A multicore emulator with a profiling infrastructure for Transactional Memory on FPGA

Nehir Sönmez

Department of Computer Architecture

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Doctor of Philosophy in Computer Architecture

July 15, 2012

To my mother Ülkü, my father Mustafa, my brother Rüzgar (who will do much better than me) and to all my friends that I also had to leave thousands of kilometers away.

Acknowledgements

First of all, I have to thank my compatriot Don Oriol "Beekeeper" Arcas, without whom I would still be halfway. I am also very grateful to my advisors Dr. Osman S. Ünsal and Dr. Adrián Cristal for their endless help and support and giving me the freedom to let me work on what I was interested in. I am also thankful to Gökhan Sayilar and Philipp Kirchhofer for their valuable reinforcements as loyal Beekeepers. Throughout this research, I have been very fortunate to receive indispensible help and guidance from Dr. Tim Harris, Dr. Satnam Singh, Prof. Roberto Hexsel, Prof. Arvind and Prof. Mateo Valero.

I am also indebted to all my coleagues and friends (my second family) from Barcelona Supercomputing Center for their constant help and support during this research. Thanks to, in order of appearance: Srdjan Stipic, Sasa Tomic, Ferad Zyulkyarov, Oriol Prat, Paul Carpenter, Marco Galuzzi, Cristian Perfumo, Sutirtha Sanyal, Milos Milovanovic, Vladimir Gajinov, Gokcen Kestor, Petar Radojkovic, Vladimir Subotic, Vladimir Marjanovic, Vladimir Cakarevic, Maja Etinski, Enrique Vallejo, Chinmay Kulkarni, Azam Seyedi, Vasilis Karakostas, Gulay Yalcin, Vesna Smiljkovic, Adrià Armejach, Ibrahim Hur, Nebojsa Miletic, Javier Arias, Otto Pflucker, Milovan Djuric, Milan Stanic, Timothy Hayes, Ivan Ratkovic, Nikola Markovic, Daniel Nemirovsky, Ege Akpinar, Oscar Palomar and to all those that I might have forgotten to mention here. I also have to thank Asif I. Khan, Myron King and the rest of the CSG at MIT, as well as Miquel Pericàs, Roberto Gioiosa, Ruben Titos and Paolo Meloni for their help and inspiration. Finally, thanks to my family for their patience and my friends for always being my best escape from work[1].

[1]The cover image is an 8-core BeeFarm design mapped on a Virtex-5 FPGA.

Abstract

Since the first "core mitosis" in processor market in 2005, the multicore era has implied a drastic change in computer architecture. For many decades, Moore's law [95] had dictated that processor frequencies would be doubled every 18 months, which indeed caused a thousand-fold increase during this time. Due to the memory wall and the power wall both hit, in order to offer higher performance, mainstream manufacturers were forced to place multiple processors on a silicon die, whereas before Moore's law had them pushing for higher-frequency massive superpipelined cores with high single-thread performance. This new direction demands better expressiveness of thread-level parallelism (TLP) and suitable ways of providing concurrency in programming a shared-memory Chip Multiprocessor (CMP). "Multicore architectures are an inflection point in mainstream software development because they force developers to write parallel programs" [3].

To program these larger and scalable parallel architectures, easier methods and abstractions for the efficient use of parallelism are essential. Traditional mechanisms such as lock-based thread synchronization, which are tricky to use and non-composable, are becoming less likely to survive. Consequently, the use of atomic instructions in lock-free Transactional Memory (TM) is a serious candidate to being the future of concurrent programming. TM is a programming paradigm for deadlock-free execution of parallel code that provides optimistic concurrency by executing transactions atomically: in an all-or-none manner. In case of a data inconsistency, a conflict causes the transaction to be aborted without committing its changes, and restarted as if no state change had occurred.

Nowadays, TM is being seen as one of the most promising ways of the parallel programming revolution, ensuring deadlock-free transactional code segments to run atomically, saving the programmer from explicitly dealing with locks. However, how TM guarantees such as atomicity and deadlock-freedom should be provided to the programmer has been a very active research topic for the last two decades.

While TM was so actively investigated, the past decade has also seen a shift of interests from using software simulation for evaluating new research ideas, to hardware emulation and prototyping in architectural design space exploration, using programmable FPGAs (ie. reconfigurable computing). Recent advances in multicore computer architecture research were being hindered by the inadequate performance of software-based instruction set simulators which led many researchers to consider the use of FPGA-based emulation. The primary reason for using an FPGA-based simulator is to achieve a significantly faster simulation speed for multicore architecture research, compared to the performance of software instruction set simulators. A secondary reason is that a system that uses only the FPGA fabric to model a multicore processor may have a higher degree of fidelity, since no functionality is implemented by a magical software routine.

This thesis attempts to bring together these two recent topics by presenting a flexible Transactional Memory environment on a prototype that is realized on FPGA fabric. For this, we develop a 16-core MIPS-compatible shared memory CMP system with Transactional Memory support, based on the Plasma open source soft processor core [113]. We present the design and implementation of the TMbox system, which features an emulation system of up to 16 MIPS soft processor cores interconnected with a bi-directional ring bus, running at 50 MHz on a Virtex5-155t FPGA of the BEE3 prototyping platform [36]. TMbox is a completely modifiable architecture implementing the first publicly-available multicore prototype with support for Hardware-, Software- and Hybrid TM. It was written in various common design

languages, and enables modifying the complete stack, down from the ISA, through the software toolchain, up to the optimized concurrent code. With our infrastructure, fast execution and quick performance evaluation can be made possible for studies in computer architecture.

Additionally, we build the first comprehensive infrastructure to profile Hybrid TM systems, an extensive visualization environment that enables examining complete transactional executions in detail. The profiling and visualization system of the TMbox enables in depth inspection of any kind of event, either triggered by the out-of-the-way profiling hardware or by a very low overhead software routine. It creates Paraver-like [19] multi-threaded traces, which help to correctly evaluate complex parallel executions as non-disruptively as possible. It is shown to aid in (i) porting programs to appropriately make use of Hybrid TM, (ii) discovering bottlenecks such as serialization, killer transactions and repetitive aborts, as well as (iii) depicting different program phases.

The result is a fast and flexible reconfigurable multicore architecture with a very useful profiling and visualization tool that serves as an efficient feedback mechanism. Although in this thesis the focus is on TM behavior, our infrastructure can be easily modified and extended to many other directions and to other topics of interest in novel computer architecture research.

Contents

List of Figures

Chapter 1

Introduction

1.1 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are the first reconfigurable computing fabrics that were proposed half a century ago by Gerald Estrin as a "fixed-plus-variable structure computer" [41]. FPGAs are part of many of the systems and devices that we use today, such as automobiles, consumer electronics, home appliances, aircraft, or medical devices. Due to their ability to be reprogrammed, FPGAs were initially used for rapid prototyping of complex digital systems, and later on, as co-processors to speed up computations. In these implementations, compute-intensive tasks or user-defined instructions could be mapped on fabric in compile-time, [49; 50; 87; 136] or frequently used loops and tasks could get mapped on reconfigurable fabric dynamically at run-time [88].

Many researchers have often used FPGA technology to accelerate computing applications. The performance achieved by these configurable machines can be up to one or two orders of magnitude greater than general purpose processor-based counterparts. Configurable computers are proven to be the fastest in fields such as RSA decryption, DNA sequence matching, signal processing, microprocessor emulation and cryptography. Here, the fast and parallel execution on the reconfigurable chip has to compensate for the communication and transfer overheads that occur while bringing the data into the FPGA and sending it back. Generally, the more the data parallelism and the fewer the dependencies between the data, the better the performance of an FPGA implementation would be.

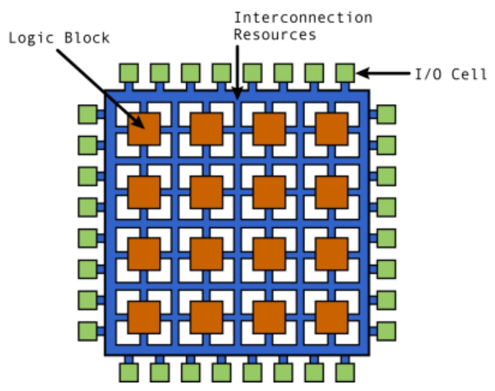


Figure 1.1: A generic FPGA [140].

A typical FPGA consists of an array of configurable logic blocks distributed across the entire chip in a large matrix of programmable interconnections, with programmable Input/Output (I/O) blocks at the periphery [132] (Figure 1.1). A logic block is an FPGA unit of area that includes N-input lookup tables (LUTs)[1] and D flip-flops. To simplify, an N-input LUT is a memory unit that, when programmed appropriately, can perform any Boolean function of up to N inputs [134], effectively emulating the logical functionality of the digital circuit. On the FPGA, configurable logic blocks (CLBs), which are made up of "slices" of LUTs are organized in an array and are used to build combinatorial and synchronous logic designs. Each CLB also has carry logic to help build fast, compact ripple-carry adders and multiplexers to help cascade multiple LUTs into larger logic structures. Each CLB element is tied to a switch matrix that accesses the general routing interconnection. Each Input/Output Block (IOB) in an FPGA offers input and output buffers and flip-flops. The programmable interconnect routes CLB/IOB output signals to other CLB/IOB inputs. It also provides low-skew clock lines that run around the FPGA, and horizontal long lines near each CLB. Although various ideas might get implemented differently by different vendors, the structures of common, non-specialized FPGAs are usually quite similar.

