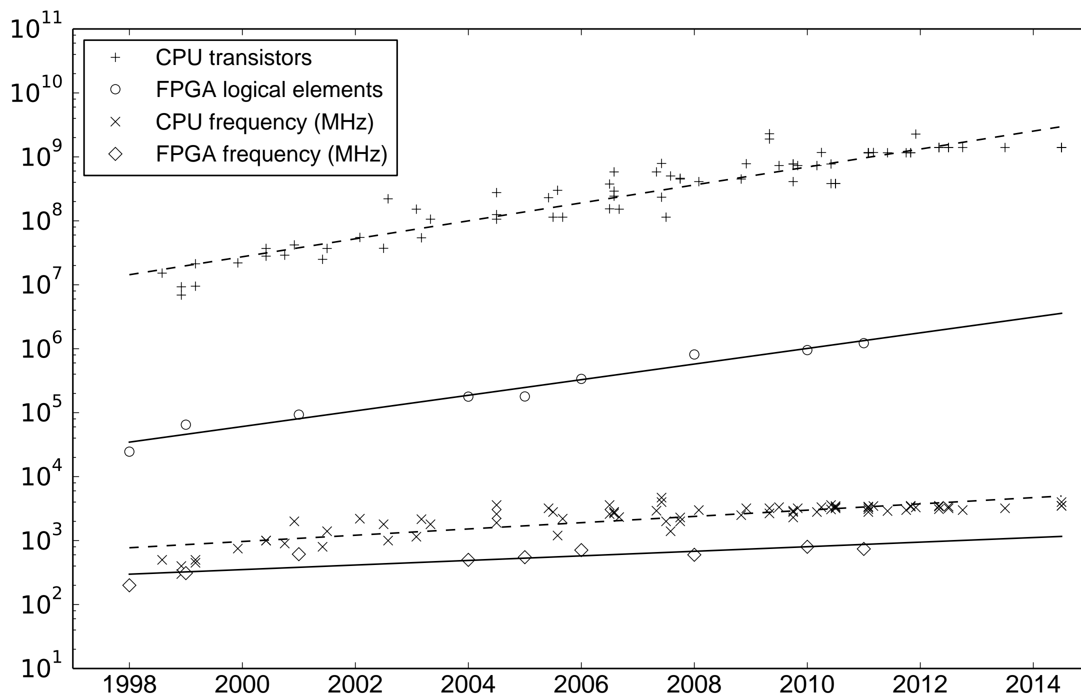


Figure 1.2: CPU trends vs. FPGA trends. CPU data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. FPGA data collected by Shannon *et al.* [111]. Trend lines are showed as dashed (CPU) and solid (FPGA) lines.

Two factors determine the performance of FPGAs, compared to traditional computers. Traditional Von Neumann computers use a central unit to process streams of software and data, which can become a bottleneck [29]. Moreover, traditional computers offer the illusion of a flat memory space, which can be accessed randomly by the software. Maintaining this illusion at reasonable performance rates requires complex architectural optimizations, such as cache hierarchies and prefetching mechanisms. However, the memory wall dominates the performance of CPU-based computers.

In contrast, in FPGAs the computation does not need to be centralized, but spread over thousands of small hardware units, which can be run in parallel. Data is streamlined through this grid of processing elements. In data-flow computers, the performance depends on the availability of data and the latency of the computing units.

1.2 FPGA-based Computer Architecture Prototyping

FPGAs are programmed using tools similar to those in computer aided hardware design, and once programmed they can emulate digital circuits. Thus, not only FPGAs can be used as high-performance accelerators, but also as hardware prototyping platforms.

Reconfigurable devices are great platforms for multicore prototyping, and FPGA-based simulators can overcome the limitations of software-based simulators.

The available logic elements allow to emulate small and medium multicores of some tens of cores per chip. Dedicated hardware, such as RAMs and DSPs, allows implementing caches, memories or arithmetical units efficiently in terms of area and performance. In addition, the ecosystem of available *soft cores* or Intellectual Property (IP) cores, *i.e.* hardware units that can be included in FPGA designs, has been growing as FPGAs are being adopted by wider communities. These IP cores can be commercial or open-source, and they range from simple arithmetic units to microprocessors or encryption modules. Reusing these modules improves the productivity when implementing large designs like multicore architectures.

Some FPGAs also contain *hard cores*, or on-chip non-reconfigurable hardware units. These units can be dedicated hardware for common tasks, such as Static RAMs, Digital Signal Processing (DSP) units, and hard Ethernet and PCI Express cores. Including them saves the effort of implementing the functionality using reconfigurable logic, which is area-expensive and less efficient compared to silicon circuits. FPGAs can also contain general-purpose hard cores, such as microprocessors, which converts the device into an heterogeneous architecture.

The hardware designs can described with Hardware Description Languages (HDL). These languages are based on Register-Transfer Level (RTL) abstractions to describe synchronous digital circuits. Verilog and VHDL are the most prominent HDLs, widely used in the hardware design industry and *de facto* standards.

Nevertheless, FPGAs also have limitations, and the existing FPGA-based multicore pro-

totypes may not suit all the needs of computer architects: legacy Hardware Description Languages (HDL) have a steep learning curve and have low productivity; FPGA designs are difficult to debug and verify; and many CPU prototypes are either too big for multicore research, or closed source.

1.3 Simulation and Accuracy

We must define what we understand by computer architecture simulation and its *accuracy*, and the existing approaches to this problem. In Table 1.1 we introduce the different types of software-based (SAME) and FPGA-based (FAME) simulators available, following the taxonomy by Tan *et al*. [117] and their Experiments per Day performance metric. **The Beefarm, TMbox and Bluebox prototypes shown in the table are derived works from this thesis.**

The term *emulator* denotes a process that provides the similar results as another process, the target model, by different means. The term *simulator* refers to a process that provides similar results as another process by imitating its internal behavior. In Section 2.3 we discuss about this possible taxonomy, and why we use the general term *simulator*. In both cases, the objective is the same, but the methods differ.

There exist two fundamental properties of computer architectures that simulators must model. On one hand, the functional behavior is *what* the architecture does, *i.e*. the transformations to the state of the computer. A functional simulator is usually utilized to validate the *correctness* of an architecture. On the other hand, the timing behavior is *when* the operations are performed, and for how long. **Some simulators only have a functional model; others allow to modify both models independently (all the SAME tools, and the abstract, multithreaded and hybrid FAME tools).**

Simulators have different scopes. For instance, a *microarchitectural simulator* will just simulate the processing unit, while a *full-system simulator* will also simulate other components of the system, like the memory hierarchy or the I/O hardware. **The column FS in Table 1.1 indicates which simulator types are full-system.**

Some simulators only support user-space instructions, and are targeted to study the behavior of the architecture from the point of view of application execution (in some cases, a single application). These simulators do not model in detail, or do not model at all, the interferences between applications, or the effects of the operating system and hardware

Table 1.1: Multiprocessor architecture simulator comparison

Type*	Host *f* (MHz)	EPD†	CA	FS	ISA	Examples	Extras‡
Industrial SAME	0.01	0.43	Yes	Yes			
Statistical SAME	0.20	8.64	No	Yes	x86	MARSS [103], gem5 [36]	
Interval SAME	20.00	864.00	No	No	x86	Sniper [45], Zsim [108]	
Direct FAME	100.00	11.97	Yes	Yes	MIPS, RISC-V	BERI [130], Sodor [48]	D
	50.00	11.93	Yes	Yes	MIPS	Beefarm/TMbox	P
Direct/decoup. FAME§	100.00	11.97	Yes	Yes	MIPS	Bluebox	PD, PV
Decoupled FAME	50.00	24.00	Yes	Yes		Green Flash [117]	
Abstract FAME	65.00	40.00	Yes	Yes	MIPS, Alpha, PowerPC	HAsim [104] Arete [83]	PD
Multithr. FAME	90.00	128.00	Yes	Yes	SPARC	RAMP Gold [116]	D
Hybrid SAME/FAME	100.00	¶	No	Yes	SPARC	ProtoFlex [54]	D

*SAME and FAME taxonomy from Tan *et al.* [117].
†Experiments per day as defined by Tan *et al.* [117]: number of possible 1-second experiments with variations on the architectural parameters, limited by the performance of the tool. For Direct FAME, a 2-hour synthesis time per experiment was assumed. For other FAME tools, the maximum number of experiments that do not require re-synthesis is shown [117]. SAME tools do not require re-compiling; frequencies obtained from [45, 108].
‡Abbreviations for Debugging, Profiling, Precise Debugging and Precise Verification.
§Bluebox is a Direct FAME with decoupled debugging and verification.
¶In hybrid SAME/FAME simulators, the functional model and the timing model are split between the host and the FPGA, which acts as an accelerator. The possible experiments per day depend on what part runs on the host, and thus can be modified without recompiling the hardware side.

management over the execution.

The techniques used to simulate architectures vary, and we discuss them in Section 2.2. In general, simulators are used because it would be too expensive to implement the real hardware for each new architecture or extension, so cheaper technologies and techniques are utilized. Software programs use many operations to simulate a single operation, resulting in long simulation times. In addition, the inherent parallelism of hardware is difficult to be simulated with software, which in the era of multicores increases the simulation workload. **Statistical models can be used to obtain the same result with a high probability of correctness (Statistical SAME). Another technique is to parallelize the simulation**

and synchronize the execution threads periodically (Interval SAME). In both cases, errors appear, but are tolerated in small amounts.

FPGA-based models follow different approaches. Direct FAME tools simulate the functional and timing models on the FPGA. Decoupled FAME tools may require more FPGA-cycles to simulate a single target model cycle, which simplifies implementation. Abstract FAME tools separate the functional and timing models, allowing for independent modifications. Multithreaded FAME tools run multiple functional and timing models on a single FPGA design, time-multiplexing the simulation. Finally, Hybrid SAME/FAME tools run one of the functional or timing models on a SAME tool, accelerating the other model on a FAME tool.

Regarding the term *accuracy*, we understand that it is the level of fidelity between the simulator and the target architecture. In the case of functional behavior, in general both models should be equal. For timing models, the accuracy is measured in terms of execution time variations. To measure the accuracy of a simulator, it is often configured like an existing architecture and the variations in execution time for different benchmarks are reckoned. State-of-the-art software simulators report timing results within an error range of 10% (Section 2.2). **We use the term cycle-accuracy to refer to a null variation in terms of clock cycles.** The column CA in Table 1.1 shows which tools are cycle-accurate.

We also show some extra features that FAME tools support, like debugging, profiling and verification mechanisms. As we explain below, our prototypes support non-obtrusive profiling, precise debugging and precise verification. Other FAME tools only support non-precise debugging (the debug actions interfere with the execution).

Finally, also in Section 2.2 we discuss the suitability of validating simulators with existing architectures. The accuracy of the simulator can be compared to an existing model, but it is not clear that the same level of accuracy will remain when used for a new, non-existing architecture. Thus, can non-cycle-accurate simulators be used to validate new architectures? We believe that cycle-accuracy should be pursued for computer architecture research. In Chapter 8 we discuss different approaches and proposals to architecture simulation.

In Chapter 2 we discuss in detail the trends that favored the emergence of multi-core architectures (Section 2.1), the limitations of software simulators (Section 2.2), and the advantages and limitations of FPGA-based prototypes (Section 2.4).

1.4 Thesis Motivation

We observe how multicore and heterogeneous architecture research is being hindered by the low performance of software-based multicore simulators. Application-level-only results with 10% of errors are assumed as normal in the computer architecture community. As we defend in this thesis, FPGAs could help overcoming these limitations. **Nevertheless, there are three main reasons that may prevent the adoption of FPGAs for computer architecture research:**

1. The synthesis of hardware requires long compilation times, which disrupts the development process and reduces the experiments/day ratio [117].

2. Computer architects are reluctant to use FPGA development languages and tools because of their low productivity and steep learning curves [45, 108].

3. The FPGA technology appears as a black box to designers, being difficult to profile, debug and verify compared to software solutions.

In this thesis we do not cover the first problem. But it can be alleviated using novel methodologies, like FPGA overlay architectures [41], which simplify hardware design providing pre-synthesized, coarse build blocks. We believe that overlay architectures are complementary to the second problem, hardware design languages and tools, and probably will converge in the future.

The low productivity of legacy hardware description languages and tools has become a hot topic in the academy and the industry. As a result, in the past decade several new languages and tools have been proposed to simplify system-level hardware design. Currently, there exist several FPGA-based multicore models ready to use. But many of them have been developed using low-level languages, not suitable for large multicores. Multicore prototypes described in high-level languages may reduce the implementation time of novel architectures. In addition, many of the available soft cores for FPGA model large architectures. Such models are appropriate for microarchitectural simulation, but only a few of them can be fit in a single FPGA. Novel multicore research requires tens or hundreds of cores to be simulated.

Another problem of the existing FPGA prototypes is related to the accessibility of the designs. Commercial CPU prototypes are reliable platforms for industrial applications. But

their source code is not available and it becomes harder for researchers to develop architectural changes. In this sense, the most popular software simulators in the academia are open source.

The languages and tools used to implement the architectures involve an additional issue: the development methodology. The physical constraints imposed by the FPGA technology, which is a central topic in FPGA design, are known at the latter stages of the development process. This development process is partitioned in tens of phases, involving dozens of tools and orthogonal constraints (time closure, area, power). High-level languages and tools may worsen this situation, adding more layers of abstraction to the process. For instance, many high-level languages generate low-level descriptions in legacy languages, which adds another stage to the development process for the sake of productivity. Thus, it is becoming increasingly difficult to relate the physical problems to the original high-level description. At the same time, high-level languages may bring the solution to this problem, allowing for more flexible development methodologies and smarter tools.

Finally, gathering results and verifying their correctness are central tasks in architectural simulation. However, once configured, FPGAs appear as black boxes to the developer and are complex to inspect and debug. Profiling, debugging and verification still are complex tasks on FPGA-based multicore simulators.

1.5 Thesis Contributions

We extended two existing softcores to obtain three multicore prototypes, Beefarm, TMbox and Bluebox. In each of them we have focused on different aspects of the problematics introduced before. Beefarm and TMbox are closely related, both adapted from an existing open-source design and TMbox being an improvement and specialization of Beefarm. Bluebox represents a major leap following the lessons learned from Beefarm and TMbox. Thus, one of the possible ways to divide this thesis is in a first part, including Beefarm and TMbox (Chapter 3, Chapter 4 and Chapter 5), and a second part, including Bluebox and the related studies (Chapter 6, Chapter 7 and Appendix A).

The main characteristics of our three prototypes are shown in Table 1.2. In a single FPGA, we could fit up to 8 and 16 cores (Virtex 5 FPGA technology), and 24 cores (Virtex 7 FPGA technology), respectively. **All of them implement the well-known MIPS architec-**

Table 1.2: Multicore prototypes developed during this thesis

Prototype	CPU stages	ISA	Cores	FPGA tech.	Lang.	Kernel	Features
Beefarm	3	MIPS I	8	Virtex 5	VHDL	HAL[1]	STM
TMbox	3	MIPS I	16	Virtex 5	VHDL	HAL[1]	HyTM, profiling
Bluebox	5	MIPS I	24	Virtex 7	BSV	Linux	Debugging, verif.

[1] Hardware Abstraction Layer.

ture and a coherent memory hierarchy, and their hardware is completely modifiable.

1.5.1 Implementing an Open-Source Multicore: Beefarm and TMbox

In our first prototype we attempted to answer the following questions: Are FPGA-based multicores faster than popular software simulators? Can a prototype be developed with minimum effort and cost? Can it be used for actual computer architecture research?

Beefarm is the answer to these questions. The implementation strategy was designed to minimize the development time and costs, while preserving maximum accessibility to the code. One option would be using commercial CPU prototypes for FPGA, like Xilinx Microblaze [7] and Altera Nios [9]. However, they are closed-source and it is difficult to alter their microarchitecture. We decided to take an open-source CPU and integrate the necessary off-the-shelf open-source components. That would save development time while guaranteeing full source code access. **In Beefarm (Chapter 3), we demonstrated how to reuse already-existing open-source components, and efficiently using the FPGA's specialized hardware units.** We extended an open-source core with virtual memory, coherency and multicore synchronization.

As a use case for architecture research, we developed Beefarm to investigate its applications for Transactional Memory (TM), a programming paradigm for parallel applications without locks. We explain TM in detail in Section 3.2, which can be implemented in software only (STM), hardware only (HTM) or in a mixed mode (Hybrid TM). Our research group has an strong background on this topic, and using FPGAs was both an ideal use case for this thesis and an enriching technology for other researchers from our group. As a result, Beefarm was designed with the necessary extensions to run TM applications on 8 cores with coherent caches.

We compared Beefarm with a state-of-the-art simulator. Even though Beefarm was not optimized for high performance, we outperformed the software simulator with a speedup of 8x. For the high-precision mode of the software simulator, the speedup reached 20x. One of the main factors of this superior performance is the massive parallelism of FPGAs, which in contrast to software simulators scales with the number of cores.

FPGAs can be great tools for simulating digital circuits, like computer architectures. But their technology is better suited for certain hardware components than others. For instance, long and wide buses are highly inefficient in FPGAs. This was one of the main bottlenecks in Beefarm, whose shared bus limited the number of cores that could be mapped to the FPGA.

In TMbox (Chapter 4), we addressed the architectural problems of Beefarm and further extended it to support Hybrid TM. That is, a TM implementation that benefits both from software and hardware mechanisms. The shared bus was replaced with a ring bus, whose short core-to-core connections map well to the FPGA fabric. The ring bus had an innovative coherence mechanism based on *backwards destructive invalidations,* explained in Section 4.2.1. **Thanks to the small CPU adapted from Beefarm and the FPGA-friendly ring bus we could map up to 16 cores in a single Virtex 5 FPGA.**

We developed TMbox as an the first FPGA-based multicore with Hybrid TM support. Among the contributions of this thesis there is the extension of the MIPS ISA with new TM instructions, and the necessary modifications to the memory system to implement the TM mechanisms. **We also extended the GCC compiler suite to support our new machine instructions.** The Hybrid TM was inspired by the AMD Advanced Synchronization Facility (ASF) [53]. We adapted a popular TM library, TinySTM [65], to support our new extensions. This way, we could run three popular benchmarks with all the different flavors of TM (software, hardware and hybrid). **TMbox helped to investigate Hybrid TM, supporting the hypothesis that it can outperform STM- and HTM-only solutions.** Thus, we validated the idea of this thesis that FPGA-based prototypes can help computer architects to verify their proposals.

However, the results obtained with TMbox were limited to absolute numbers, such as execution time. The dynamic nature of TM, where the interactions between the atomic transactions determine the performance of the system, required greater insight, but the black-box nature of FPGAs complicated the observation of the internal state of the software and the hardware.

We investigated the possibility of real-time, non-obtrusive profiling of our Hybrid TM system (Chapter 5). Based on TMbox, we could fully profile TM applications on real time though a PCI connection. In this thesis we developed the profiling mechanism that allowed us to graphically display the behavior of the transactions, discovering pathologies and improvement opportunities.

The profiling system had two main design specifications. First, being non-obtrusive, *i.e.* not altering the behavior of the studied application. And second, being able to profile not only hardware events, but also software and mixed software-hardware events. For this purpose, we instrumented the TM mechanism and designed a message format to output architecture events through a PCI bus. **The events were cycle-accurate, the hardware events had zero overhead, and the software events incurred in much less overhead than a software profiling solution.** The level of detail was much higher than the software profiler. For instance, our system was able to profile all the data affected by a transaction and the interactions between transactions. In the most detailed profiling level, our system only incurred in 14% of overhead. We extended GCC again to support new instructions for software-level profiling.

We also implemented a converter to transform our traces into the Paraver [106] format. Paraver is a tool to visualize architectural events from supercomputers. Thanks to TMbox and the profiling system, TM researchers from our group could identify pathologies in the applications, optimization opportunities, and behavioral differences between the results from regular TM benchmarks and the results from synthetic TM benchmarks.

1.5.2 High-level Methodologies, Debugging and Verification: Bluebox

From the experience with Beefarm and TMbox we learned some valuable lessons. One lesson was that the legacy hardware design languages that we used in Beefarm and TMbox were limiting our productivity. A second lesson was the need of a standard software stack. In Beefarm and TMbox we implemented a simple hardware abstraction layer with a subset of the standard C library. Our tests were similar to those in many software simulators: user-level only. The influence of the system operations (interrupts, exceptions, virtual memory, *etc.*) was unknown.

Regarding the low-level languages, we performed a detailed comparison of the new high-level languages and tools (Appendix A). We started classifying them and choosing

representative candidates. Then, we performed a quantitative and qualitative comparison using data-processing algorithms and analytical baselines. The lack of systematic studies comparing hardware description languages and tools was surprising, especially since the appearance of many mature proposals. This comparison helped us to choose a high-level language, Bluespec SystemVerilog (BSV) [1], to replace the legacy ones. One of the deciding factors was the flexibility of BSV to integrate other languages, or extending its simulator.

We also decided that supporting the Linux Kernel and a standard software stack was necessary. Not only for the credibility of the prototype, but also because standard software libraries and tools reduced the implementation and debugging time. It can be exasperating for developers having errors in a prototyping platform and not knowing if it is caused by the hardware or the software. Using the Linux Kernel and the standard C library, one of the variables was fixed.

We applied such lessons in a new prototype, Bluebox (Chapter 6). We redesigned our architecture, using high-level tools during the implementation. As in Beefarm and TMbox, we did not start from scratch, but we reused a basic MIPS CPU in BSV. This original CPU was targeted for educational use, lacking basic functionalities like a multiply unit or all the interruption/exception mechanism (in MIPS known as Coprocessor 1). We implemented the necessary functionalities to support the full MIPS I ISA. We also ported the Linux Kernel to our platform, being able to boot multicore GNU/Linux environments. The cache hierarchy of Bluebox is more realistic than those from Beefarm and TMbox, having direct-mapping level 1 caches a set-associative level 2 cache. **We also used a bigger FPGA, being able to map up to 24 cores in a single chip.** We preserved the ring topology, which allowed us to easily add extra nodes to the network, like serial port I/O units.

Nevertheless, using high-level tools may increase the distance between the abstract design and the final hardware, making FPGA designs even harder to debug and verify. At the same time, high-level languages and tools allow for new techniques to solve these problems. **For this reason, for Bluebox we developed a new design methodology that allowed us to perform debugging and verification on each of the three design stages that we defined: functional simulation, timing simulation, and FPGA simulation.**

Our design methodology has a strong accent on non-obtrusive debugging and verification in all the design stages, both the simulation on the host or the real execution on the FPGA. **In our methodology, the designers can debug or verify the multicore system in a non-obtrusive way using software-class tools and metaphores.** This includes with

graphical user interfaces, interfaces to other languages like Python for third-party front-ends, and support for popular software debuggers like GDB. **This support is system-wide, allowing the user to debug and verify all the CPUs both in user or system level.**

One key concept in our methodology is reusing the functional model in subsequent stages of design. For instance, a simple software model can be used to design the functionality. This software model is later used in the hardware design simulated on a host, and in the FPGA in a live execution. In Section 6.3 we describe how we use elastic buffers to verify the design, and how we instrument the BSV simulator allowing for external control and state inspection by a debugger. In the FPGA execution stage, we isolate the design in a gated clock domain to preserve the cycle accuracy. In Section 6.4 we describe the techniques that allow for non-obtrusive debugging, preserving the memory and I/O delays and zero-overhead verification of 99.5% of the instructions. **Using 24 cores, we can verify 100% of the instructions, including system events like interruptions and exceptions, with 7x slowdown. That is, verifying 17 million instructions per second with virtually no behavioral interferences.**

We also developed a technique to analyze the impact of the FPGA technology over one of these languages (Chapter 7). As we said, the physical constraints imposed by the FPGA technology are not known at design time, but after a long build process. The different paradigms in both ends of the design flow (*e.g.*, FPGA gates *vs.* Bluespec SystemVerilog) make it difficult for the user to map the physical symptoms to the high-level causes. **Our technique can automatically estimate the area, performance and power characteristics of the high-level models when synthesized on a given FPGA platform.** One of the side benefits of this method is that it can compare the effects of placing the design on different FPGA devices.

With this methodology, we analyze the effects of mapping the high-level descriptions of the Bluebox CPU and a Reed-Solomon decoder over three different FPGA devices, relating the timing, area and power results to the original high-level components in BSV. **For this purpose, we developed a tool to automatically analyze the BSV intermediate representation, non-obtrusively instrumenting the resulting low-level hardware description, and finally estimating the physical properties of the FPGA gates and wires over these components.**

1.6 Acknowledgements

Unless otherwise specified, all the contributions in this thesis have been made by the author solely. The multicore prototypes have been adapted from existing softcores, Plasma and the educational core sMIPS. In both cases they have been extended to support the MIPS I ISA and become a multiprocessor system-on-chip running an operating system. All the modifications and extensions to the initial designs, except for the contributions from other authors listed below, have been implemented by the author. These modifications that are the outcome of the author's work, include: implementing the TLB and MMU units; the interrupt and exception mechanism; the atomic memory synchronization instructions (LL/SC); the timing and I/O hardware, in our systems packed in Coprocessor 2, including the UART and keyboard drivers for Bluebox; the shared and ring buses; adapting the memory hierarchy to support partial-word accesses and associative sets; the coherency mechanism used in TMbox; the DDR and clocking infrastructure for Virtex 7 used in Bluebox; the TM instructions and hardware; the profiling instructions and hardware; the conversion engine from TMbox events to Paraver traces; the modifications to the C compiler; adapting the TM library to our Hybrid TM mechanism and to the MIPS architecture; developing the hardware abstraction layer for Beefarm; porting the Linux Kernel to Bluebox, including adapting the R3000 Linux ports to a multiprocessor; the Bluespec SystemVerilog module analyzer and the Verilog annotation mechanism; the development methodology for Bluebox; the gated clock mechanism for Bluebox and its elastic I/O buffers; and the checkpointing, debugging and verification mechanisms in Bluebox.

We want to credit the contributions of other PhD students and researchers to this thesis. Dr. Nehir Sönmez is both a director and a prominent co-author. He guided the adaptation of Beefarm to the BEE3 infrastructure (DDR, clocks, *etc*.), including the PCIe hardware and software drivers used in Chapter 5, and he learned and literally brought Bluespec SystemVerilog to Barcelona (and probably to Spain). He designed and analyzed the TM tests that we made in Beefarm and TMbox, including the comparison between STM, HTM and Hybrid TM in Section 4.4 and the optimization opportunities and the analysis of pathologies shown in Section 5.4. Gokhan Sayilar, from Sabanci University and now at UT-Austin, developed the shared FPU for Beefarm. Philipp Kirchofer, from the Karlsruhe Institute of Technology extended the memory bus to send profiling messages, performed stress tests of the profiling system and provided a software profiling baseline. These projects have been

altruistically advised by Dr. Roberto Hexsel (Universidade Federal do Paraná), Dr. Satnam Singh (Microsoft Research, now Google) and Dr. Ibrahim Hur (BSC, now at Intel).

Dr. Mikel Luján and his research group at the University of Manchester, especially Dr. Mohsen Ghasempour, Dr. Geoffrey Ndu, Dr. John Mawer, Dr. Wei Song and Dr. Javier Navaridas, developed some of the implementations found in the language comparison in Appendix A. Bluebox has been adapted from an educational prototype developed at the MIT by Dr. Arvind and his research group. The power analysis technique and the Reed-Solomon decoder implementation described in Chapter 7 have been developed by Dr. Abhinav Agarwal (MIT, now Oracle). He, Dr. Myron King and Dr. Asif Khan greatly helped with their advise to the development of Bluebox.

The research leading to the results presented in this thesis has been supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625, TIN2008-02055-E, and by the UPC project TIN2012-34557. It has also been funded by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and the European Commission FP7 projects VELOX (216852) and AXLE (318633), and by the Short Term Scientific Mission grant from the European Cooperation in Science and Technology Action IC1001 (EuroTM).

1.7 Thesis Organization

This thesis is organized as follows. In Chapter 2 we discuss the limitations of software simulators and the challenges of FPGA-based simulators. In Chapter 3, we present our experience while developing Beefarm. In Chapter 4 we extend and improve Beefarm to support Hybrid TM. In Chapter 5 we implement profiling support for TMbox. In Chapter 6 we present Bluebox, a multicore prototype designed with a high-level language. Bluebox supports debugging and verification during all the design process. In Chapter 7 we show how to map the technology impact over the original high-level description of Bluebox and another design on 3 different FPGAs. In Chapter 8 we compare our work with the current state of the art. Chapter 9 concludes this dissertation, and Chapter 10 contains a relation of all the publications derived from this thesis. Finally, in Appendix A we provide a detailed comparison of four high-level languages and tools.

2 Background

In this chapter we discuss how the recent trends in computer architecture favored the emergence of multicore architectures, and the limitations of software simulators under this new paradigm. We briefly introduce FPGAs, and we show how these devices can improve the research of parallel and heterogeneous architectures. We conclude the chapter summarizing the current problems of FPGA-based multicore prototypes, which motivate this thesis.

2.1 The Rise of Multicore Architectures

In the last decades, microprocessor-based computing has been driven by two fundamental scaling trends. On one hand, the number of transistors in a silicon chip has been doubling every 18 months, a trend observed by Gordon E. Moore [94]. On the other hand, Robert H. Dennard also observed that the power density of transistors would remain proportional to the area, but not the number of transistors [60]. This means that current and voltage also scaled linearly with size. Combined, Moore's Law and Dennard scaling allowed implementing larger and more power-efficient computer architectures.

For each new generation of transistors, the performance of microprocessors was increased in two ways. First, Dennard scaling would favor higher switching frequencies, which would mean faster microprocessors. And second, the ever-growing number of transistors per chip could be used to increment the Instructions Per Cycle (IPC), thanks to techniques such as pipelining, memory caching, branch prediction and out-of-order execution. Thus, the performance in terms of Instructions Per Second ($IPS = IPC \times f$) would increase faster than the Moore's law predicted, without requiring significant changes to the software [101].

However, during the last decades several *performance walls* were hit: the memory wall, due to the sub-optimal performance of memory units compared to microprocessors (also known as Von Neumann bottleneck [29]); the Instruction-Level Parallelism (ILP) wall, because it was becoming increasingly difficult to extract more parallelism from sequential software; and the power wall, caused by Dennard scaling's failure[1].

As shown in Figure 2.1, after year 2005 it was clear that the frequency could not be raised further, mainly due to the power wall. While Moore's Law continued to offer more transistors per chip, Dennard scaling could not be the main source of performance anymore. Thus, the new challenge for computer architecture was to use the ever-growing number of transistors to improve the IPC per Watt ratio.

One immediate effect was the adoption of Chip-multiprocessor (CMP) architectures, or multicores. Duplicating the cores in a single chip allowed to use all the available transistors, while providing power-efficient performance gains: for highly parallel algorithms, linear speedup could be delivered with no frequency increase. Under this new paradigm, instead of higher ILP levels, transparent to the programmer, the goal was Thread-Level Parallelism (TLP).

In contrast to ILP, it was not easy for compilers and processors to automatically exploit TLP. Instead, it required modifying the operating system and the applications. Parallel programming paradigms were developed [61, 78, 115] to simplify these tasks. However, there exist classes of algorithms that cannot be parallelized efficiently in homogeneous multicores, for instance because these algorithms contain long sequential regions. In such cases,

[1] Dennard scaling assumed that the Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) voltage and current could be scaled linearly to the size of the transistor. However, 40 years of size reduction guided by Moore's Law have lead to unexpected physical effects, not foreseen by Robert H. Dennard in 1974. These physical effects result in an increased sub-threshold leakage, which now represents a significant percentage of the total power requirements [40].

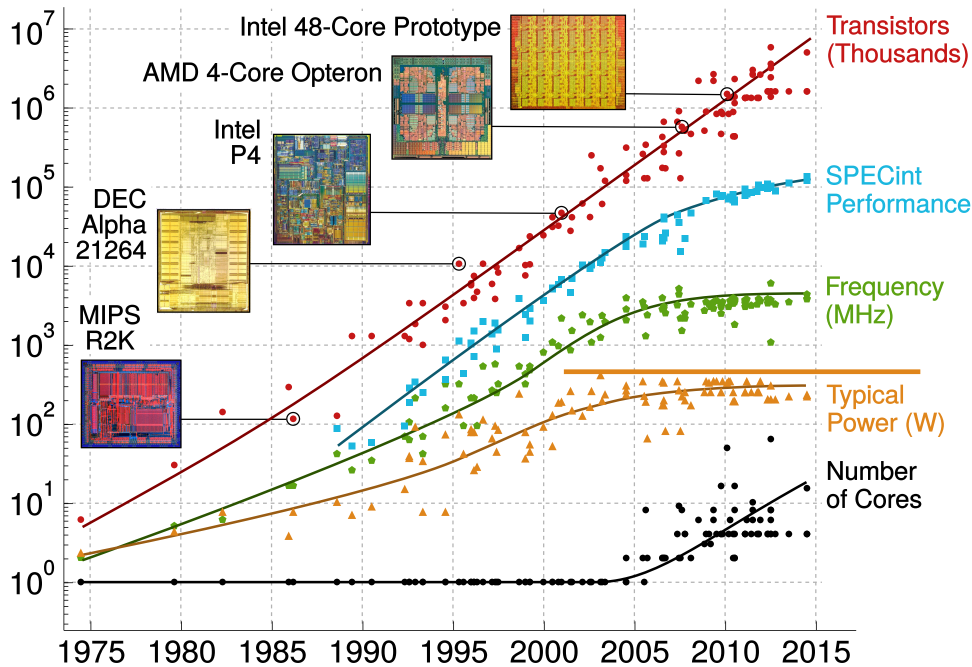


Figure 2.1: Trends in computation. Data collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten. Data until 2015 courtesy of professor Christopher Batten.

replicating a processor will not result in a higher performance. Actually, some projections estimate up to 50% of *dark silicon* in multicore chips at 8 nm in 2024, *i.e.* half of the chip will remain unused by the software [62]. In other words, parallel performance in homogeneous multicores does not follow Moore's Law.

In order to make a more power-efficient use of the transistors and overcome the limitations of homogeneous multicores, asymmetrical or heterogeneous architectures have been proposed in the recent years. Heterogeneous architecture is a broad concept referring to architectures with dissimilar processing units, especially when in the same chip. For instance, there exist multicores with different processor models supporting the same Instruction Set Architecture (ISA) [69], multicores supporting several ISAs [70], or general purpose processors combined with specialized hardware accelerators [14, 109]. It must be noted here that the difference between a specialized processor and a programmable hardware accelerator is becoming very subtle.

2.2 The Limitations of Software Architectural Simulators

Novel computer architecture research is strongly based on empirical evidence. Architectural simulators play a key role when validating new proposals and designs. Software implementations can be developed and extended rapidly. The software can be easily debugged and profiled, using typical software tools.

However, software simulators suffer from three performance bottlenecks. First, each simulated hardware event requires several software instructions. Second, the inherent massive parallelism of hardware is executed in a sequential microprocessor. Third, simulating a multicore on a single core may linearly increase the workload and degrade the performance.

Software simulators rely on the sequential coherency of software to guarantee the correctness of the simulation. Parallelizing the simulation requires a fine-grain synchronization between the simulated events on different processing units, making it impractical. Coarse-grain synchronization may improve the performance, but it derives in some degree of inaccuracy. Likewise, replacing fine-grain simulation events with coarse-grain events guided by statistical models also reduces the fidelity of the simulation.

There are many popular and recent proposals of software multicore simulators. gem5 [36] and MARSS [103] are full-system multicore simulators that use statistical models to speedup the execution, reaching speeds of up to 0.8 MIPS for 8 cores. Sniper [45] and ZSim [108] perform interval simulation but they do not model the full system and running multiple applications can be complicated. Sniper can simulate 16 cores at 2 MIPS, with the average errors of simulation within 25%. ZSim can simulate up to 314 MIPS when simulating a 1000-core system. None of them is cycle-accurate.

There are three drawbacks when using software simulators that are not cycle-accurate:

1. Accuracy is traded off for performance, introducing subtle differences in the simulation.

2. This performance gain is based on replacing cycle-level simulation with architectural-level statistical models, which exhibit similar architectural events, but may have a different cycle-level behavior.