FPGAs are canonical examples of a configurable device. On an FPGA, all

[1]Throughout this thesis, we utilize Xilinx terminology, since we worked with their generously donated FPGAs.

layers already exist on the chip and connections are either created or removed and the LUTs are programmed to implement the desired functionality with the aid of Computer-Aided Design (CAD) tools. The functional computation to be implemented on the hardware is defined by a set of configuration bits, generated by the CAD tools, which describes how each gate and wire (interconnect) should behave. Consequently, FPGAs can perform any computational task that fits in the machine's finite state and computational capacity that is set by its operational resources. In such, the designers themselves have the advantage of emulating an Integrated Circuit (IC) on top of a programmable chip. Although there are benefits like having a low NRE (Non-Recurring Engineering) cost and a lower time-to-market period compared to ASICs (Application-Specific Integrated Circuits), it also presents some drawbacks such as the medium-high cost per unit and slower performance than ASICs.

The term configurable is used to refer to architectures where the active circuitry can perform any of a number of different operations, but the functionality cannot be changed from cycle to cycle[1]. The register-rich nature of FPGA chips do not solely consist of an array of look-up tables and flip-flops, but also include on-chip RAM blocks and fast hardware DSP (Digital Signal Processing) units, making them perfect candidates for processor design.

1.1.1 Intellectual Property Cores

In order to keep production and development expenses as low as possible, while complying with various design constraints, embedded devices often come in the form of a System-on-Chip (SoC) in a core-based system design. A SoC is a concept that integrates the use of pre-designed, pre-verified, re-usable silicon circuitry, called Intellectual Property (IP) cores, to be used as building blocks for large and complex applications interconnected by a network-on-chip (NoC) [32] on an Integrated Circuit (IC). So, rather than developing every sub-system from scratch, the system is composed by integrating various cores with the re-use of previously deployed ones (Figure 1.2).

[1]Although this is changing with new FPGAs that include static memories and 3D stacking technology [68].

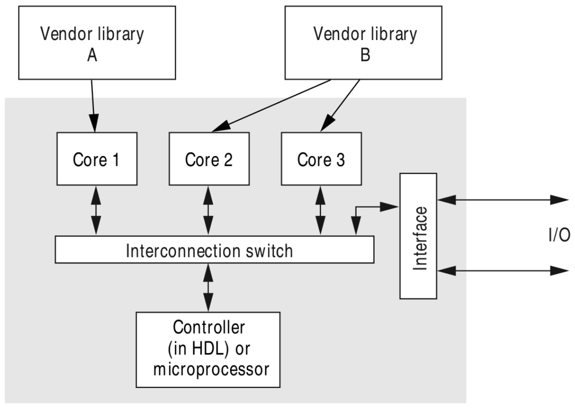


Figure 1.2: Generic system-on-chip (SoC) architecture [89].

SoCs have altered the way commercial, off-the-shelf components are sold: as Intellectual Property (IP) cores, processor-level components with behavioral, structural, or physical descriptions, rather than actual Integrated Circuits. Examples of common IP cores range from a thousand-gate analog circuit blocks to memory controllers, peripheral devices such as MAC (Media Access Control), Ethernet, UART (Universal Asynchronous Receiver / Transmitter), or PCI (Peripheral Component Interconnect) bus controllers, to million-gate processor cores [31]. IP cores can be split into two main categories: soft and hard cores, as discussed next.

1.1.2 Soft and Hard IP Cores

Although programmable logic saves development cost and time over increasingly complex ASIC designs, FPGAs have started becoming popular over the past two decades as the unit price has dropped and gate count per chip has reached numbers that allow for the implementation of more complex applications and soft cores. A soft core consists of a synthesizable HDL (Hardware Description Language) description that can be re-targeted to different semiconductor processes, rather than being a fixed part of the chip circuitry. HDLs such as VHDL (Very High Speed Integrated Circuit Hardware Description Language), Verilog, SystemC and Bluespec increase the range of options available to designers by enabling hardware implementation with the flexibility that language-based design provides, allowing designers to implement efficient soft Intellectual Property cores.

Hard IP cores are physical descriptions that involve the implementation of a silicon-level circuit within the device fabric. A hard core includes layout and technology-dependent timing information and is ready to be included into a system. These components are products of the specific technology used, are proprietary IPs by definition, and are subject to patents and copyrights. Depending on the design constraints, the designer can use the already optimized and synthesized hard cores, or to comply with the constraints, adapt the soft (or firm) cores to specific limitations. Typical hard cores included in a modern FPGA may include DSP blocks, Block RAMs, or hard processor IPs.

Since soft cores do not target a specific technology, they are inherently more flexible in function and implementation than hard cores. On the other hand, hard core developers can afford to spend more time optimizing their implementations to be used in many designs. For a SoC that requires the highest performance in current process and design technology, a full-custom hard core is better at meeting these needs by using latches, dynamic logic, 3-state signals and custom memories. However, they are not flexible or parameterizable like soft cores and do not accept modifications and customizations.

All of the benefits and characteristics of soft IP cores are realized by soft processor cores implemented within FPGA components. Two proprietary soft processor core examples are the Xilinx Microblaze and Altera Nios II [66; 71]. Both soft processor cores are 32-bit Harvard bus architecture (separate data and instruction memories) Reduced Instruction Set Computer (RISC) systems with 32 general-purpose registers. Many open source soft processor cores, which accept modifications have also been made available, especially in the last decade [2; 47; 52; 106; 113].

The typical design flow facilitated by CAD tools includes the synthesis of the Register Transfer Level (RTL) code written in an HDL into logical gates, later its translation for the selected technology, mapping of units and their placement onto the FPGA, the routing of all signals and finally the generation of the bitstream to be loaded on the FPGA (Figure 1.3).

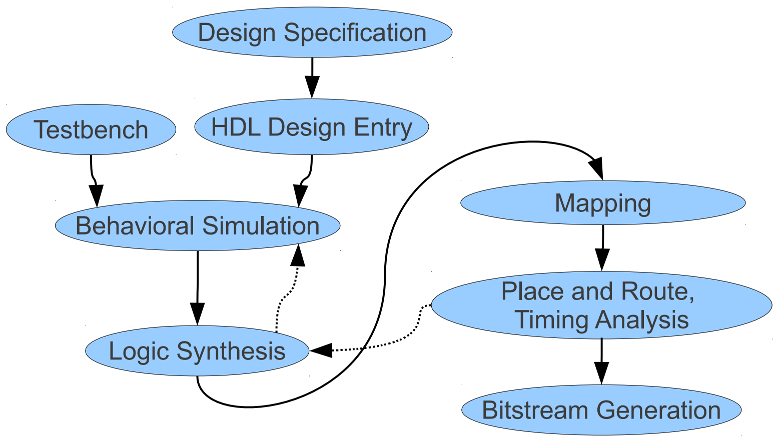


Figure 1.3: FPGA Design Flow

1.1.3 FPGA use for computer architecture investigation

With the always-increasing frequencies of typical uni-processors, the investigation of architectural schemes have been realized by software-based microarchitectural simulators, such as Simplescalar, PTLsim, Simics or M5 [12; 114]. These sequential simulators are expressive and it's relatively easy and fast to make changes to the system in a high-level environment. However, little effort has been made to parallelize or accelerate these programs which turn out to be slow to simultaneously simulate the multiple cores of a typical multiprocessor of the current era of chip multi-cores. This has caused the computer architecture community to consider performing emulations on reconfigurable fabrics as an alternative to using software-based simulations.

The low performance of legacy architectural simulator software for investigating new generation Chip Multi-Processor (CMP) architectures has been addressed in a few ways: the development of new parallel simulators [92], parallelization efforts for sequential simulators [114], acceleration using GPUs [110], and prototype/emulation implementations on reconfigurable fabric [26; 34]. FPGAs were proven successful in accelerating simulations working in concert with a host computer [22], as well as FPGA-only multi-core MPSoC implementations [125].

1.1.3.1 Current Overview

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 configurable logic cells each generation on a rapidly growing process technology. Another opportunity comes from the already-tested Intellectual Property (IP) cores. There are various open-source synthesizable Register Transfer Level (RTL) models of various x86, MIPS, PowerPC, SPARC architectures that can run at up to a hundred MHz. These models can already include detailed specifications for multi-level cache hierarchy, out-of-order issue, speculative execution, Floating Point Units (FPU), and branch prediction. These are excellent resources to start building a credible multicore system for any kind of architectural research. Furthermore, various IPs for incorporating UART, SD, Floating Point cores, Ethernet or DDR controllers are also easily accessible [104].