3. The correctness of the statistical models is matched against real hardware. However, there is no reason for those models to continue being valid when new features are

Table 2.1: Terminology used in this thesis.

Architectural model	Taxonomy by Tan *et al.* [117]	Terminology in this thesis
Software-based emulator	SAME simulator	Software-based simulator
FPGA-based simulator	Direct FAME simulator	FPGA-based simulator
FPGA-based emulator	Decoupled FAME simulator	

added to the system [83], which is the purpose of computer architecture research in the first place.

We believe that statements 1 and 3 limit the suitability of fast, non-cycle-accurate software tools for new architectural research proposals. Such models are only accurate when simulating existing hardware. In addition, when verifying the correctness of the new microarchitectural modifications, the idea expressed in statement 2 may hide software and hardware errors that only appear under certain timing conditions. In Section 6.5.2 we show some examples of such behavior.

2.3 FPGA-based Simulators and Emulators

The concept of *emulator* refers to *imitating* a process, that is, obtaining the same behavior over time, but using different operations. Software-based emulators imitate hardware by using software algorithms and statistical models.

In contrast, *simulators* are processes that try to replicate not only the results, but also the operations. FPGA-based multicore prototypes are real digital circuits that cannot fake the hardware operations. However, there also exist software-based emulators accelerated with FPGAs [52, 54, 116], or FPGA-based emulators [83, 104]. Thus, the difference between simulation and emulation is subtle for FPGA-based tools.

In the taxonomy proposed by Tan *et al.* [117], any multicore model is equally referred to as a *simulator*, and labeled as Software Architecture Model Execution (SAME) and FPGA Architecture Model Execution (FAME). FAME simulators which actually are emulators are labeled as *decoupled FAME simulators*. Following this example, in this thesis we use the terms *software-based simulator* and *FPGA-based simulator*, even when we may actually refer to emulators. In Table 2.1 we show the correspondence between the actual model, the taxonomy by Tan *et al.*, and the terminology used in this thesis.

2.4 The Limitations of FPGA-based Prototyping

It is a general consensus in the community that FPGA-based simulators are precise and fast [45, 108]. In Chapter 3 we performed simple experiments showing the superior performance of FPGAs when doing precise full-system simulations. However, their adoption is limited by their reduced usability compared to software simulators, and when discussing this alternative the higher development cost is argued as a detriment [45, 108].

We believe that the complexity of FPGA-based simulator development arises from three main sources. First, FPGA development has low productivity, which can be addressed with novel high-level languages and tools. Second, hardware models require long compilation times, which may be reduced from hours to minutes using overlay architectures[1]. Third, it is notoriously difficult to inspect and debug the hardware models, especially once running on the FPGA.

The size of modern FPGAs allowed to implement large hardware designs, highlighting the limitations of HDLs. In order to improve the productivity, new languages and tools have been proposed. High-level HDLs are adulterated versions of HDLs, extending their syntax or including new functionalities to improve the productivity. One example is SystemVerilog, which extends Verilog with new data types, improved modularity and verification features. Most high-level languages and tools generate an equivalent design in Verilog or VHDL, which become an intermediate representation that hardware CAD tools can understand.

Bluespec SystemVerilog (BSV) is a commercial language based on guarded atomic rules. The hardware is not described as a synchronous digital circuit, but as atomic actions that are statically scheduled. Its syntax is similar to SystemVerilog, and it has functional constructs inherited from the Haskell functional language. BSV simplifies the expression of hardware as a bounded data-flow network, *i.e.* a data-flow network with blocking FIFO channels. Such a paradigm is convenient for the data-flow nature of hardware, in contrast to the sequential nature of software.

High-Level Synthesis (HLS) follows a different approach than HDLs: hardware is synthesized from a software algorithm. HLS compilers accept software programs, usually written in C or a dialect of C, and a functionally equivalent Verilog or VHDL model is produced. These tools are targeted to software programmers that have no experience in hardware

[1] Overlay architectures are development environments based on predesigned hardware blocks. User designs can be translated into combinations of such blocks in very short times.

design, especially when using the FPGAs as accelerators of existing software applications. However, producing efficient hardware from a software description can be as challenging as parallelizing a sequential application for multicore execution.

In particular, the synthesis of FPGA configurations from hardware descriptions can be time-consuming. The hardware model must pass a series of stages –namely, synthesis, mapping and routing– that may take hours to finish, especially for large FPGAs and large designs. In addition, these stages involve non-trivial low-level details, such as resource usage and timing closure. And finally, profiling, debugging and verifying the hardware designs can be challenging.

3

An Open-Source FPGA-based Multicore

3.1 Introduction

This chapter reports on our experience of designing and building an 8-core cache-coherent shared-memory multiprocessor system on FPGA called Beefarm, which has been implemented on the BEE3 [58] infrastructure to help investigate support for Transactional Memory [74, 95, 120].

The inherent advantages of using today's FPGA systems are clear: multiple hard/soft processor cores, multi-ported SRAM blocks, high-speed DSP units, and more and more configurable fabric of logic cells each and every generation on a more rapidly growing process technology than ASIC. Another advantage of using FPGAs are the already-tested and readily-available Intellectual Property (IP) cores. There are various open-source synthesizable Register Transfer Level (RTL) models of x86, MIPS, PowerPC, SPARC, Alpha architectures. These flexible soft processor cores are excellent resources to start building a credible multicore system for any kind of architectural research. Recently, thanks to the emerging communities, many IP designs for incorporating UART, SD, Floating Point Unit (FPU), Eth-

ernet or DDR controllers are easily accessible [10]. Furthermore, RTL models of modern processors have also been developed by chip manufacturers [110, 125], while designs even tend to spread among multiple FPGAs, as in the example of Intel Nehalem.

On-chip Block RAM (BRAM) resources on an FPGA, which are optionally pre-initialized and with built-in ECC, can be used in many configurations; such as:

- RAM or SRAM: For implementing on-chip instruction/data cache, direct-mapped or set associative; cache tags, cache coherence bits, snoop tags, register file, multiple contexts, branch target caches, return address caches, branch history tables, debug support tables for breakpoint address/value registers, count registers or memory access history.
- Content Addressable Memory (CAM): For reservation stations, out-of-order instruction issue/retire queues, fully associative TLBs.
- ROM: Bootloader, look-up tables.
- Asynchronous FIFO: To buffer data between processors, peripherals or coprocessors.

Special on-chip DSP blocks can be cascaded to form large multipliers/dividers or floating-point units. Complete architectural inspection of the memory and processor subsystems can be performed using statistic counters embedded in the FPGAs without any overhead.

Although FPGA-based multiprocessor emulation has received considerable attention in the recent years, the experience and tradeoffs of building such an infrastructure from these available resources has not yet been considered. Indeed, most of the emulators developed were either (i) written from scratch, (ii) using hard cores such as PowerPC, or (iii) using proprietary closed-source cores such as the Microblaze.

Therefore, in this work we choose a new approach: we take an already existing freely-available uniprocessor MIPS core called Plasma [11] and we heavily modify and extend it to build a full multiprocessor system designed for multicore research. To obtain the Honeycomb core, the basic building block for the Beefarm: we designed and implemented two coprocessors, one providing support for virtual memory using a Translation Lookahead Buffer (TLB), and another one encapsulating an FPU; we optimized the Plasma to make better use of the resources on our Virtex-5 FPGAs; we modified the memory architecture to enable virtual memory addressing for 4 GB; we implemented extra instructions to better support

exceptions and thread synchronization (load-linked and store conditional) and we developed the BeelibC system library to support the Beefarm system. Additionally, we designed coherent caches and developed a parameterizable system bus that accesses off-chip RAM through a DDR2 memory controller [118]. Finally, we developed a run-time system and compiler tools to support a programming environment rich enough to conduct experiments on Software Transactional Memory (STM) workloads.

A hypothesis we wish to investigate is the belief that an FPGA-based emulator for multicore systems will have better scalability for simulation performance compared to software-based instruction set simulators. We check this hypothesis using our flexible Beefarm infrastructure with designs ranging from 1 to 8 cores. The results show performance speedups of up to 6x compared to the well-known, cycle accurate M5 software simulator running on a fast host.

The key contributions of this work are:

- A description of the Beefarm multiprocessor system on the BEE3 platform with explanations to justify our design decisions, extensions and discussions on the tradeoffs and an analysis of the FPGA resource utilization of our approach.
- Experimental results for three benchmarks investigating support for Transactional Memory and an analysis of the performance and scalability of software simulators versus our Beefarm system.
- A description of different strategies to implement efficiently a given well-known functionality. Focusing on floating-point support, we provide experimental results and discuss the tradeoffs of each solution.
- An experience reporting the pros and cons of using FPGA-based multicore emulation and identification of specific challenges that need to be overcome to better support this approach in the future.

The next section explains how the Plasma core was modified to design the Honeycomb core, how the Beefarm architecture was implemented on the BEE3 platform, and the software stack, specifically with regard to research on Software Transactional Memory (TM). Section 3.4 compares executions of three STM benchmarks on our platform with the M5 software simulator. Section 3.5 presents different approaches to implement floating-point

support and Section 3.6 describes our experience in building the Beefarm. Section 3.7 concludes and describes future work.

3.2 Transactional Memory

In parallel and concurrent computing under shared memory models, some software blocks must be executed atomically to guarantee a correct execution. These situations include updating shared data structures, or accessing critical hardware. Non-synchronized processing units may not guarantee such atomic behavior. For this reason, synchronization mechanisms play key roles in multicore and heterogeneous architectures.

A common synchronization mechanism is a lock, which allows software threads to specify mutually-exclusive software regions. Being the basis of other synchronization techniques, locks can affect the performance of parallel applications. In addition, a correct and efficient lock usage may require a great programming effort. Conversely, software bugs caused by incorrect lock usages, *i.e.* “data races”, can be non-deterministic and hard to debug.

Transactional Memory [78] is a programming paradigm designed to avoid using locks when writing parallel code. Lock-based techniques are *pessimistic schemes* because software threads are only allowed to access atomic data structures if their behavior will be guaranteed to be correct in any case. In contrast, TM is an *optimistic scheme* because it does not restrict the concurrent access data structures, and conflicts are managed dynamically.

In Figure 3.1, a vector element is updated randomly. When using coarse grain locks (Figure 3.1a), two threads may not access the vector concurrently, to avoid conflicting updates. This would only happen a small fraction of time, and this pessimistic schema sequentializes the accesses. In Figure 3.1b, fine-grain locks are used to protect per-element accesses. This implementation allows concurrent accesses to non-conflicting elements, but requires one lock per element. Fine-grain locking can become challenging for complex data structures and large algorithms, deriving into data races. TM removes the necessity of locks, allowing the programmer to identify which software instructions must be executed atomically. The TM mechanism will abort only conflicting updates, resulting in the same behavior as the optimal fine-grain locking schema.

The great advantage of TM over locks is that atomicity is managed automatically, and

```
int v[N];
lock l;

void update() {
  lock(l);
  v[rand()]++;
  unlock(l);
}
```

(a) Coarse grain locking.

```
struct elem {
  int val;
  lock l;
}

struct elem v[N];

void update() {
  int i = rand();
  lock(v[i].l);
  v[i].val++;
  unlock(v[i].l);
}
```

(b) Fine-grain locking.

```
int v[N];

void update() {
  atomic {
    v[rand()]++;
  }
}
```

(c) TM-based atomic block.

Figure 3.1: Lock-based vs. TM-based atomic blocks.

a safe parallel execution is guaranteed. In lock-based applications the correctness highly depends on the programmer, while in TM-based programming atomic code regions just need to be defined by the user. In addition to programming simplicity, TM also guarantees progress, *i.e.* no dead locks (execution stopped) may occur.

Data accesses inside an atomic software region TM are speculative, and they are made effective as a whole transaction when exiting the region. When two software threads access the same data, one of the transactions is aborted and the changes are discarded. There exist different policies for data versioning (how to handle the original and the transactional data) and conflict detection (eager –as soon as possible– or lazy –when the transactions are committed). These policies and the implementation of the TM mechanism determine the performance, as well as the maximum size of the transactions.

There are some limitations that most TM implementations cannot overcome. The size of the transactions is mostly determined by the number of different data addresses per transaction that the system can manage. In addition, transactions must be completely reversible, which implies that transactions cannot perform some software or hardware actions. For instance, most TM implementations cannot deal with system calls or I/O. Typically, these are then marked as irrevocable transactions, *i.e.* they cannot be rolled back.

There exist two main approaches to TM. Software Transactional Memory (STM) was the first mechanism to be implemented. It uses memory buffers to store the data versions, and a software library manages the transactions and their conflicts [79, 112]. Hardware

Transactional Memory (HTM) uses dedicated hardware to implement speculative memory accesses [22, 74, 78, 95]. STM is flexible and it does not require any dedicated hardware, but it can generate a significant execution overhead. HTM may deliver the same performance as fine-tuned lock-based schemes, but the hardware capacity greatly constrains the transaction size. Hybrid Transactional Memory (HyTM) systems rely on HTM for better performance, but they can fall back to STM under special circumstances, such as size limitations or transaction jams [44].

3.3 The Beefarm System

This section introduces the architectural and implementation details of the Beefarm system, a bus-based multiprocessor version of the popular MIPS R3000 designed for the BEE3 FPGA platform. Our architectural and design decisions not only show the experience of implementing a multicore emulator from a popular soft core design in a modern FPGA platform, but also provide an example of the variety of available resources that are ready to be used in current reconfigurable systems. We are not interested in using large multithreaded soft cores like OpenSPARC (64-bit) and Leon3 (32-bit SPARC) because we concentrate on designing our own TM-capable multiprocessor emulator by modifying and upgrading a popular soft core, and reflecting on our experience.

3.3.1 The Plasma Core

The synthesizable MIPS R2000-compatible soft processor core Plasma was designed for embedded systems and written in VHDL. It has a configurable 2-3 stage pipeline (no hazards), a 4 KB direct-mapped L1 cache, and can address up to 64 MB of RAM. It was designed to run at a clock speed of 25 MHz, and it includes UART and Ethernet cores. It also has its own real-time operating system with some support for tasks, semaphores, mutexes, message queues, timers, heaps and a TCP/IP stack. The Plasma core is a suitable choice because it is based on the popular MIPS architecture, it is complete and it has a small area footprint on the FPGA. Such RISC architectures with simpler pipelines are both more easily customizable and require fewer FPGA resources compared to a deeply-pipelined superscalar processor, so they are more appropriate to be integrated into a larger multiprocessor SoC. By using a known architecture like MIPS, we also get to (i) have easy access to information about

this well-known Instruction Set Architectur (ISA), (ii) use a standard cross-compiled GCC toolchain and meanwhile (iii) keep it efficient thanks to RISC architecture.

Although the original Plasma core is suitable for working with diverse research topics, it has some limitations that makes it unsuitable as the processing element of the Beefarm system. It does not support virtual memory (MIPS Coprocessor 0), exceptions and floating point arithmetic (MIPS Coprocessor 1). In addition, although the design is acceptable for the low speed and area requirements, we can not fit more than a couple of cores on the older Altera and Xilinx chips, for which the core is designed. Furthermore, the design only infers optional on-chip resources for the register file, and there is no support for multiprocessing or coherent caches.

3.3.2 The BEE3 Platform

The BEE3 hardware prototyping platform contains four Virtex5-155T FPGAs, each one with 24,320 total slices, 212 36-KBit Block RAMs, and 128 Digital Signal Processing (DSP) units. Ethernet, SD card, Compact Flash and RS232 are also available on the BEE3 rack. Each FPGA controls four DDR2 DIMMs, organized as two independent 2-DIMM memory channels of up to 4 GB each. The DDR2 controller [118] manages one of the two channels per FPGA. The design that is written in Verilog includes an instance of a small processor called TC5 which calibrates the controllers and serves requests. Using one controller provides sequential consistency for our multicore described in this work, since there is only one address bus, and reads are blocking and stall the processor pipeline. The controller design occupies a small portion of the Virtex5-155T (around 2%).

3.3.3 The Honeycomb core: Extending Plasma

Figure 3.2 shows the block diagram for the Honeycomb processor, where instructions and data words are 32-bit wide, and data can be accessed in bytes, half words (2 bytes) or words (4 bytes). The processor is implemented in a 3-stage pipeline with an optional cycle for data access instructions. In a typical ALU instruction, during the first stage the opcode is fetched and decoded, and the registers are read. The opcode is converted into a 60-bit control word that sets up the arithmetic units and prepares the memory controller. The arithmetic results and the calculated address are passed to the next stage. The TLB translation is processed

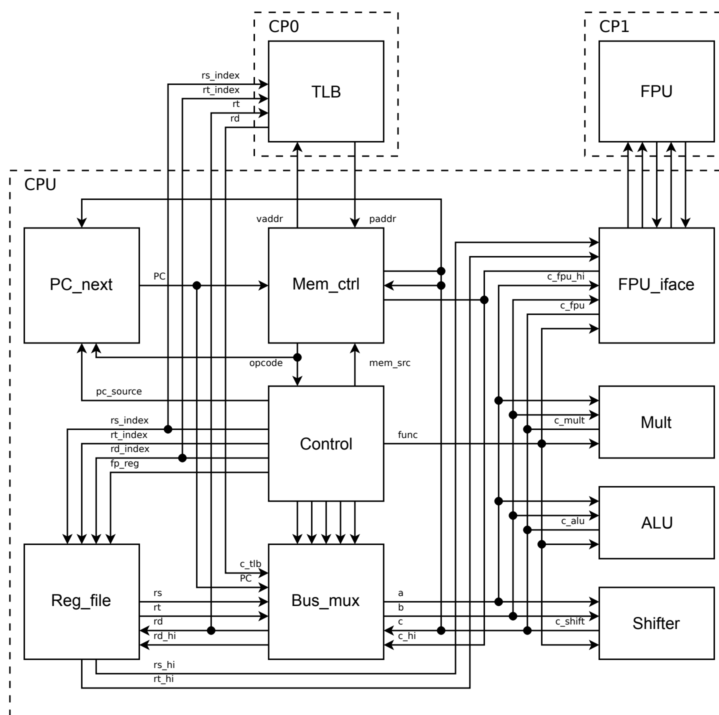


Figure 3.2: The Honeycomb processor functional block diagram, with the Coprocessor 0 (CP0) for virtual memory and exceptions and the Coprocessor 1 (CP1) for floating point arithmetic. Some wire names are hidden for simplicity.

before accessing the caches using a half-cycle shifted clock.

During load or store operations, instruction fetch and data access are performed sequentially. This requires two cycles and the CPU remains stalled during one additional cycle. Instruction and data caches are blocking and cache misses can take various cycles.

Coprocessor 0: The MMU

In order to provide support for virtual memory, precise exceptions and operating modes, we implemented a MIPS R3000-compatible 64-entry TLB (called CP0), effectively upgrading the core from an R2000 to an R3000, which we named Honeycomb.

The CP0 of the Honeycomb controls a 64-entry, fully-associative TLB. It provides memory management and exception handling intercepting the memory control unit datapath, translating virtual addresses to physical addresses or raising an exception signal in case of a page miss/fault. Each entry contains two values: The 20 highest bits of the physical address, which will replace the corresponding ones in the virtual address, and the 20 highest

bits of the virtual address which are used as a matching pattern.

There exist various approaches to implement an efficient CAM on FPGAs, with different trade-offs between read and write access times, resource usage, and the technology utilized [42], where general-purpose Look-Up Tables (LUTs) or on-chip block memories can be used. The use of LUT logic is inappropriate for medium and large CAM elements, and the time-critical nature of this unit makes multi-cycle access inappropriate since it must translate addresses each cycle on our design. Only the approach based on RAM blocks fitted the requirements: We implemented this unit with on-chip BRAM configured as a 64-entry CAM and a small 64-entry LUT-RAM memory. Virtual patterns that are stored in the CAM give access to an index to the RAM that contains the physical value. It does not need a dedicated pipeline stage because it is controlled by a half-cycle shifted clock that performs the translation in the middle of the memory access stage. This 6-bit deep by 20-bit wide CAM occupies four BRAMs and 263 LUTs.

Coprocessor 1: Double-Precision FPU

Another lack of the original Plasma is floating point arithmetic support. Since the initial Intel 8087 design in 1980, a floating point unit is an integral part of a modern computing system architecture, and starting in the early and mid-1990s many microprocessors for desktops and servers have had more than one FPU. The MIPS 3010 FPU implemented in Coprocessor 1 (CP1) can perform IEEE 754-compatible single and double precision floating point operations. The 3010 FPU designed with Xilinx Coregen library cores takes 5520 LUTs and 14 DSP units, and can perform FP operations and conversions in variable number of cycles (between 4 and 59). We used only 4 of the 6 integer-double-float conversion cores to save space.

This optional MIPS CP1 has 32x32-bit FP registers and a parallel pipeline and in our design the FPU instructions stall the CPU. The general purpose register file was extended to include these registers, implemented as LUTRAM. For double precision, two registers represent the low and high part of the 64-bit number. This kind of access adds complexity to the design and it was necessary to replicate the registers to allow a flexible 32 (single precision) or 64-bit (double precision) read/write access each cycle. The trade-offs of this implementation are discussed in Section 3.5.

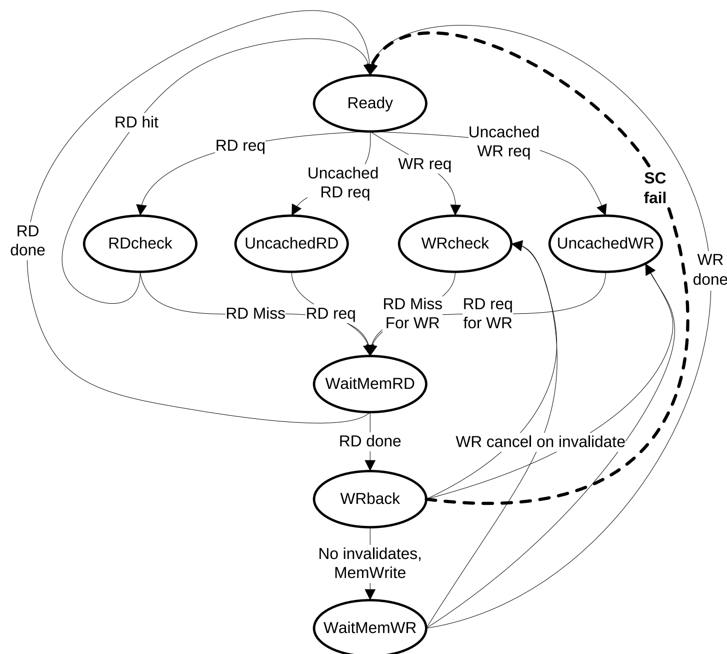


Figure 3.3: The cache state diagram, with the special transition added for LL/SC support highlighted.

Memory Map and ISA Extensions

We redesigned the memory subsystem which could originally only map 1 MB of RAM, to use up to 4 GB with configurable memory segments for the stack, bootloader, cache, debug registers, performance counters and memory-mapped I/O ports. Furthermore, we extended the instruction set of the Honeycomb with three extra instructions borrowed from the MIPS R4000 ISA:

- ERET (Exception RETurn), to implement more precise exception returns that avoid branch slot issues.

- LL (Load-Linked) and SC (Store Conditional), which provide hardware support for typical synchronization mechanisms such as Compare and Swap (CAS) or Fetch and Add (FAA). This is useful for Software Transactional Memory support, as we detail in Section 3.3.7.

3.3.4 The Beefarm System Architecture

Honeycomb's 8 KB write-through L1 cache design that supports the MSI cache coherency protocol for both data and instructions in 16-byte, direct-mapped blocks, uses two BRAMs for data and another one for storing the cache tags. The BRAM's dual-port access support makes it possible to serve both the CPU and the bus requests in a single cycle. Reads and writes are blocking, and coherence is guaranteed by the snoopy cache invalidation protocol that we implemented.

The cache state machine is shown in Figure 3.3. It remains in *Ready* state while the CPU does not issue any requests, for example because it is stalled during a multiplication operation. The requests are processed by one of the four next states, depending if they are reads or writes, and if they require cached access or the special MIPS uncached mode that accesses directly the memory. In this latter case, cache memory is not accessed, all the accesses are threated as misses. The access mode depends on the virtual address and is decided by the Memory Management Unit in the CPU. Read and write misses are collected by the *WaitMemRD* state, that issues a read request to the main memory, and if in cached mode it stores the data in a new line in the cache. *WRback* modifies the data line and sends the new value to the main memory; the response comes in form of an invalidation that informs the requesting cache that the write was performed correctly and instructs other caches to remove that line.

The memory arbiter will only choose one of two write operations in conflict (i.e. to the same address). The corresponding invalidation signal will cause the losing cache to fetch the line again and retry the write operation. If the write request was caused by a Store Conditional instruction, the write is not retried and the cache simply sends a failing signal to the CPU (see the special transition highlighted in the cache state diagram with dashed lines).

The caches designed are interconnected with a central split-bus controlled by an arbiter, as shown in Figure 3.4. Write accesses are processed in an absolute order by the system bus, and caches snoop on this bus to invalidate entries that match the current write address of the DDR controller. While this protocol is not considered to be the most efficient among the bus-based snooping protocols, it can perform invalidations as soon as the writes are issued on the write FIFOs of the DDR and it serves to find an adequate balance between efficiency and resource usage. More complex caches that demand a higher resource usage

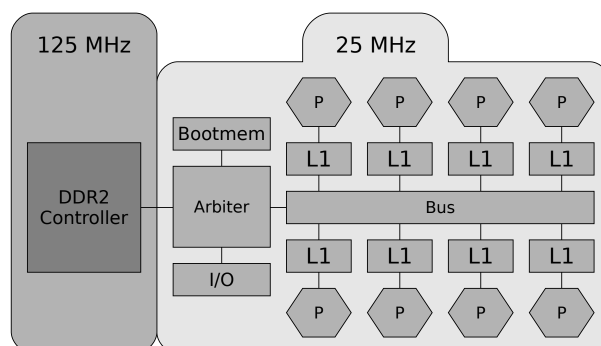


Figure 3.4: The Beefarm multiprocessor system.

would make it difficult to implement a large multiprocessor given the limited resources present on chip.

The bus arbiter implemented interfaces the FIFOs of the DDR controller, serving requests from all processors following a round-robin scheme. The boot-up code is stored in a BRAM connected to the arbiter and mapped to a configurable region of the address space so that the arbiter either accesses the DDR memory or the Bootmem in BRAM depending on the address range. I/O ports are also mapped, and the lowest 8 KB of physical memory give access to the cache memory, becoming a useful resource during boot-up when the DDR memory is not initialized yet. Furthermore, the cache can be used as stack thanks to the uncached execution mode of MIPS. Such direct access to cache memory is useful for debugging, letting privileged software to read and even modify the contents of the cache.

The arbiter, the bus, caches and processors can run at a quarter of the DDR frequency (25 - 31.25 MHz), the CPU's shallow pipeline being the main cause of this upper bound on the clock. Although the bus and cache frequencies could be pushed to work at 125 MHz, or at an intermediate frequency, it was not desirable to decouple this subsystem from the processor because partitioning the system in many clock domains can generate tougher timing constraints and extra use of BRAM to implement asynchronous FIFOs or extra circuitry to prepare signals that cross different clock domains. Further optimizations to the Honeycomb are certainly possible by clocking faster all special on-chip units and including this extra circuitry.

Around eight Honeycomb cores (without an FPU) could form a Beefarm system on one Virtex5-155T FPGA, however the system bus can become a bottleneck not only during system execution, but also when placing and routing the design, so in this work we only present

results for up to 4 CPU cores with FPUs or 8 CPU cores without FPU support.

To use all four FPGAs on the BEE3 infrastructure and to have a distributed shared memory multiprocessor of up to 40 Honeycomb cores, we are implementing a distributed directory scheme that will provide system-wide memory coherency. This ongoing work is described in Section 3.7.

3.3.5 FPGA resource usage

One of the objectives of the design is to fit the maximum number of cores while supporting a reasonable number of features. The components have to be designed to save the limited LUTs and conservatively use BRAMs and DSPs, both to allow for more functionality to be added later on, and to reduce the system complexity. This is necessary to keep the frequency of the clock high and thus the performance, meanwhile reducing the synthesis and place and route time.

The Honeycomb core without an FPU occupies 5712 LUTs on a Virtex-5 FPGA including the ALU, MULT/DIV and Shifter units, the coherent L1 cache, the TLB and the UART controller, a comparable size to the Microblaze core. Figure 3.6 shows the LUT occupation of the CPU's components. The functional blocks on the Honeycomb can be categorized in three groups:

- **Compute-intensive (DSP):** The compute-intensive structures in the Plasma core (*e.g.* ALU, MULT/DIV, Shifter, FPU) originally fit the third category since the ALU is actually designed as a combinatorial circuit, while the MUL/DIV/Shifter units take 32 cycles to iterate and compute. These are good candidates to take advantage of hard DSP units. The ALU can be mapped directly onto a DSP while a MULT can be generated with Xilinx Coregen in a 35x35 multiplier utilizing an acceptable 4 DSP and 160 LUT space. The shifter can also benefit from these 4 DSPs thanks to dynamic opmodes, however, a 32-bit divider can take anywhere between 1100 LUTs to 14 DSP units. In general, it is not clear what the most optimal way of combining all these operations is to have a minimal design. In Section 3.5 we discuss different options for floating-point support. However the optimal way of realizing a compact FPU is left as future work.

- **Memory-intensive (BRAM/LUTRAM):** In the second category (*e.g.* Regfile, Cache,

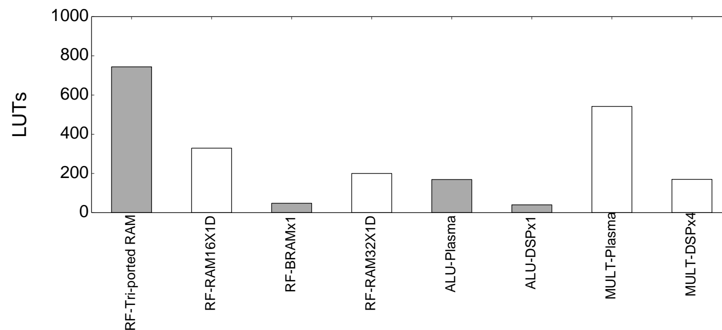


Figure 3.5: Some of the available options (in 5-LUTs) to implement the Register File, the ALU and the Multiplier unit. The lighter bars indicate the choices in the original Plasma design.

TLB) the TLB is designed in a CAM as explained in Section 3.3.3, and the cache and the cache tags in BRAMs. For the Reg_File, the original Plasma design selects between instantiating 4-LUT distributed RAMs (RAM16), behaviorally describing a tri-ported RAM, or using a BRAM. The use of BRAM is inefficient since it would use a tiny portion of a large structure, and the tri-ported RAM infers too many LUTs, as seen in Figure 3.5. When distributed LUTRAM is inferred, each 5-LUT can act as a 32-bit register on the Virtex-5, enabling two reads and one write per cycle assuming one of the read addresses is the write address. There are a few options to enable two reads and a write to distinct addresses on each CPU cycle: (i) to do the accesses in two cycles on one register file, using one of the input addresses for reading or writing when needed, (ii) to clock the register file twice as fast and do the reads and writes separately, or (iii) to duplicate the register file to be able to do two reads and a write on distinct addresses on the same cycle. Although we currently use the third approach, our design accepts either configuration. Other groups have proposed latency insensitive circuits which save resources by accessing the register file in a few cycles[123].

- **LUT-intensive:** Units that implement irregular case/if structures or state machines, *e.g.* PC_next, Mem_ctrl, control, bus_mux, TLB logic, system bus and cache coherency logic. This category demands a high LUT utilization; one striking result in Figure 3.6 is that the cache coherence circuitry occupies roughly half of the LUT resources used by the Honeycomb. Such complex state machines do not map well on reconfigurable fabric, however synthesis results show that the Honeycomb core, when a similar speed-grade Virtex-6 chip is selected, performs 43.7% faster than the Virtex-5 version, so such irregular behavioral descriptions can be expected to perform faster

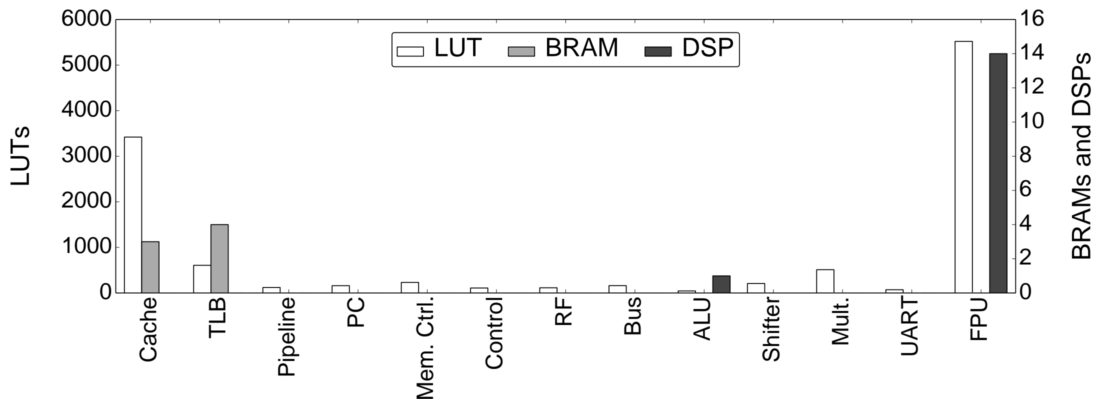


Figure 3.6: LUT (left axis, dark bars) and BRAM and DSP (right axis, clear bars) usage of Honeycomb components. The FPU uses more LUTs than the rest of the units of the core together (CPU and cache).

as the FPGA technology advances each generation.

The most crucial issue to keep in mind while developing a multicore emulator on reconfigurable platforms is undoubtedly to match better the underlying FPGA architecture. BRAMs and DSP units must be used extensively, and more regular units that match a compute-and-store template rather than complex state machines must be fashioned. Unlike the cache coherence protocol and the shared system bus that map poorly, compute-intensive units and the register bank are good matches for distributed memory that use 5-LUTs, although one still can not do a single cycle 3-ported access. In general, we believe that caches are an important bottleneck and a good research topic for multicore prototyping. There is little capacity for larger or multi-level caches on our FPGA, and it would not be easy at all to provide high levels of cache associativity.

3.3.6 The Beefarm Software

Since we are not running a full Linux with all system calls implemented, we can not use the GNU C standard library libC, so we developed a set of system libraries called BeelibC for memory allocation, I/O and string functions. Many groups exercise falling back to a host machine or a nearby on-chip hard processor core to process system calls and exceptions [54, 116]. A MIPS cross-compiler with GCC 4.3.2 and Binutils 2.19 is used to compile the programs with statically linked libraries. The cores initially boot up from the read-only Bootmem that initializes the cores and the stack and then loads the RTOS kernel code into memory either from the serial port or from the SD card onto the DDR. The SD card,

which is currently interfaced by software bit-banging can transfer roughly a MB of data per minute, however an SD core can be incorporated for faster data transfer. Although Ethernet might be another fast option, we deemed the additional complexity too potentially risky at least for the moment while developing the initial design. We also plan to port an RTOS with multiprocessing and threading support such as the eCos or RTEMS to our design. We currently let all cores initialize and wait on a barrier, which is set by CPU0. Another option that reserves the CPU0 for I/O is also implemented.

We use the Xilinx ISIM and the Mentor Graphics ModelSim for offline functional or post Place and Route (PnR) simulation. Real-time debugging on Xilinx chips is done with Xilinx Chipscope Pro, for which we apply various triggers and hardware debug registers. All results were obtained using 64-bit Xilinx ISE 12.2 running on RHEL5.

3.3.7 Investigating TM on the Beefarm

One of the most attractive proposals for shared-memory CMPs has been the use of atomic instructions in Transactional Memory (TM), a new programming paradigm for deadlock-free execution of parallel code without using locks, providing optimistic concurrency by executing atomic transactions in an all-or-none manner. In case of a data inconsistency, a confict occurs and one of the transactions has to be aborted without committing its changes, and restarted. Transactional Memory can be implemented in hardware (HTM) [74, 95], which is fast but resource-bounded while requiring changes to the caches and the ISA, or software (STM) [19, 65] that can be flexible running on stock processors at the expense of weaker performance. Specifically, we are interested in the intermediate approaches, or Hardware-assisted STM (HaSTM) which by architectural means aims to accelerate a TM implementation that is controlled by software.