FPGAs can be good alternatives to implement complex computer circuitry. On-chip Block RAM (BRAM) resources on an FPGA which are optionally pre-initialized or with built-in ECC can be used in many configurations, such as (i) RAM or SRAM; for implementing direct mapped or set associative on-chip instruction/data cache, cache tags, cache coherence bits, snoop tags, register file, multiple contexts, branch target caches, return address caches or branch history tables, (ii) CAM; for reservation stations, out-of-order instruction issue/retire queues or fully associative TLBs, (iii) ROM; for bootloader or lookup tables, or (iv) asynchronous FIFO; to buffer data between processors, peripherals or coprocessors[51]. BRAM capacity, which does not occupy general-purpose Look-Up Table (LUT) space or flip-flops could be used to implement debug support tables for breakpoint address/value registers, count registers or memory access history. The available dedicated on-chip DSP blocks can be cascaded to form large multipliers/dividers or FPUs. Complete architectural inspection of the memory and processor subsystems can be performed using statistics counters embedded in the FPGAs without any overhead.

Many vendors provide large FPGA programming boards and high-end FPGA prototyping boxes with preferential pricing for academia. FPGAs have already been proposed to teach computer architecture courses for simple designs as well as for more advanced topics [52; 91; 127]. 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 [22; 26; 34; 81; 98; 125]. It is also known that processor vendors make use of FPGAs to better investigate their new designs in a rapid way [117; 135].

1.1.3.2 Choice of architecture

Although FPGA-based multiprocessor emulation has received considerable attention in the recent years, the experience and tradeoffs of building such an infrastructure from the already-available resources and IP cores has not yet been considered in depth. Indeed, most of the infrastructures developed were either (i) written from scratch using higher level HDLs, such as Bluespec [67], (ii) using hard cores such as PowerPC, or (iii) using proprietary cores such as the Microblaze [38].

One direction is to choose a well-known architecture like MIPS and enjoy the commonly-available toolchains and library support, as we demonstrate throughout this thesis. Although supporting a minimal OS might be acceptable depending on the objectives, a deeper software stack could have many advantages by providing memory protection, performing scheduling, aiding debugging, file system support, etc. Full OS support can be accomplished either by highly detailed design implementations on the FPGA, or with hybrid approaches, where the core functionality is retained in FPGA hardware and it cooperates with a nearby host computer that can serve (i) system calls and exceptions, (ii) infrequent or slow running instructions and/or (iii) I/O operations, instead of implementing everything inside the FPGA [26]. There also exist commercial simulator accelerators like Palladium and automated simulator parallelization efforts that take advantage of reconfigurable technology [108]. In this thesis, we model the entire multiprocessor system on FPGA logic (Figure 1.4).

This flexible experimental systems platform on an FPGA offers a multiprocessor System-on-Chip (SoC) implementation that: (i) can be configured to integrate

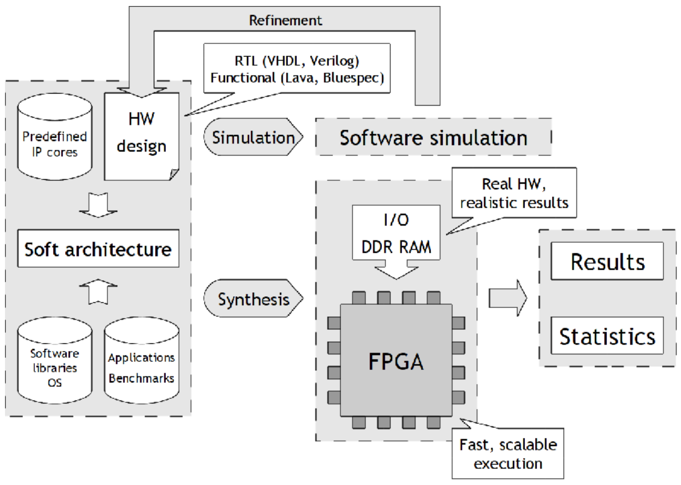


Figure 1.4: An overview of FPGA emulation approach.

various Instruction Set Architecture (ISA) extensions and hardware organizations, (ii) fit and scale well for large designs of tens of processor cores, (iii) offer high enough performance to run full benchmarks in acceptable timeframes, (iv) run at least some minimal OS and (v) must provide with credible results. The applications that run on real hardware should provide the researcher with fast, wide-ranging exploration of HW/SW options and head-to-head comparisons to determine the strade-offs between different implementations [6].

1.2 Transactional Memory

For the exchange of data among multiple threads, shared memory is a common and convenient IPC (Inter-Process Communication) paradigm to provide all processors with a single view of memory. However, some form of synchronization between the processes that are storing and fetching information to and from the shared memory region is required. For enforcing limits on access to a shared resource, locking is the most commonly used synchronization mechanism. Locking is simple to use, however it has many problems: Simple coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks, priority inversion or data races. Many scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races.

Some 30 years ago, Lomet proposed an idea to support atomic operations in programming languages, similar to what had already existed in database systems [86]. Nowadays, based on this idea, the proposal that has drawn the most attention for programming shared-memory CMPs has been the use of Transactional Memory (TM), an attractive paradigm for the deadlock-free execution of parallel code without using locks [64; 80; 115].

TM-based algorithms can be expected to run slower than ad-hoc non-blocking algorithms or fine-grained lock based code, but TM is as easy as using coarse-grained locks: one simply brackets the critical section that needs to be atomic! Using atomic blocks in TM simplify writing concurrent programs because when a block of code is marked, the compiler and the runtime system ensure that operations within the block appear atomic to the rest of the system [59]. TM schemes attempt to optimistically interleave and to execute all transactions in parallel. A transaction is committed if and only if any other transaction has not modified the section of the memory which its execution depended on. As a consequence, the programmer no longer needs to worry about manual locking, deadlocks, race conditions or priority inversion [60].

1.2.1 Flavors of TM

The Transactional Memory approach allows programmers to specify transaction sequences that are executed atomically, by encapsulating critical sections inside the `atomic{}` construct. TM implementations have to ensure that all operations within the block either complete as a whole, or automatically rollback as if they were never run. The underlying TM mechanism has to automatically detect data inconsistencies and aborts and restarts one or more transactions. If there are no inconsistencies, all the side effects of a transaction have to be committed as a whole.

Transactional Memory can be implemented in dedicated hardware (HTM) [20; 96], which is fast but resource-bounded, while it might require changes to the caches and the Instruction Set Architecture (ISA). On the other hand, Software TM (STM) [4; 44] 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 software when hardware resources are exceeded [33]. Another approach is Hardware-assisted STM (HaSTM), which by architectural means aims to accelerate a TM implementation that is controlled by software. By leaving the policy to software, different experimentations on contention management, deadlock and livelock avoidance, data granularity and nesting can be accomplished. HaSTM does not implement any TM semantics in hardware, but provides mechanisms that accelerate an STM, which may also have uses beyond TM.In transactional applications, a conflict occurs when two simultaneously running transactions access the same memory location and one of the accesses is a write. A TM implementation may detect conflicts either eagerly or lazily. In the eager approach, conflicts are detected immediately as soon as they occur, whereas in lazy conflict detection, they are detected at a later time of the transactional execution (e.g. at commit time).

Transactional write operations in a TM system can be buffered (lazy versioning) or done in-place (eager versioning). With buffered writes, the speculative values of the memory references are stored in a thread local buffer/cache, and only written to memory when the transaction successfully commits. With in-place writes, the TM system logs all original values for rolling back in case of an abort, and writes the speculative values to memory. Therefore, in buffered update TMs, commits are more expensive, whereas in eager update TMs, aborts tend to be more costly.

Furthermore, a TM implementation may also differ based on the granularity at which conflicts are detected. Typically TM implementations detect conflicts at word, cache line or object granularity. The choice of the granularity involves design and performance tradeoffs. Word and cache line granularity are more suitable for HTMs and non-garbage-collected low-level programming languages such as C. Per-object conflict detection is more suitable for managed STMs in object-oriented languages such as Java or C#.

Conflict resolution determines which transaction(s) are aborted at what point in the execution when a conflict is detected. Conflict resolution may follow immediately after a conflict is detected (i.e. eager conflict resolution) or at a later moment of the transactional execution (i.e. lazy conflict resolution), for instance at commit time. Once it is time to resolve the conflict, the TM system, or a contention manager [118] may choose to block the transactional execution until the other conflicting transaction commits or aborts, or delaying and assigning a back-off time for the abort. In case of a cyclical dependency between two or more waiting transactions, the TM system might choose to abort all transactions to avoid deadlock [58].

In the past two decades, there have been a flurry of proposals of different flavors of semantics and implementations, such as the first Transactional Memory system proposed in hardware (HTM) [64], that of a software-only implementation [119], hybrid approaches [33; 83], those that propose hardware conflict resolution, hardware signatures to track read and write sets, those that timestamp transactions, realize hardware-assisted TM, other hybrid mechanisms, and many more. Today, TM is still a topic of very active investigation.

1.3 Thesis Contributions

FPGA emulation/prototyping of multicores has been receiving a lot of research attention. Recently, FPGA emulators and prototypes of many complex architectures of various ISAs have been proposed. However, only a few of these are on research on Transactional Memory. Furthermore, only a few implement/scale to an interestingly-large number of processor cores. Additionally, the majority of these proposals are based on proprietary or hard processor cores, which imply rigid pipelines that can prevent the researcher from modifying the ISA and the microarchitecture of the system.

Therefore in this thesis, we choose a new approach: We reuse an already existing MIPS [76] processor core called Plasma [113] and we modify and extend it to build a full multiprocessor system designed for multicore research on Software-, Hardware- and Hybrid Transactional Memory. More particularly, the contributions of this thesis are made up of three parts:

- Beefarm STM: Reports on our experience of designing and building an eight core cache-coherent shared-memory multiprocessor system on FPGA called BeeFarm, to help investigate support for Software Transactional Memory [54; 96; 129]. Towards this goal, we have successfully ported an STM library and ran TM applications on the BeeFarm.

- TMbox HybridTM: Features an MPSoC built to explore trade-offs in multi-core design space and to evaluate recent parallel programming proposals such as Hybrid Transactional Memory. For this work, we evaluate a 16-core Hybrid TM implementation based on the TinySTM-ASF proposal on a Virtex-5 FPGA [73] and we accelerate three benchmarks written to investigate TM trade-offs. Our flexible system, comprised of MIPS R3000 compatible cores interconnected by a ring network, is easily modifiable to study different architecture, library or Operating System extensions. TMbox is the first implementation on FPGA in the literature with support for Sofware-, Hardware- and Hybrid TM.

- TMbox Profiling: Multi-core prototyping additionally presents a good opportunity for establishing low overhead and detailed profiling and visualization in order to study new research topics. In this direction, we design and implement a low execution, low area overhead profiling mechanism and a visualization tool for observing Transactional Memory behaviors on FPGA. We demonstrate the usefulness of such detailed lightweight examination of SW/HW transactional behavior to appropriately port applications to Hybrid TM and to accelerate them. Thanks to its ability to rapidly run and visualize full multi-threaded benchmarks, the TMbox with profiling support can point out pathologies such as repetitive aborts, killer transactions and starvation and to depict the phased behavior in full benchmarks with very low instrumentation overheads.

In this thesis, we explain how we devised a flexible infrastructure to run and to inspect Transactional Memory systems and applications in a rapid way using reconfigurable computing technology. As a result, TMbox is the only publicly available multicore prototype with extensive support for Hardware-, Software-

and Hybrid Transactional Memory and a low overhead profiler and visualizer for analyzing extremely detailed behavior of multi-threaded transactional programs. Chapter 2 presents the design and implementation of the Beefarm STM, an initial implementation with Software TM support. Chapter 3 contains a description of the TMbox Hybrid TM implementation and Chapter 4 shows the low overhead profiling and visualization infrastructure designed for TMbox. A discussion of tradeoffs of using reconfigurable computing and future trends are in Chapter 5, as well as thesis conclusions. Appendices 1, 2 and 3 contain the list of TMbox instructions, registers, and a table of abbreviations, respectively.

The work in this thesis was highly facilitated by three other students. UPC PhD student Oriol Arcas helped in implementing the MIPS CoProcessor0, cache modifications, the software stack, the TM Unit, the profiling infrastructure, and others that are described in detail in his Master's thesis [9]. Additionally, Gokhan Sayilar (from Sabanci University, now a PhD student at UT-Austin) helped us to implement an efficient FPU for the BeeFarm. Philipp Kirchhofer from the Karlsruhe Institute of Technology implemented the HTM part of the profiling infrastructure, as well as the initial profiling data analysis software [79]. Besides the mentors Osman S. Unsal, Adrian Cristal and Mateo Valero, we received precious support from Ibrahim Hur (BSC, now at Intel) and from Satnam Singh (MSRC, now at Google), our FPGA guru.

Chapter 2

The BeeFarm STM platform

Abstract

This chapter reports on our experience of designing and building an eight core cache-coherent shared-memory multiprocessor system on FPGA called BeeFarm, to help investigate support for Transactional Memory [54; 96; 129]. Towards this goal, we have ported TinySTM [44], a lightweight and efficient word-based STM library implementation in C and C++, and ran TM benchmarks on the BeeFarm. Our approach is through taking a MIPS-based open-source uniprocessor soft core, Plasma, and extending it to obtain the BeeFarm infrastructure for FPGA-based multiprocessor emulation. We discuss various design tradeoffs and we demonstrate superior scalability through experimental results compared to a traditional software instruction set simulator, the M5 [12].

2.1 Introduction

In this chapter, we take a popular MIPS uniprocessor core called Plasma [113] and we extend it to build a full multiprocessor system designed for multicore research for Software Transactional Memory. For this, we also provide our infrastructure with compiler tools to support a programming environment rich enough to conduct experiments on Transactional Memory workloads. An hypothesis we wish to investigate is the belief that an FPGA-based emulator for multicore systems will have better scalability compared to software-based instruction set simulators. We check this hypothesis using our flexible BeeFarm infrastructure with designs ranging from 1 to 8 cores and obtaining performance speedups of up to 8x, comparing to the M5 simulator.

2.1.1 Contributions

The key contributions of this chapter are:

- A description of extending the Plasma open source processor core for implementing the Honeycomb processor.
- From the Honeycomb processor cores, building the cache coherent BeeFarm multiprocessor system on the BEE3 platform [36].
- Experimental results for three benchmarks investigating support for Transactional Memory and an analysis of the performance and scalability of software simulators versus hardware emulation/prototyping through using the BeeFarm system.

The next section explains how the Plasma core was modified to design the Honeycomb core, and later to build the BeeFarm soft multicore, and the software stack that supports running STM applications. Section 2.3 compares executions of three STM benchmarks on our platform with the M5 software simulator. Section 2.4 discusses other related research, while Section 2.5 concludes this chapter.

2.2 The BeeFarm System

The BeeFarm system is a bus-based multiprocessor implementation of the well-known MIPS R3000 architecture, inspired by the Plasma soft processor core [113], designed for running on the BEE3 hardware prototyping platform. Particularly, the objective is to reuse a complete and small soft processor IP core, and to be able to fit as many cache coherent cores as possible into a TM-capable multicore emulator. RISC architectures with simpler pipelines can be customized more easily and require less FPGA resources compared to a deeply-pipelined superscalar processor, so they are more appropriate to be integrated into a larger multiprocessor SoC. For this reason, using large, multithreaded soft cores like OpenSPARC (64-bit) [31] were omitted. We chose to use the Plasma in this work, mainly since (i) it is based on the popular MIPS architecture [94], (ii) it is complete and (iii) it has a relatively small area footprint on the FPGA. Using the Leon 3 (32-bit SPARC core) [1] and the miniMIPS [55] as the main processor soft IP core were the other two acceptable options with similar advantages and disadvantages.

2.2.1 The Plasma soft core

The synthesizable MIPS R2000-compatible soft processor core Plasma was designed for embedded systems and written in VHDL [113]. It features 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 includes UART and Ethernet cores. It also has its own real-time operating system (RTOS) with some support for tasks, semaphores, mutexes, message queues, timers, heaps etc.

In a typical ALU instruction, the program counter (PC) passes the current instruction address to the memory control unit, which fetches the 32-bit opcode from cache or from memory, when needed. Cache accesses pause the CPU and can take various cycles in case of a miss. In the next stage, the opcode received is passed to the control unit that converts it to a 60-bit control word and forwards it to the appropriate entities through a central multiplexer (bus-mux).

2.2.2 The Honeycomb core: Extending Plasma

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. These include the lack of virtual memory support (implemented in MIPS as the Coprocessor 0), precise exceptions and synchronization mechanisms. Furthermore, there is no support for floating point arithmetic (MIPS Coprocessor 1), multiprocessing capabilities or coherent caches.

The successor architecture to the MIPS R2000 ISA is the MIPS R3000, featuring a 5-stage pipeline and a co-processor for managing virtual memory. Later, the MIPS R4000 developed implemented a 64-bit pipeline. To be more resource efficient, we did not opt for a 64-bit datapath, however, we believe that this will be a necessity for future multicore emulators that can make use of bigger and more advanced FPGAs. Therefore, to build the Honeycomb core, we effectively upgraded the MIPS R2000-compatible Plasma to a MIPS R3000-compatible soft processor core. For enabling this, we made several changes to the Plasma soft core:

- Design and implementation of two coprocessors: CP0 that provides support for virtual memory using a Translation Lookaside Buffer (TLB), and CP1 encapsulating an FPU,
- Optimization of the cores to make better use of the resources on our Virtex-5 FPGAs, where it can run at twice the frequency (50 MHz),
- Memory architecture modifications to enable virtual memory addressing for 4 GB and caches of 8 KB,
- Implementation of extra instructions to better support exceptions and thread synchronization (load-linked and store-conditional) ,
- Added coherent caches and developed a parameterizable system bus that accesses off-chip RAM through a DDR2 memory controller [126],
- Development of system libraries for memory allocation, I/O and string functions, as in [123].

On the Honeycomb processor (Figure 2.1) 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 stage for data access instructions (Figure 2.2).