We provide a framework that could be easier for conducting architectural studies particularly to study HaSTM designs. We believe that the Beefarm system can provide an ideal environment to implement detailed per-atomic block profiling infrastructure to better visualize the overheads related to TM and the properties of transactional benchmarks. We can use detailed profiling to construct hardware conflict tables that aid contention management where each transaction can perform lookups in a transaction conflict table and may decide to stall until a previously-known-to-conflict transaction commits to start processing. This can reduce the wasted work caused by a high amount of conflicts and aborts. Additionally,

```
$CAS_SC_FAIL:
     ll    $v0, 0($a0)
     bne   $v0, $a1, $CAS_END
     nop
     move  $t0, $a2
     sc    $t0, 0($a0)
     beqz  $t0, $CAS_SC_FAIL
     nop
$CAS_END:
     jr    $ra
```

Figure 3.7: Compare and Swap using LL/SC in MIPS assembly.

we can use CAMs to implement Bloom filters that track read/write sets.

Towards this goal, we have successfully ported an STM library and ran TM applications on the Beefarm. We present the results of running TM applications in Section 3.4. TinySTM [65] is a lightweight and efficient word-based STM library implementation in C and C++. It differentiates mainly by its time-based algorithm and lock-based design from other STMs such as TL2 and Intel STM. By default, it compiles and runs on 32 or 64-bit x86 architectures, using the atomic_ops library to implement atomic operations. We modified it to support Compare and Swap (CAS) and Fetch and Add (FAA) primitives for the MIPS architecture using LL/SC instructions (Figure 3.7).

3.4 Comparison with SW Simulators

3.4.1 Methodology

The multiprocessor system presented in this work was designed to speed up multiprocessor architecture research, to be faster, more reliable and more scalable than software-based simulators. Its primary objective is to execute real applications in less time than popular full-system simulators, although it is not possible to run as fast as the actual ASIC. Therefore our tests:

- Measure the performance of the simulator platform, *not* the performance of the system simulated. What is relevant is not the simulated processor's speed, but the time that the researcher has to wait for the results and its reliability.

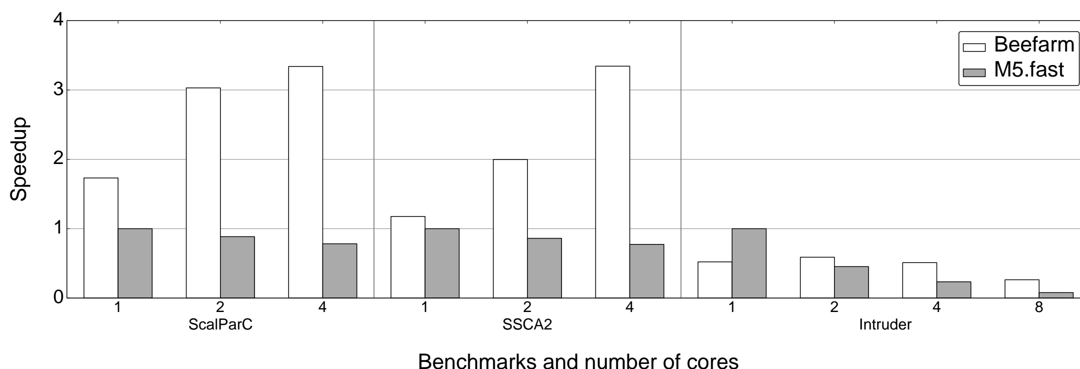


Figure 3.8: Beefarm vs M5: Speedups for ScalParC, SSCA2 and Intruder (normalized to single core M5 run).

- Abstract away from library or OS implementation details, so that external functions like system calls would not significantly affect the results of the benchmark.
- Can be easily ported to different architectures, avoiding architecture-specific implementations like synchronization primitives.
- Pay special attention to the scalability of the emulation, targeting a key weakness of multiprocessor software simulators. The emulations are not affected by the number of processors in other ways than the usual and expected from a reliable simulator (memory bandwidth, traffic contention, cache protocols, etc.).

M5 [37] is a well-known "full-system simulator" in computer architecture research which can simulate an arbitrary number of Alpha processors with complex architectural details like caches and buses, and can easily be modified. We believe that despite the fact that MIPS and Alpha are distinct architectures this can be a fair comparison; both architectures are 32-bit RISC featuring 32 integer registers, operate on fixed-size opcodes and the only operations that access the memory are load and store, etc. We believe that the choice of comparing with M5 is sensible since our main aim is to measure and compare the scalability of the software simulator and the Beefarm multicore emulation infrastructure.

To obtain precise measurements of the execution time of the M5, we added a precise 64-bit hardware cycle counter to measure the total execution time, not the "simulated time". We executed the test in the M5 compiled with the maximum optimizations and with the minimum timing and profiling options (fast mode), and additionally for ScalParC in a slower profiling mode with timing. The compilers used to obtain the test programs for the Beefarm

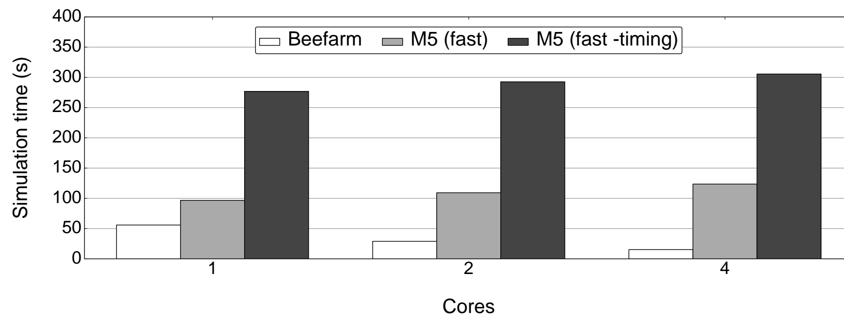


Figure 3.9: Beefarm vs. M5: ScalParC simulation time.

and the M5 both use GCC version 4.2, compiled with the -O2 optimization level or -O3 when possible. The host machine where we ran the M5 is an Intel Xeon E5520 server with 2x quad-core processors running at 2.26 GHz with 64 GB of DDR3 RAM and 8 MB of L3 cache memory.

3.4.2 Beefarm Multicore Performance with STM Benchmarks

To test multicore performance with STM benchmarks running on the Beefarm, we have run ScalParC, a scalable TM benchmark from RMSTM [82] and two TM benchmarks from the STAMP suit [93] which are very commonly used for TM research. We modified ScalParC with explicit calls to use the TinySTM library. In our experiments, ScalParC was run with a dataset with 125K records, 32 attributes and 2 classes, SSCA2 was run with problem scale 13 and Intruder with 1024 flows.

The results that are normalized to the single-core M5 executions show that while the Beefarm can scale in a near-linear way, the M5 simulator fails to scale and the performance rapidly degrades as the core counts are increased. Figure 3.8 shows that the gap opens with more cores and with only four, the Beefarm just needs fifteen minutes to run the ScalParC benchmark, an eightfold difference. This would be the common case when the abort ratio between the transactions are low and little work is repeated, so the benchmark itself is scalable. SSCA2 is another scalable benchmark that is able to benefit from the inherent parallelism of the FPGA infrastructure, and the good performance of the floating point unit. The two-core Beefarm takes about half of the runtime of the M5 and it shows better scalability with more cores.

Intruder is a very high abort rate integer-only benchmark that scales poorly, and this can

be seen on both M5 and Beefarm results. It performs worse on the Beefarm for single processor runs, however for more than two cores, it runs faster than the M5, whose scalability again degrades rapidly. We are able to run Intruder with 8 CPUs because this design does not use an FPU, however our performance is lower in this case. In this sense a hardware FPU which takes roughly the space of a single CPU core is clearly a worthy investment for the case of this particular benchmark. Other available hardware kernels such as a quicksort core would be a very useful accelerator for this particular benchmark, and such specialized cores/execution kernels could further push the advantages of multicore emulation on reconfigurable platforms.

A single core ScalParC run on the Beefarm takes around 9 hours when a soft-float library is used and about an hour when an FPU is used as Figure 3.9 depicts. Modeling timing is very costly on software simulators, but in the future we'd like to adopt the pipelined and partitioned approaches for timing that were proposed for FPGA emulators [104, 116].

The results of the STM benchmarks show that our system exhibits the expected behavior, scaling well with more cores and thus reducing the time that the researcher has to wait to obtain results. In other words, the simulated time and the simulation time are the same on our FPGA-based multicore emulator. The software-based simulator suffers from performance degradation when more cores are simulated and fails to scale. As seen on the Intruder example, certain configurations (eg. without any hardware accelerators like the FPU) for a small number of cores could result in software simulators performing faster than FPGA devices. Mature simulators that take advantage of the superior host resources could still have advantages over FPGA emulators for simulations of a small number of cores.

3.5 Efficient Implementations for Floating-point Support

Most of the software applications and benchmarks assume that the underlying architecture will provide floating-point support. This assumption can cause the programmer not to try to optimize the resource usage. But in the case of floating-point hardware, a complete set of calculation, comparison and conversion operations can consume 5520 LUTs, the resources equivalent to one of our Honeycomb processors (5712 LUTs). **Optimizing this kind of unit becomes of paramount importance, especially when the emulation technology has a critical limitation in resource consumption.** In this section we present different

strategies to save resources related to floating-point operations.

The first solution is to not implement floating-point support. It has been presented in the previous section, where our architecture has been adapted accordingly to the characteristics of the application. As the Intruder benchmark does not need any floating-point calculations, removing such units allowed us to double the number of processors. This principle can be extended to applications that make unnecessary use of floating-point operations. An example is the use of real numbers for statistic purposes (e.g. execution time), which most of the times could be implemented using integer formats.

If floating-point operations cannot be avoided, other optimizations can be applied. One important waste of resources is caused by the arbitrary use of the floating-point precision formats or floating-point operations. For the former, reducing from a double to a single precision format will optimize the speed and reduce the size of the hardware units. For the latter, rewriting the algorithms to avoid some arithmetic operations would also reduce the number of floating-point units used. For example, our floating-point unit implements 4 arithmetic operations, one comparator and the conversion from/to integer logic. It would be desirable that the architecture could be adapted to the needs of the application, instead of providing unnecessary operations.

Another way to free hardware resources is to emulate floating-point calculations using integer arithmetic. Many software libraries exist for this purpose, among whom we have chosen the SoftFloat library [76]. This library guarantees results conforming the IEEE 754 standard, which defines 32-bit (single precision) and 64-bit (double-precision) floating-point formats.

The software can interface the libraries in two ways:

- At program level. The programmer can hard-code the operations using the library primitives, or let the compiler instrument the floating-point operations and convert them into well defined semantics. For example, GCC defines a standard set of function calls that the programmer will have to provide. We used this approach to create wrappers that interface the SoftFloat library and convert between GCC's types and functions and the library ones.

- At system level. The hardware can throw exceptions when unimplemented instructions are detected, like floating-point instructions. Then the operating system can emulate by software the offending instruction using a software library. In this ap-

proach a minimal hardware collaboration is needed. The advantage of this solution is that floating-point support is transparent to the application. But it can be less efficient than the previous approach, because context switching and exception handling will add some overhead.

One variation of the floating-point instruction interception can be the use of an external host to perform the calculations. The FPGA device can communicate with an external computer through a serial port, an Ethernet interface or a PCI channel. Then the operating system sends the intercepted floating-point opcodes to the host, and passes the results received. This solution also implies the additional hardware to communicate with external devices and the external host infrastructure.

A third solution consists of sharing the floating-point units between processors. This would reduce the availability of floating-point operations and increase the latency, but not necessarily cause a deep impact to the architecture's performance. Program profiling can show the intensity and distribution of the floating-point calculations. Then applications can be classified depending on these characteristics, and a small penalty can be paid in those cases where floating-point operations are sparse. This strategy can be refined through software and hardware collaboration, scheduling the accesses to the shared units.

To study the impact of these approaches, we modified our architecture to replace the private FPUs with shared units. In this new architecture, the CP1 unit does not contain the FPU but it behaves as a proxy for the remote, shared FPUs. Each FPU is associated to a cluster of processors interconnected through a shared split-bus, independent from the memory bus. An arbiter selects which request to serve in a round-robin fashion. This flexible implementation allows us share an arbitrary number of FPUs among an arbitrary number of processor clusters.

In Figure 3.10 we compare the performance of four different versions of a 4-processor BeeFarm system: the version with 4 FPUs, one for each processor; with 2 shared FPUs (2 clusters of 2 CPUs with one FPU in each); with a single, shared FPU; and without an FPU, using software emulation. We can observe that the performance is not always proportional to the amount of resources: With 2 and 1 FPUs the performance loss remains within 30%, compared to a full set of private FPUs.

From these results we could state that for ScalParC the optimal configuration could be sharing two FPUs, which would only cause a 9% performance reduction compared to the

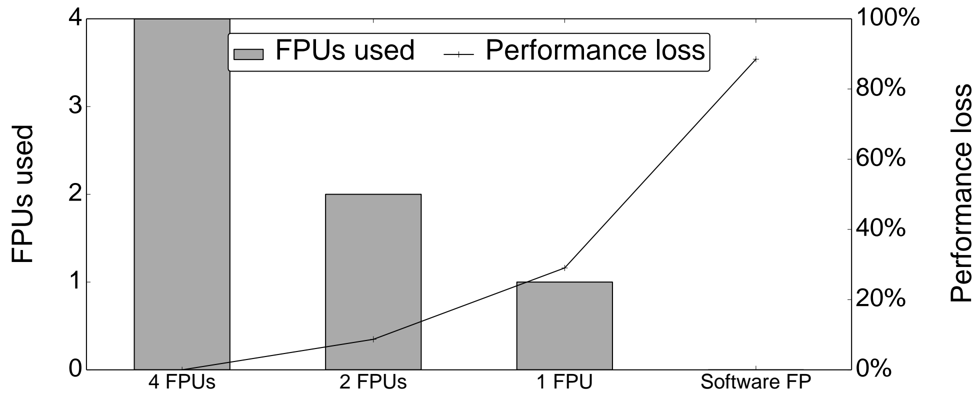


Figure 3.10: Resource usage (bars) and performance loss (line) for four versions of BeeFarm with 4 cores executing the ScalParC benchmark: (i) software floating-point emulation, (ii) 1 shared FPU, (iii) 2 shared FPUs and (iv) 4 non-shared FPUs. Performance speedup is normalized to BeeFarm with software emulation.

version with non-shared FPUs. We think that this methodology should be used for other kinds of hardware units that are not always necessary and could be shared. At the same time, complementary techniques like software optimization or emulation can be applied with the help of the programmer or more sophisticated tools. Finally, the solutions presented here can be applied to a wide range of instructions, from floating-point calculations to system call emulation.

3.6 The Experience and Trade-offs in Hardware Emulation

Although we achieved good scalability for our simulation speeds with respect to the number of processor cores, we have observed several challenges that still face the architecture researcher that adopts FPGA-based emulation. These include:

Place and route times can be prohibitively long, although newer synthesis tool versions have started to make use of the host multithreading capabilities. In the case of adding a simple counter to the design for observing the occurrence of some event, the resynthesis, mapping, placing and routing of an 8-core Beefarm takes 2-3 hours on our 8-core server. slightly modifying the online debugging core [3], *i.e.* adding a new signal to be inspected, also requires a complete re-do of Map and P&R, which can be very inconvenient. One solution to this problem might be to try exploit floor planning tools (*e.g.* PlanAhead) or the explicit use of layout constraints (*e.g.* Xilinx's RLOCs) to floorplan the processor cores and other components which could significantly improve place and route run-times. Us-

ing FPGAs, mapping and placement issues are a lot more relevant compared to software simulators, since actual hardware with area and timing constraints has to be designed.

Another issue is the low level of observability offered by online debugging tools like ChipScope plus the resource overhead. This problem could be mitigated by the development of an application specific debug framework that is tailored to capturing information about multiprocessor systems. Hardware simulators such as Modelsim are also of indispensable help when designing a circuit.

We also observe an impedance mismatch between the speed of the off-chip DDR memory which runs much faster than the internal processing elements. This mismatch could be exploited by using the fast external memory to model multiple independent smaller memories which would better support architecture research where each core has its own local memory. Alternatively, one controller on each FPGA can be dedicated to model secondary-level caches, since such large caches can not be realized on our FPGA.

Although the Virtex-5 FPGAs that we use in this work do not allow for implementing a greater number of cores or large multi-level caches, new FPGAs that support 6-LUTs on 28 nm technology double the total number of gates available on chip while allowing the designer to have access to up to six MB of Block RAM capacity and thousands of DSP units. Such abundance of resources might be more suitable for building larger and faster prototypes on reconfigurable infrastructures.

Opencores [10] can be a very important catalyst for the development of other emulation initiatives in the academia. The openFPGA consortium aims to standardize core libraries, APIs and benchmarks for FPGA computing. While consortia such as RAMP can be driving forces for FPGA technology in the computer architecture community, more support and tools are needed to make the process mainstream. We need readily-usable interfaces, programming and debugging tools, and more helper IP cores. The available parameterizable IPs that are already tested should be used whenever possible. For that and other reasons we are planning to publicly release our hardware and software source code.

Finally, other researchers have advocated the use of higher level hardware description languages to improve programmer productivity. In Chapter 6 we introduce an equivalent multicore implemented in a high-level language.

3.7 Conclusions

In this chapter, we have described a different roadmap in building a full multicore emulator: By heavily modifying and extending a readily available soft processor core. We have justified our design decisions in that the core be small enough to fit many on a single FPGA while using the on-chip resources appropriately, flexible enough to easily accept changes in the ISA, and mature enough to run system libraries and a well-known STM library. We have presented an 8-core prototype on a FPGA prototyping platform and compared performance and scalability to software simulators for three benchmarks written to explore tradeoffs in Transactional Memory.

Our hypothesis was that an FPGA-based prototype would have a simulation speed that scaled better with more modeled processor cores than a software-based instruction set simulator. The Beefarm architecture shows very encouraging scalability results which helps to support this hypothesis. For small numbers of cores we find that software-based instruction set simulators are still competitive, however as more cores are used, the gap between the software and hardware based approaches widens dramatically. This is illustrated clearly with the ScalParC example where for 4 cores the Beefarm system outperforms the M5 simulator in fast mode by 8x.

Our experience showed us that place and route times, timing problems and debugging cores are problematic issues working with FPGAs. We have also identified parts of a typical multiprocessor emulator that map well on FPGAs, such as processing elements, and others that map poorly and consume a lot of resources, such as a cache coherency protocol or a large system bus. We have described different strategies to minimize the resource consumption of one of these units, floating-point support.

4 Hybrid Transactional Memory on FPGA

4.1 Introduction

In this chapter we present TMbox, a multicore prototype with architectural extensions to support Hybrid Transactional Memory (TM, explained in Section 3.2). From our experience implementing Beefarm, we take some architectural decisions like a new interconneciton topology.

Today's FPGA systems can integrate multiple hard/soft processor cores, multi-ported SRAM blocks, high-speed DSP units, and programmable I/O interfaces with configurable fabric of logic cells. With the abundance of pre-tested Intellectual Property (IP) cores available, nowadays it is possible to prototype large architectures in a full-system environment which allows for faster and more productive hardware research than software simulation. Over the past decade, the RAMP project has already established a well-accepted community vision and various novel FPGA architecture designs [52, 54, 85, 104, 116, 127]. Another advantage of FPGA emulation over software simulation is the reduced profiling overhead and the possibility for a variety of debugging options.

One direction is to choose a well-known architecture like MIPS and enjoy the commonly-available toolchains and library support. In this prototype, we implemented our own thin hardware abstraction layer to provide a minimal subset of the standard C library.

Transactional Memory can be implemented in hardware (HTM) [49, 95], which is fast but resource-bounded while usually requiring changes to the caches and the ISA, or software (STM) [65] which can be flexible, run on off-the-shelf hardware, albeit at the expense of lower performance. To have the best of two worlds, there are intermediate Hybrid TM (HyTM) proposals where transactions first attempt to run on hardware, but are backed off to SW when HW resources are exceeded, and Hardware-assisted STM (HaSTM) which by architectural means aims to accelerate a TM implementation that is controlled by software [47, 56].

Despite the fact that FPGA emulators of many complex architectures of various ISAs have been proposed, only a few of these are on TM research, and only up to a small number of cores. Furthermore, the majority of these proposals are based on proprietary or hard processor cores, which imply rigid pipelines that can prevent an architect from modifying the ISA and the microarchitecture of the system.

In this chapter, we present TMbox, a shared-memory CMP prototype with Hybrid TM support. More specifically, our contributions are as follows:

- A description of the first 16-core implementation of a Hybrid TM that is completely modifiable from top to bottom. This implies convenience to study HW/SW tradeoffs in topics like TM.
- We detail on how we construct a multicore with MIPS R3000-compatible cores, interconnect the components in a bi-directional ring with backwards invalidations and adapt the TinySTM-ASF hybrid TM proposal to our infrastructure.
- Experimental results and comparisons of STM, HTM and Hybrid TM performance for three TM benchmarks designed to investigate trade-offs in TM.

The next section presents the TMbox architecture, Section 4.3 explains the Hybrid TM implementation, Section 4.4 discusses the limitations and the results of running three benchmarks on TMbox. Section 4.5 concludes this chapter.

Table 4.1: LUT occupation of components

PC_next	Mem_ctrl	Control	Reg_File	Bus_mux	ALU	Shifter	MULT	Pipeline	Cache	TLB	TM_unit	Bus_node	DDR_ctrl	UART
138	156	139	147	155	157	201	497	112	1985	202	1242	619	1119	77

Total: 6946

4.2 The TMbox Architecture

The basic processing element of TMbox is the Honeycomb CPU core, which was implemented by heavily modifying and extending the Plasma soft core [11]. The development philosophy and experience of this core has been thoroughly explained in Chapter 3. The Honeycomb core (without an FPU and the DDR controller) occupies 5704 LUTs (Table 4.1) on a Virtex-5 FPGA including the ALU, MULT/DIV and Shifter units, the coherent L1 cache and the UART controller, a comparable size to the Microblaze core.

The BEE3 hardware prototyping platform contains four Virtex5-155T FPGAs, each one with 98K LUTs, 212 BRAMs, and 128 DSP blocks. Each FPGA controls four DDR2 DIMMs, organized in two channels of up to 4 GB each. The DDR2 controller [118] manages one of the two channels, performs calibration and serves requests, occupying a small portion of the FPGA (around 2%). Using one controller provides sequential consistency for our multicore since there is only one address bus, and loads are blocking and stall the processor pipeline.

4.2.1 Interconnection

On each FPGA, we designed and implemented a bi-directional ring as shown in Figure 4.1. Arranging the components on a ring rather than a bus requires shorter wires which eases placement on the chip, relaxing constraints, and is a simple and efficient design choice to diminish the complexities that arise in implementing a large crossbar on FPGA fabric. Apart from increased place and route time, longer wires would lead to more capacitance, longer delay and higher dynamic power dissipation. Using a ring will also enable easily adding and removing shared components such as an FPU or any application-specific module, however this is out of the scope of this work.

CPU requests move counterclockwise; they go from the cores to the bus controller, *e.g.*

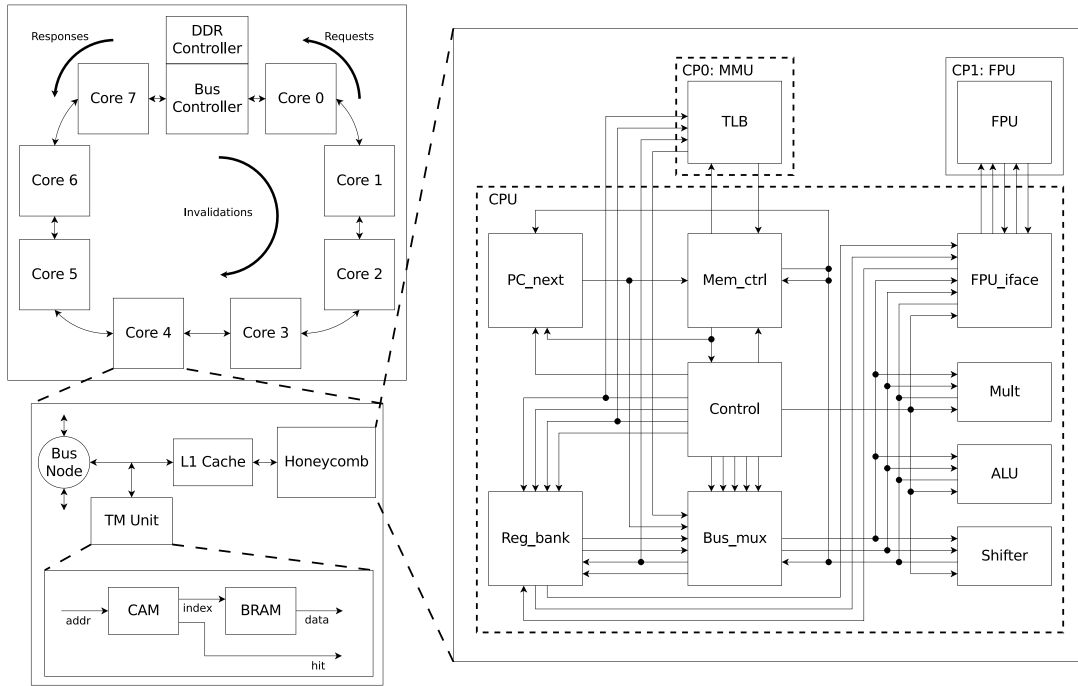


Figure 4.1: An 8-core TMbox infrastructure showing the ring bus, the TM Unit and the processor core.

$CPU_i \rightarrow CPU_{i-1} \rightarrow ... \rightarrow CPU_0 \rightarrow$ Bus Ctrl. Requests may be in form of load or store, carrying a type field, a 32-bit address, a CPU ID and a 128-bit data field, which is the data word size in our system. Responses also go in the same direction; from the bus controller to the cores, *e.g.* Bus Ctrl. $\rightarrow CPU_n \rightarrow CPU_{n-1} \rightarrow ... \rightarrow CPU_{i+1} \rightarrow CPU_i$, and use the same channel as requests, carrying responses to the read requests served by the DDR Ctrl. On the opposite channel, moving clockwise are backwards invalidations caused by write requests which move from the Bus Ctrl. towards the cores in the opposite direction, *e.g.* Bus Ctrl. $\rightarrow$ CPU_0 - ... $\rightarrow CPU_{i-1} \rightarrow CPU_i$. These carry only a 32-bit address and a CPU ID field. When a write request meets an invalidation on any node, it gets cancelled. The caches on each core also snoop and discard the lines corresponding to the invalidation address. We detail how we extend this protocol for supporting HTM in the next section.

4.3 Hybrid TM Support for TMbox

TinySTM [65] is a lightweight and efficient word-based STM library implementation in C and C++. It differentiates from other STMs such as TL2 and Intel STM mainly by its time-

Table 4.2: HTM instructions for TMbox

Instruction	Description
`XBEGIN (addr)`	Starts a transaction and saves the abort address (addr) in TM register `$TM0`. Also, it saves the contents of the `$sp` (stack pointer) to TM register $TM1.
`XCOMMIT`	Commits a transaction. If it succeeds, it continues execution. If it fails, it rolls back the transaction, sets TM register `$TM2` to *ABORT_CONFLICT*, restores the `$sp` register and jumps to the abort address.
`XABORT (20-bit code)`	Used by software to explicitly abort the transaction. Sets TM register `$TM2` to *ABORT_SOFTWARE*, restores the `$sp` register and jumps to the abort address. The 20-bit code is stored in the TM register `$TM3`.
`XLB, XLH, XLW,` `XSB, XSH, XSW`	Transactional load/store of bytes, halfwords (2 bytes) or words (4 bytes).
`MFTM (reg), (TM_reg)`	Move From TM: copies the value of a TM register to a general purpose register.

based algorithm and lock-based design. By default, it compiles and runs on 32 or 64-bit x86 architectures, using the atomic_ops library to implement atomic operations, which we modified to include Compare and Swap (CAS) and Fetch and Add (FAA) primitives for the MIPS architecture using load-linked and store conditional (LL/SC) instructions. TinySTM-ASF is a hybrid port that enables TinySTM to be used with AMD's HTM proposal, ASF [53], which we modified to work with TMbox. This version starts the transactions in hardware mode and jumps to software if (i) hardware capacity is exceeded, (ii) there is too much contention or (iii) the application explicitly requires it. Our hardware design closely follows the ASF proposal with the exception of nesting support.

A new processor model (-march=honeycomb) was added by modifying GCC and GAS (the GNU Assembler). This new Instruction Set Architecture (ISA) includes all the R3000 instructions plus RFE (Return from Exception), LL, SC and the transactional instructions in Figure 4.2. All GNU tools GAS, ld, objdump, *etc*. were modified to work with these new instructions.

To enable hardware transactions, we extended our design with a per-core TM Unit that contains a transactional cache that only admits transactional loads and stores. By default

it has a capacity of 16 data lines equaling a total of 256 bytes. If the TM cache capacity is exceeded, the transaction aborts setting the TM register `$TM2` to *ABORT_FULL* after which the transaction falls to software and restarts.

A transactional LD/ST causes a cache line to be written to the TM Unit. An invalidation of any of the lines in the TM Unit causes the current transaction to be aborted. Modifications made to the transactional lines are not sent to memory until the whole transaction successfully commits. The TM Unit provides a single cycle latency for performing lookups on the transactional read/writesets. A Content Addressable Memory (CAM) is built using LUTs to enable asynchronous reads and since BRAM-based CAMs grow superlinearly in resources. Two BRAMs store the data that is accessed by an index provided by the CAM. The TM Unit can additionally serve LD/ST requests on an L1 miss if the line is found on the TM cache.

4.3.1 Instruction Set Architecture Extensions

To support HTM, we augmented the MIPS R3000 ISA with the new transactional instructions listed in Table 4.2. We have also extended the register file with four new transactional registers, which can only be read with the `MFTM` (move from TM) instruction. `$TM0` register contains the abort address, `$TM1` has a copy of the stack pointer for restoring when a transaction is restarted, `$TM2` contains the bit field for the abort (overflow, contention or explicit) and `$TM3` stores a 20-bit abort code that is provided by TinySTM, *e.g.* abort due to malloc/syscall/interrupt inside a transaction, or maximum number of retries reached, *etc*.

Aborts in TMbox are processed like an interrupt, but they do not cause any traps, instead they jump to the abort address and restore the `$sp` (stack pointer) in order to restart the transactions. Regular loads and stores should not be used with addresses previously accessed in transactional mode, therefore it is left to the software to provide isolation of transactional data if desired. LL/SC can be used simultaneously with TM provided that they do not access the same addresses.

Figure 4.2 shows an atomic increment in TMbox MIPS assembly. In this simple example, the abort code is responsable for checking if the transaction has been retried a maximum number of times, and if there is a hardware overflow (the TM cache is full), and in this case jumps to an error handling code (not shown).

```
    LI    $11, 5          // set max. retries = 5
    LI    $13, HW_OFLOW   // reg 13 has err. code
    J     $TX

$ABORT:
    MFTM  $12, $TM2       // check error code
    BEQ   $12, $13, $ERR // jump if HW overflow
    ADDIU $10, $10, 1     // retries++
    SLTU  $12, $10, $11   // max. retries?
    BEQZ  $12, $ERR2      // jump if max. retries

$TX:
    XBEGIN($ABORT)        // Provide abort address
    XLW   $8, 0($a0)      // Transactional LD
    ADDi  $8, $8, 1       // a++
    XSW   $8, 0($a0)      // Transactional ST
    XCOMMIT               // if abort go to $ABORT
```

Figure 4.2: TMbox MIPS assembly for `atomic{a++}` (NOPs and branch delay slots are not included).

4.3.2 Bus Extensions

To support HTM, we add a new type of request, namely *COMMIT_REQ*, and a new response type, *LOCK_BUS*. When a commit request arrives to the DDR, it causes a backwards *LOCK_BUS* message on the ring which destroys any incoming write requests from the opposite direction, and locks the bus to grant exclusive access to perform a serialized commit action. All writes are then committed through the *time window* created, after which the bus is unlocked with another *LOCK_BUS* message, resuming normal operation. More efficient schemes can be supported in the future to enable parallel commits [49].

4.3.3 Cache Extensions

The cache state machine reuses the same hardware for transactional and non-transactional loads and stores, however a transactional bit dictates if the line should go to the TM cache or not. Apart from regular cached read/write, uncached accesses are also supported, as shown in Figure 4.3. Cache misses first make a memory read request to bring the line and wait in `WaitMemRD` state. In case of a write operation, the `WRback` and `WaitMemWR` states manage the memory write operations. While in these two states, if an invalidation arrives to the same address, the write will be cancelled. In case of a store-conditional instruction,

the write will not be re-issued, and the LL/SC will have failed. Otherwise, the cache FSM will re-issue the write after such a write-cancellation on invalidation.

In case of processing a transactional store inside of an atomic block, an incoming invalidation to the same address will cause an abort and possibly the restart of the transaction. Currently our HTM system supports lazy version management: the memory is updated at commit-time at the end of transactions, as opposed to having in-place updates and keeping an undo log for aborting. We also provide eager conflict detection which implies that data inconsistencies are detected immediately. Aborts due to conflicts can only happen if an address in the TM cache gets invalidated during the transaction execution (between `XBEGIN` and `XCOMMIT`/`XABORT`).

To support HTM, the cache state machine is extended with three new states, `TMbusCheck`, `TMlockBus` and `TMwrite`. One added functionality is to dictate the locking of the bus prior to committing. Another job is performing burst writes in case of a successful commit which runs through the `TMwrite` → `WRback` → `WaitMemWR` → `TMwrite` loop. The `TMwrite` state is also responsible for the gang clearing of all entries in the TM cache and the writeset entries in cache after a commit/abort. To enable this, address entries that are read from the TM Unit are sent to L1 cache as invalidation requests, after which the TM cache is wiped out in preparation for a new transaction.

4.4 Experimental Evaluation

TMbox can fit 16 cores in a Virtex-5 FPGA, occupying 86,797 LUTs (95% of total slices) and 105 BRAMs (49%). In this section, we first examine the limitations of our implementation, we then discuss the results of three TM benchmarks.

4.4.1 Architectural Benefits and Drawbacks

On the TM side, the performance of our best-effort Hybrid TM is bounded by the size of the transactional cache of the TM unit. Although for this work we chose to use a small, 16-entry TM cache, larger caches can certainly be supported on the TMbox on larger FPGAs (keeping in mind the extra area overhead introduced).