The Honeycomb core was designed to run on the BEE3 hardware prototyping platform which contains four Virtex5-155T FPGAs, each one with 24,320 slices of 6-LUTs, 212 BRAMs, and 128 DSP units. Four DDR2 memories are controlled by each FPGA, organized in two channels of up to 4 GB each. The DDR2 controller [126] manages one of the two channels per FPGA using a small processor called TC5 (also used in Beehive V5 [127]). It performs calibration and serves requests, and occupies a small portion (around 2%) of the Virtex5-155T FPGA. 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.

A new processor model (-march=honeycomb) was added by modifying GCC and GAS (the GNU Assembler). This new ISA includes all MIPS R3000 instructions with the addition of RFE (Return from Exception), Load Linked and Store Conditional. All GNU tools (GAS, ld, objdump) were modified to work with these new instructions.

Coprocessor 0: The MMU: In order to support 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 provides memory management and exception handling intercepting the memory control unit datapath. Each entry contains two values: The 20 highest bits of the physical address, which replace the corresponding ones in the virtual address, and the 20 highest bits of the virtual address, which are used as a matching pattern. More details can be found in [9].

There exist various approaches to implement an efficient Content Addressable Memory (CAM) on FPGAs, with configurable read/write access times, resource usage, and the technology utilized, where general-purpose LUTs or on-chip block memories can be used [16]. The use of LUT logic is inappropriate for medium and large CAMs, and the time-critical nature of this unit makes multi-cycle access

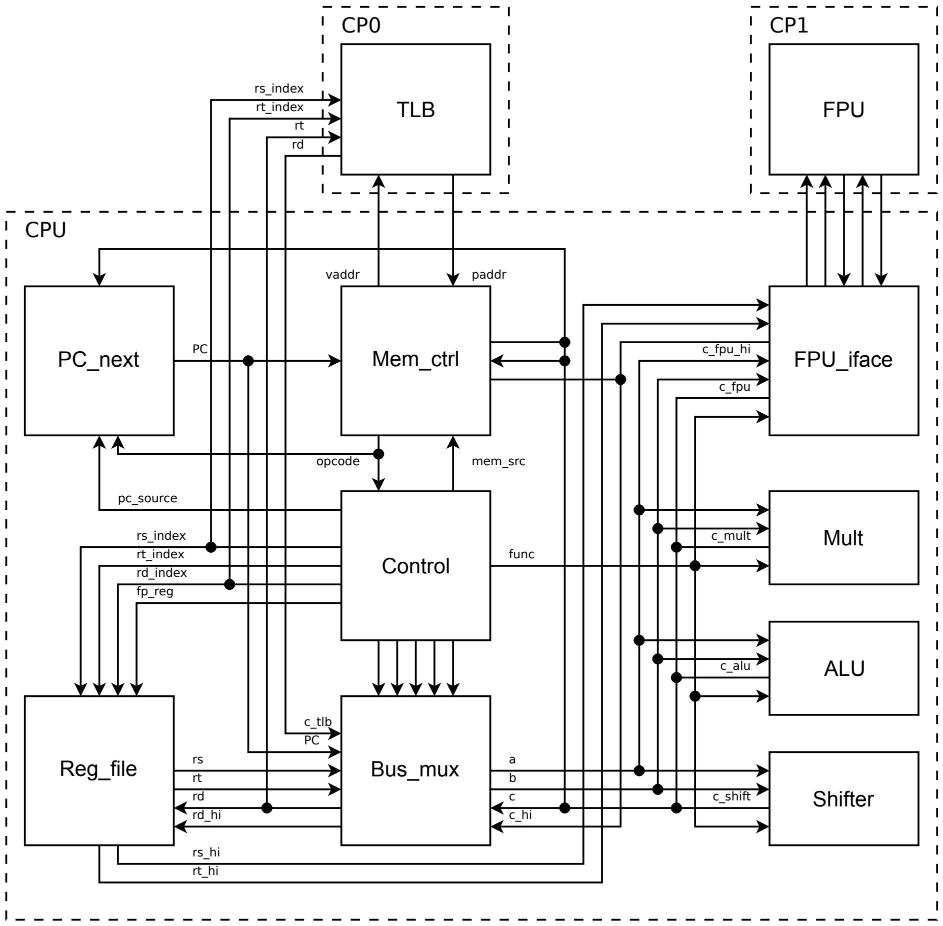


Figure 2.1: The Honeycomb processor functional block diagram, with the Coprocessor 0 (CP0) for virtual memory and exceptions and the Coprocessor 1 (CP1) for floating point arithmetic.

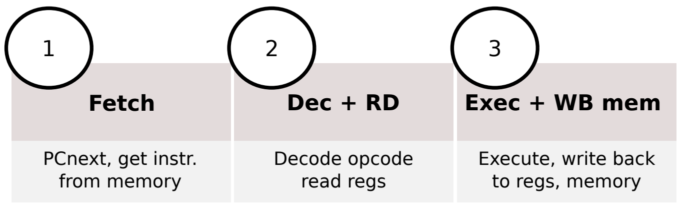

1
Fetch
PCnext, get instr. from memory
2
Dec + RD
Decode opcode read regs
3
Exec + WB mem
Execute, write back to regs, memory

Figure 2.2: The Honeycomb pipeline.

inappropriate, since it must translate addresses each cycle on our design. Only the approach based on RAM blocks fitted the requirements: This unit was implemented with on-chip BRAM configured as a 64-entry CAM and a small 64-entry LUT-RAM [70]. Virtual patterns that are stored in the CAM give access to an index to the RAM that contains the physical value. It is controlled by a half-cycle shifted clock that performs the translation in the middle of the memory access stage, so it does not require a dedicated pipeline 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, an integral part of a modern computing system architecture. The MIPS 3010 FPU implemented in Coprocessor 1 (CP1) can perform IEEE 754-compatible single and double precision floating point operations. It was designed using Xilinx Coregen library cores, takes up 5520 LUTs and 14 DSP units, performing FP operations and conversions in variable number of cycles (4–59). 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. The integer register file was extended to include FP registers implemented as LUT-RAM. For double precision, two registers represent the low and high part of the 64-bit number and the register file was replicated to allow 64-bit (double precision) read/write access each cycle [116].

Memory Map and ISA Extensions: We redesigned the memory subsystem that 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. We also provided with **LL** (Load-Linked) and **SC** (Store Conditional) instructions, which provide hardware support for synchronization mechanisms such as Compare and Swap (CAS) or Fetch and Add (FAA). This is essential for providing Software TM support, as we detail in Section 2.2.5.

Honeycomb's 8 KB write-through L1 cache design that supports the MSI cache coherency [63] (unified data and instructions) in 16-byte, direct-mapped

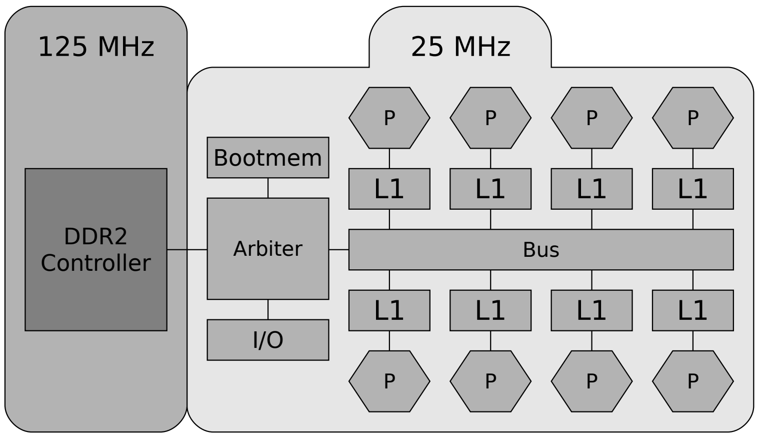


Figure 2.3: The BeeFarm multiprocessor system.

blocks. It uses 2 BRAMs for storing data and another two for the cache tags. The BRAM's dual-port access enables serving both CPU and bus requests in a single cycle. Reads and writes are blocking, and coherence is guaranteed by the snoopy cache invalidation protocol that was implemented. 2.4

2.2.3 The BeeFarm System Architecture

The caches designed for the BeeFarm are interconnected with a central split-bus controlled by an arbiter, as shown in Figure 2.3. The caches snoop on the system bus to invalidate entries that match the current write address, where write accesses are processed in an absolute order. This protocol can perform invalidations as soon as the writes are issued on the write FIFOs of the DDR. This serves to find an adequate balance between efficiency and resource usage. More complex caches that demand a higher resource usage 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. I/O ports are also mapped, and the lowest 8 KB of physical memory give access to the cache memory, becoming a useful resource

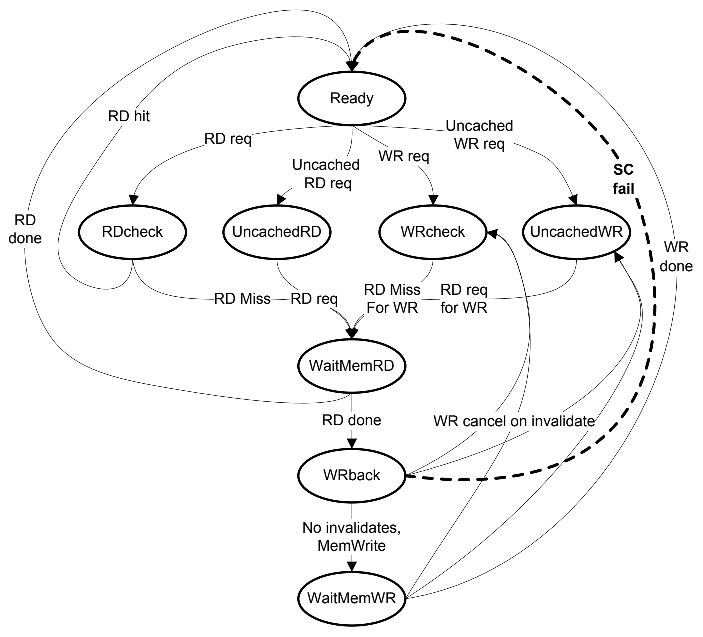


Figure 2.4: Cache state diagram.

during boot-up when the DDR is not yet initialized. Furthermore, the cache can be used as the 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/fifth 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, 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 such extra circuitry.