In pure HTM mode, all 16 lines of the TM cache can be used for running the transaction in hardware, however the benchmark can not be run if there are larger transactions that do

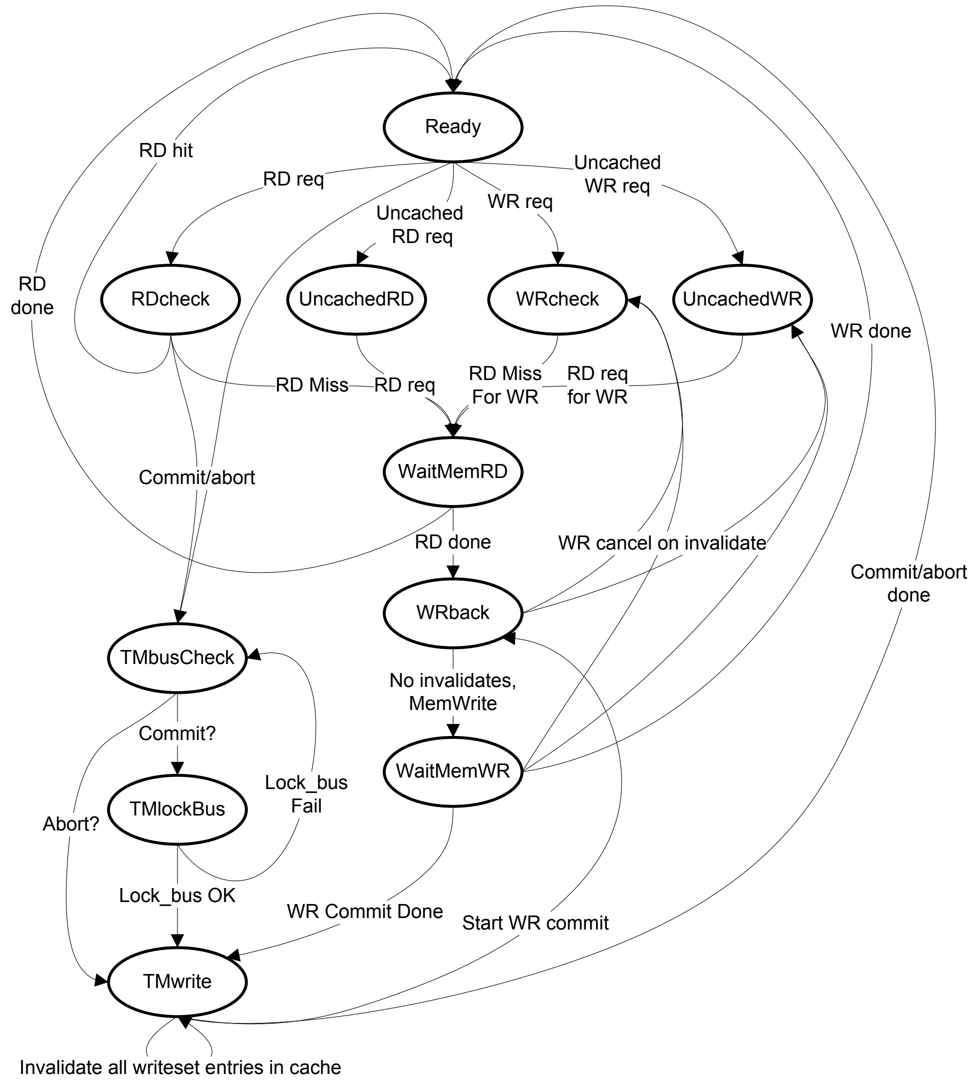


Figure 4.3: Cache state diagram. Some transitions (LL/SC) are not shown for visibility.

Table 4.3: TM Benchmarks Used in TMbox

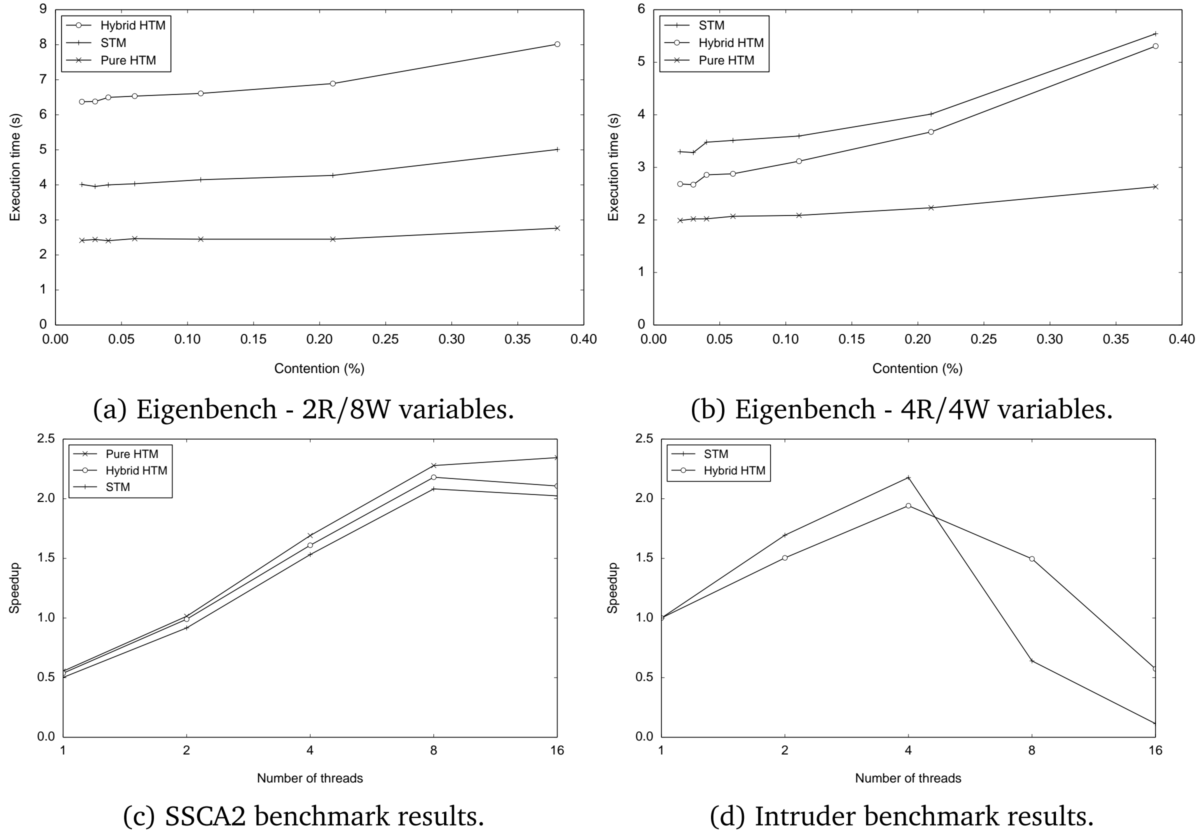
TM Benchmark	Description
Eigenbench [80]	Highly tunable microbenchmark for TM with orthogonal characteristics. We have used this benchmark (2000 loops) with (i) r1=8, w1=2 to overflow the TM cache and vary contention (by changing the parameters a1 and a2) from 0–28%, and (ii) r1=4 and w1=4 to fit in the TM cache and vary the contention between 0–35%.
Intruder [93]	Network intrusion detection. A high abort rate benchmark, contains many transactions dequeuing elements from a single queue. We have used this benchmark with 1024 attacks.
SSCA2 [93]	An efficient and scalable graph kernel construction algorithm. We have used problem scale = 12.

not fit the TM cache, since there is no hardware or software mechanism to decide what to do in this case. The largest overhead related to STM is due to keeping track of transactional loads and stores in software. The situation can worsen when the transactions are large and there are many aborts in the system.

In hybrid TM mode, it is desired to commit as many transactions as possible on dedicated hardware, however when this is not possible, it is also important to provide an alternative path using software mechanisms. All transactions that overflow the TM cache will be restarted in software, implying all work done in hardware TM mode to be wasted in the end. Furthermore to enable hybrid execution, TinySTM-ASF keeps the lock variables inside the TM cache as well. This results in allowing a maximum of 8 variables in the read/writesets of each transaction as opposed to 16 for pure HTM. Of course this is true provided that neither the transactional variables, nor the lock variables share a cache line, in which case, in some executions we observed some transactions having a read/writeset of 9 or 10 entries successfully committing in hardware TM mode.

On the network side, the ring is an FPGA-friendly option: we have reduced the place and route time of an 8 core design to less than an hour using the ring network, where before it took us more than two hours using a shared crossbar for interconnection. However, each memory request has to travel as many cycles as the total number of nodes on the ring plus the DDR2 latency, during which the CPU is stalled. This is clearly a system bottleneck: using write-back caches or relaxed memory consistency models might be key in reducing the number of messages that travel on the ring to improve system performance.

On the processor side, the shallow pipeline negatively affects the operating frequency of the CPU. Furthermore larger L1 caches can not fit on our FPGA, however they could



(a) Eigenbench - 2R/8W variables.

(b) Eigenbench - 4R/4W variables.

(c) SSCA2 benchmark results.

(d) Intruder benchmark results.

Figure 4.4: SSCA2 and Intruder benchmarks.

be supported on larger, newer generation FPGAs, which would help the system to better exploit locality. Decoupling the instructions and data by having separate caches would also be a profitable enhancement.

4.4.2 Experimental Results

Eigenbench is a synthetic benchmark that can be tuned to discover TM bottlenecks. As Figure 4.4a shows, the transactions in EigenBench with 2R+8W variables overflow (since TinySTM-ASF keeps the lock variables in the transactional cache) and get restarted in software, exhibiting worse performance than STM. However, the 4 read-4 write variable version, shown in Figure 4.4b, fits in the cache and shows a clear improvement over STM.

In the SSCA2 results presented in Figure 4.4c, we get a 1-8% improvement over STM because this benchmark contains small and short transactions that can fit in the transac-

tional cache. Although Intruder (Figure 4.4d) is a benchmark used frequently for TM, it is not a TM-friendly benchmark, causing a high rate of aborts and non-scalable performance. However, especially with 16-cores, our scheme achieves in (i) discovering conflicts early and (ii) committing 48.7% of the total transactions in hardware, which results in almost 5x superior performance compared to direct-update STM, which has to undo all changes on each abort. We were unable to run this benchmark on pure HTM since it contains memory operations like malloc/free inside transactions that are complex to run under HTM and are not yet supported on TMbox.

The three benchmarks can benefit from our hybrid scheme because they do not run very large transactions, so most of the fallbacks to software caused are due to repeated aborts or mallocs inside transactions. For SSCA2, we see good scalability for up to 8 cores, and for Intruder for 4 cores. The performance degradations in STM for Intruder are caused by the fact that the STM directly updates the memory and as the abort rates increase, its performance drastically decreases. Furthermore, compared to the sequential versions of the benchmarks, the TM versions perform in the range of 0.2x (Intruder) to 2.4x (SSCA2). These and other scalability problems can be caused by architectural issues such as the limits of the ring bus, serialized commits or the particular coherency mechanism.

4.5 Conclusions

We have presented a 16-core prototype on a FPGA providing hardware support and accelerating a modern TM implementation running significant benchmarks that are widely used in TM research. We implemented the necessary ISA extensions, compiler support and microarchitectural mechanisms to support Hybrid TM.

The results agree with our insights and findings from other works [93]: Hybrid TM works well when hardware resources are sufficient, providing better performance than software TM. However, when hardware resources are exceeded, the performance can fall below the pure software scheme in certain benchmarks. The good news is that Hybrid TM is flexible; a smart implementation should be able to decide what is best by dynamic profiling. This is an interesting example of research opportunities unveiled by this FPGA prototype.

We have also shown that a ring network fits well on FPGA fabric and using smaller cores can help building larger prototypes.

5 Profiling and Visualization on FPGA

5.1 Introduction

In the previous chapter we have extended a FPGA-based multicore prototype to support Transactional Memory (TM, introduced in Section 3.2). In this chapter we upgrade TMbox with profiling support, allowing for full state inspection with minimal overhead.

TM support can be provided by flexible but slower software libraries (STM), or by using fast, dedicated hardware (HTM). An HTM system is usually bound by some sort of capacity constraints, *i.e.* the hardware can only handle transactions with specific characteristics. Hybrid TM aims to provide the best of two worlds. Transactions first attempt to run on the dedicated TM hardware and fall back to software when it is not possible to complete the transaction in hardware (e.g. when resources are exceeded) [56]. ATLAS and later Configurable TM were the first systems to support hardware TM on FPGA [81, 127]. Hardware acceleration proposals by using Bloom filters for TM were also investigated [38, 86], as well as our Hybrid TM platform for FPGA, TMbox, which has been described in Chapter 4.

But another issue remains: Performance and scalability are both important for a suc-

cessful adoption of TM. Profiling executions in detail is absolutely necessary to have a correct understanding of the qualities and the disadvantages of different implementations and benchmarks. A low-overhead, high-precision profiler that can handle both hardware and software TM events is required. Up to now, no comprehensive profiling environment supporting STM, HTM and Hybrid TM has been developed.

Due to its flexibility and extensibility, an FPGA is a very suitable environment for implementing profiling mechanisms and offers a unique advantage based on three main aspects. Firstly, compared to a software simulator, there are no overheads in simulation time due to the special hardware added. Moreover, FPGAs emulate real hardware interfacing real storage or communication devices and exhibit a higher degree of fidelity than software simulators. Second, the relative area overhead for implementing extra profiling circuitry can be very low, and the throughput high, as we demonstrate. Third, because of its customizability, we are free to extend the architecture with new application-specific instructions for profiling. We use this flexibility to reduce the software overheads of the profiling calls added to the programs, as we will show with the new `event` instruction.

Using these advantages in utilizing FPGAs for multicore prototyping, we address three main issues:

- STM application profiling can suffer from high overheads, especially with higher levels of detail (e.g. looking into every transactional load/store). Such behavior may influence the application runtime characteristics and can affect and alter the interactions of threads in the program execution, producing unreliable conclusions.

- Hardware extensions for a simulated HTM system and a software API was suggested by the flagship HTM-only profiling work, TAPE [50]. It is useful for pinpointing and optimizing undesired HTM behaviors, but incurs some overhead due to API calls and saving profiling data to RAM. We argue that using an FPGA platform, hardware events can come for free.

- Visualizing executions in a threaded environment can be an efficient means to observe transactional application behavior, as was looked into in the context of an STM in C# [136]. A profiling framework facilitates capturing and visualizing the complete execution of TM applications, depicting each transactional event of interest, created either by software or by hardware.

The purpose of this work is to address these shortcomings and to develop a complete monitoring infrastructure that can accept many kinds of software and hardware transactional events with low overhead in the context of a Hybrid TM system on FPGA. This is the first study to profile and visualize a Hybrid TM scenario, with the capabilities to examine in detail how hardware and software transactions can compliment each other. For gathering online profiling information, first we describe (i) profiling hardware that supports generating TM-specific hardware events with zero execution overhead, and (ii) an extension to the Instruction Set Architecture (ISA) called the `event` instruction that enables a low, single-cycle overhead for each event generated in software. Later, a post-processing tool that generates traces for the threaded visualization environment Paraver [106] is engaged. The resulting profiling framework facilitates to visualize, identify and quantify TM bottlenecks: It allows to understand the interaction between the application and the TM system, and it helps to detect bottlenecks and other sub-optimal behavior in the process. This is very important for optimizing the application for the underlying system, and for designing faster and more efficient TM systems.

Running full TM benchmarks, we compare different levels of profiling and their overheads. Furthermore, we show visualization examples that can lead the TM programmer/designer to make better and more reliable choices. As an example, we demonstrate how using our profiling mechanism the Intruder benchmark from STAMP [93] can be ported to best utilize Hybrid TM resources. Such a HW/SW event infrastructure can be easily modified to examine in detail full complex benchmarks in any research domain.

The next section presents the design objectives and the extensions made to TMbox, a Hybrid TM implementation on FPGA. Section 5.3 explains the infrastructure that implements the profiling mechanism in order to produce meaningful HTM/STM events and to process them offline on a host. Section 5.4 presents overhead results running TM applications and example traces illustrating the features of our profiling mechanism. In Section 5.5 we conclude this chapter.

5.2 Design Objectives

To get a complete overview of TM behavior, it is vital to have a system with low impact on application runtime characteristics and behavior, otherwise the optimization hints gathered

could cause a misguided attempt to ameliorate the system. Since the profiling infrastructure will be designed on actual hardware, we cannot implement unrealistic behavior, and the new circuitry has to map well on the reconfigurable fabric, with minimal overheads. We made three key design choices to get low execution and low area overhead and not to disturb placing and routing on the FPGA:

- Non-intrusively gather and transfer runtime information by implementing the monitoring hardware separately. Build the monitoring infrastructure only by attaching hardware hooks to the existing pipeline.
- Make use of the flexibility of the ISA and the GCC toolchain to add new instructions to the ISA to support STM events with low profiling overhead.
- Use little area on the FPGA by adding minimal extra circuitry, without widening the buses or causing extra routing overheads. To transfer the events non-disruptively, instead of adding a new events network, we utilize the idle cycles on an already-existing network.

5.2.1 TMbox Architecture

The TMbox system has been introduced in Chapter 4. This multicore features the best-effort Hybrid TM proposal ASF [53], which is used with TinySTM [65], a lightweight word-based STM library. The transactions are first started in hardware mode with a *start tx* instruction. A transactional load/store causes a cache line to be written to the special TM Cache. *Commit tx* ends the atomic block and starts committing it to memory. An invalidation of any of the lines in the TM Cache causes the current transaction to be aborted (Figure 5.1). Transactions are switched to software mode when (i) TM Cache capacity, which is by default 16 cache lines (256 bytes) is exceeded, (ii) the abort threshold is reached because of too much contention in the system or (iii) the application explicitly requires it, e.g. in case of a system call or I/O inside of a transaction. In software mode, the STM library is in charge of that transaction, keeping track of read and write sets and managing commits and aborts. This approach enables using the fast TM hardware whenever it is possible, but meanwhile to have an alternative way of processing transactions that are more complex or too large to make use of the TM hardware.

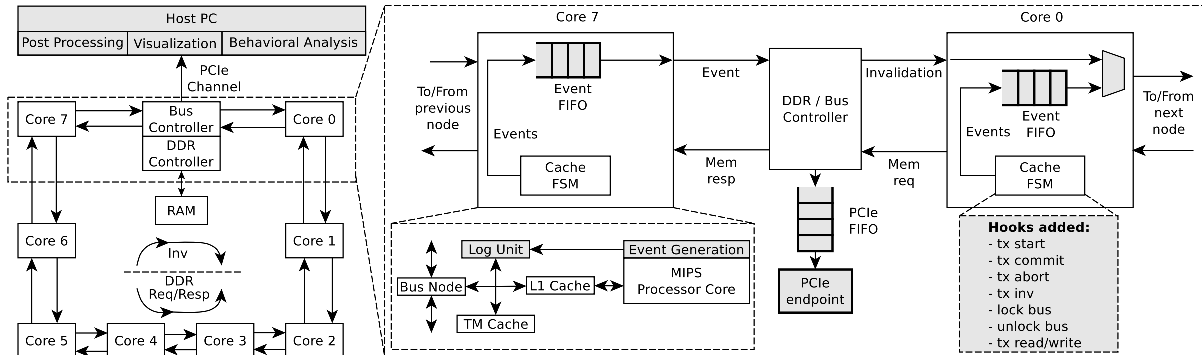


Figure 5.1: An 8-core TMbox system block diagram and modifications made to enable profiling (shaded).

The hardware TM implementation supports lazy commits: Modifications made to the transactional lines are not sent to memory until the whole transaction is allowed to successfully commit. However, TinySTM supports switching between eager and lazy commit and locking schemes as we will look into with software transactions in Section 5.4.2.

A bidirectional ring bus interconnects the CPUs of the TMbox. This design decision allows for an FPGA-friendly implementation: Short wires ease the placement on the chip and relax timing and routing constraints (a property that we do not want to break). It also helps keeping the caches coherent; CPU requests and DDR responses move counter-clockwise, whereas invalidation signals that are generated by writes to the DDR move in the opposite direction (Figure 5.1). Whenever a write request meets an invalidation to the same address on any node of the ring bus, it gets cancelled. Meanwhile, the caches on each core also snoop and discard the lines corresponding to the invalidation address, effectively enabling system-wide cache coherency.

5.2.2 Network reuse

To cause as little area and routing overhead as possible, we discard the option of adding a dedicated network for events. Instead, we choose to piggyback on an existing network. More particularly, we utilize the idle cycles on the less-frequently-used invalidation bus. However, we do not want to disturb the execution by causing extra network congestion, so we give a lower priority to profiling events by first buffering them and transferring them only when a free cycle on the bus is detected. This way, the profiling packets do not disrupt the traversal of the already-existing invalidation packets in any way.

Although this approach might be somewhat specific to the TMbox architecture, we believe that the methodology of always first buffering the created events, and injecting them in the network only on a free slot could be applied to different network types, as well. Future work could address how to implement similar functionality on a different network type, such as a mesh or a tree. A disadvantage of this approach is that the fixed message format of the invalidation ring bus has to be matched. Another drawback is watching out for buffer overflows. The next section explains in detail how the design decisions affected the way the TMbox system was modified to support creating, propagating, transferring and post-processing timestamped TM events.

5.3 The Profiling and Visualization Infrastructure

The profiling and visualization framework consists of performing three steps on the FPGA and the final step on the host computer. First, the TM behavior of interest is decomposed into a small, timestamped event packet containing information about the change of state. Second, the event is propagated on the bus to the central Bus Controller node. Third, from the node, it is transferred on the fly by PCI Express (PCIe) to a host computer. Finally, the post-processing application running on the host parses all event packets and re-composes them back to meaningful, threaded events with absolute timestamps, and creates the Paraver trace of the complete application.

5.3.1 Event specification and generation

HTM events

The event generation unit (Figure 5.1) monitors the TM states inside the cache Finite State Machine (FSM) of the processor, generating events whenever there is a state change of interest, e.g. from *tx start* to *tx commit*. Figure 5.2 shows the format of an event in a detailed way. The timestamp marks the time when an event occurred, and is delta-encoded: only the time difference in cycles between two consecutive events is sent. This space-efficient encoding allows a temporal space of about a million cycles (20 ms @ 50 MHz) between two events occurring on a processor. The event data field stores additional data available for an event, for instance the cause of an abort (e.g. capacity overflow, software-induced,

Message Header		Message Data		
2 bits	4 bits	20 bits	4 bits	4 bits
Message Type	CPU Sender ID	Timestamp (δ-encoded)	Event Type	Event Data

Figure 5.2: Event format for the profiling packets.

invalidation). Due to the 4-bit wide event type, we can define up to 16 different event types. Some of the basic event types defined for hardware transactions include: *tx start*, *tx commit*, *tx abort*, invalidation, lock/unlock ring bus (for performing commits). These hardware events come with zero execution overhead, since the profiling machinery works in parallel to the cache FSM. Our infrastructure supports easily adding and modifying events, as long as there is a free event type encoding available in hardware.

The fact that we can only use 20 bits for the timestamp in order to match the predefined message format can cause wraparounds, so the Paraver threads can fail to be properly synchronized. To address this, we added an extra event type that is very rarely used. When it detects a timestamp counter overflow, in the next event, it also sends the number of idle timestamp overflows occurred along with the timestamp. Although the bus and the event messages could also be widened, we opted for modifying the existing hardware as little as possible to accomplish as low overhead as possible. This is also the reason why we eliminated the option of having a separate bus only for the events.

Extending the ISA with STM events

For generating low overhead events from software, an `event` instruction was added to the processor model by modifying the GNU Compiler Collection (GCC) and the GNU Binutils suite (GAS and objdump). The `event` instruction creates STM events with a similar encoding to the HTM events, supporting up to 16 different software events that are implemented in special event registers. Little hardware with a small area overhead of 32 LUTs/core had to be added: extending the opcode decoder, some extra logic for bypassing the data, and multiplexing it into the event FIFO. More complex processor architectures might need to be more heavily modified to add new instructions and registers.

However, the ability to create such precise events from software with single-instruction overhead is a very powerful tool for closely inspecting a variety of behaviors. Software events can be modified simply by storing the wire/register/bus values of interest in event

registers and by reading them from software with an `event` call.

Similar to the "free" hardware events discussed earlier, the events generated in software also utilize the same event FIFO. However, software events have some execution overhead: one instruction per event. In the next section, we compare execution overheads of this approach to software-only events created on a commodity machine, and demonstrate that utilizing the `event` instruction actually contributes to a smaller overhead in runtime.

5.3.2 Event propagation on the bus

A logging unit captures events sent by the event generation unit located in each core. Here, the event is timestamped using delta encoding and enqueued in the event FIFO. As soon as an idle cycle is detected on the invalidation bus, the event is dequeued and transferred towards the bus controller.

To prevent a disturbance of program runtime behavior, the profiling events are classified as low-priority traffic on the invalidation ring bus. So, invalidation packets always have higher priority. Consequently, when the ring bus is busy transferring invalidation messages, it is necessary to buffer the generated events. To keep the events until a free slot is found, event FIFOs (one BRAM each) were added to each core, as shown in Figure 5.1.

The maximum rate at which an invalidation can be generated on the TMbox is once every three cycles. The DDR controller can issue a write every three cycles, which translates into an invalidation message that has to traverse the whole bus. Therefore, for an 8-core ring setup, the theoretical limit of starting to overflow into the event FIFOs is when one event is created by all cores every 12 cycles. Using a highly contended shared counter written in MIPS assembly, we observed that the FIFOs never needed to have more than 4 elements. This is the worst case behavior: TM programs written in high level languages incur further overhead through the use of HTM/STM abstraction frameworks and thus would actually exhibit a smaller pressure on the buffers of the monitoring infrastructure.

Changing the network type for the system would imply the need to modify the infrastructure to look for and to use empty cycles or to add another data network for events which would come with routing issues and area overhead. While with a dedicated event bus this step would have been trivial, better mapping on the FPGA requires a lower cost approach. Therefore, we reuse the already-available hardware and only incurring area overhead by placing FIFOs to compensate for traffic congestion.

5.3.3 Transfer of events to the host

To transfer the profiling packets, we use a PCIe connection that outputs data at 8 MB/sec, coupled with a large PCIe output FIFO placed to sustain temporary peaks in profiling data bandwidth. In our executions, we did not experience overflows and lost packets, although the throughput of the PCIe implementation is obviously limited. A suitable alternative for when a much greater amount of events are created (e.g. at each cache miss/hit), might be to save to some large on-chip DDR memory instead of transferring the events immediately. However this memory should preferably be apart from the shared DDR memory of the multicore prototype, for reasons of non-disruptiveness. The profiling data might reach sizes of many MB, so saving the profiling data on on-chip BRAMs is not a viable option.

5.3.4 Post-processing and execution regeneration

After the supervised application has terminated, and all events have been transferred to the host machine, they are fed in to the Bus Event Converter. This program, which we implemented in Java, (i) parses the event stream, (ii) rebuilds TM and application states, and (iii) generates statistics that are compatible for visualizing with Paraver [106]. The mature and scalable program Paraver was originally designed for the processing of Message Passing Interface (MPI) traces, which we adapted to visualize and analyze TM events and behavior. Our post-processing program converts the relative timestamps to absolute timestamp values and re-composes the event stream into meaningful TM states. At this point, additional states can also be created, depending on the information acquired through the analysis of the whole application runtime. This removes the need to modify the hardware components to add and calculate new states and events during the execution, and allows for a more expressive analysis and visualization.

5.4 Experimental Evaluation

In order to demonstrate the low overhead benefits of the monitoring framework proposed, we ran STAMP [93] applications using Eigenbench [80], a synthetic benchmark for TM mimicry. STAMP is a well-known TM benchmark suite with a wide range of workloads, and Eigenbench imitates its behavior in terms of number and size of transactions, read/write

Table 5.1: Area overhead per processor core and the tracked events in different profiling options

Profiling Type	Area Overhead (per CPU core)	Actions Tracked
STM-only (x86 host)		SW start tx, SW commit tx, SW abort tx
STM-only	32 5-LUTs + 1 BRAM	SW start tx, SW commit tx, SW abort tx
HTM-only	129 5-LUTs + 1 BRAM	HW start tx, HW commit tx, HW abort tx, lock bus, unlock bus, HW inv, HW tx r/w, HW PC
Hybrid TM (CG)	129 5-LUTs + 1 BRAM	HTM-only + STM-only
Hybrid TM (FG1)	129 5-LUTs + 1 BRAM	Hybrid TM (CG) + SW tx r/w + tx ID
Hybrid TM (FG2)	129 5-LUTs + 1 BRAM	Hybrid TM (FG1) + SW inv + SW PC

sets and many other orthogonal characteristics. We used the parameters for five STAMP benchmarks provided by the authors of Eigenbench. We compare runtime overheads of the profiling hardware to an STM-only implementation which generates runtime event traces in a way comparable to our FPGA framework. This version called STM (x86) tracks each transactional start, commit, and abort events in TinySTM running on a Westmere[1] server. The events are timestamped and placed in a buffer, which is written to a thread-local file handle.

Along with STM and HTM profiling, we engage three levels of Hybrid TM profiling to enable both light and detailed profiling options. The coarse-grained (CG) version features the typical HTM and STM events (Table 5.1). Besides the most common *start tx*, *commit tx* and *abort tx* events, we also look at invalidation events and the overheads of locking/unlocking the bus for commits (part of the HTM commit behavior of TMbox). Additionally, there are two fine-grained profiling options that are implemented through the `event` mechanism in software. FG1 includes tracking all transactional reads and writes, also useful for monitoring readset and writesets. It also keeps transaction IDs, which are needed for dynamically identifying atomic blocks and associating each transactional operation with them.

In addition, the maximum profiling level FG2 that we implemented features source code identification, a mechanism for monitoring conflicting addresses and their locations in the code. For enabling this, a `JALL` (Jump And Link and Link) instruction was added to the MIPS ISA. This extends the standard `JAL` instruction by storing an additional copy of the return address, which is kept as a reference to be able to identify the Program Counter (PC)

[1] OS is Linux version 2.6.32-29-server (Ubuntu 10.04 x86_64).

of the instruction that is responsible of the subsequent transactional read/write operations in TinySTM. This way, a specific event with that unique PC is generated by the transactional operations in these subroutines, effectively enabling us to identify the source code lines with low overhead.

5.4.1 Runtime and area overhead

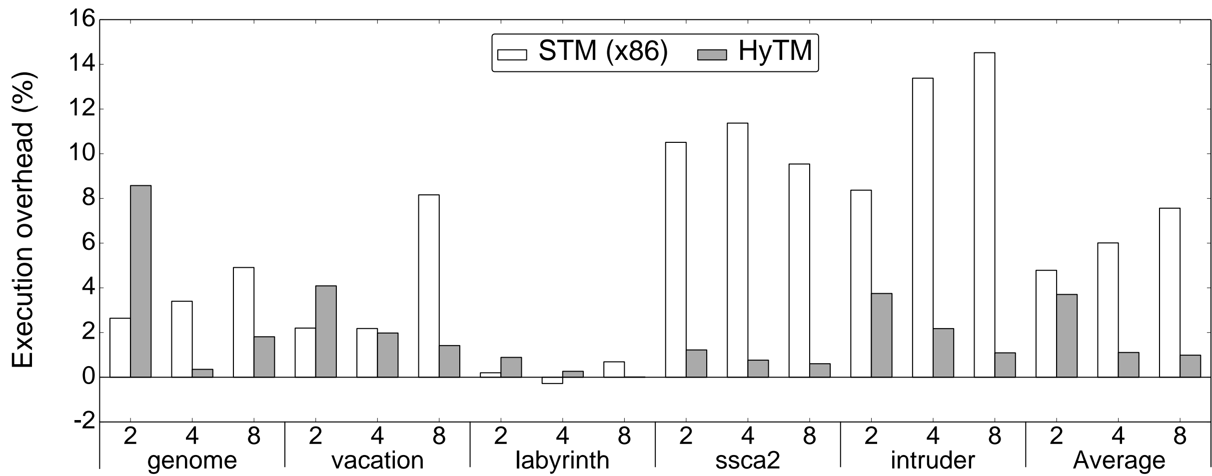
In Figure 5.3a, STM (x86) profiling overhead was compared to our FPGA framework with the same level of profiling detail (STM-only). The overhead introduced by the FPGA implementation is less than half of the STM (x86) overhead, on average. This is largely due to adding the `event` instruction to the ISA to accomplish a single instruction overhead per software event. Please note that if the transactional read and write events were tracked additionally, we would expect a larger slowdown for STM (x86).

Figure 5.3b shows the extra overhead that our FPGA profiler causes by turning on all kinds of TM profiling capabilities. Almost half a million events were produced for some benchmarks. With the highest level of detail, the average profiling overhead was less than 6% and the maximum 14%.

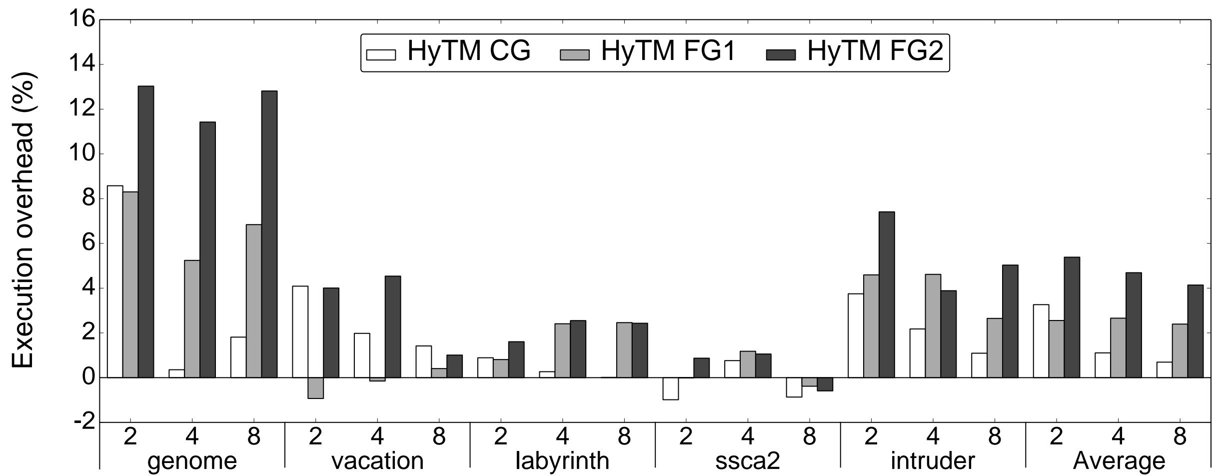
When the transactions can be run on the dedicated hardware, as in the case of SSCA2, the overall profiling overhead is lower. This is because hardware events come “for free” and less work has to be done in software, where there is some overhead. Therefore, the success of the Hybrid TM drives that of the TM profiling machinery. The higher the percentage of transactions that can complete in hardware, the faster and more efficient the execution of the TM program is, and the more lightweight is the profiling. Conversely, Genome has many transactions that never fit the dedicated TM hardware and exhibit higher overheads. For future work, we want to investigate new techniques that could allow zero runtime overhead in STM profiling. This would also avoid the interferences caused by profiling in the normal program behavior, which we observed in the case of Vacation.

Interesting cases of low Hybrid TM CG overheads appear when running Genome with 4 threads and SSCA2 with 2 and 8 threads. This behavior is due to the specific Eigenbench parameters for STAMP benchmarks, eg. Genome’s parameter values for CPU 3 are huge and cause the application to behave much differently for 4 threads than for 2 threads.

The inclusion of profiling hardware to TMbox results in a 2.3% increase in logic area, plus the memory needed to implement the event FIFO (1 BRAM) for each processor core.



(a) Runtime overhead (%) for STM (x86) vs. STM (FPGA), in different core counts and applications. (avg. 20 runs).



(b) Runtime overhead (%) for different Hybrid TM profiling levels, core counts and applications. (avg. 20 runs).

Figure 5.3: STAMP-Eigenbench Benchmark Overheads

The fixed area overhead of the PCIe endpoint plus the PCI_FIFO occupies 3978 LUTs and 30 BRAMs. The critical path was not affected by these changes.

5.4.2 Improvement Opportunities

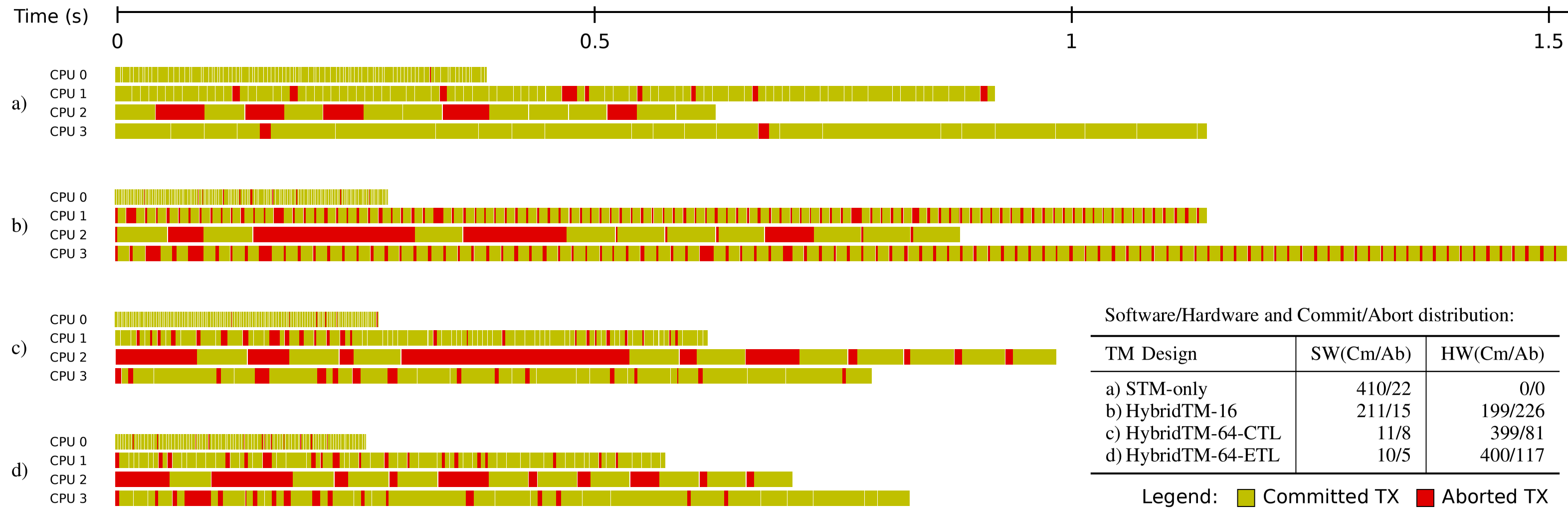
In this section, we present sample Paraver traces for the Intruder benchmark to demonstrate how our low overhead profiling infrastructure can be useful in analyzing TM benchmarks and systems. First, we run the Eigenbench-emulated Intruder and suggest a simple methodology to improve the application's execution for the appropriate usage of TM resources. Next, to visualize TM behavior and pathologies from real application characteristics, we depict some example traces running the actual, non-emulated Intruder benchmark from the STAMP suite.