Finally, around eight Honeycomb cores (without an FPU) could form a BeeFarm system on one Virtex5-155T FPGA, although the system bus can become a bottleneck not only during system execution, but also when placing and routing the design.

2.2.4 FPGA resource utilization

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 5,712 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 [71]. Figure 2.5 shows the LUT occupation of the CPU's components. The functional blocks on the Honeycomb can be categorized in three groups:

- **Compute-intensive (DSP):** Eg. ALU, MULT/DIV, Shifter, FPU. These are good candidates to take advantage of hard DSP units and in the Plasma core originally fit the third category, since the ALU is a combinatorial circuit, while the MUL/DIV/Shifter units take 32 cycles to iterate and compute. 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 LUTs. The shifter can also benefit from these 4 DSP thanks to dynamic opmodes, however, a 32-bit divider can take anywhere between 1,100 LUTs to 14 DSP units: The optimal way of combining all these operations to have a minimal design is not yet clear, and would be interesting to look into as future work.

- **Memory-intensive (BRAM/LUT-RAM):** Eg. Reg_Bank, Cache, TLB. The TLB is designed in a CAM, and the cache and the cache tags in BRAMs. For the Reg_Bank, 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

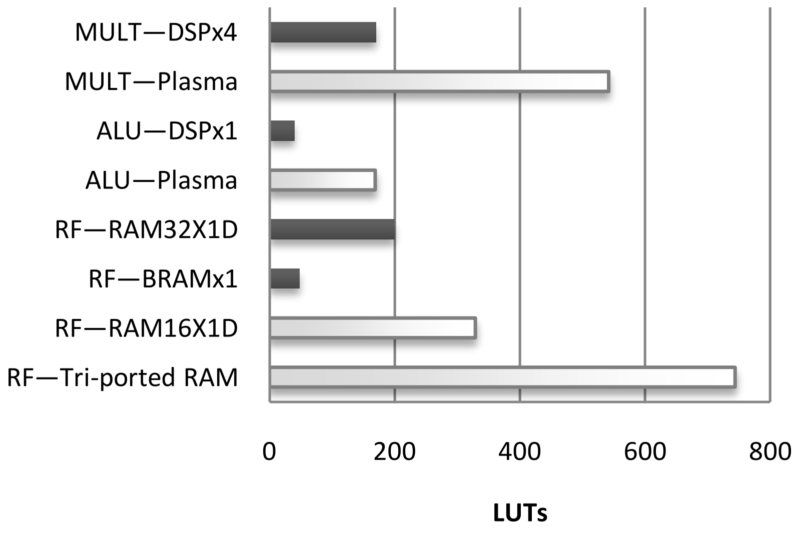


Figure 2.5: Some of the available options (in 6-LUTs) to implement the Register File, the ALU and the Multiplier unit. The lighter bars indicate the choices already present in the original Plasma design.

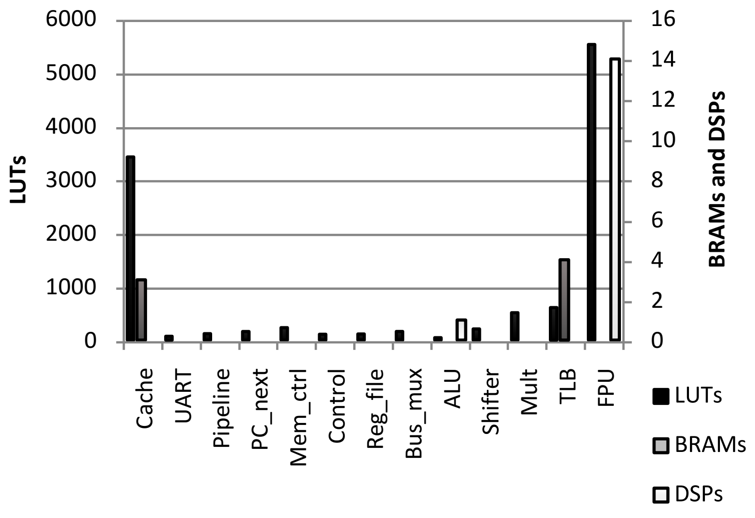


Figure 2.6: LUT and BRAM usage of Honeycomb components.

it would use a tiny portion of a large structure, and the tri-ported RAM infers too many LUTs, as seen in Figure 2.5. When distributed LUT-RAM is inferred, each 6-LUT can incorporate a 32-bit register on the Virtex-5, enabling two reads and one write operation per cycle, assuming that 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 multiple cycles [133].

- **LUT-intensive:** Eg. implementing irregular case/if structures or state machines: 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 2.5 is that providing cache coherency 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 still be expected to perform faster as the FPGA technology advances.

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 6-LUTs, although it is not possible to perform a 3-ported access in a single-cycle. BRAMs and DSP units must be used carefully, to better match the underlying FPGA architecture. Regular units that match a compute-and-store template rather than complex state machines must be fashioned. In general, we believe that caches are a bottleneck and a good research topic for multicore prototyping. There is little capacity for larger or multi-level

L1 Cache Size	slice Regs	slice LUTs	BRAMs
4 KB	19	26	3
8 KB	40	47	4
16 KB	75	80	6

Table 2.1: Area occupied by L1 caches of different sizes

caches on our FPGA, and it would not be easy at all to provide high levels of cache associativity.

There are a total of 212 Block RAMs on our Virtex 5 FPGAs. The DDR controller utilizes 5, the Bootmem uses 2, and the TC5 tiny processor needs another 3. Figure 2.1 shows the distribution of FPGA resources for different L1 cache sizes. For supporting hardware transactions later on the next chapter, we also want to reserve BRAMs for implementing a TM cache. As a rule of thumb, we don't want to use up more than 50% of the hard FPGA units. As the number of cores go up, placement and routing would be more difficult to perform. Therefore, we choose to use caches of 8 KB each, using 4 BRAMs per core for implementing the L1 cache. The cache design is parameterizable, and accepts other configurations.

2.2.5 The BeeFarm Software Stack

Since our system can not directly utilize the GNU C standard library libC and we want to avoid the complexities of running a full Linux with all system calls implemented, we developed a set of system libraries called BeeLibC for memory allocation, I/O and string functions.