Intruder-Eigenbench

Figure 5.4 shows the four traces of a simple refinement process using our profiling mechanism. By running the STAMP application Intruder with 4 CPUs, we attempt to derive the best settings both in hardware and in software for running this application in Hybrid TM mode. The program has to complete a total of 410 transactions on four threads. Around 300,000 events are generated in the highest profiling mode for this benchmark. On overall, a 24.1% improvement in execution time was observed when moving from STM-only to Hybrid TM-64-ETL. This final version of Intruder is able to utilize both the TM hardware and the software TM options better.

Intruder-STAMP

To pinpoint real application behavior, the actual non-emulated Intruder from STAMP was ran with 128 attacks [93]. Intruder is an interesting benchmark in the sense that (i) it contains a mix of short and long transactions that can sometimes fit in the dedicated transactional hardware, and other times overflow, (ii) typically has a high abort rate which is interesting for TM research, (iii) exhibits real transactional behavior, such as I/O operations inside transactions, and (iv) demonstrates phased behavior, which shows an inherent advantage of our visualization infrastructure.



Software/Hardware and Commit/Abort distribution:

TM Design	SW(Cm/Ab)	HW(Cm/Ab)
a) STM-only	410/22	0/0
b) HybridTM-16	211/15	199/226
c) HybridTM-64-CTL	11/8	399/81
d) HybridTM-64-ETL	10/5	400/117

Figure 5.4: Improving Intruder-Eigenbench step by step from an STM-only version to utilize Hybrid TM appropriately.
STM-only: Here, the TM application is profiled with FG1 level profiling, where a count of the number of read/write events for each transaction is kept. By analyzing the profiled data, we discover a certain repetition of small transactions which can benefit from HTM acceleration. However, there also exist very large transactions, suggesting that an HTM-only approach is not feasible.
Hybrid-TM-16: Introduces HTM with a 16-entry TM Cache per processor core, so this trace depicts both STM and HTM events. CPU 0 seems to have benefited from using HTM (as shown in the table in Figure 5.4), although there are still some small transactions on other CPUs that might fit if the hardware buffers are increased in size. Please note that a poorly-configured Hybrid TM can end up showing worse performance than an STM.
Hybrid TM-64-CTL: Uses a larger, 64-entry TM Cache. Here, CPUs 1 and 3 also start utilizing HTM efficiently, causing the software transactions to reduce in number. However, CPU 2 suffers from long aborting transactions and a large wasted work. Looking at the available software TM options, we switch from Commit Time Locking (CTL) to Encounter Time Locking (ETL) to discover some conflicts early and to decrease the abort overheads.
Hybrid TM-64-ETL: On overall, a 24.1% speedup in execution time when moving from STM-only to Hybrid TM-64-ETL was observed. Although there are only 3 fewer aborts in software now, they cause much less wasted work, helping the application to run faster to completion.

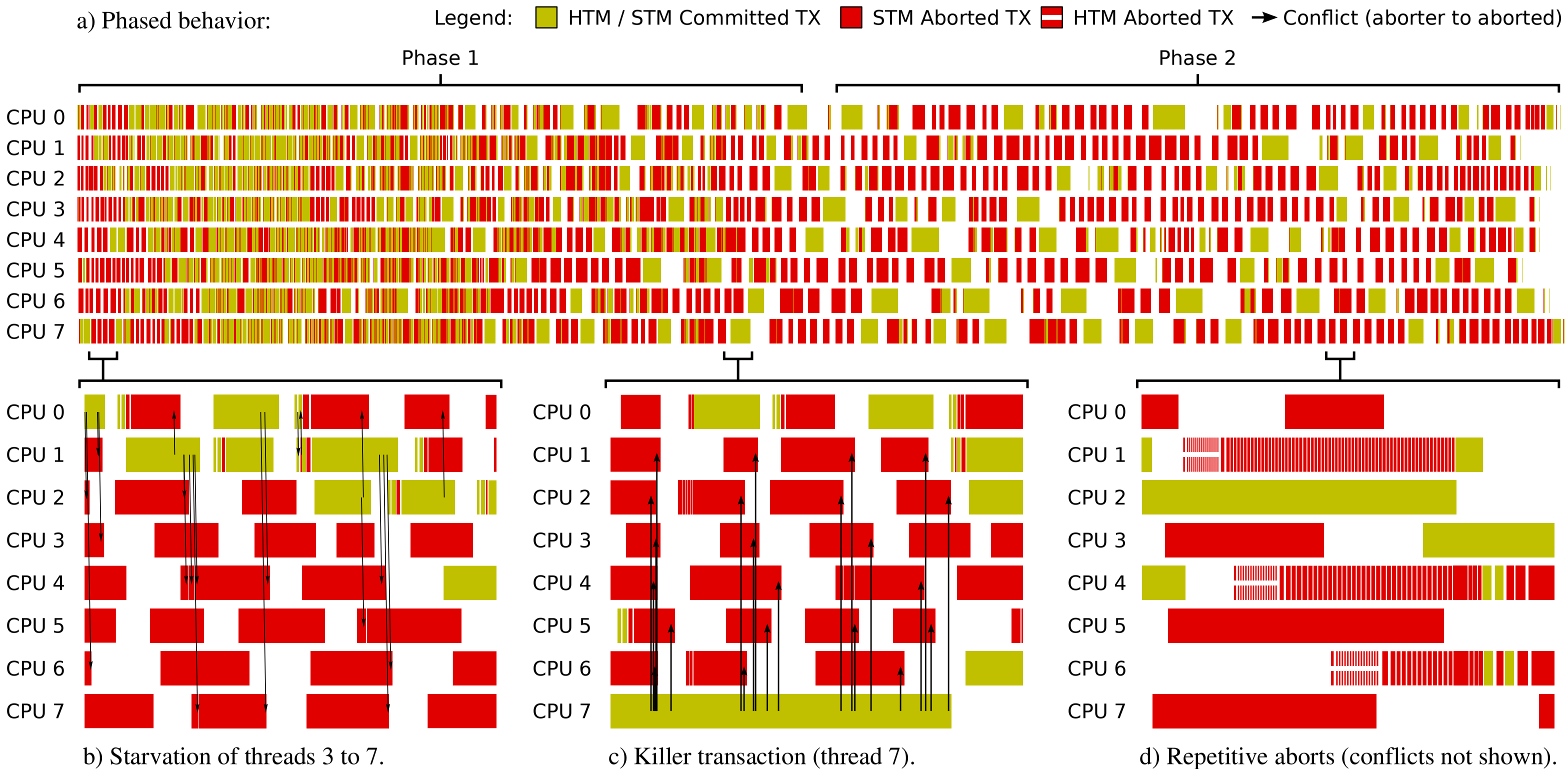


Figure 5.5: Example traces showing phased behavior and transactional pathologies in Intruder, a network intrusion detection algorithm that scans network packets for matches against a known set of intrusion signatures.
The benchmark consists of three steps: capture, reassembly, and detection. The main data structure in the capture phase is a simple queue, and the reassembly phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session.
a) Phased behavior: A program does not always exhibit the same behavior throughout its execution, and in terms of TM might show different phases of high aborts, shorter transactions, or serialization. In the first half of this trace, there are more transactions and parallelism. Here, the packets are being constructed by all the threads and there is not enough complete data to process for detection. In the second half of the execution, complete packets are ready for the detector function, which generates less (but larger) transactions in number, which results in more conflicts among them. Dynamic switching mechanisms would be suitable for treating adequately phased behavior in TM applications.
b) Starvation: A clear example of starvation on CPUs 3, 5, 6 and 7 (towards the beginning of the benchmark) is shown.
c) Killer transaction: Illustrates a single transaction (CPU 7) aborting six others. After it commits, other CPUs can finally take the necessary locks and start committing successfully.
d) Repetitive aborts: Demonstrates the pathology of repetitive aborts and its effect on the execution, as in [39]. Finding the optimal abort threshold (to switch to STM mode) could be important in such cases.

Figure 5.5 describes and depicts phased behavior and some examples of different pathologies that can be discovered thanks to the profiling framework. Some solutions to these problems include rewriting the code, serialization, taking pessimistic locks or guiding a contention manager that can take appropriate decisions.

Our infrastructure could perform additional actions automatically. Some transactions are more suitable to run in software and others in hardware. Detecting HW/SW partitioning can avoid the wasted work caused by the transactions that are sure to abort in HTM mode. Automatic switching the STM mechanisms (CTL vs. ETL, or lazy vs. eager versioning), either statically or dynamically, could be achieved by looking at how early the aborts happen, transaction sizes and other relevant data. Additionally, by modifying the application software and the post-processing application, and adding events of interest, various advanced profiling information can be reached by analysis. Some examples are to draw sets/tables of conflicting `atomic{}` blocks, or read/write sets. Profiling the reasons of the aborts could help contention management schemes. Such advanced STM profiling was studied in Java [23] and Haskell [105].

5.5 Conclusions

An FPGA, for its flexibility in programming and its speed, is a convenient tool for the customization of hardware and application-specificity. Based on this, we have built the first profiling environment capable of precise visualization of HTM, STM and Hybrid TM executions in a multi-core FPGA prototype. We have used a post-processing tool for events and Paraver for their interactive visualizations. Taking into consideration non-intrusiveness and low overhead, the extra hardware added was small but efficient. It was possible to run STAMP TM benchmarks with maximum profiling detail inside the 14% overhead limits. On average, we incurred half the overhead of an STM-only software profiler. Our infrastructure also proved successful to port the Intruder benchmark to use Hybrid TM and get a speedup of 24.1%, and to detect bottlenecks and transactional pathologies.

The profiling framework could be easily adapted to work for any kind of multicore profiling and visualization, and with other state-of-the-art shared memory hardware proposals such as speculative lock elision, runahead execution or speculative multithreading.

6

High-level Debugging and Verification

6.1 Introduction

During the implementation of Beefarm (Chapter 3) and TMbox (Chapters 4 and 5) we have learned valuable lessons. One is the need of high-level languages and tools for hardware development, along with a methodology that reduces the distance between the different stages of the design work flow. Moreover, debugging and verification are time-consuming tasks that slow down the development process. In this chapter we want to address these issues and test our solutions in a new prototype, Bluebox.

The simulation workload linearly grows with the number of simulated cores, leading to unaffordable simulation times. Speeding up the execution through parallelization usually requires either relaxing the correctness requirements, or replacing the computation of the simulation with approximated statistical models. In both cases, the accuracy of the simulation may be affected.

New software simulators have been proposed to face these challenges [36, 45, 103, 108]. These new-generation tools are targeted to simulate tens to hundreds of cores, with

simulation errors within 25% and simulation performance ranging from 0.2 to 314 Million Instructions per Second (MIPS). Different techniques have been proposed, from statistical models that elide the computations [36, 103] to interval simulation based on key architectural events [45, 108].

However, such non-cycle-accurate software simulators have several limitations, as we have discussed in Section 2.2, making them unsuitable for simulating novel computer architectures. At the same time, FPGA-based simulators can overcome these limitations.

As we discussed in Section 2.4, it is a general consensus in the community that FPGA-based simulators are precise and fast, but when discussing this alternative the higher development cost is argued as a detritment [45, 108]. We believe that the complexity of FPGA-based simulator development arises from three main sources:

1. The low productivity in FPGA development, which can be addressed with High-Level Synthesis tools and new generation Hardware Description Languages (HDL).

2. Long compilation times, which may be reduced from hours to minutes using overlay architectures.

3. The notorious difficulty of inspecting and debugging hardware models, especially once running on the FPGA.

Regarding the languages and tools, we performed an empirical and qualitative evaluation, described in Appendix A, of four representative high-level hardware description languages and tools. The results of our comparison show that such new approaches to hardware design can obtain similar performance to legacy languages with less implementation effort. Among the examined languages and tools, we have chosen Bluespec SystemVerilog. The reasons for this election are summarized in the next section.

In this chapter we want to address the latter issue, debugging and verification. For that purpose, we developed a design methodology, along with non-obtrusive debugging and verification mechanisms, for high-level software-based and FPGA-based simulation of multicore models. We implemented applied this methodology in Bluebox, a RISC multicore system written in Bluespec SystemVerilog. Our platform is highly customizable, boots the Linux kernel and supports the whole MIPS I ISA. We provide three different simulation modes for the three different stages of the development: a fast, functional software sim-

ulation for architecture validation; a slow, cycle-accurate software simulation for timing validation; and a fast, cycle-accurate FPGA-based simulation mode.

In this work we make the following contributions:

- We developed Bluebox, a multicore that boots Linux and supports the whole MIPS I ISA.
- We extend the Bluespec SystemVerilog simulator to implement system-level and user-level debugging, checkpointing and verification.
- As we discuss in Section 6.5.1, host-assisted verification is not feasible. For that reason, we implement non-obtrusive debugging, based on a gated clock domain, and verification on the FPGA. Our techniques respect the cycle-accuracy of the simulation and the I/O, and we use hardware to accelerate 99.5% of the verification.
- We run a 24-core simulation, achieving a full-system verification performance of 17 MIPS.

The rest of this chapter is organized as follows. In the next section we present the Bluebox multicore architecture and its three simulation modes. In Section 6.3 we extend the host-based simulation to implement debugging and checkpointing functionalities. In Section 6.4 we describe how to implement FPGA-based microarchitectural debugging and verification mechanisms. In Section 6.5 we evaluate the performance of our solution. In Section 6.6 we conclude this chapter.

6.2 The Bluebox Synthesizable Multicore

We developed this architecture both as a research platform and as a proof of concept of our debugging and verification techniques. The objectives when designing Bluebox were:

1. To simulate as many cores as possible.
2. To support a popular operating system and a standard Instruction Set Architecture (ISA).
3. To be completely modifiable (i.e., no private units that cannot be seen or modified by the researcher).

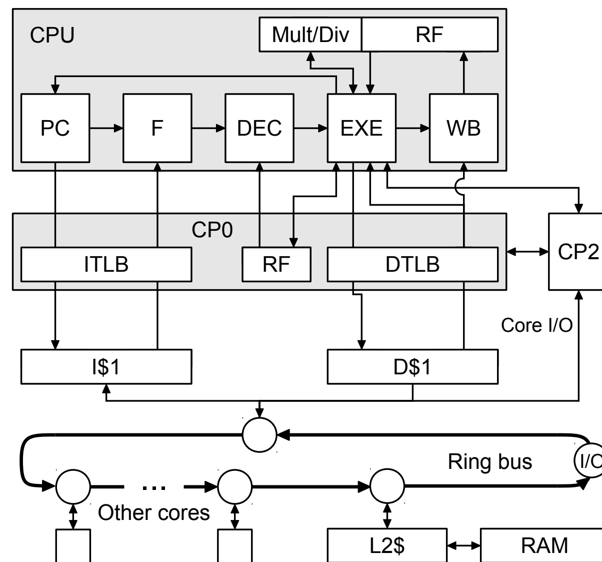


Figure 6.1: A Bluebox core with the CPU, instruction and data caches. In the MIPS I ISA, Coprocessor 0 manages the exceptions and virtual memory. We chose to dedicate Coprocessor 2 to timers, I/O and inter-processor messaging.

4. To be implemented in a high-productivity language.

Bluebox is a multicore architecture that supports the whole MIPS I ISA. We started from the educational softcore sMIPS, a 5-stage, 32-bit RISC processor, and we added arithmetic and multiplication/division units and a unified Memory Magagement Unit (MMU) with exceptions and interrupts, in addition to atomic instructions for synchronization. These minimal characteristics ensure a reduced footprint while still being able to run any standard application and boot the Linux Kernel 2.6. At this moment the core does not include a floating point unit because such units consume a significant amount of resources in FPGAs, but this functionality is emulated by Linux natively. We are considering implementing area-efficient solutions such as shared FPUs between cores.

Each CPU has independent instruction and data caches. The memory hierarchy implements a MSI (Modified-Shared-Invalid) coherency protocol. The CPU and the caches conform one core, as shown in Figure 6.1. A level 2 cache manages the coherency. All the cores are interconnected using a ring bus. We chose this topology because its short interconnects from core to core are better suited for FPGAs, which increases the frequency and eases placing and routing.

One important aspect of Bluebox is that most of its characteristics are parameterizable. Most of the instruction subsets of the CPU can be disabled to reduce the footprint, such as multiply/divide; byte, half-word and unaligned memory accesses; virtual memory; and exceptions and interrupts. The pipeline supports several optimizations which can be enabled, like branch delays (a characteristic of the MIPS ISA), a branch predictor, and bypassing values between stages, which reduces the execution cycles but could decrease the frequency. The size of the caches, the size of the cache lines and the number of ways of the level 2 cache are configurable as well.

We implemented the architecture with the commercial language Bluespec SystemVerilog (BSV) [1]. It was convenient for our purposes because its rule-based, data-flow nature gives the designer a tight control over the hardware model, and it allows extending the simulation with software as we will see in next section.

We also considered other alternatives, like C-based High-Level Synthesis (HLS) languages, such as Xilinx Vivado or LegUp [43]. But currently it is difficult to express the cycle model with these tools.

Our platform can be simulated in three modes, representing the three stages of the development process:

- `funcsim`: A functional and fast software simulator, for verification and validation.
- `timesim`: A detailed but slow simulation on host.
- `fpgasim`: A fast and detailed simulation on the FPGA.

The `funcsim` model is a single-stage CPU without memory caches. We used this functional simulator to validate the subsequent stages of the development, as in [99, 121]. `funcsim` can be used for fast functional verification of new research hardware and software features as well.

In the next section we will detail the first two modes, consisting of software simulation on a host machine. FPGA-based simulation will be covered in Section 6.4.

6.3 Software Extensions For High-Level Debugging

Once the architecture was validated with `funcsim`, we implemented a BSV model of Bluebox. The BSV compiler can generate a Verilog model, which can be mapped to a FPGA, and

a cycle-accurate C++ simulation model. However, this latter software simulation model does not allow direct inspection of the state elements, such as memories or registers. It supports `$display` commands, like Verilog, which can print information during software simulations in a similar fashion as `printf` in C programs. But this debugging level is not sufficient for complex hardware systems like multicore architectures.

For this reason, we extended the BSV C++ simulation model with new features, in order to facilitate a software-development-like experience. Verilog and BSV support a programming interface to call software functions from within the simulation. We used this interface to extend the BSV simulator and produce `timesim`, a set of C++ extensions over the BSV simulator.

6.3.1 Externalizing The State

The state of the simulation at any moment is determined by several architectural elements. These elements are stored in BSV primitives, such as memories and registers, which cannot be queried by external software. We identified such elements and moved them to the C++ extensions of `timesim`. Then we replaced the BSV state primitives, such as registers and memories, by wrapper modules with compatible ports. Accessing these wrappers generates C++ calls to the external, software-managed state.

The elements managed by `timesim` are:

- The CPU register file.
- The MIPS multiply/divide unit registers.
- The MIPS Coprocessor 0 register file, used for virtual memory, exceptions and interrupts.
- The Translation Look-aside Buffer (TLB) entries.
- The main RAM.

We excluded other elements, like the contents of the caches, in this version of Bluebox. However, the same methodology could be applied to inspect them.

6.3.2 Debugging And Profiling

In this section we describe how we use these extensions. The external C++ calls from within the simulation have another side effect: the BSV simulator yields the execution to the external function, which pauses the execution because it is single-threaded. We used this effect to implement debugging functionalities to the BSV simulator.

The main debugging breakpoint is in the execution stage of the CPU, where the changes of the current instruction are applied to the state. Before this happens, an external C++ function is called, which yields the control to the user through an interface. The user can inspect the contents of the state managed by `timesim`, and send commands like advance one instruction, run freely or stop on a given address. This kind of environment is familiar to programmers, and allows the designer to abstract from the low level details and focus on the architectural elements such as instructions and CPU registers.

Currently, the `funcsim` and `timesim` modes of Bluebox support several kinds of interfaces, like terminal, ncurses (graphical terminal), a Python shell, and a GDB server. The user interfaces show helpful information like the disassembly of the current instructions and the contents of the register file. The GDB interface allows the popular GDB debugger to connect to `funcsim` and `timesim` and control the execution of the hardware model as if it was a multithreaded application (each core appears to GDB as if it was a software thread). Some debuggers allow source-code-level debugging, abstracting from the machine code of the simulation. This feature is very helpful when debugging applications with complex macros and build systems like the Linux Kernel.

All the events collected by the extensions can also be saved into traces and visualized in a graphical way. `funcsim` and `timesim` allow to store special events and later visualize the trace using Paraver [106], a tool to graphically visualize supercomputing execution traces. Bluebox can generate different Paraver traces, including:

- The software threads created by Linux. This trace required software-side collaboration, which represented a single line of code in the Linux scheduler and an overhead of 1 cycle each time a software process is scheduled.
- The execution mode of a CPU over time, distinguishing between normal execution and special modes like interrupt serving or virtual memory management.
- The exact software function that was executed each cycle. This trace is very detailed,

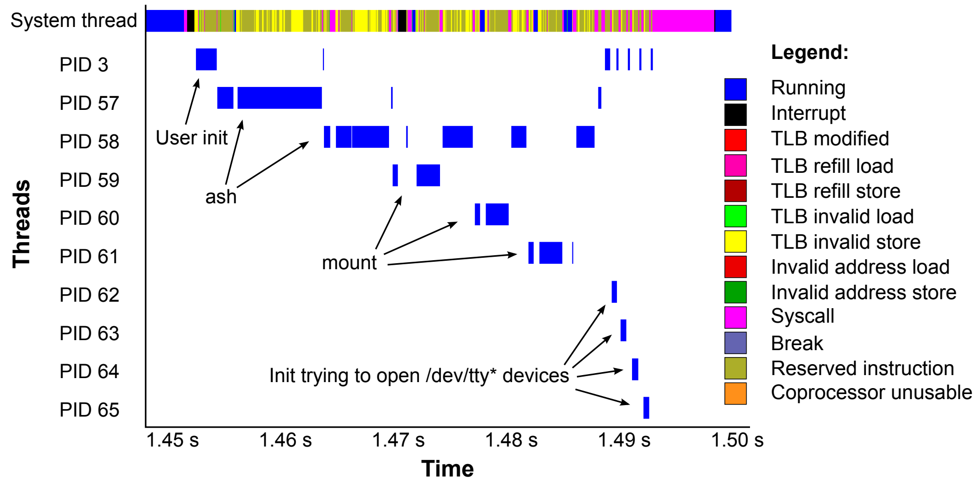


Figure 6.2: Execution trace using Paraver, displaying Linux threads and the execution mode of the CPU.

and helps the designer to understand the execution path of the hardware until an error appears.

In Figure 6.2 we show an example Paraver trace of a Bluebox core booting the Linux Kernel.

We also implemented some features which simplify the debugging of the multicore, such as the capacity of loading binary applications. Traditionally, hardware designers use Verilog commands to load memory contents from plain text files, which requires some simulation cycles and transforming the data into the desired hardware state. In some FPGA design environments the contents of the memories can be hardcoded in the description of the hardware, which requires recompiling the model each time the contents are modified. `funcsim` and `timesim` can preload the memories, like the RAM or the register file, at the beginning of the simulation from complex formats such as Executable and Linkable Format (ELF) files, the common Linux executable format.

The ELF files may also contain debugging information, which helps `funcsim` and `timesim` to offer advanced information to the user. Examples of such information include the name of the current function being executed, or the name of the data variable being accessed by load and store instructions. Moreover, we implemented a call stack decoder that shows the source-code lines which lead to the current instruction.

6.3.3 Checkpointing

The detailed simulation with `timesim` is faster than the detailed simulation of Verilog code thanks to the rule-based nature of BSV. But being a cycle-accurate simulation of hardware which can be synthesized, it becomes impractical for large multicore models: `timesim` can execute up to 0.1 MIPS when simulating one core, but the performance falls dramatically when the number of simulated cores grows.

To avoid long simulation times, we implemented a checkpointing mechanism in `timesim` which allows to save the state of the simulation at any given point. This state is saved to a file, and can be loaded later on to resume the simulation from that point with no functional differences. Moreover, the hardware model can be modified between simulation sessions. For instance, if a bug is detected in a later moment of the simulation, a checkpoint can be saved before arriving to the failing cycle, and different hardware models can be tested with the same initial state. Another possibility is avoiding tedious simulations, like booting the Linux Kernel, and resuming the simulation directly when applications can be executed.

This is possible because most of the state is managed by `timesim`. But during the execution of the hardware model there are intermediate stages where part of the state is in hardware units that are not managed by our extensions. These include, for instance, the instructions in the pipeline and the dirty data lines in the data caches, which have been modified but not committed to memory yet. For this reason, the simulation must be driven to a new stable "epoch" where all the state in intermediate hardware has been flushed to units managed by `timesim`.

For this purpose, `timesim` can instruct the CPU pipeline to finish the execution of the current instructions and stop issuing more requests. After doing so, the CPU starts sending special commands to the L1 and L2 data caches to flush any dirty data lines to the main RAM. This behavior required small hardware extensions to the CPU and the caches, which are not present in `fpgasim` in this version.

After these steps, the state managed by `timesim` is coherent with the state of the simulation, and it is saved into a file. Later on, the user can load this checkpoint and the state is preloaded into the hardware elements, which allows resuming the simulation from the previous functional point. Flushing the pipeline and the data caches when saving the simulation, and refilling when resuming it, requires some simulation cycles because this mechanisms are implemented in hardware. This timing differences are minimal, but per-

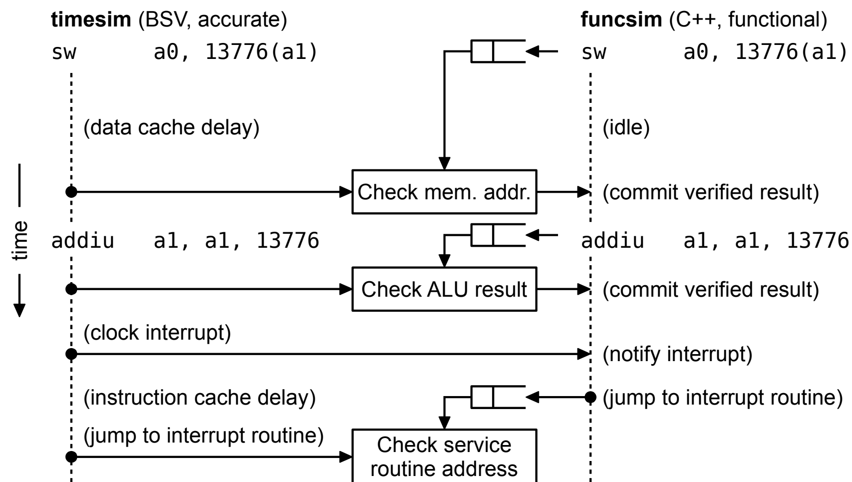


Figure 6.3: Verification of `timesim` using `funcsim`. Functional results are buffered and checked when timing-aware results are available.

fect checkpointing would require `timesim` to manage all the state of the simulation.

To optimize the size of the checkpoint file, the `timesim` RAM manager keeps a registry of modified 4-KB pages. When saving the memory, only such pages are stored. When loading a checkpoint, only the pages in the checkpoint file are loaded to the memory. The hardware mechanisms that allow flushing the buffers of the system could also be synthesized as hardware in the FPGA. But in this version of Bluebox this is not possible yet, especially because saving the contents of the entire DDR RAM can be challenging.

6.3.4 Verification

Errors in the software or the hardware may not exhibit symptoms immediately. For example, the result of a buggy arithmetic operation in the CPU may be sent to the memory. After a few million cycles, when this result is retrieved and used to calculate an instruction address, it will cause a TLB exception. Debugging the origin of this error can be slow and tedious, especially in a multicore running an operating system.

To implement fine-grain, instruction-level verification, we integrated `funcsim` into `timesim` as a functional verifier. In this simulation mode, `funcsim` and `timesim` run in parallel and the result of each instruction is compared. The much lower simulation cost of `funcsim` only represented a modest time overhead for `timesim`. Both simulation modes run independently, but `funcsim` does not have a notion of time and must be synchronized with

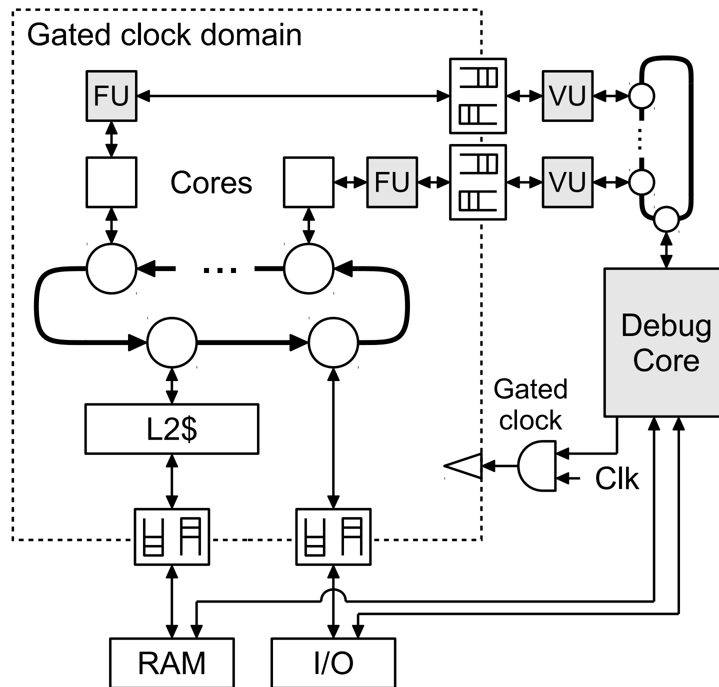


Figure 6.4: Debugging and verification infrastructure of `fpgasim`, with the Debug Core, Filter Units (FU) and Verification Units (VU).

the timing model of `timesim`. For that purpose, we implemented the elastic verification mechanism shown in Figure 6.3, where the immediate results of `funcsim` are queued, and checked when the timing-aware results of `timesim` are generated. If both versions match, the result is committed to the shared state that is managed by `timesim`.

6.4 FPGA-based, On-Chip Debugging And Verification

The BSV hardware model used in `timesim` can be synthesized into Verilog and mapped on to an FPGA. We call this simulation mode `fpgasim`. However, in this mode the software extensions of `timesim` are not available, and any debugging techniques must be implemented as part of the hardware model. In Figure 6.4, we show the architecture of the debugging and verification system.

In `fpgasim` the cores generate a packet of data for each instruction, containing basic debugging and verification information, such as the address of the instruction or the arithmetic results, if any. The system can process this information in two modes: debugging and

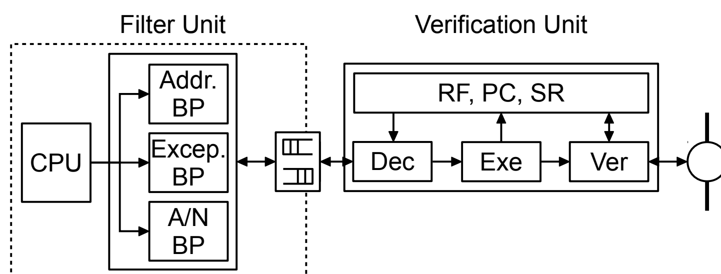


Figure 6.5: Detail of the Filter Unit (FU) and Verification Unit (VU). In debugging mode, the FU filters instructions that do not match current breakpoint type, and the VU bypasses all the information. In verification mode, the FU is disabled (*filter none*) and the VU decodes, executes and verifies simple instructions in one cycle.

verification. In the former, the multicore can be cycle-stepped, and breakpoints can be set thanks to Filter Units (FU). In the latter, all the instructions executed by the all the cores are verified by Verification Units (VU). In Figure 6.5, we show the details of the FU and the VU.

A simplified Bluebox core, the Debug Core, manages this infrastructure. We disabled all the features that were not necessary, such as the virtual memory, to ensure a minimal footprint in terms of resources. This minimal functional subset of the core is the basic unit of trust in our system, and its correctness was verified with `funcsim`.

The whole multicore is isolated in a gated clock domain. If we only paused the CPU, but not the whole system, the locking of the pipeline could hide the delays of the microarchitecture, which would appear as a zero-delay operation. The interfaces to external clock domains, such as the DDR controller or I/O, are protected by synchronization queues.

The gated clock domain protects the internal consistency of the multicore, but the delays that were originated outside cannot be preserved. For instance, pausing the system may hide the latency of the DDR memory. To avoid external delay simulation loss of accuracy, we enriched the cross-domain queues with a latency feedback mechanism. This mechanism is aware of the delay of each request and response. If the execution is paused and some delay is missed, it artificially holds the data to emulate the missed cycles, rebalancing the simulation.

6.4.1 The Debugging Mode

In this mode, the Debug Core runs a debugging software and can send breakpoint commands to the FU located next to each core, inside the gated clock domain. The VU just bypasses the data that was not filtered by the FU.

There are three types of breakpoints:

- Address breakpoints: The information generated by the CPUs is dropped, letting the cores run freely until the address matches. The information is sent to the Debug Core, which pauses the simulation and notifies the user.
- Exception breakpoints: The same behavior as address breakpoints, but the condition is a given exception code (e.g., interrupt, TLB, etc.).
- All/None breakpoints: Either all the information is dropped, letting the core run freely, or all the information is submitted to the Debug Core, which allows cycle-stepping.

6.4.2 The Verification Mode

In verification mode, the Debug Core runs the `funcsim` software model and the FUs are in *filter none* mode, bypassing all the information. We placed the hardware VUs outside the gated clock domain. These dedicated hardware units accelerate the verification of simple instructions, such as arithmetic operations or branches.

These *fast* instructions represent 99.5% of the execution and are checked in one cycle, while the *slow* instructions, mainly related to exceptions and virtual memory, are verified by software in the Debug Core.

Each VU has an instruction decoder and an execution unit. A minimal subset of the state, consisting in the register file, the program counter and the state register is present in the units. The verification proceeds as follows:

1. If a fast instruction is received, it is verified in the VU. If correct, the local state is updated and the information is dropped.
2. If a slow instruction is received or the verification fails, the VU sends the information to the Debug Core and the simulation is stopped.

3. The software in the Debug Core will exchange messages with the VU to obtain the last valid state from their local subset.

4. The instruction is verified by software, and the updated state is sent back to the originating VU.

5. Finally, a continuation message is sent to the VUs and the simulation is resumed.

We included the Coprocessor 0's Status Register (which controls virtual memory and exceptions) in the VU because it is frequently read. This reduced the slow instructions by 30%. Multiplications and divisions are also performed by software. The designer could extend the verification to memory caches, for instance tracking requests and responses.

The Debug Core, when verifying, must have a copy of the state in case an exception is detected. Making such copies requires many cycles. We implemented a simple Transactional Memory [77] scheme to optimize the execution, based on a small journal FIFO, which can undo up to 16 recent writes to the data cache.

6.5 Experimental Evaluation

In this section we will evaluate the performance of the FPGA-based simulator, `fpgasim`. We mapped from 1 to 24 cores to a Xilinx VC709 board, running at a maximum speed of 80 MHz. The 24-core version consumes 77.75% of the logic elements, each core requiring a 2.18% (7345 LUTs, including the CPU and the caches). The verification units represent a total area overhead of 9.80%.

We run the synthetic, integer-only Dhrystone benchmark (one copy in each core) to measure the performance of the multicore and the verification system. Running a floating-point benchmark would force Linux to emulate the FPU through exceptions, which would represent an abnormally-high number of system instructions to verify.

In Figure 6.6 we observe that the multicore reaches a peak performance of 120 MIPS with 16 cores. This data shows that the simulated architecture has performance bottlenecks in the CPU and in the memory hierarchy. Being an FPGA-based simulator, it should be easy to further implement dedicated analysis hardware, such as performance counters, and investigate more advanced interconnection topologies.

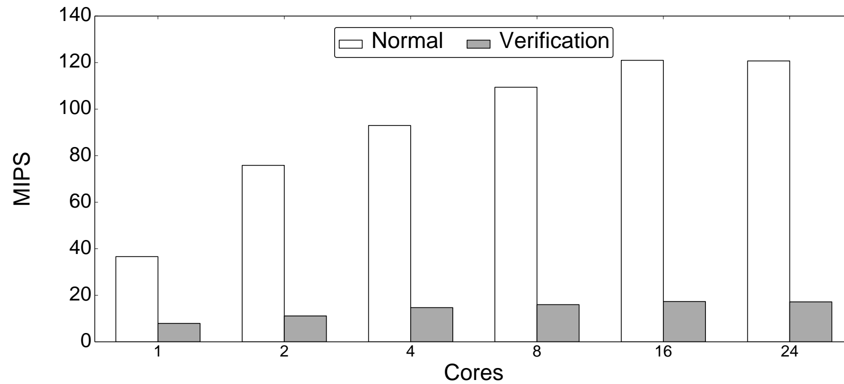


Figure 6.6: Performance of FPGA-based simulation in normal mode and in verification mode.

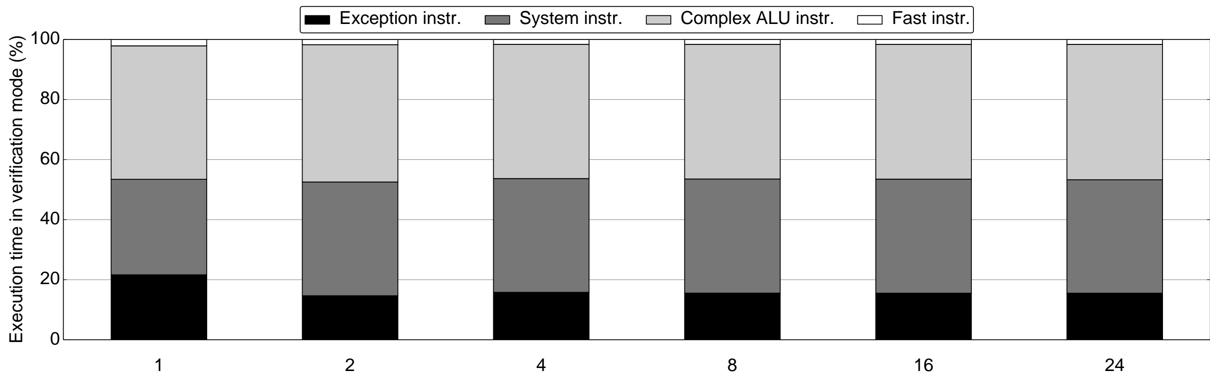


Figure 6.7: Breakdown of time spend on each type of verified instructions. *Fast* instructions are verified in hardware, *slow* in software.

The verification units scale at the same rate as the multicore, reaching a maximum performance of 17 MIPS for 16 cores. The slowdown compared to the non-verified version ranges from 4x to 7x. In Figure 6.7 we show a breakdown of the simulation time in verification mode. The verification of ALU instructions represents a 38% of the simulation time. These instructions are mainly multiply and divide operations, which Dhrystone uses intensively. We did not consider it useful to verify hardware divisions with a similar hardware divider. The system instructions, mainly virtual memory operations, represent the 0.27% of the instructions and 44% of the verification time.

6.5.1 The Suitability Of On-Chip Verification

The instructions that cannot be verified by hardware represent 0.5% of the instructions and require 78% of the execution time. Each slow instruction can take between 600 and

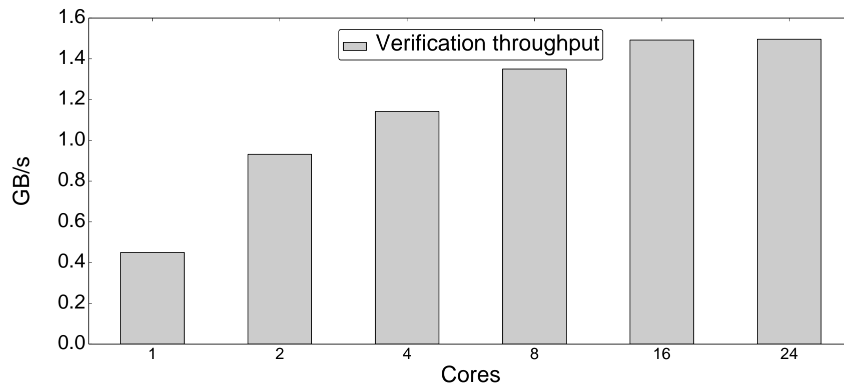


Figure 6.8: Verification data processed by `fpgasim` (GB/s).

1500 cycles of multicore time. Not verifying the system instructions and the exception events would reduce the verification overhead by a 56%, speeding up the verification performance by 2.3x. But developing a full-system simulator, capable of verifying operating system events, was an important feature for us.

Choosing between a self-contained, versatile solution like the on-chip Debug Core or other alternatives is an interesting research problem. For example, if an I/O interface is not required by the simulated system, the software verification can be offloaded to a host. In Figure 6.8 we can observe that the amount of verification data generated by the CPUs, which reaches a maximum peak of 1.5 GB/s. If the VC709 PCI Express interface was used, and with a measured, round-trip transfer delay of 25 μs, sending an instruction to a host for verification would be equivalent to 2000 Debug Core cycles at 80 MHz (without including the software verification time on the host).

Thus, off-chip, host-assisted verification is not worthwhile if advanced techniques are not implemented. For instance, verification instructions should be aggregated to compensate the latency, and the verification should be performed in blocks. The only problem with this approach is that slow instructions are not always consecutive, which makes it challenging to pack them. On the other hand, on-chip solutions have very high bandwidth and low latency.

Using embedded hard cores would not require such complex modifications. For instance, the Xilinx Zynq 7000 FPGA has a measured latency from CPU to BlockRAM between 40 and 74 ns [18], and its embedded ARM dual core runs at 720 MHz. In this platform between 3 and 6 Debug Core cycles are needed to obtain a verification request. However, a Zynq 7000 has 36% less logic resources than a Virtex 7, and the simulated multicore would

be smaller.

6.5.2 Use Cases

We want to illustrate the usefulness of the methods previously described with our own experience during the development of Bluebox. In particular, we detected some timing errors in a functionally correct hardware. One of those errors was related to the MIPS Load-Linked (LL) and Store Conditional (SC) instructions, which allow atomic writes to the memory and are essential to multicores. Under certain conditions, the LL address could be mapped to the same L2 cache line as the SC instruction, which caused an always-failing atomic update. We fixed that problem adding associative cache sets to the L2 cache.

Another problem was related to the CPU's pipeline. If an instruction fetch entered the pipeline right before updating a TLB entry, it would miss the new virtual address and potentially generate a false-positive TLB error. Such problem required extra control logic to avoid instruction memory requests during TLB updates. Those two cases exemplify the necessity of a cycle-accurate verification system that does not interfere with the simulation behavior.

6.6 Conclusions

Based on the lessons learned from our previous prototypes, in this chapter we presented debugging and verification techniques for host-based and FPGA-based multicore simulation. We showed how to implement non-obtrusive, full-system debugging and verification on the FPGA. We implemented those techniques in a reference architecture, Bluebox. Bluebox simulates up to 24 cores with cycle-accuracy at 120 MIPS, and perform full-system verification at 17 MIPS (including OS and I/O instructions).

7 Characterizing the High-level Descriptions

Existing commercial and academic design tools are good at synthesis, placement and routing of hardware designs for different implementation platforms. Usually such tools provide very detailed reports, but only on gross and macro-level resource consumption and performance metrics. Even if the detailed reports are analyzed, the resources are closely tied to the implemented circuit. Therefore, it is difficult to obtain a breakdown of area, delay and power metrics for the modules and sub-modules in the high-level design source.

Architecture designers require tools to automatically generate such a breakdown, as typical design iterations in the design cycle are always limited to specific blocks and modules. FPGAs are an important part of the high-performance ecosystem, as accelerators in heterogeneous architectures or high-throughput custom machines. In both cases they require deep design-space exploration [34] and architectural refinements [96] to appropriately map onto the target technology with the required constraints. There is an urgent need to improve and enhance the feedback from downstream synthesis tools to inform high-level design decisions.

In this chapter we present a methodology to analyze high-level Bluespec SystemVer-

ilog designs in order to generate module and sub-module-level metrics for area, delay and power.

7.1 Methodology

We divided our methodology in two steps. Firstly, the design is analyzed and annotated. Then, after passing through the synthesis flow, the resulting circuit is analyzed and compared to the original, abstract design. The user can apply architectural solutions to implementation problems, or even automatic architectural optimizations could be possible. And as the architectural and low-level information is known by the tool, automatic optimizations could be applied.

Using Figure 7.1 to illustrate the process (an Euclidean Greatest Common Divisor with 32-bit numbers), it proceeds as follows:

1. The user describes the hardware model in BSV (Figure 7.1a).

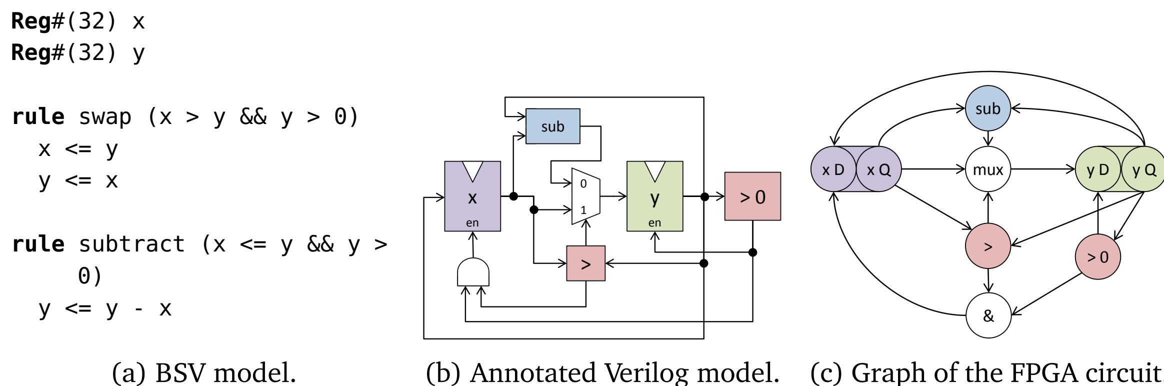
2. It is compiled into Verilog, so that it can be understood by the FPGA's tools. But during such conversion it loses critical architectural information (hierarchy of modules, rules, methods, *etc*.).

3. We automatically analyze the BSV description, identifying the architectural information. Based on this information, we annotate the flat Verilog description in an innocuous manner (Figure 7.1b).

4. After synthesizing the annotated Verilog model, these annotations remain. We automatically analyze the reports from the FPGA's tools (Figure 7.1c. We use this new information to enrich the original BSV description.

We extract three types of information from the FPGA reports: area (resource usage), performance (delays between gates), and power. For that purpose, we treat the FPGA circuit as a Directed Acyclic Graph (DAG) for computation purposes. The nodes of the graph are FPGA cells, and the edges are FPGA nets. The Verilog annotations. For the area estimation, we count the number of annotated FPGA cells.

For the performance estimation, we analyze the DAG (Figure 7.2a). The weight of the edges are the delays between cells. The combinatorial paths are graph's paths between



(a) BSV model. (b) Annotated Verilog model. (c) Graph of the FPGA circuit.

Figure 7.1: Example of our methodology applied to a GCD module, from the architecture to annotated circuits.

registers. The annotations referring to a certain high-level element (Figure 7.2b) allow us to identify the contribution of such element in the combinatorial paths. The algorithm to gauge the performance impact finds all the paths that cross such element (Figure 7.2c) and partitions them into connected sets (Figure 7.2d) to simplify the computation. Then, the algorithm reports if such high-level element affects any system delay (the slowest paths in the system, which affect the global performance, as in Figure 7.2e). It also estimates the inner delay of the high-level element, that is, the delay of a given BSV rule (Figure 7.2f).

It is important to note that breaking down combinatorial paths is important to identify what high-level components must be optimized. Our tool allows the designer to identify if a certain BSV rule is dominant in a combinatorial path.

Regarding the power consumption, our methodology also enables us to gauge the static power (proportional to the area) and the dynamic power (proportional to the area and the behavior). The static power of each high-level element is the total static power of its annotated FPGA cells.

The dynamic power of a high-level element is the total dynamic power of its annotated cells (the power required by its digital activity). We can estimate this activity profiling the high-level model. The dynamic activity of combinatorial paths is related to changes in their inputs (register's output). Profiling the high-level BSV model, we can estimate how each BSV rule may change the contents of the digital registers, generating dynamic activity on combinatorial paths.

Although this metrics are not totally accurate, it gives us a power score to identify the most power-consuming high-level elements.

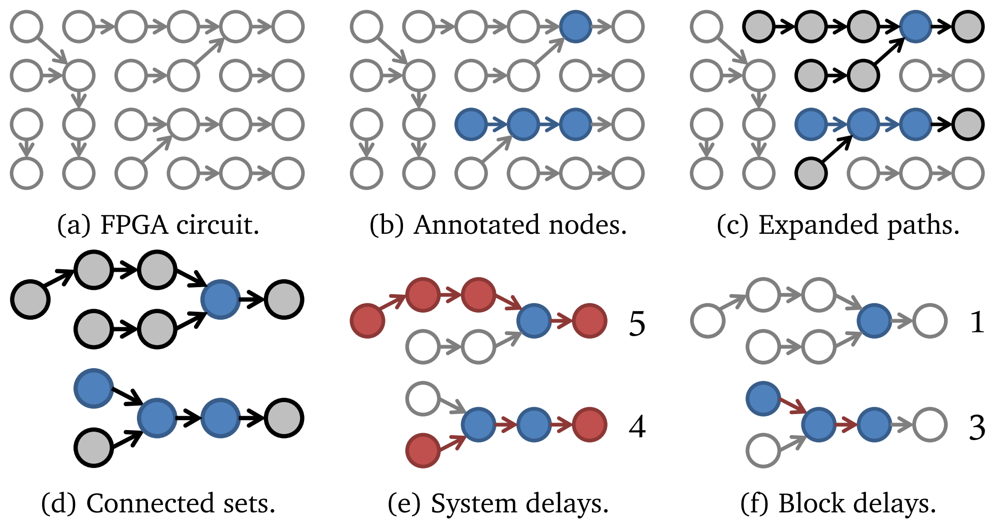


Figure 7.2: Algorithm used to calculate the block delay. Nodes are FPGA cells, arrows are combinational paths.

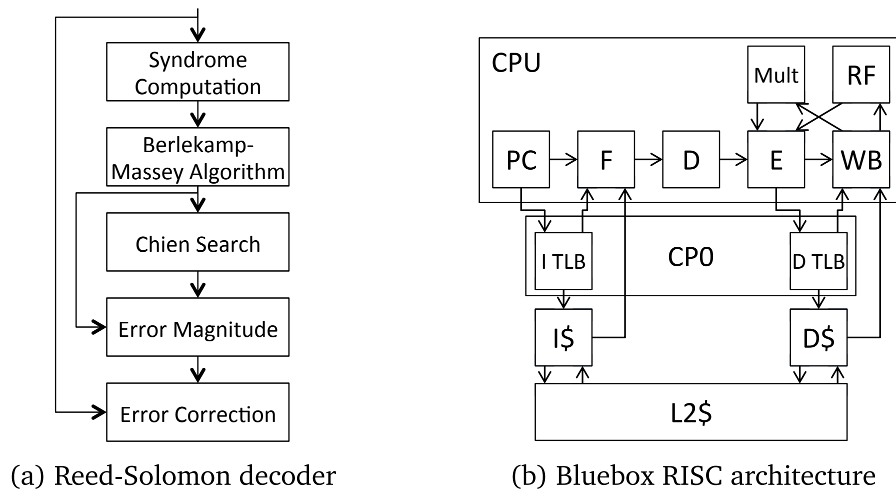
7.2 Results

In this section we describe the architectures chosen to demonstrate our methodology and the results obtained using our tools. The two test cases are:

Reed-Solomon: This design is a parameterized Reed-Solomon error correction decoder which meets the throughput requirement for use in an 802.16 wireless receiver. This architecture was developed by Agarwal *et al*. [20]. The decoding algorithm is composed of several steps, each of which is implemented as a separate module shown in Figure 7.3a. Dynamic activity, used for determining the power score metrics of the design, was generated using a testbench that feeds input data with errors at 50% of the maximal correctable rate.

Bluebox: This design is the 32-bit RISC microarchitecture that we introduced in Chapter 6. Figure 7.3b shows the main components consisting of a multiply unit, coprocessor 0 (implementing data and instruction TLBs), independent instruction/data L1 caches, and a unified, N-way L2 cache. This 5-stage processor can boot the GNU/Linux kernel. We used the dynamic activity generated during booting of Linux to generate the power score metrics of the Bluebox design.

We implemented these designs on three different FPGA devices: Spartan 6 XC6SLX45T-3FGG484, Virtex 5 XC5VLX155T-2FF1136 and Virtex 7 XC7VX485T-2FFG1761. We used



(a) Reed-Solomon decoder

(b) Bluebox RISC architecture

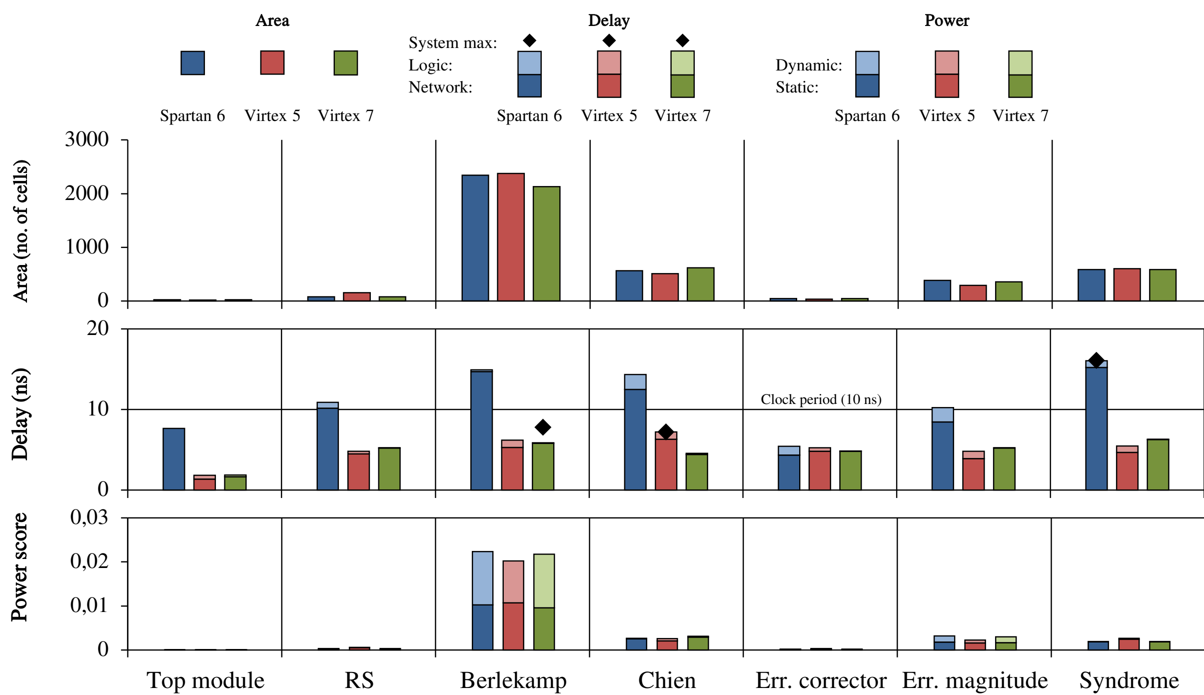
Figure 7.3: The two example models analyzed.

the Xilinx ISE and Vivado tools 14.4 to synthesize, place and route the Verilog hardware model. We also used these tools to export the final circuits. All the designs were targeted at 100 MHz on all the FPGAs.

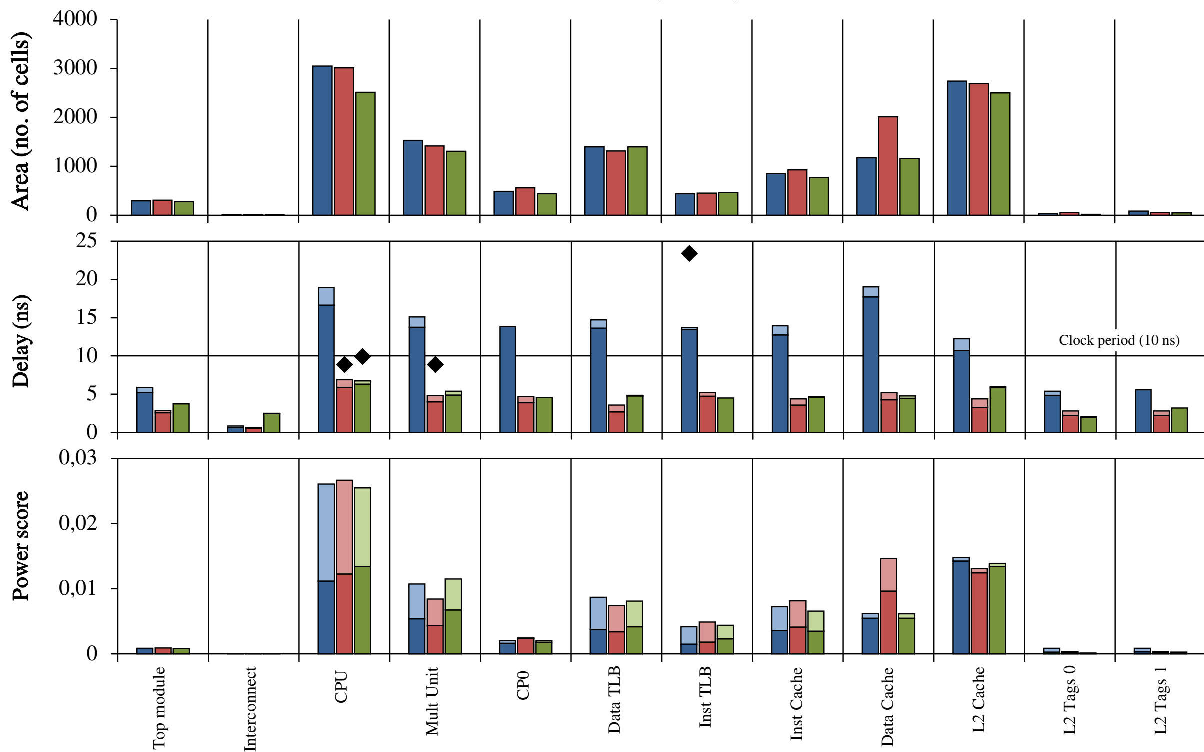
The results for both models are shown in Figure 7.4. The bar colors indicate the FPGA device: Spartan 6 – blue, Virtex 5 – red, and Virtex 7 – green. Three metrics are displayed:

- The area results show the number of cells per architectural module.
- The delay charts show the longest block delay. This metric measures the maximum contribution of each block to any path that crosses it. The darker components are the network part of the delay, and the lighter are the logic part. The diamond-shaped mark over the columns indicates the maximum delay of the system, and which block or blocks contribute to it. For both designs we can observe the significance of the network in the total delay.
- The power scores are also decomposed in two: static power score (darker shade) and dynamic power score (lighter shade).

Our framework produces results for every module and sub-module unit, but we grouped some results to simplify the charts.



(a) Reed-Solomon area, delay and power metrics.



(b) Bluebox area, delay and power metrics.

Figure 7.4: Area, delay and power metrics on three Xilinx FPGAs: Spartan 6 (blue), Virtex 5 (red) and Virtex 7 (green).

7.2.1 Reed-Solomon decoder

As shown in Figure 7.3a, Reed-Solomon decoder implementation consists of five main modules - Syndrome, Berlekamp, Chien, Error Magnitude and Error Corrector. Figure 7.4a shows the breakdown of metrics for these modules, as well as for the decoder module (RS) itself, which acts as a wrapper around these component modules, and for the top module which deals with Input-Output to memory. From an algorithmic perspective, Berlekamp step is the most computationally intensive part of the decoding process, and accordingly we see that this module has the maximum area in all three FPGA platforms. Error correction step involves the minimum computation as it simply removes the computed error values from the received data, thus contributing to minimal area. The other three component modules have similar moderate area usage.

For delay metrics, implementations on Virtex 5 and Virtex 7 platforms easily meet the required 100 MHz clock frequency with critical paths located mostly in Chien and Berlekamp modules respectively. However, the Spartan 6 implementation is unable to achieve this due to long computational operations in Berlekamp, Chien and Syndrome, with Syndrome contributing the critical path. The power metrics roughly track in similar ratios as with the area metrics. One important point to notice is that most of the dynamic power consumption in the decoder is contributed by the Berlekamp module. Dynamic power comprises up to 50% of Berlekamp's power consumption (in the case of Virtex 5) while other blocks' power consumption is mainly the static power of the FPGA resources used. This highlights the importance of this module for the decoder design, and suggests design refinement for reduced area as well as the use of power reduction techniques for reducing unnecessary dynamic activity (*e.g.* clock gating).

7.2.2 Bluebox

In Bluebox there are 12 modules. These modules correspond to some of the architectural units shown in Figure 7.3b. The results of the missing submodule architectural units, such as the pipeline stages, are included in their parent modules.

As with the previous case, the area and power metrics are very similar across all tested FPGA platforms, seen in Figure 7.4b. In general, area metrics seem to follow a descending trend from Spartan 6 to Virtex 7. This is a result of the FPGA architectures being different in these devices. For instance, Spartan 6 slices (a group of two LUTs) have one carry chain

output [131], which can be used to implement fast carry chain arithmetic operations. The Virtex 5 and Virtex 7 slices have two independent carry chains [132], which allow implementing more arithmetic operations with fewer LUTs. This is especially clear in the Virtex 7 area results.

The data cache requires more resources in Virtex 5 than in the other devices. The area report showed that the data cache module required 500 more registers in Virtex 5. We observed that while specifying the same architecture, the synthesis tool did not infer the data cache RAM unit correctly for Virtex 5, and implemented the cache memory using registers instead of using efficient on-chip BlockRAMs. We argue that such portability problems make necessary not only the development of platform-neutral synthesis tools and languages, but also cross-platform analysis tools like ours.

The delay results differ for Spartan 6, which was unable to achieve timing closure for a target clock period of 10 ns. The area of the design has an important impact on the performance of the design. High resource usage congests the network and makes it difficult for the router to achieve the timing goals. Bluebox occupies about 30% of the Spartan 6 device, much higher than the other two devices. In a congested device the network delays are high, even if it can fit the design. In addition, the logic delays of Spartan 6 cells are higher than the high-performance Virtex 5 and 7 LUTs. For instance, delay of a 6-input LUT in a Spartan 6 device can be ∼200 ps, whereas in Virtex 5 it is ∼80 ps and in Virtex 7 ∼40 ps. We can observe that the critical path of the Spartan 6 implementation is caused by the instruction TLB. In Virtex 5, we show two maximum delay marks, one over the CPU and another over the multiply unit. This means that the critical delay starts at the execution stage of the CPU and ends at the multiply unit. The delay report, along with the delay value, also includes the path that caused it and what architectural elements are contributing. In Virtex 7, the maximum delay is caused by the execution stage of the CPU.

For power consumption, it is seen that similar blocks dominate in all three platforms. These are L2 Cache, Execute block and the Multiplier. The dominance of the L2 Cache comes due to it being the largest block by far, thus having the largest static power dissipation. The computationally intensive Execute and Multiply blocks have a lot of logic and see a lot of dynamic activity. Beyond these three blocks we start seeing differences between the platforms. Virtex 5 has the Decode unit at relatively higher power consumption than even the Multiply unit. These differences arise due to the different availability of DSP arithmetic resources in the 3 FPGAs, different number of multiplexers generated for large data storage,

and different levels of power and area optimizations implemented in the platforms.

7.3 Automatic Architectural Optimization

The architecture defined by the user determines the performance, area and power consumption of the final FPGA circuit. The way that the architecture is synthesized, placed and routed can optimize these metrics, but they are always constrained by the architectural decisions. Thus, we believe that significant changes of these results can only be achieved through high-level, architectural decisions. For instance, the results in Figure 7.4b suggest several modifications at architectural level: reducing the number of entries of the L2 cache can improve area and power metrics. Splitting the Execute stage of the pipeline in two would break the combinational path crossing the data TLB and the data cache.

Currently these architectural optimizations are performed by the designer, under the guidance of the reports produced by the synthesis tools. Like the designer, our framework has knowledge about the architectural design and the synthesis reports. This knowledge enables the tool to implement technology-guided architectural changes.

The quality and impact evaluation methodology that we present in this work relies on two fundamental components. One is describing the hardware architecture using a high-level design language, such as Bluespec or another HLS language. The other is the methodology to project the technology problems to the architecture, as we described in the previous sections. But automatic architectural optimization requires additional components. The framework must distinguish the characteristics of each architectural unit, so that these parameters can be modified to meet the constraints imposed by the technology. For instance, the optimization tool should be able to modify the cache policies or the size of some units. The user should be able to put some constraints over those variations, informing what quality minimums must be preserved when modifying the architecture. Information about the target technology can complement these optimization inputs, allowing the tool to apply different strategies.

In future, using the methodology that we present in this work we will investigate the possibilities and challenges that automatic architectural optimization presents to hardware designers.

7.4 Conclusions

In this chapter we proposed a methodology to estimate the impact of the final FPGA technology over the original HLS model. In particular, we devised a methodology to automatically gauge the area, performance and power requirements of two high-level Bluespec SystemVerilog designs when implemented on three different FPGA platforms. Our methodology can identify the contribution of single high-level elements, which allows the user to identify the critical components of the architecture. In addition, the different effects of each FPGA device can be compared.

We believe that this automatic analysis opens the door to automatic architectural optimizations.

8 Related Work

During the development of this thesis the FPGA community has greatly advanced, with the emergence of new tools and device families. Thus, the prototypes and techniques developed within this thesis or by other researchers must be analyzed in their temporal context.

8.1 FPGA-based Multicore Models

In the last 15 years the FPGAs have grown enough to contain a full 32-bit processor. Since then, many academical proposals have appeared implementing FPGA-based CPU models. In Table 8.1 we show a selection of such projects. The first proposals were single-core soft-cores, mainly due to the reduced size of the FPGAs. Later on, multiprocessor systems were emulated or simulated, and even large architectures with multithreaded and out-of-order execution were implemented. It is also notable how the design languages have evolved from the legacy VHDL and Verilog to high-level languages like BSV or Chisel. Regarding the implemented ISAs, small RISC architectures dominate, again due to the limited capacity of FPGAs. For a taxonomy of FPGA-based architectural simulators, Tan et al. [117]

Table 8.1: FPGA-based Designs for Hardware Prototyping

Year	Name	Type	ISA	Model	Language
2000	OpenRISC [17]	CPU	OpenRISC	Softcore	Verilog
2001	Plasma [11]	CPU	MIPS	Softcore	VHDL
2002	Microblaze [7]	CPU	Microblaze	Softcore	*
2004	Nios II [9]	CPU	Nios	Softcore	*
2004	MiniMIPS [8]	CPU	MIPS	Softcore	VHDL
2004	LEON3 [5]	CPU	SPARC	Softcore	VHDL
2007	FAST [52]	CPU	PPC/x86	Accelerator	BSV
2007	RAMP Red (ATLAS) [127]	SMP	PPC	Hardcore	Verilog
2007	RAMP Blue [85]	SMP	Microblaze	Softcore	*
2009	Protoflex / Simflex [54]	SMP	SPARC	Accelerator	BSV
2010	RAMP Gold [116]	SMP	SPARC	Simulator	SystemVerilog
2010	BeehiveV5 [119]	SMP	Beehive	Softcore	VHDL
2011	HAsim [104]	CPU	MIPS/Alpha	Simulator	BSV
2011	**Beefarm**†	SMP	MIPS	Softcore	VHDL
2012	Arete [83]	SMP	PPC	Simulator	BSV
2012	BERI / CHERI [130]	CPU	MIPS	Softcore	BSV
2012	**TMbox**†	SMP	MIPS	Softcore	VHDL
2014	Sodor [48]	CPU	RISC-V	Softcore	Chisel
2014	Rocket [88]	SMP	RISC-V	Softcore	Chisel
2015	**Bluebox**†	SMP	MIPS	Softcore	BSV

*Source code not available.
†Proposals introduced in this thesis.

provide an excellent review of FPGA Architecture Model Execution (FAME) versus Software Architecture Model Execution (SAME).

In this thesis three multicore prototypes have been developed. Their main goals were:

1. To be all-modifiable by the researchers;
2. to be cycle accurate;
3. executing a well-known ISA;
4. and having a small footprint to fit as many cores as possible into the FPGA.

Currently there are no clear proposals that fulfill these characteristics. The commercial CPUs from Xilinx, Microblaze [7], and Altera, Nios [9], are closed-source and researchers cannot modify their microarchitecture. Despite their ISAs are similar to MIPS, their adoption as research arhictectures beyond the FPGA community is low. The alternative Open-

RISC [17], despite being open source, suffers from the same problem. Moreover, the Beehive [119] platform and its RC5 CPU were specifically designed for the BEE3 FPGA platform, and its exotic ISA complicates its adoption. We believe that introducing new ISAs hinders the portability of the implementations.

For this reason we chose to implement the MIPS I ISA. It is the RISC architecture par excellence, based on very simple instructions easy to understand and implement. For our first two implementations, Beefarm (Chapter 3) and TMbox (Chapter 4), we adapted the MIPS-like Plasma [11] core. Other proposals like miniMIPS [8] had similar characteristics: small footprint, open source and well-known ISA.

RISC-V [126], a free RISC ISA aimed to become the standard for computer architecture education and research, would be our choice today. It is no coincidence that David A. Patterson, co-author with John L. Hennessy of the educational DLX ISA used in the popular book *Computer Architecture: A Quantitative Approach* (which inspired the commercial architectures MIPS and SPARC), is involved into the design of RISC-V. Likewise, the Sodor [48] and Rocket [88] softcores, which implement the RISC-V ISA using the Chisel language [28], also fulfill our design goals. Unfortunately, these three proposals (RISC-V, Chisel and Sodor / Rocket) evolved in parallel to this thesis. This case is a clear example of redundant research due to lack of tools and standards, which hopefully RISC-V and Chisel will help to avoid in the future. Regarding Rocket, it is similar to Bluebox in most of its characteristics: developed in a high-level language, both implement a well-defined RISC ISA and can boot modern operating systems like the Linux kernel. In the case of Sodor, it is a single-core educational implementation with no virtual memory support. Regarding Rocket, while it has superior performance and more features, like full hardware floating point support, Bluebox implements unobtrusive system verification on chip.

The LEON3 [5] core is also open source, but the large OpenSPARC V8 architecture is prohibitive for multicore designs. In Chapter 3 we discuss the experience of upgrading and adapting an open-source softcore. The only previous study on adapting an available softcore onto a commercial FPGA platform has been the LEON3 core on the BEE2 platform, however the report by Wong [129] does not include many details or experience on the feasibility of this approach.