Unlike the BeeFarm, many groups exercise falling back to a host machine or a nearby on-chip hard processor core to process system calls and exceptions [26; 125]. 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 support, which was implemented by software

```
$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 2.7: Compare and Swap using LL/SC in MIPS assembly

bit-banging can transfer roughly a MB of data per minute. Alternatively, an SD soft core can be incorporated for even faster data transfer. Although Ethernet might be another fast option, we deemed the additional complexity too potentially risky while developing the initial design. It might also be useful to port an RTOS with multiprocessing and threading support such as the eCos [40] or RTEMS [99] to multicore emulators. We currently let all cores initialize and wait on a barrier, which is set by CPU0. Another option that reserves the CPU0 only for I/O is also implemented.

TinySTM is an STM library that differentiates mainly by its time-based algorithm and lock-based design from other STMs, such as TL2 and Intel STM [4; 39]. 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 through the use of LL/SC instructions borrowed from the MIPS R4000 architecture [25]. Figure 2.7 shows hos LL/SC instructions can be composed to form a Compare and Swap operation.

2.3 Comparison with SW Simulators

2.3.1 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.
- Abstract away from library or OS implementation details, so that external functions like system calls do 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, a key weakness of software multiprocessor simulators. Our emulations are inherently 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).

M5 [12] is a well-known and easily modifiable "full-system simulator" that can simulate an arbitrary number of Alpha processors with complex architectural details like caches and buses. We believe that despite the fact that MIPS and Alpha are distinct architectures, this can be a fair comparison to measure and compare the scalability of the software simulator and the BeeFarm multicore emulator. Both architectures are 32-bit RISC, featuring 32 registers, operate on fixed-size opcodes and the only operations that access the memory are load and store. We used a configuration that models a DEC Tsunami system with an in-order, 5-stage pipeline Alpha 21164 CPU, with 64-entry data TLB, 48-entry instruction TLB, 2-cycle L1 cache access and 10-cycle L2 cache access times [48].

To obtain exact measurements of the execution time of the M5, we added a precise 64-bit hardware cycle counter to measure the total execution time which we compare against BeeFarm hardware counters. 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 and the M5 both use GCC version 4.2, compiled with the -O2 optimization level or -O3 when possible on 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.

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 [69], for which we apply various triggers to read our hardware debug registers. All results were obtained using 64-bit Xilinx ISE 12.2 running on RHEL5.

2.3.2 Single Core Performance

The Honeycomb core, despite of its smaller cache, performs in comparable speeds to some older well-known architectures running the Dhrystone 2.1 benchmark [138], as seen on Figure 2.8. The results are in VAX MIPS, the unit of measurement of Dhrystone performance. In particular, Honeycomb has a better performance than M5 when the total execution time of the simulator (wall clock time) is compared. OVPsim is a fast simulator that is used to accelerate the development of embedded software, simulating the MIPS32 architecture [103]. Both software simulators were run on the 2x4-core server previously described.

2.3.3 Multicore Performance using STM Benchmarks

To test multicore performance with STM benchmarks running on the BeeFarm, we have run ScalParC, a scalable TM benchmark from RMS-TM [77], as well as

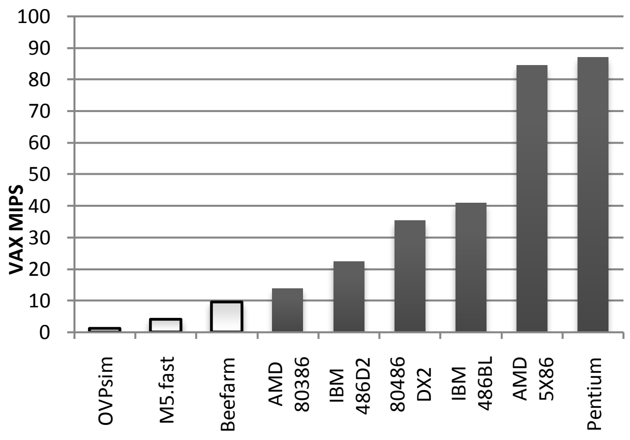


Figure 2.8: Dhrystone 2.1 performance comparison of BeeFarm, M5 simulator and others.

Intruder and SSCA2 TM benchmarks from STAMP [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 2.9 shows that the gap opens with more cores and with only four, the BeeFarm with FPU just needs fifteen minutes to run the ScalParC benchmark, an eightfold difference. A single core ScalParC run on the BeeFarm took around 9 hours when a soft-float FP library is used [62] and a little more than an hour when an FPU is used (not shown).

The scalability of our hardware is more obvious when the abort ratio between the transactions are low and little work is repeated, so the benchmark itself is scalable. SSCA2 also benefits from the inherent parallelism of the FPGA infrastructure and the good performance of the FPU: The two-core BeeFarm takes about half of the runtime of the M5 and it shows better scalability with more cores. In this sense our FPU which takes roughly the space of a single CPU core

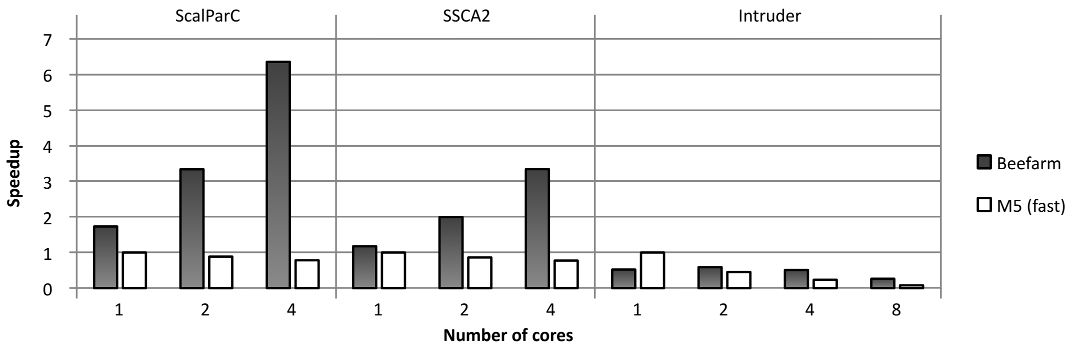


Figure 2.9: BeeFarm vs M5: Speedups for ScalParC, SSCA2 and Intruder (normalized to single core M5 execution).

is clearly a worthy investment for the case of this particular benchmark. Other available hardware kernels such as a quicksort core [17] 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.

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 the FPU.

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 an 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.

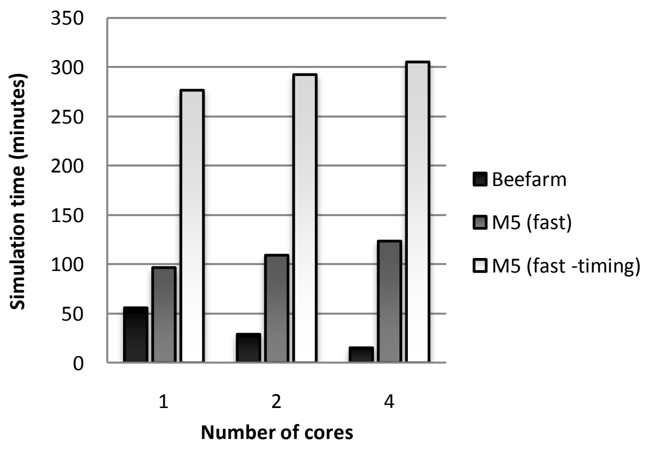


Figure 2.10: BeeFarm vs M5: ScalParC simulation time.

2.4 Related Work

Some classic works using MIPS and FPGAs include the GARP reconfigurable array which is used as an accelerator to a MIPS processor [61], and DartMIPS an early paper on implementing the MIPS R3000 architecture on an FPGA [42]. On a similar vein, RPM features a multiprocessor design with SPARC processors coupled with cache and memory controllers that are implemented on FPGA [102].

Some of the recent multicore prototyping proposals such as RAMP Blue [81] implement full ISA on RTL and require access to large FPGA infrastructures, while others such as the Protoflex [26; 27] can use single-FPGA boards with SMT-like execution engines for simulator acceleration. Although multithreading would lead to a better resource utilization on FPGAs, on this initial study it was not implemented. There is a wide variation of ISAs such as larger SMP soft cores [34], hard cores [137] and small and unconventional cores like the TC5 [127], for prototyping shared memory as well as message-passing schemes [6]. Our work differs from these approaches in the sense that we model the cores only on reconfigurable logic and we effectively upgrade a full ISA open source soft processor core to better fit the architecture of modern FPGAs and to be able to closely examine STM applications and implementations.

The only previous study on adapting an available soft core onto a commercial FPGA platform has been the LEON-3 core on the BEE2, however the report

Design name	Description
`Protoflex/Simflex(CMU)` [26]	16-way ultraSPARC SMP, FPGA provides acceleration (interleaved multi-context execution engine), SW SIMICS for I/O, syscall simulation (BEE2 board)
`Hasim (MIT)` [34]	MIPS R10K/Alpha in Bluespec, 4-way OoO, with separate functional/timing partitions.
`UT-FAST(UT)` [22]	PowerPC/x86 in Bluespec. AMD desktop+Virtex4VLX200 via hypertransport
`RAMP Red(ATLAS)` [98; 137]	On-chip PPC405 hard cores, 2 per board, 8-32 boards (Virtex2Pro30), runs Montavista Linux, Transactional Memory support
`RAMP Blue` [81]	Microblaze cores w/ crossbar-on-chip, message-passing on distributed memory, 21 BEE2 boards–1000 processors, Runs uclinux
`RAMP White` [6]	PPC hard cores on V2P30, runs Linux, cache coherent CMP
`RAMP Gold` [125]	64 in-order SPARC V8 on 8 BEE3s. Multiple HW threads on target model, separate functional/timing partition
`BeehiveV5` [127]	TC5 RISC processors, message-passing on Virtex-5 FPGA, using C and C#

Table 2.2: FPGA-based Multiprocessor Designs for Hardware Prototyping

by Wong [139] does not include many details or their experiences on the feasibility of this approach. As for comparison of hardware emulation with software simulators, RAMP Gold [125] compares a SPARC V8 ISA design with three different Simics configurations: Functional, cache, and timing. They obtain up to 250x speedup for 64 cores running Splash-2 benchmarks. The ATLAS design compares 8 on-chip PowerPC hard cores (100 MHz) with HTM support on five TM applications with a execution-driven simulator at 2 GHz to show 40-200x speedup. Their experience also reflects that higher level caches and high associativity were problematic to implement on current FPGAs [137]. It might be worth investigating novel implementations that could reduce the size of the FP logic, for example (i) reusing FP units to also do integer calculations, (ii) combining them as in [131], or (iii) sharing them between the CPU cores as done in RAMP Blue [81]. Furthermore, few works have addressed implementing cache coherent multiprocessors on FPGAs, some examples exist for MIPS [74] and PowerPC systems [78].