The Research Accelerator for Multiple Processors (RAMP) project was a joint effort from UC Berkeley, UT Austin, MIT, CMU, Stanford and companies like Xilinx and IBM. It is the most notable effort in the academy to implement FPGA-based multicore simulators and

emulators. It is a conglomerate of different projects and "RAMP flavors", from FPGA-based accelerators to full prototypes. Protoflex [54] uses a single FPGA fabric to multiplex multiple execution contexts, "transplanting" the functional simulation from a software application to the FPGA. The software application orchestrates the simulation, *i.e.* it simulates the timing model, while the FPGA accelerates the functional simulation. Conversely, in FAST [52] the functional model is simulated in the host while the timing model is accelerated in the FPGA, however it is a single-core simulator.

RAMP Gold [116] runs both the timing model and the functional model on the FPGA. The functional model is multithreaded, allowing to multiplex the execution of 64 cores using a single FPGA implementation. Compared to a software-only simulator, it can obtain up to 250x speedup for 64 cores running Splash-2 benchmarks. Protoflex requires vast resources and a software simulator, while RAMP Gold can be run on an inexpensive FPGA. While these approaches are performance-intensive, the current implementations are not fully cycle-accurate. Protoflex uses statistical sampling with FPGA acceleration. RAMP Gold assumes a magic crossbar to interconnect the cache hierarchies, although a realistic interconnect could be implemented.

In contrast to Protoflex and RAMP Gold, HAsim [104] and Arete [83] use *A-Ports* or *Latency-Insensitive Bounded Data-flow Networks*, which decouple FPGA cycles from model cycles while guaranteeing cycle-accuracy. HAsim uses time multiplexing, which seems to not scale well. Arete can model up to 8 PowerPC cores with 4 FPGAs at 55 MIPS. Both are implemented in the high-level language BSV.

Nevertheless, the reduced time-to-results offered by Protoflex or RAMP Gold (and potentially by HAsim and Arete) comes with a downside. Both implementations are not prototypes, but models run by a specific framework. As Tan *et al.* say [117]:

> It is both a strength and a weakness of Decoupled FAME [a full prototype in FPGA] that the full target RTL is modeled. The strength is that the model is guaranteed to be cycle accurate. Also, the same RTL design can be pushed through a VLSI flow to obtain reasonable area, power and timing numbers from actual chip layout. The weakness is that designing the full RTL for a system is labor-intensive and rerunning the tools is slow.

We differ from Tan *et al.* in considering that implementing the full RTL is labor-intensive. The methodology that we have presented in Chapter 6, in addition to the use of high-

level languages, reduces the development time and the testing time. Abstract modeling frameworks can also be labor-intensive: the custom modeling frameworks may require implementing from scratch the target model, which can be prohibitive for many researchers. In Beefarm, we have shown how to adapt an open-source softcore (Chapter 3). In addition, we have imported already-existing IP like multiply units or UART cores. Although DSLs could be developed for the simulation frameworks, allowing to easily specify functional and timing models and reducing the development time, still those models would be trapped in a domain-specific language.

In summary, if for software simulators there exist a trade-off between precision and performance, for FPGA-based models there may exist a trade-off between performance and portability.

The BERI/CHERI [130] processors have notable coincidences with Bluebox. These soft-cores, used to research capability-based memory protection, are also implemented in Bluespec SystemVerilog, supporting the 64-bit MIPS ISA and being able to boot an unmodified FreeBSD kernel. To our knowledge, these designs are not multicores but they could be another interesting research platform. Although, we have not synthesized the design and we do not know if 64-bit architecture requires significant more resources than the 32-bit Bluebox, which would limit the number of cores in a single FPGA. The source code is available online.

Regarding the reduction in resource utilization, it is worth investigating novel implementations that could reduce the size of the floating point logic, for example reusing FP units to also do integer calculations or combining them as in [122]. An example of shared FPUs between the CPU cores, as done in previous Section 3.5, is given in RAMP Blue [85].

Ring networks are suggested as a better architecture for shared memory multiprocessors in [32] and bi-directional bus has been used in [100], but as far as we know using backwards-propagating write-destructive invalidations as in TMbox is a novel approach. Beehive makes use of a uni-directional ring where messages are added to the head of a train with the locomotive at the end [38].

8.2 Transactional Memory on FPGA

Some work have been published in the context of studying Transactional Memory on FPGA prototypes. ATLAS is the first full-system prototype of an 8-way CMP system with PowerPC hard processor cores with TCC-like HTM support [127]. It features buffers for read/write sets and per-CPU caches that are augmented with transactional read-write bits. A ninth core runs Linux and serves OS requests from other cores. There also exist work on TM for embedded systems [81], and Bloom filter implementations to accelerate transactional reads and writes [38, 86].

Kachris and Kulkarni describe a basic TM implementation for embedded systems which can work without caches, using a central transactional controller on four Microblaze cores [81]. Pusceddu et al. present a single FPGA with support for Software Transactional Memory [107]. Ferri et al. propose an energy-efficient HTM on a cycle-accurate SW simulator, where transactions can overflow to a nearby victim cache [66].

Recent work that also utilizes MIPS soft cores focuses on the design of the conflict detection mechanism that uses Bloom filters for an FPGA-based HTM [86]. TMACC [47] accelerates software transactional memory for commodity cores. The conflict detection mechanism uses Bloom filters implemented on an FPGA. This accelerates the conflict detection of the STM. Moderate-length transactions benefit from the scheme whereas smaller transactions do not. The TM support for the Beehive FPGA multiprocessor stores transactional data in a direct-mapped data cache and overflows to a victim buffer [38]. Bloom filters are also used for the conflict detection. Damron et al. present Hybrid Transactional Memory (HyTM) [56]. An approach that uses best-effort HTM to accelerate transactional execution. Transactions are attempted in HTM mode and retried in software. The HTM results are based on simulation.

8.3 Profiling

The aforementioned TM systems lack a comprehensive support for the profiling of transactions. These works on TM profiling are presented next.

Chung et al. gather statistics on the behavior of TM programs on hardware and describe the common case behavior and performance pathologies [39].Ansari et al. present a framework to profile and metrics to understand Software TM applications [23]. Although

the presented metrics are transferable to our approach, the implementation of a software profiling framework in Java differs significantly from our hardware-based implementation.

The programming model of the underlying TMbox is comparable to the TCC model [74]. The monitoring techniques used in this work are in some parts comparable to the TAPE [50] system. Major differences include the use of multiple ring buses in the TMbox system, compared to a switched bus network with different timing characteristics and influences on HTM behavior. Further, the HW support for profiling with TAPE incurs an average slowdown of 0.27% and a maximum of 1.84%. Our system by design has zero HTM event overhead.

The tracing and profiling of non-TM programs has a long tradition as well as the search for the optimal profiling technique [31]. SW techniques for profiling, targetting low overhead, have been researched [67, 98], alongside of OS support [134], and HW support for profiling [59, 135]. Further, techniques to profile parallel programs using message passing communication have been developed [113]. In nearby research fields Faure et al. describe an event-based distributed monitoring system for software and hardware malfunction detection [64].

Up to now, a comprehensive profiling environment for hybrid TM systems has not been proposed. Previous approaches either lack the ability to profile TM programs or are designed for a specific hardware or software TM system. As a consequence, these approaches cannot capture the application's behavior comprehensively.

An application running on a Hybrid TM system may transition between HW and SW execution modes. These changes can only be tracked and understood by a dedicated solution, such as the framework presented here.

8.4 Comparing and Analyzing High-Level Languages and Tools

An exhaustive survey of HLS tools has been published recently by Daoud *et al.* [57], describing a plethora of HLS tools from the last 30 years, but it does not provide any empirical evaluation or comparison between them. Bacon *et al*.'s classification of HLS frameworks [30] is also very rich in technical details, but no direct comparison is made between different implementations.

Many studies focus on a single language in order to evaluate its benefits compared to

RTL. Bachrach *et al.* [28] present Chisel and compare it against Verilog to demonstrate that better performance with less lines of code can be obtained when implementing a RISC CPU. Cornu *et al.* [55] compare a genetic sequence algorithm on Impulse C and RTL implementations. They show a 4.2× speedup over hand-written RTL code, and state that HLS may provide higher performance than RTL because higher amounts of optimization can be obtained from high-level design, while low-level optimization is less efficient for the designer. Agarwal *et al.* [20] compare BSV against a C-based HLS tool, implementing a complex Reed-Solomon decoder. They show that the latter could have limited performance and high resource usage, while BSV can obtain better performance, requiring similar resources as a commercial IP. Meeus *et al.* [91] provide an overview of many HLS tools (mostly C, Matlab and BSV), and a qualitative comparison of the Sobel edge detector image processing algorithm. They mainly focus on tool comparison, on metrics like area, learning curve, and the ease of implementation, but not on performance.

Hammami *et al.* [73] present a comparison of 4 benchmark designs (two filters, FFT and ray casting), which are implemented on 3 C-based HLS (ImpulseC, Handel-C and SystemC). The paper does a detailed comparison of area, throughput and tool variation, and it also looks into how the tools deal with concurrency and pipeline extraction. Virginia *et al.* [124] include three different C-to-VHDL compilers (ROCCC, SPARK and their DWARV proposal) and compare a large subset of kernels using metrics such as throughput-per-slice, as well as readability, writing effort, supported ANSI-C subset, testability and hardware knowledge required. They conclude that the restrictions on the supported C subset directly influence the rewriting effort that is needed to make certain kernels compatible with the compilers.

Windh *et al.* [128] provide an extensive and detailed analysis of Xilinx Vivado HLS, Altera OpenCL, Bluespec SystemVerilog, LegUp and ROCCC. This survey, subsequent to our comparison, also compares quantitatively and qualitatively the languages and tools. While it does not provide analytical baselines like we did in our work, it is an excellent and up-to-date reference for comparing the state of the art in hardware description tools.

Hara *et al.* [75] propose a complete benchmark suite for HLS, with complex and diverse applications. However, it is specific to C-based languages. Mueller *et al.* [97] presented sorting networks and the same median operator with sliding window that we use in this work.

Regarding the characterization of hardware models, Yan *et al.* [133] presented an estimation model that provides an area-delay tradeoff for chosen applications and FPGA plat-

forms. However, it is aimed primarily at design partitioning of VLIW and Coarse-Grained reconfigurable architectures, while our work aims at modeling any custom hardware design. Modeling frameworks like McPAT [90] are able to estimate design metrics for a wide variety of processor configurations and implementation technologies, but are limited to pre-defined architectural parameters and can not be used on arbitrary designs. Amouri *et al*. [21] proposed a method to accurately measure and validate the leakage power distribution in FPGA chips using a thermal camera. These extremely accurate results can be used within our methodology for modeling architectural power consumption. Li *et al*. [89] proposed a fine-grained power model for interconnects and LUTs in an FPGA implementation targeting sub-100 nm technology. However, correlating high-level design blocks to the FPGA power estimates requires additional analysis to keep track of how resources are allocated in each synthesis, placement and routing process, as well as individual activity and trace generation for various component blocks. Our technique provides this analysis.

We have shown how to implement our methodology for rule-based languages. There exist similar approaches for HLS tools, which convert C code into hardware. In particular, Xilinx Vivado [12] estimates the performance of C constructs.

8.5 On-chip Verification and Debugging

DIVA [27] introduced dynamic verification, augmenting an out-of-order CPU with a small in-order verification CPU. Results were checked at commit time. This technique was targeted to silicon prototype verification. Argus [92] has microarchitectural verification units. Both are targeted to functional verification, and cycle-accuracy is not strictly preserved as in Bluebox. Other FPGA-based tools also have debugging support, however only HAsim and Arete are cycle-accurate like ours, but they do not preserve I/O delays. In the case of BERI, the BSV-C interface is also used, in their case for emulating I/O peripherals.

The inherent limitations of software simulators have been discussed in Section 2.2. In contrast, Bluebox can perform full-system, cycle-accurate simulations, and accuracy is also preserved with newly introduced changes.

Regarding the verification of `timesim` with `funcsim`, Nikhil et al. [99] call this method “tandem verification”. Tomić et al. [121] also implemented fast, functional modules to verify architectural elements in software simulators, such as hash maps to test data caches.

9 Conclusions

Since this thesis was started, FPGAs have evolved two generations, becoming almost four times bigger and 20% faster. Newer models embed hard CPUs, with fast PCI and Ethernet interfaces. Designers can describe their hardware using high-level languages such as Bluespec SystemVerilog and Chisel, and high-performance accelerators can be generated from C or OpenCL codes. At the same time, we are observing how the performance of homogeneous multicores cannot go around the power wall. It is an exciting time for the FPGA community, and we predict that FPGAs will play an important role in the computing ecosystem in the close future.

The research of novel multicore and heterogeneous architectures is necessary, and it requires powerful simulation tools. In Section 2.2 we have discussed limitations of software simulators. When precision is traded off for performance, as state-of-the-art software simulators do, the results obtained after architectural modifications may not be representative.

In this thesis we have implemented three FPGA-based multicore prototypes. We used these prototypes to study novel architectural extensions, such as Transactional Memory, and we investigated the advantages that reconfigurable devices offer to computer architects.

As we have shown in Chapter 3, FPGAs can overcome the limitations of software simulators. For high-precision simulations, FPGA architectural simulators perform several orders of magnitude faster than well-known software simulators. New-generation software simulators can simulate hundreds of cores at stunning frequencies, but a considerable price must be paid in terms of precision. **We also have shown how FPGA-based multicore architectures can be implemented using off-the-shelf open-source components.**

Novel architectural extensions can be studied using FPGAs. In Chapter 4 we have implemented the first 16-core FPGA-based Hybrid Transactional Memory system, TMbox. We replaced the shared bus from Beefarm with a ring bus with backwards destructive invalidations, which is better suited to the FPGA's characteristics. Using a TM interface similar to AMD ASF [53], TMbox supports the state-of-the-art TinySTM library [65].

Monitoring particular architectural events is a central task of simulators. Software simulators can profile the simulation using "magical" software routines, at the cost of decreasing the simulation performance. FPGAs can provide dedicated hardware for the same purpose. In Chapter 5 we profiled the transactional memory system of TMbox. **Our profiling mechanism can monitor most of the events at no cost, and record very detailed traces with an execution overhead within 14%.** On average, we incurred half the overhead of an STM-only software profiler. We have shown how to utlize our tool to observe bottlenecks and discover pathologies in well-known TM benchmarks.

The significant effort required to debug and verify FPGA models can discourage some computer architects from using them. In Chapter 6 we developed a development methodology that provides debugging and verification support during different stages of the design process. We developed Bluebox, a multicore prototype described in the high-level language Bluespec SystemVerilog. Bluebox supports the Linux Kernel, and its three different simulation modes (functional, timing and full-FPGA) can be debugged and verified. We implemented a 24-core version, achieving full-system verification at 17 MIPS.

The research developed during this thesis has been applied to other projects. For instance, the Bluebox processor has been used in database acceleration [25]. The HW/SW approach, similar to the Bluebox HW/SW verification system, allowed us to implement a gigabyte-level sorting engine with minimal development time.

In Appendix A we explored the different high-level languages and tools currently available. We compared them qualitatively and quantitatively using simple data-processing benchmarks. We found that many of them achieve the same performance as hand-written

Verilog hardware models. Our survey is the first one that includes representative candidates from the main classes of hardware description languages and tools. We also developed the methodology shown in Chapter 7 to gauge the impact of the FPGA technology over the abstract hardware description. We implemented a tool that applies this methodology automatically for Bluespec SystemVerilog hardware models.

Our conclusion after using several hardware design tools and languages is that the academy and the industry are on the right track, but still far from a typical software development experience. While C-to-gates translation can become popular to accelerate algorithms using FPGAs, cycle-accurate hardware design languages and tools are still necessary. In this sense, high-level languages like Bluespec SystemVerilog [1] or Chisel [28] can pioneer high-level hardware design. In order to reach broader audiences, native debugging, profiling and verification mechanisms are mandatory during all the design flow, especially when running on the FPGA (as we show in Chapter 6). Many tools and techniques that we developed in this thesis are natural mappings from the software development paradigm to the hardware design paradigm, which can explain why computer architects are still reluctant to use FPGAs.

The segmentation of the design stages, with different tools and languages in each one, complicates the development experience. We addressed this issue in Chapter 7, but there is a lot of room for further research. We also believe that the newer open-source tools, architectures and designs, like Chisel [28], LegUp [43], RISC-V [126], Sodor [48] or BERI [130], in addition to low-cost FPGA devices, can stimulate the adoption of FPGAs for research groups with limited resources. Usage of FPGAs, as well as research on FPGA design tools, can greatly benefit from open licenses, as we demonstrate in Chapter 3 building a multicore from open-source IP.

Finally, the FPGA technology still has difficulties hosting a popular out-of-order x86 CPU. Most FPGA-based simulators model RISC processors and a limited number of cores. The inner reconfigurable fabric has problems modeling some architectural components, like content-addressable memories or floating point units, but it is particularly good at RAMs. The addition of more specialized units may improve the capacity of FPGAs when modeling computer architectures. The I/O subsystems are also particularly difficult to setup, and with the emergence of heterogeneous CPU-FPGA chips this topic will become critical. In Chapters 5 and 6 we realized how difficult it can be to interconnect a FPGA with a host, and the practical limitations of off-FPGA computation (Section 6.5.1).

9.1 Research Opportunities

As we said, the high-level synthesis community is thriving. FPGA vendors want to bridge the gap between expert hardware developers and regular software programmers. High-level languages and tools may become a good opportunity for computer architects to develop their novel architectures in an efficient and precise way.

From our work, many research opportunities await. A promising research direction are interconnection networks for FPGA: long and wide buses are expensive to implement in FPGA's switched networks. From our results in Chapter 7, the underlying FPGA wires generate 70% of the total delay. New 14 nm FPGA generations have been released recently that address this problem using register re-timing techniques [16].

The new hardware description languages and tools increase the productivity, but they lack many desirable features. In this thesis we investigated a few, such as profiling, debugging and verification. Our automatic characterization of the technology impact, shown in Chapter 7, opens the door to automatic architectural optimizations: the high-level tools can modify architectural parameters in response to the physical constrains (area, performance and power). Currently, such architectural optimizations are manually applied by the designer, while the synthesis tools try to optimize the hardware at circuit level. Architectural changes, like changing the topology of the bus or varying the number of lines in a cache, requires high-level knowledge that our tool possesses.

From our survey on high-level hardware languages and tools, we strongly believe that a common methodology is needed to compare them. We expect more languages to be proposed in the next years, as the high-level synthesis field gains more users. For instance, a source-code sizing metric should be developed to compare languages belonging to paradigms so different, beyond just counting the lines of code.

10 Publications

The contents of this thesis have led to the following publications:

Nehir Sonmez, **Oriol Arcas,** Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Satnam Singh and Mateo Valero. **From Plasma to BeeFarm: Design experience of an FPGA-based multicore prototype.** In *Reconfigurable Computing: Architectures, Tools and Applications (ARC)*, pages 350–362, 2011.

Nehir Sonmez, **Oriol Arcas,** Otto Pflucker, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, Satnam Singh and Mateo Valero. **TMbox: A Flexible and Reconfigurable 16-Core Hybrid Transactional Memory System.** In *19th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 146–153, 2011.

Oriol Arcas, Philipp Kirchhofer, Nehir Sönmez, Martin Schindewolf, Osman S. Unsal, Wolfgang Karl and Adrián Cristal. **A low-overhead profiling and visualization framework for Hybrid Transactional Memory.** In *20th International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2012.

Oriol Arcas, Nehir Sonmez, Gokhan Sayilar, Satnam Singh, Osman S. Unsal, Adrian Cristal, Ibrahim Hur and Mateo Valero. **Resource-bounded multicore emulation using Beefarm.** In *Microprocessors and Microsystems (MICPRO)*, vol. 36, num. 8, pages 620–631, 2012.

Nehir Sonmez, **Oriol Arcas,** Osman S. Unsal, Adrián Cristal and Satnam Singh. **TM-box: A Flexible and Reconfigurable Hybrid Transactional Memory System.** In *Multicore Technology: Architecture, Reconfiguration, and Modeling*, 2013, CRC Press.

Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sonmez, Mohsen Ghasempour, Adria Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal and Mikel Lujan. **An Empirical Evaluation of High-Level Synthesis Languages and Tools for Database Acceleration.** In *24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2014.

Oriol Arcas-Abella, Adrián Cristal and Osman S. Unsal. **High-Level Debugging and Verification for FPGA-Based Multicore Architectures.** In *23rd International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015.

The following publications are related but not included in this thesis:

Myron King, Asif Khan, Abhinav Agarwal, **Oriol Arcas** and Arvind. **Generating infrastructure for FPGA-accelerated applications.** In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2013.

Oriol Arcas-Abella, Adrià Armejach, Timothy Hayes, Görker Alp Malazgirt, Oscar Palomar, Behzad Salami and Nehir Sönmez. **Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory**, In *Computing in Science Engineering (CiSE)*, volume 18, issue 1, pages 80–87, 2016.

High-level Hardware Design Languages

In recent years, several new approaches have been proposed to lower the complexity of hardware development and to make it more attractive to software developers. The most prominent approach is through the use of High-Level Synthesis (HLS) languages and tools, which translate software languages, often C and it variants, into low-level Register-Transfer Level (RTL) descriptions [30, 55, 57]. HLS languages are gaining popularity as they have the potential of "opening FPGAs to the masses". Consequently, FPGA/EDA vendors are increasingly adopting and supporting them. The ease of programmability, performance, resource usage and efficiency can vary from one HLS technology to another, and usually there is a tradeoff between these characteristics.

A.1 Evaluation of HLS Languages and Tools

HLS tools fill the gap between low-level RTL and high-level algorithms, raising the level of abstraction and effectively hiding the low-level details from the designer. Each proposal stresses a different characteristic (eg., productivity, learning curve, versatility, performance,

etc.) resulting in various tradeoffs among them.

To classify HLS, we adopted the taxonomy of Bacon *et al*. [30] which defines HLS as any language or tool that includes a high-level feature which RTL does not have. Their classification has five categories: HDL-like languages, CUDA/OpenCL frameworks, C-based frameworks, high-level language-based frameworks and model-based frameworks. In the following subsections, we describe the first four groups and select one language from each for our evaluation. Model-based frameworks are not included in our study because of their specificity to particular domains (eg., DSP modeling).

- **HDL-like HLS: Bluespec SystemVerilog.** The first category comprises of modern HDL-like languages, which borrow features from other programming languages to create a new one. This is the case with SystemVerilog [13] and the rule-based Bluespec SystemVerilog (BSV) [1]. We have chose BSV because it is a radically different approach to hardware description, based on guarded rules and syntax inherited from SystemVerilog. In this paradigm, hardware designs are described as data-flow networks of guarded atomic rules. Actions in rules are executed in a transactional manner: state changes happen all-at-once when the rule is fired. Parallelism is achieved through concurrent execution of non-conflicting rules.

- **CUDA/OpenCL HLS: Altera OpenCL.** Open Computing Language (OpenCL) is an open industry standard for programming heterogeneous computing platforms (a host CPU, GPU, DSP or FPGA). It is based on standard ANSI C (C99) with extensions to create task-level and data-level parallelism. Altera's SDK for OpenCL (AOCL) [15] exploits parallelism in data-independent threads, or "work items" in OpenCL speak. AOCL translates the software description into a pipelined hardware circuit, where each stage of the pipeline executes a different thread. This approach is less versatile than general-purpose C compilers, but can be more efficient for data-flow and streaming applications, which is one of the drawbacks of other HLS.

- **C-based HLS: LegUp.** The other categories in Bacon *et al.*'s classification are frameworks that target subsets, or extensions of already-existing software languages. In most of the cases, the designers adopt a popular language to smoothen the learning curve. The most prominent group is based on C: LegUp [43], ROCCC [71] and Impulse C (specialized in stream programming) [4] support C subsets. xPilot [51] (now

Xilinx Vivado [12]) and Calypto Catapult C [2] also accept C++ and SystemC. We included LegUp in our evaluation for two reasons. First, (i) it is open-source, and (ii) we believe that the synthesis mechanism is similar to those used in other C-based HLS tools. LegUp compiles LLVM [87] code into Verilog. The C functions are converted into Finite State Machines (FSM). Local and global variables are stored in shared memories (Block RAMs or external DDR), and are accessed by the FSMs.

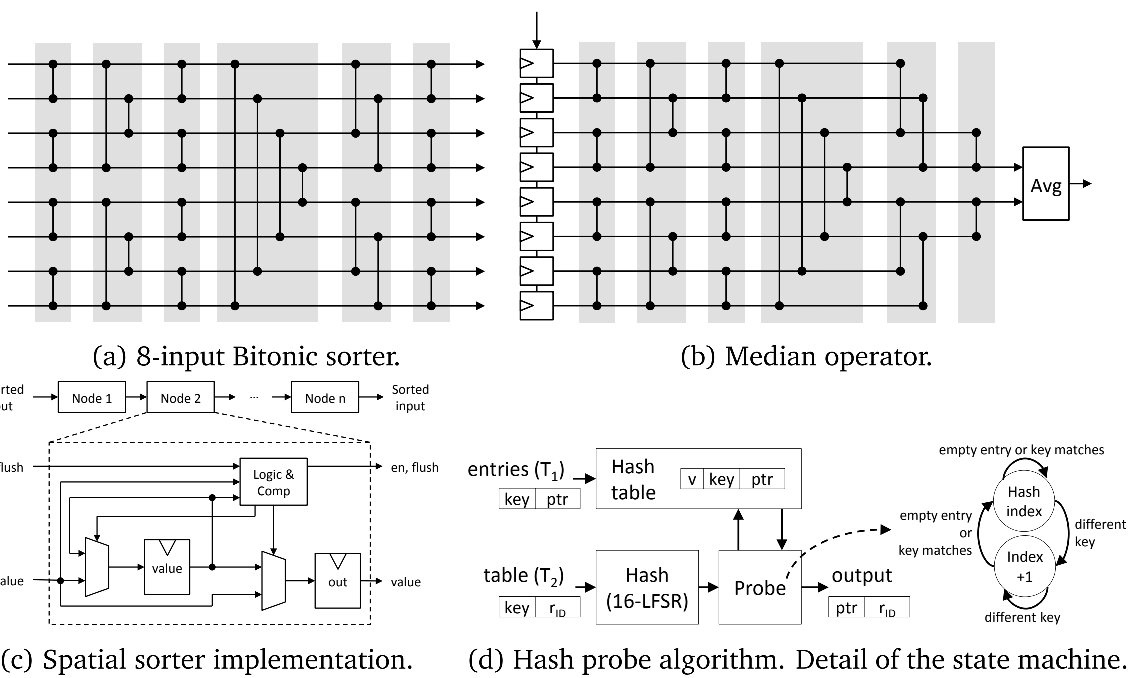
- **High-level Language Frameworks: Chisel.** This last group includes frameworks that translate high-level languages (other than C) into hardware. Some examples are the event-driven Esterel [35], Kiwi [114] (C#) and Lime [26] (Java). Chisel [28] is based on the functional language Scala (which is based on Java), and therefore targeted to high-productivity. The hardware designs are pure Scala applications and can be synthesized into C++ simulators or Verilog RTL descriptions. The framework is made with Scala, and provides basic data types, structures and language constructs. However, the interconnection of the elements (ie., modules, wires, registers) is done in an RTL-like manner.

A.1.1 Studied Database Algorithms

Our comparisons focus on three common and time-consuming database operations: sorting, aggregation and joins. The inherent parallelism of sorting makes it suitable for efficient FPGA implementations [68]. We chose two sorting algorithms that are suitable for hardware implementation, namely bitonic and spatial sorters. For aggregation, we implemented the median operator with a sliding-window, also used in [97]. Finally, for table joins, we included a hash probe algorithm to accelerate hash join operations [72].

- **Bitonic Sorter** A bitonic sorter is a type of a sorting network [33] particularly efficient in hardware, consisting of multiple levels of compare-and-exchange units. The sorting is performed in $O(\log^2 n)$ time complexity, and requires $O(n \log^2 n)$ comparators that can be pipelined, increasing the frequency and the throughput. Figure A.1a shows an 8-input bitonic sorter.

- **Spatial Sorter** The spatial sorter [102] is composed of an array of sorting registers, each of which does a compare and swap operation [84]. As seen in Figure A.1c, the



(a) 8-input Bitonic sorter.

(b) Median operator.

(c) Spatial sorter implementation.

(d) Hash probe algorithm. Detail of the state machine.

Figure A.1: The Bitonic sort, spatial sort and median operator algorithms. In sorting networks the horizontal arrows are input values, and the vertical lines are the Knuth compare-and-exchange operator (↕). Sorting stages are shaded in gray.

main ingredients of a sorter node are a comparator, two registers and two multiplexers. New elements are inserted at the beginning of the sorter array. Elements are sorted as they are inserted into the array of registers, as in the bubble sort algorithm. The spatial sorter has a worst case time cost of $2n$ cycles to sort an input set of size n.

- **Median Operator** Aggregation operations, which reduce a stream of data to a single result, are ubiquitous in database queries. We implemented the median operator as in Mueller *et al.* [97], expressed by the CQL [24] query Q_1:

  ```
  SELECT median(v) FROM S [ Rows 8 ].
  ```
  (Q_1)

  This expression describes a median operator over a sliding window of 8 elements. Figure A.1b depicts our implementation using a bitonic sorter. For every cycle and for each new element inserted into the window, a new median value is calculated. Since only the median values of the sorter are used, some comparators and registers are optimized away by the synthesis tool. We used a 16-input window for evaluating this algorithm.

- **Hash Probe** Database operations that join two tables using a common column are frequent and time-consuming. We considered the θ-join, defined in Equation A.1:

  $$T_1 \bowtie_\theta T_2 = \sigma_\theta(T_1 \times T_2) \tag{A.1}$$

  That is, selecting some rows, based on the join function θ, among the product of tables T_1 and S_2. The hash join algorithm performs θ-joins using a hash-table to match coincident rows. In contrast to previous algorithms that require more computational power, hash probe hardware essentially consists of a memory that is randomly accessed and some control logic. In Figure A.1d we show the main elements of the hardware model. The `Probe` state machine implements the hash table probe algorithm. The Linear Feedback Shift Register (LFSR) implements the hash function.

```
module mkSorter (SortBox#(n,t));
  ...
  rule do_bitonic_sort;
    Integer stage = 0;
    for (Integer ai=2; ai <= valueOf(n); ai=ai*2)
      for (Integer bi=ai; bi > 1; bi=bi/2) begin
        let x = regs[stage];
        let y = (stage+1 < valueOf(num_stages)) ?
          regs[stage+1] : destW;
        for (Integer i=0; i < valueOf(n); i=i+bi)
          for (Integer j=0; j < bi/2; j=j+1)
              begin
            let k2 = bi!=ai?(i+j+bi/2):(i+bi-j-1)
                ;
            Bool swap = compareData(x[i+j], x[k2
                ]);
            y[i+j] <= (swap ? x[k2] : x[i+j]);
            y[k2] <= (swap ? x[i+j] : x[k2]);
          end
        stage = stage + 1;
      end
  endrule
  ...
endmodule
```

(a) Example Bluespec SystemVerilog code

```
__kernel
__attribute__((
  reqd_work_group_size(
      WORKGROUP_SIZE,1,1)))
void bitonic(
    __global int2* restrict
        input_data,
    __global int2* restrict
        output_data) {
  const unsigned group =
      get_group_id(0)*16;
  int2 temp[16];
  #pragma unroll
  for (unsigned i = 0; i < 16; ++i)
    temp[i] = input_data[group+i];
  compAndSwap(&temp[0], &temp[1]);
  compAndSwap(&temp[2], &temp[3]);
  compAndSwap(&temp[4], &temp[5]);
  compAndSwap(&temp[6], &temp[7]);
  ...
  #pragma unroll
  for (unsigned i = 0; i < 16; ++i)
    output_data[group+i] = temp[i];
}
```

(b) Example Altera OpenCL code

```
volatile int input_data[16];
volatile int output_data[16];
int main() {
  int i, temp[16];
  for (i = 0; i < 16; ++i)
    temp[i] = input_data[i];
  compAndSwap(&temp[0], &temp[1]);
  compAndSwap(&temp[2], &temp[3]);
  compAndSwap(&temp[4], &temp[5]);
  compAndSwap(&temp[6], &temp[7]);
  ...
  for (i = 0; i < 16; ++i)
    output_data[i] = temp[i];
  return 0; }
```

(c) Example LegUp code

```
class Bitonic(val n:Int,k:Int,d:Int)
    extends Module {
  val io = new BitonicIfc(n, k, d)
  ...
  val subu = Vec.fill(2) {
    Module(new Bitonic(n/2, k, d)).io }
  for (t <- 0 until n/2 ) {
    subu(0).in(t) := io.in(t)
    inputs0(t) := subu(0).out(t)
    subu(1).in(t) := io.in(t+n/2)
    inputs0(t+n/2) := subu(1).out(t)
  }
  ...
}
```

(d) Example Chisel code

Figure A.2: Bitonic sorter code snippets for each HLS framework

A.1.2 Programming Experience Evaluation

In this section, we share our experience implementing the algorithms using the different languages. For sorting algorithms, the input data is 32-bit key – 32-bit value pairs (ie., 16 × 64 = 1,024 bits in size). The median operator uses 32-bit key inputs. The hash probe uses 16-bit key – 32-bit value pairs, and a hash table with 64K × 64-bit buckets (512 KB) and a load factor of 0.6. The size of (T_1) is 400 MB and the size of (T_2) is 600 MB. None of the tools required more than 30 seconds to generate the Verilog descriptions.

Using **Bluespec SystemVerilog** all the algorithms were substantially easy to describe using the data-flow, rule-based paradigm of BSV. The advanced evaluation system of BSV handles well the recursive definition of the bitonic sorter. The BSV models are fully parametrized, and can generate hardware models for arbitrary input sizes, as well as to perform a different number of comparisons at each pipeline stage. In Figure A.2a, we show one of the submodules of the bitonic sorter in BSV.

Implementing hash probe in BSV proved to be more difficult than in Verilog. Obtaining an optimal scheduling in BSV can sometimes require some extra effort from the designer, due to the strict sequentially consistent paradigm of the language, which might not be obvious to designers with an RTL background.

Altera OpenCL is strongly influenced by GPU programming. However, our implementation of the bitonic sorter is slightly different from the typical GPU implementation to enable us to fully exploit parallelism on the FPGA. Figure A.2b shows a snippet of the OpenCL code. The code to be accelerated ("kernel" in OpenCL speak) contains the hard-coded compare-and-exchange operations of the 16-input bitonic sorter. We had to hard-code the 16-input version of the bitonic sorter, as we did for LegUp.

The spatial sorter was more difficult to implement than the bitonic sorter, as in LegUp, even with the multithreading capabilities of OpenCL. The calculation of the median operator is performed by work-groups of threads. As work-groups are mutually independent, each one needs to have its own input buffers to avoid conflicts. Similarly to LegUp, the hash probe algorithm was ported easily to OpenCL. But in this case, we used a sequential implementation and relied on AOCL to pipeline the design and to optimize memory accesses.

We found that AOCL has a steeper learning curve than LegUp. The programmer needs considerable knowledge about underlying OpenCL concepts such as work-groups and work-

items. On the other hand, the efficiency obtained can be much higher for some classes of database problems, as we describe in the next section.

The main advantage of pure-C HLS tools like **LegUp** is that the learning curve is very smooth, as most programmers are already familiar with C. On average, the algorithms required very few lines of code. Moreover, most already-existing algorithms written in C can be ported to an FPGA almost seamlessly. However, obtaining an efficient implementation requires experience using the tools, as well as prior knowledge on the target technology to later optimize it. In addition, a substantial rewriting of the initial code may be required.

In Figure A.2c, we show the most interesting fragments of our C version of the bitonic sorter. The LegUp code, which can be compiled as regular C code and executed on any processor, is completely straightforward to a C programmer. The resulting binary is functionally equivalent to the hardware generated by LegUp. This feature is interesting for fast simulation and debugging, as well as for the migration of software kernels to an FPGA. On the other hand, the C language has a limited evaluation system, the C preprocessor, based on conditional directives and macros.

We had to learn how to correctly describe the hardware using C. For instance, input and output buffers (`input_data` and `output_data`) are marked as `volatile` to indicate the compiler not to try optimizing away memory accesses. Instead of operating over data in the `main` function the data is copied into a temporal buffer. This will make the compiler read all the input data and optimize the sorting over the temporal buffer. The spatial sorter, very natural to express in any HDL, is not well suited for C-based HLS. We found that the multi-threaded nature of the algorithm, where independent sorting units exchange data, is very difficult to express in C. The implementation of the median operator was more straightforward. Storing only the median value (the average of the two middle values of the sorted set) allows the LegUp compiler to optimize away the extra computation and the LLVM compiler to trim the unused data paths.

We saw that hash probe is a very suitable algorithm to be expressed in C-based HLS, yielding the best overall performance results for LegUp, as shown in the next section.

The programming model of **Chisel** is very similar to RTL languages. Hardware units are defined and interconnected in an imperative way. However, for evaluating the code, all the high-level constructs of the Scala language are available. In Figure A.2d we show an example code snippet of our bitonic implementation in Chisel.

As in the case of BSV, almost all the examples were very easy to express as parameter-

Table A.1: Analytical models for the resource usage of the studied algorithms.

	Stages (S)		Registers		Logic	
Algorithm	Value	Complexity	Value	Complexity	Value	Complexity
bitonic	$\sum_{i=1}^{\log n} i$	$O(log^2 n)$	nSw	$O(nw\log^2 n)$	$\frac{nSw}{2}$	$O(nw\log^2 n)$
spatial	n	$O(n)$	$2nw$	$O(nw)$	$2nw$	$O(nw)$
median	$\sum_{i=1}^{\log n} i$	$O(\log^2 n)$	†	$O(nk\log^2 n)$	‡	$O(nk\log^2 n)$

Note: k = key bits, v = value bits, $w = k + v$, S = stages. Median only uses k.
† $nSk - 2k\sum_{i=0}^{\log n-2}\left(\frac{n}{2} - 2^i\right)$
‡ $\frac{nSk}{2} - k\sum_{i=0}^{\log n-3}\left(\frac{n}{4} - 2^i\right)$

izable implementations, and the high-level constructs produced very succinct code. One special case was the Chisel implementation of hash probe, which was easier than BSV (as we did not run into control flow issues) and Verilog (as we were able to use higher level constructs). Being a subset of Scala, Chisel is a good language for developers with some background on Java. It supports advanced features like polymorphism and parametrized modules. However, we believe that some features would improve productivity even further. For instance, BSV-like implicit condition handling would simplify the control logic of designs.

A.2 Empirical Evaluation and Comparison

In this section, we compare the empirical results against analytical models and hand-written Verilog models. We targeted the implemented algorithms to an Altera Stratix V 5SGXA7 FPGA, with the same synthesis options. We used the Quartus "Early Timing and Area Estimates" flow to compile the designs.

The performance of an HLS can be seen as the combination of the algorithmic performance of the hardware designed and the I/O performance. In this section, we concentrate on algorithmic performance, however the I/O performance also has to be taken into account when choosing an HLS. The biggest advantage of using Altera OpenCL is its ability to automatically generate I/O interfaces with the host. Bluespec provides libraries for interfaces such as Ethernet or PCIe. LegUp allows interfacing a soft CPU core (hybrid flow) and Chisel does not have support for I/O interfacing yet.

Table A.2: Evaluation of the four algorithms on different programming environments.

	Registers		Logic			Memory	F_{max}	Throughput		
Implementation	FF	% incr.	LUT	% incr.	ALM*	(Kbits)	(MHz)	MB/s	%	LoC
bitonic analytical	10,240	0.00%	2,560	0.00%						
bitonic Verilog	10,250	0.10%	2,640	3.13%	7,210.8	0.00	311.43	38,016.36	100.00%	134
bitonic BSV	10,250	0.10%	2,640	3.13%	6,997.5	0.00	313.97	38,326.42	100.82%	57
bitonic Chisel	10,272	0.31%	2,649	3.48%	5,571.0	0.00	314.96	38,447.27	101.13%	114
bitonic LegUp	4,210		5,180		3,973.4	0.00	211.86	1,034.47	2.72%	101
bitonic OpenCL	38,455		5,221		15,842.6	361.38	307.12	1,317.21	3.46%	140
STL sort C++ (host CPU)							2,300.00	570.42	1.50%	3
spatial analytical	2,048	0.00%	640	0.00%						
spatial Verilog	2,081	1.61%	641	0.16%	1,359.5	0.00	341.30	1,301.96	100.00%	98
spatial BSV	2,112	3.13%	1701	165.75%	1,081.0	0.00	343.52	1,310.42	100.65%	181
spatial Chisel	2,112	3.13%	720	12.50%	1,053.0	0.00	345.30	1,317.21	101.17%	87
spatial LegUp	1,115		823		612.5	0.50	309.12	3.13	0.24%	28
spatial OpenCL	26,059		14,667		15,072.3	877.84	236.85	660.53	50.73%	66
median analytical	4,544	0.00%	2,240	0.00%						
median Verilog	4,555	0.24%	2,352	5.00%	4,009.5	0.00	302.76	1,154.94	100.00%	159
median BSV	4,554	0.22%	6,168	175.36%	3,359.5	0.00	334.67	1,276.66	110.54%	70
median Chisel	4,577	0.73%	2,351	4.96%	3,321.5	0.00	338.98	1,293.11	111.96%	132
median LegUp	10,449		5,262		3,781.4	0.47	174.98	34.25	2.97%	97
median OpenCL	19,366		7,590		9,309.5	190.06	312.10	920.60	79.71%	84
median C++ (host CPU)							2,300.00	836.10	72.39%	6
hash probe Verilog	995		174		327.5	3,136.00	174.06	995.98	100.00%	66
hash probe BSV	1,150		166		365.5	3,136.00	181.46	1,038.32	104.25%	124
hash probe Chisel	1,020		179		333.5	4,096.00	171.59	981.85	98.58%	83
hash probe LegUp	345		397		262.5	4,096.00	302.85	61.90	6.22%	50
hash probe OpenCL	35,536		21,854		19,175.6	3,876.08	270.19	2.14	0.21%	59
hash probe C++ (host CPU)							2,300.00	433.32	43.51%	18

*ALM (Adaptive Logic Module): Altera's basic cell blocks, with an 8-input fracturable LUT and four 1-bit registers.

A.2.1 Analytical Analysis

To have a concrete baseline for comparison, we first performed an analytical analysis of the expected FPGA resource utilization for determining the number of registers and LUTs needed to implement our algorithms.

In an efficient implementation, the manually-optimized Verilog model should directly match the analytical model. The resource usage of the BSV and Chisel designs is also expected to be very close to the Verilog and analytical baselines. The LegUp and the AOCL models follow a different computational paradigm that would be very difficult to model, so we will not compare them against the analytical models. Similarly, no analytical model was devised for the resource usage of the hash probe design, which is mostly made up of control logic.

In Table A.1, we show the number and complexity of the stages, registers and combinational logic of each algorithm. The parameter n represents the size of the sorting set. The parameters k and v represent the key and value sizes in bits (and $w = k + v$). It can be seen that the Bitonic sorter needs $O(nw\log^2 n)$ registers and $O(nw\log^2 n)$ combinational LUTs (for the comparators). In the case of the median operator, the costs are $O(nk\log^2 n)$ because only the keys are sorted, and only the middle numbers of the sorting are used, allowing to optimize away some registers and comparators, as shown in Figure A.1b. The spatial sorter requires 2 registers in each sorting unit: one for the current value and one to store the outgoing one. The hardware model requires $O(nw)$ registers and $O(nw)$ combinational LUTs.

A.2.2 Experimental Results

In Table A.2 we show the empirical results for all the languages and tools evaluated. For each algorithm, we show the resource usage, maximum frequency, estimated throughput (MB/s) and lines of code needed (LoC)[1]. The LegUp resource usage was obtained by stripping out the additional infrastructure generated by the tools, and only leaving out the algorithmic kernel. In the AOCL implementations complex I/O optimizations are implemented, like input and output buffering, resulting in an intensive resource usage. Additionally,

[1] Although we consider that LoC cannot be used as the primary criteria, and more advanced metrics should be used such as function points. The creation of a *hardware description sizing metric* adapted to HDLs and HLS is one of the possible future research directions.

RAM blocks were extensively used by the OpenCL implementations, and minimally used by LegUp, as well. The block memory usage for hash probe shows that for some implementations it was possible to optimize away the unused bits (only 49 bits of the 64-bit hash table buckets were used).

In terms of performance, our results demonstrate that HLS tools are indeed able to offer competitive performance to fine-tuned Verilog/VHDL, effectively accelerating database operations. We also implemented these algorithms in software, using the efficient Standard Template Library (STL) implementations of C++, running on a host machine with Intel Xeon E5-2630 CPU at 2.3 GHz. We made sure that the benchmarks used already-cached data, attempting to mimic ideal conditions in software. The results show that the computational power of most of the HLS implementations is significantly higher than a software version. Furthermore, we used 16-input designs in this work, while FPGAs allow bigger circuits to be implemented, and higher performance gains can be expected (along with considerable power savings compared to a high-performance CPU).

The throughput results can be thought of as being interfaced through BRAMs. For AOCL, we derived the unconstrained throughput using the de-rate factor that the compiler applies when the maximum bandwidth is exceeded. The high bandwidth achieved by bitonic could be provided by DRAM, as available in the Maxeler MAX3 platform (38.4GB/sec [6]). For other I/O interfaces that provide less throughput, some options are: (i) to generate a smaller/slower circuit that requires less bandwidth and saves unused computational power, (ii) to employ caching structures similar to ROCCC's smart buffers [71], or (iii) to use another algorithm, such as the spatial sorter instead of bitonic.

The bitonic sorter is easy to express in all languages, and delivers a speedup between 1.81x and 67.4x over the software version. The C-based HLS tools exhibit diverse behaviors. AOCL outperforms the software in most of the cases and achieves competitive results in the spatial sorter and the median operator (> 50% of the hand-coded Verilog throughput). In addition, AOCL can automatically replicate the computation units, resulting in linear speedup in our experiments, at the cost of more resource and bandwidth (in Table A.2 we only used 1 computational unit). LegUp has moderate throughput results, but requires very few resources.

For the database algorithms that we have studied, BSV and Chisel produced code that is on par with hand-optimized Verilog, yielding the best throughput-per-area ratios. Curiously, the compiler might even be able to use some extra logic and to optimize further in certain

cases, as in the median operator implementations in BSV and Chisel. In the case of BSV, the high LUT usage for the spatial sorter and the median operator (caused by the rule-based programming model) is absorbed by the ALMs and doesn't result in a higher area requirement. The register usage increase for Verilog, BSV and Chisel over the analytical models were mostly caused by control logic.

A.2.3 Discussion

With our experience and experiments, we can conclude that there is no obvious election when choosing an HLS, but an orthogonal set of characteristics that must be considered. Bluespec SystemVerilog has a steep learning curve, but provided good performance results in our experiments. Among other benefits, it guarantees tight control over the cycle accurate model and automatic flow control validation, and supports high-level, parameterizable constructs. It is a good choice to implement system-level HW models, especially for designers with a background in RTL design or in Haskell.

The Altera's OpenCL framework required succinct implementations while delivering good throughput rates on some algorithms, and was easy to use because the OpenCL standard is based on C. However, we found it difficult to parameterize the designs and the resource footprint was the highest (as a result of including a complex I/O infrastructure automatically). It is a natural choice for data-flow algorithms, especially when accelerating already-existing C kernels, but some expertise in OpenCL is required.

The LegUp HLS tool accepts generic C code. We consider it the easiest to learn and to use. Its performance results were lower than the other tools considered (which may improve in future versions). Thus, it can be considered for designers from any field with little experience using HDLs, and it allows to easily implement algorithms with lower bandwidth requirements, predominance of flow control structures, or to accelerate already-existing C code.

Finally, Chisel delivered good performance results from relatively succinct implementations. As in BSV, it retains cycle accuracy and the high-level Scala constructs allow to parameterize the code. However, it lacks a proper standard type library, and the hardware scheduling/interconnection must be done manually as in RTL-like languages. It is a good choice for designers with a strong RTL background (and some Scala knowledge), enabling them to implement system-level designs.

References

[1] Bluespec Inc. http://www.bluespec.com. Cited on page: 13, 83, 119, 124

[2] Calypto Catapult C. http://calypto.com/en/products/catapult/overview. Cited on page: 125

[3] Chipscope Pro Tool User Guide. Xilinx, Inc. (User Guide). Cited on page: 47

[4] Impulse Accelerated Technologies, Inc. http://www.impulseaccelerated.com/. Cited on page: 124

[5] LEON3 Multiprocessing CPU Core. Aeroflex Gaisler (Product Sheet). Cited on page: 108, 109

[6] Maxeler MaxCloud. Maxeler Technologies (http://www.maxeler.com/products/maxcloud/). Cited on page: 134

[7] Microblaze processor reference guide. Xilinx, Inc. (Reference Guide). Cited on page: 10, 108

[8] miniMIPS. http://opencores.org/project,minimips. Cited on page: 108, 109

[9] Nios II Gen2 Processor Reference Guide. Altera Corporation (Reference Guide). Cited on page: 10, 108

[10] OpenCores Website. www.opencores.org. Cited on page: 26, 48

[11] Plasma soft core. http://opencores.org/project,plasma. Cited on page: 26, 53, 108, 109

[12] Xilinx Vivado ESL Design. Xilinx, Inc. (http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/). Cited on page: 115, 125

[13] IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE STD 1800*, pages 1–1285, 2009. Cited on page: 124

[14] The Benefits of Multiple CPU Cores in Mobile Devices. NVIDIA (white paper), 2010. Cited on page: 19

[15] *Altera SDK for OpenCL Programming Guide Version 13.0 SP1*, 2013. Cited on page: 124

[16] Understanding How the New HyperFlex Architecture Enables Next-Generation High-Performance Systems. Altera (White Paper), 2015. Cited on page: 120

[17] OpenRISC Open-Source Architecture. Opencores (website), January 2015. Cited on page: 108, 109

[18] CPU latency to access an AXI Slave using Master AXI GP. Xilinx Inc. (Answer Record), January 2015. Cited on page: 94

[19] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. DOI: 10.1145/1133981.1133985. Cited on page: 40

[20] A. Agarwal, Man Cheuk Ng, and Arvind. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed-Solomon Decoder. *IEEE Embedded Systems Letters*, 2(3):72–76, Sept 2010. ISSN 1943-0663. DOI: 10.1109/LES.2010.2055231. Cited on page: 100, 114

[21] A. Amouri, H. Amrouch, T. Ebi, J. Henkel, and M. Tahoori. Accurate Thermal-Profile Estimation and Validation for FPGA-Mapped Circuits. In *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–60, April 2013. DOI: 10.1109/FCCM.2013.48. Cited on page: 115

[22] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. DOI: 10.1109/HPCA.2005.41. Cited on page: 30

[23] M. Ansari, K. Jarvis, C. Kotselidis, M. Luján, C. Kirkham, and I. Watson. Profiling Transactional Memory Applications. In *17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 11–20, Feb 2009. DOI: 10.1109/PDP.2009.35. Cited on page: 78, 112

[24] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2): 121–142, June 2006. ISSN 1066-8888. DOI: 10.1007/s00778-004-0147-z. Cited on page: 127

[25] O. Arcas-Abella, A. Armejach, T. Hayes, G.A. Malazgirt, O. Palomar, B. Salami, and N. Sonmez. Hardware Acceleration for Query Processing: Leveraging FPGAs, CPUs, and Memory. *Computing in Science Engineering*, 18(1):80–87, Jan 2016. ISSN 1521-9615. DOI: 10.1109/MCSE.2016.16. Cited on page: 118

[26] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and Synthesizable Language for Heterogeneous Architectures. *SIGPLAN Not.*, 45(10):89–108, October 2010. ISSN 0362-1340. DOI: 10.1145/1932682.1869469. Cited on page: 125

[27] T.M. Austin. DIVA: a Reliable Substrate for Deep Submicron Microarchitecture Design. In *32nd Annual International Symposium on Microarchitecture, 1999. MICRO-32. Proceedings*, pages 196–207, 1999. DOI: 10.1109/MICRO.1999.809458. Cited on page: 115

[28] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM. ISBN 978-

1-4503-1199-1. DOI: 10.1145/2228360.2228584. Cited on page: 109, 114, 119, 125

[29] John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978. ISSN 0001-0782. DOI: 10.1145/359576.359579. Cited on page: 3, 18

[30] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA Programming for the Masses. *Commun. ACM*, 56(4):56–63, April 2013. ISSN 0001-0782. DOI: 10.1145/2436256.2436271. Cited on page: 113, 123, 124

[31] Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994. ISSN 0164-0925. DOI: 10.1145/183432.183527. Cited on page: 113

[32] L.A. Barroso and Michel Dubois. Performance Evaluation of the Slotted Ring Multiprocessor. *IEEE Transactions on Computers*, 44(7):878–890, Jul 1995. ISSN 0018-9340. DOI: 10.1109/12.392846. Cited on page: 111

[33] K. E. Batcher. Sorting Networks and Their Applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM. DOI: 10.1145/1468075.1468121. Cited on page: 125

[34] J. Benson, R. Cofell, C. Frericks, Chen-Han Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam. Design, integration and implementation of the DySER hardware accelerator into OpenSPARC. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2012. DOI: 10.1109/HPCA.2012.6168949. Cited on page: 97

[35] Gérard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19 (2):87 – 152, 1992. ISSN 0167-6423. Cited on page: 125

[36] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sar-

dashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2): 1–7, August 2011. ISSN 0163-5964. DOI: 10.1145/2024716.2024718. Cited on page: 6, 20, 79, 80

[37] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006. ISSN 0272-1732. DOI: 10.1109/MM.2006.82. Cited on page: 42

[38] Andrew Birrell, Tom Rodeheffer, and Chuck Thacker. Hardware Transactional Memory for Beehive. Microsoft Research Silicon Valley (Technical Report), 2010. Cited on page: 63, 111, 112

[39] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 81–91, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. DOI: 10.1145/1250662.1250674. Cited on page: 77, 112

[40] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, Winter 2007. ISSN 1098-4232. DOI: 10.1109/N-SSC.2007.4785534. Cited on page: 18

[41] A. Brant and G.G.F. Lemieux. ZUMA: An Open FPGA Overlay Architecture. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96, April 2012. DOI: 10.1109/FCCM.2012.25. Cited on page: 8

[42] Jean-Louis Brelet. XAPP201: An Overview of Multiple CAM Designs in Virtex Family Devices. Xilinx (Application Note), 1999. Cited on page: 33

[43] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11,

pages 33–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. DOI: 10.1145/1950413.1950423. Cited on page: 83, 119, 124

[44] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 69–80, June 2007. DOI: 10.1145/1250662.1250673. Cited on page: 30

[45] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. DOI: 10.1145/2063384.2063454. Cited on page: 6, 8, 20, 22, 79, 80

[46] William Carter, Khue Duong, Ross H Freeman, H Hsieh, Jason Y Ja, John E Mahoney, Luan T Ngo, and Shelly L Sze. A User Programmable Reconfigurable Gate Array. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, 1986. Cited on page: 2

[47] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS XVI, pages 27–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0266-1. DOI: 10.1145/1950365.1950372. Cited on page: 52, 112

[48] Christopher Celio. Sodor processor collection. Software repository. Cited on page: 6, 108, 109, 119

[49] H. Chafi, J. Casper, B.D. Carlstrom, A. McDonald, C.C. Minh, Woongki Baek, C. Kozyrakis, and K. Olukotun. A Scalable, Non-blocking Approach to Transactional Memory. In *IEEE 13th International Symposium on High Performance Computer Architecture (HPCA 2007)*, pages 97–108, Feb 2007. DOI: 10.1109/HPCA.2007.346189. Cited on page: 52, 57

[50] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, JaeWoong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. TAPE: A Transactional Application Profiling Environment. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 199–208, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. DOI: 10.1145/1088149.1088176. Cited on page: 64, 113

[51] Deming Chen, Jason Cong, Yiping Fan, Guoling Han, Wei Jiang, and Zhiru Zhang. xPilot: A platform-Based Behavioral Synthesis System. *SRC TechCon*, 5, 2005. Cited on page: 124

[52] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3047-8. DOI: 10.1109/MICRO.2007.36. Cited on page: 21, 51, 108, 110

[53] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of AMD's Advanced Synchronization Facility Within a Complete Transactional Memory Stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. DOI: 10.1145/1755913.1755918. Cited on page: 11, 55, 66, 118

[54] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):15:1–15:32, June 2009. ISSN 1936-7406. DOI: 10.1145/1534916.1534925. Cited on page: 6, 21, 39, 51, 108, 110

[55] Alexandre Cornu, Steven Derrien, and Dominique Lavenier. HLS Tools for FPGA: Faster Development with Better Performance. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger Woods, and Tarek El-Ghazawi, editors, *Reconfigurable Com-*

puting: Architectures, Tools and Applications, volume 6578 of *Lecture Notes in Computer Science*, pages 67–78. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19474-0. Cited on page: 114, 123

[56] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. DOI: 10.1145/1168857.1168900. Cited on page: 52, 63, 112

[57] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A Survey of High Level Synthesis Languages, Tools, and Compilers for Reconfigurable High Performance Computing. In Jerzy Swiątek, Adam Grzech, Paweł Swiątek, and Jakub M. Tomczak, editors, *Advances in Systems Science*, volume 240 of *Advances in Intelligent Systems and Computing*, pages 483–492. Springer International Publishing, 2014. ISBN 978-3-319-01856-0. Cited on page: 113, 123

[58] J. Davis, C. Thacker, and C. Chang. BEE3: Revitalizing computer architecture research. *Microsoft Research*, 2009. Cited on page: 25

[59] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-level Profiling on Out-of-order Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-7977-8. Cited on page: 113

[60] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, Ernest Bassous, and Andre R. Leblanc. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions. *Proceedings of the IEEE*, 87(4):668–678, April 1999. ISSN 0018-9219. DOI: 10.1109/JPROC.1999.752522. Cited on page: 17

[61] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. *Parallel Processing Letters*, 21(02):173–193, 2011. Cited on page: 18

[62] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the End of Multicore Scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011. Cited on page: 19

[63] Gerald Estrin. Organization of Computer Systems: The Fixed Plus Variable Structure Computer. In *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM. DOI: 10.1145/1460361.1460365. Cited on page: 2

[64] E. Faure, M. Benabdenbi, and F. Pêcheux. Distributed Online Software Monitoring of Manycore Architectures. In *2010 IEEE 16th International On-Line Testing Symposium (IOLTS)*, pages 56–61, July 2010. DOI: 10.1109/IOLTS.2010.5560232. Cited on page: 113

[65] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-based Software Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. DOI: 10.1145/1345206.1345241. Cited on page: 11, 40, 41, 52, 54, 66, 118

[66] Cesare Ferri, Samantha Wood, Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. Embedded-TM: Energy and Complexity-effective Hardware Transactional Memory for Embedded Multicore Systems. *Journal of Parallel and Distributed Computing*, 70(10):1042 – 1052, 2010. ISSN 0743-7315. DOI: http://dx.doi.org/10.1016/j.jpdc.2010.02.003. Transactional Memory. Cited on page: 112

[67] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead Call Path Profiling of Unmodified, Optimized Code. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. DOI: 10.1145/1088149.1088161. Cited on page: 113

[68] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of*

Data, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM. ISBN 1-59593-434-0. DOI: 10.1145/1142473.1142511. Cited on page: 125

[69] Peter Greenhalgh. big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7. ARM (white paper), 2011. Cited on page: 19

[70] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, March 2006. ISSN 0272-1732. DOI: 10.1109/MM.2006.41. Cited on page: 19

[71] Zhi Guo, Betul Buyukkurt, Walid Najjar, and Kees Vissers. Optimized Generation of Data-Path from C Codes for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 112–117, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2288-2. DOI: 10.1109/DATE.2005.234. Cited on page: 124, 134

[72] R.J. Halstead, B. Sukhwani, Hong Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating Join Operation for Relational Databases with FPGAs. In *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–20, April 2013. DOI: 10.1109/FCCM.2013.17. Cited on page: 125

[73] Omar Hammami, Zhoukun Wang, Virginie Fresse, and Dominique Houzet. A Case Study: Quantitative Evaluation of C-Based High-Level Synthesis Systems. *EURASIP Journal on Embedded Systems*, 2008(1):685128, 2008. ISSN 1687-3963. DOI: 10.1155/2008/685128. Cited on page: 114

[74] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, June 2004. DOI: 10.1145/1028176.1006711. Cited on page: 25, 30, 40, 113

[75] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17:242–254, 2009. DOI: 10.2197/ipsjjip.17.242. Cited on page: 114

[76] John Hauser. SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html. Cited on page: 45

[77] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. DOI: 10.1145/165123.165164. Cited on page: 92

[78] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993. ISSN 0163-5964. DOI: 10.1145/173682.165164. Cited on page: 18, 28, 30

[79] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003. DOI: 10.1145/872035.872048. Cited on page: 29

[80] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–11, Dec 2010. DOI: 10.1109/IISWC.2010.5648812. Cited on page: 60, 71

[81] C. Kachris and C. Kulkarni. Configurable Transactional Memory. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pages 65–72, April 2007. DOI: 10.1109/FCCM.2007.41. Cited on page: 63, 112

[82] Gokcen Kestor, Vasileios Karakostas, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, and Mateo Valero. RMS-TM: A Comprehensive Benchmark Suite for Transactional Memory Systems. In *Proceedings of the 2Nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, pages 335–346, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0519-8. DOI: 10.1145/1958746.1958795. Cited on page: 43

[83] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind. Fast and cycle-accurate modeling of a multicore processor. In *2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 178–187, April 2012. Cited on page: 6, 21, 108, 110

[84] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0554-9. DOI: 10.1145/1950413.1950427. Cited on page: 125

[85] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System in FPGAs. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 54–61, Aug 2007. DOI: 10.1109/FPL.2007.4380625. Cited on page: 51, 108, 111

[86] Martin Labrecque, Mark C. Jeffrey, and J. Gregory Steffan. Application-specific Signatures for Transactional Memory in Soft Processors. *ACM Trans. Reconfigurable Technol. Syst.*, 4(3):21:1–21:14, August 2011. ISSN 1936-7406. DOI: 10.1145/2000832.2000833. Cited on page: 63, 112

[87] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86, March 2004. DOI: 10.1109/CGO.2004.1281665. Cited on page: 125

[88] Yunsup Lee, A. Waterman, R. Avizienis, H. Cook, Chen Sun, V. Stojanovic, and K. Asanovic. A 45nm 1.3GHz 16.7 Double-Precision GFLOPS/W RISC-V Processor with Vector Accelerators. In *40th European Solid State Circuits Conference (ESSCIRC)*, pages 199–202, Sept 2014. DOI: 10.1109/ESSCIRC.2014.6942056. Cited on page: 108, 109

[89] Fei Li, Yan Lin, Lei He, Deming Chen, and J. Cong. Power Modeling and Characteristics of Field Programmable Gate Arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(11):1712–1724, Nov 2005. ISSN 0278-0070. DOI: 10.1109/TCAD.2005.852293. Cited on page: 115

[90] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*, pages 469–480, Dec 2009. Cited on page: 115

[91] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An Overview of Today's High-level Synthesis Tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012. ISSN 0929-5585. DOI: 10.1007/s10617-012-9096-8. Cited on page: 114

[92] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 210–222, Dec 2007. DOI: 10.1109/MICRO.2007.18. Cited on page: 115

[93] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IEEE International Symposium on Workload Characterization (IISWC 2008)*, pages 35–46, Sept 2008. DOI: 10.1109/IISWC.2008.4636089. Cited on page: 43, 60, 62, 65, 71, 75

[94] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998. ISSN 0018-9219. DOI: 10.1109/JPROC.1998.658762. Cited on page: 17

[95] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture*, pages 254–265, February 2006. Cited on page: 25, 30, 40, 52

[96] R. Moussalli, W. Najjar, Xi Luo, and A. Khan. A High Throughput No-Stall Golomb-Rice Hardware Decoder. In *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 65–72, April 2013. DOI: 10.1109/FCCM.2013.9. Cited on page: 97

[97] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, August 2009. ISSN 2150-8097. DOI: 10.14778/1687627.1687730. Cited on page: 114, 125, 127

[98] Sagnik Nandy, Xiaofeng Gao, and Jeanne Ferrante. TFP: Time-Sensitive, Flow-Specific Profiling at Runtime. 2958:32–47, 2004. DOI: 10.1007/978-3-540-24644-2_3. Cited on page: 113

[99] Rishiyur S. Nikhil and Darius Rad. RISC-V at Bluespec. In *1st RISC-V Workshop*, 2015. Cited on page: 83, 115

[100] Hitoshi Oi. *Design and Performance Evaluation of the Bidirectional Ring-based Shared Memory Multiprocessor*. PhD thesis, Tampa, FL, USA, 1999. Cited on page: 111

[101] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3: 26–29, September 2005. ISSN 1542-7730. DOI: 10.1145/1095408.1095418. Cited on page: 18

[102] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, Randy Allmon, Rachid Rayess, Stephen Maresh, and Joel Emer. Triggered Instructions: A Control Paradigm for Spatially-programmed Architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. DOI: 10.1145/2485922.2485935. Cited on page: 125

[103] A. Patel, F. Afram, Shunfei Chen, and K. Ghose. MARSS: A full system simulator for multicore x86 CPUs. In *48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1050–1055, June 2011. Cited on page: 6, 20, 79, 80

[104] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer. HAsim: FPGA-based High-detail Multicore Simulation Using Time-division Multiplexing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 406–417, Feb 2011. DOI: 10.1109/HPCA.2011.5749747. Cited on page: 6, 21, 44, 51, 108, 110

[105] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, Osman Unsal, Adrián Cristal, Tim Harris, and Mateo Valero. The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-core Environment. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 67–78, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-077-7. DOI: 10.1145/1366230.1366241. Cited on page: 78

[106] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *WoTUG-18*, volume 44, 1995. Cited on page: 12, 65, 71, 85

[107] M. Pusceddu, S. Ceccolini, G. Palermo, D. Sciuto, and A. Tumeo. A Compact Transactional Memory Multiprocessor System on FPGA. In *2010 International Conference on Field Programmable Logic and Applications (FPL)*, pages 578–581, Aug 2010. DOI: 10.1109/FPL.2010.113. Cited on page: 112

[108] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. DOI: 10.1145/2485922.2485963. Cited on page: 6, 8, 20, 22, 79, 80

[109] Mike Santarini. Zynq-7000 EPP Sets Stage for New Era of Innovations. *Xcell journal*, 75(2):8–13, 2011. Cited on page: 19

[110] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, Ronak Singhal, Jim Brayton, Sebastian Steibl, and Hong Wang. Intel Nehalem Processor Core Made FPGA Synthesizable. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 3–12, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-911-4. DOI: 10.1145/1723112.1723116. Cited on page: 26

[111] Lesley Shannon, Veronica Cojocaru, Cong Nguyen Dao, and Philip H.W. Leong. Technology Scaling in FPGAs: Trends in Applications and Architectures. In *2015 IEEE 23th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, May 2015. Cited on page: 3

[112] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995. DOI: 10.1145/224964.224987. Cited on page: 29

[113] Sameer Shende, AllenD. Malony, Alan Morris, and Felix Wolf. Performance Profiling Overhead Compensation for MPI Programs. 3666:359–367, 2005. DOI: 10.1007/11557265_47. Cited on page: 113

[114] S. Singh and D. Greaves. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *16th International Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pages 3–12, April 2008. DOI: 10.1109/FCCM.2008.46. Cited on page: 125

[115] J.E. Stone, D. Gohara, and Guochun Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3): 66–73, May 2010. ISSN 1521-9615. DOI: 10.1109/MCSE.2010.69. Cited on page: 18

[116] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović. RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 463–468, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0002-5. DOI: 10.1145/1837274.1837390. Cited on page: 6, 21, 39, 44, 51, 108, 110

[117] Zhangxi Tan, Andrew Waterman, Henry Cook, Sarah Bird, Krste Asanović, and David Patterson. A Case for FAME: FPGA Architecture Model Execution. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 290–301, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. DOI: 10.1145/1815961.1815999. Cited on page: 5, 6, 8, 21, 107, 110

[118] Chuck Thacker. A DDR2 controller for BEE3. Microsoft Research, 2009. Cited on page: 27, 31, 53

[119] Chuck Thacker. Beehive: A many-core computer for FPGAs (v5). Microsoft Research Silicon Valley (Technical Report), 2010. Cited on page: 108, 109

[120] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. EazyHTM: Eager-lazy Hardware Transactional Memory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, New York, NY, USA, 2009.

ACM. ISBN 978-1-60558-798-1. DOI: 10.1145/1669112.1669132. Cited on page: 25

[121] Saša Tomić, Adrián Cristal, Osman Unsal, and Mateo Valero. Using Dynamic Runtime Testing for Rapid Development of Architectural Simulators. *International Journal of Parallel Programming*, 42(1):119–139, 2014. ISSN 0885-7458. DOI: 10.1007/s10766-012-0208-7. Cited on page: 83, 115

[122] S. Tsen, S. Gonzalez-Navarro, M. Schulte, B. Hickmann, and K. Compton. A Combined Decimal and Binary Floating-Point Multiplier. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2009)*, pages 8–15, July 2009. DOI: 10.1109/ASAP.2009.28. Cited on page: 111

[123] M. Vijayaraghavan and Arvind. Bounded Dataflow Networks and Latency-Insensitive circuits. In *7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design (MEMOCODE '09)*, pages 171–180, July 2009. DOI: 10.1109/MEMCOD.2009.5185393. Cited on page: 38

[124] Arcilio J Virginia, Yana D Yankova, and Koen LM Bertels. An empirical comparison of ANSI-C to VHDL compilers: SPARK, ROCCC and DWARV. In *Annual Workshop on Circuits, Systems and Signal Processing ProRISC*, pages 388–394, 2007. Cited on page: 114

[125] Perry H. Wang, Jamison D. Collins, Christopher T. Weaver, Blliappa Kuttanna, Shahram Salamian, Gautham N. Chinya, Ethan Schuchman, Oliver Schilling, Thorsten Doil, Sebastian Steibl, and Hong Wang. Intel Atom Processor Core Made FPGA-synthesizable. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '09, pages 209–218, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. DOI: 10.1145/1508128.1508160. Cited on page: 26

[126] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.0. EECS Department, University of California (Technical Report UCB/EECS-2014-52), 2012. Cited on page: 109, 119

[127] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. A Practical FPGA-based Framework for Novel CMP Research. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 116–125, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-600-4. DOI: 10.1145/1216919.1216936. Cited on page: 51, 63, 108, 112

[128] S. Windh, Xiaoyin Ma, R.J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W.A. Najjar. High-Level Language Tools for Reconfigurable Computing. *Proceedings of the IEEE*, 103(3):390–408, March 2015. ISSN 0018-9219. Cited on page: 114

[129] Timothy Wong. LEON3 port for BEE2 and ASIC implementation. http://cadlab.cs.ucla.edu/software_release/bee2leon3port/. Cited on page: 109

[130] J. Woodruff, A.T. Markettos, and S.W. Moore. A 64-bit MIPS Processor Running FreeBSD on a Portable FPGA Tablet. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, page 1, Sept 2013. DOI: 10.1109/FPL.2013.6645630. Cited on page: 6, 108, 111, 119

[131] Xilinx Inc. Spartan-6 FPGA Configurable Logic Block User Guide. http://www.xilinx.com/support/documentation/user_guides/ug384.pdf, . Cited on page: 104

[132] Xilinx Inc. Virtex-5 FPGA User Guide. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, . Cited on page: 104

[133] Leipo Yan, Thambipillai Srikanthan, and Niu Gang. Area and Delay Estimation for FPGA Implementation of Coarse-grained Reconfigurable Architectures. *SIGPLAN Not.*, 41(7):182–188, June 2006. ISSN 0362-1340. DOI: 10.1145/1159974.1134677. Cited on page: 114

[134] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 15–26, New York, NY, USA, 1997. ACM. ISBN 0-89791-916-5. DOI: 10.1145/268998.266640. Cited on page: 113

[135] C.B. Zilles and G.S. Sohi. A programmable Co-Processor for Profiling. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 241–252, 2001. DOI: 10.1109/HPCA.2001.903267. Cited on page: 113

[136] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrián Cristal, Ibrahim Hur, and Mateo Valero. Discovering and Understanding Performance Bottlenecks in Transactional Applications. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 285–294, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. DOI: 10.1145/1854273.1854311. Cited on page: 64