Transactional Memory has already drawn a lot of attention in the research community as a new easy-to-use programming paradigm for shared-memory multicores. However, so far mostly preliminary work has been published in the context of studying TM on FPGA-based multicore prototypes. The ATLAS emulator (RAMP) proposed an 8-way CMP system with PowerPC hard cores, and read-/write set buffers and caches augmented with transactional read-write bits and TCC-type TM support, and a ninth core for running Linux [98; 137]. Another work presented two Microblaze [71] cores with TCC support on a small Xilinx Spartan-3 board [90]. A TM implementation that targets embedded systems which can work without caches, using a central transactional controller interconnecting four Microblaze cores was explained in [75]. The feasibility of a DSM (distributed shared memory) TM implementation was investigated by sketching an architecture using Altera Nios processors in 53.

Recent work that also utilizes MIPS soft cores focuses on the design of a conflict detection mechanism that uses Bloom filters [14] for a 2-core FPGA-based HTM, however they do not consider/detail any design aspects on their infrastructure. They derive application-specific signatures that are used for detecting conflicts in a single pipeline stage. The design takes little area, reducing false conflicts, and this approach is a good match for FPGAs because of the underlying bit-level parallelism used for signatures [84].

2.5 BeeFarm Conclusions

In this work, we have described a different road map 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 processor core must 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've presented an 8-core prototype on a modern FPGA prototyping platform and compared performance and scalability to software simulators for three benchmarks written to explore tradeoffs in Transactional Memory.

The BeeFarm architecture shows very encouraging scalability results, which helps to support the hypothesis that an FPGA-based emulator would have a simulation speed that scales better with more modelled processor cores than a software-based instruction set simulator. For a small number of cores, we find that software-based instruction set simulators are still competitive, however the gap widens dramatically as more cores are used, as ScalParC runs suggest, where for 4 cores the BeeFarm system outperforms the M5 simulator (in fast mode) by around 8x.

A system that uses the FPGA fabric to model a multicore processor may have a higher degree of fidelity than a software simulator, since no functionality is implemented by a magical software routine. By interfacing a real DDR controller and real I/O, FPGAs have to implement real hardware with timing and area constraints. 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 are working on a ring architecture that can substantially reduce routing congestion to fit more Honeycomb cores on an FPGA.

FPGAs also need to provide tri-ported RAM support for the benefit of hardware architecture research. The impedance mismatch with an off-chip DDR RAM could be turned into interesting opportunities, such as emulating multiple smaller RAMs or second-level caches. Other future work of interest to us is on how to use hard DSP blocks optimally to combine all fixed-point and floating-point operations onto the same DSPs. Using specialized cores for floating point or even quicksort can push the speed limits higher in hardware multicore emulation. For the design and implementation of a memory directory [5] that could enable the use of the complete 4-FPGA infrastructure, using a higher level language e.g. Bluespec, which has been successfully used by other architecture research groups could help improve the productivity.

2.5.1 Publications

Published work related to the material in this chapter can be found in:

- Nehir Sönmez, Oriol Arcas, Gokhan Sayilar, Adrian Cristal, Ibrahim Hur, Osman Unsal, Satnam Singh and Mateo Valero, "From Plasma to Bee-Farm: Design Experience of an FPGA-based Multicore Prototype", Proc. 7th International Symposium on Applied Reconfigurable Computing (ARC 2011), Belfast (United Kingdom), March 2011.

- Oriol Arcas, Nehir Sönmez, Gokhan Sayilar, Satnam Singh, Osman Unsal, Adrian Cristal, Ibrahim Hur and Mateo Valero, "Resource-bounded Multicore Emulation Using Beefarm", to appear in Elsevier MICPRO Journal, 2012.

Chapter 3

TMbox: A Flexible and Reconfigurable Hybrid Transactional Memory System

Abstract

In this chapter, the design and implementation of TMbox is presented. The TMbox is an MPSoC built to explore trade-offs in multi-core design space and to evaluate recent parallel programming proposals such as Transactional Memory (TM). It is based on the Honeycomb core described in the previous chapter, and can fit a 16-core Hybrid Transactional Memory implementation based on the TinySTM-ASF proposal on a Virtex-5 FPGA. The flexible system, comprised of MIPS R3000 compatible cores interconnected in a ring network, is easily modifiable to study different architecture, library or Operating System extensions. In this chapter, three benchmarks written to investigate TM trade-offs are accelerating using hardware TM support in TMbox.

3.1 Introduction

FPGAs have already proved useful to study topics in computer architecture, however very little work exists in TM implementations, and none implementing Hybrid TM, which engages both STM and HTM concepts to have the best of both.

Additionally, the shared bus of the BeeFarm STM had some problems: it was implemented with long and wide wires that tighten the placement and routing constraints. It consequently did not allow more than 8 processors to share the same crossbar at reasonable frequencies. To ease the placement on chip, relax the constraints, we designed and implemented a bi-directional ring for our infrastructure.

3.1.1 Contributions

The main objectives are providing our system with HTM and finally Hybrid TM support, and fitting a greater number of cores using a more FPGA-friendly ring network. The software stack is similar to that of the BeeFarm, presented in the previous chapter. More specifically, our contributions in this chapter 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 emerging topics like TM.
- We detail on how we construct a multi-core with MIPS R3000 compatible cores, interconnect the components in a bi-directional ring with backwards invalidations and adapt the TinySTM-ASF hybrid TM on our infrastructure.
- We present experimental results for three TM benchmarks designed to investigate trade-offs in TM.

The next section presents the TMbox architecture, Section 3.3 explains the Hybrid TM implementation, Section 3.4 the results of running three benchmarks on TMbox and Section 3.5 sketches a 4-FPGA version of TMbox for the BEE3

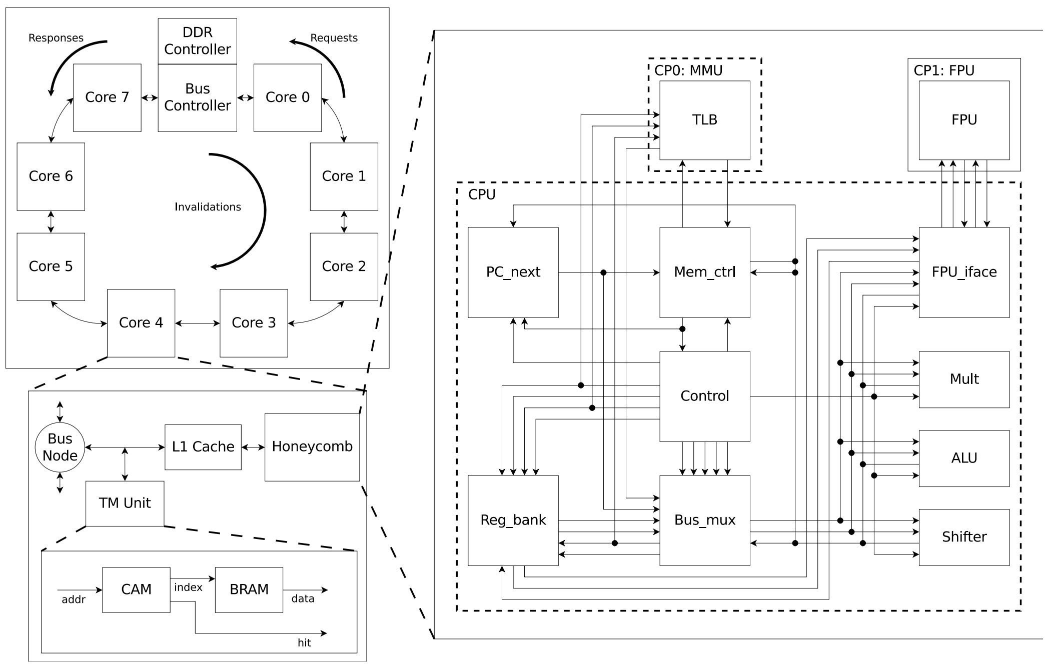


Figure 3.1: An 8-core TMbox infrastructure showing the ring bus, the TM Unit and the processor core.

platform. Related work can be found in Section 3.6, and Section 3.7 concludes this chapter.

3.2 HybridTM on TMbox

The TMbox system features the best-effort Hybrid TM proposal ASF [24], which is used with TinySTM [44], 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 4.1). 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.

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 [122]. However, TinySTM supports switching between eager and lazy commit and locking schemes as we will look into with software transactions later in Section 4.4.2.

The R3000-compatible CPU cores used in the TMbox are based on the Plasma core. They feature a 3-stage pipeline, coherent, direct-mapped, write-through L1 caches of 8 KB, extra instructions to support exceptions and thread synchronization (load-linked and store conditional), extensions to the MIPS ISA to support Hardware, Software and Hybrid TM execution and extensive debugging facilities. The system can address up to 4 GB of DDR2 RAM and features libraries for memory allocation, I/O and string functions.

3.2.1 The bi-directional ring bus

The shared bus of the BeeFarm caused tough placement and routing constraints, as well as disallowing to fit more than 8 soft processors on the FPGA. To ease the placement on chip and to relax the constraints, a bi-directional ring network was implemented, as shown in Figure 3.1. It is a simple and efficient design choice to diminish the complexities that arise in implementing a large crossbar on FPGA fabric. To interconnect the processor cores, arranging the components on a ring rather than a shared bus requires shorter wires which allows for an FPGA-friendly implementation: Short wires ease the placement on the chip and relax timing and routing constraints. 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. Debugging is also easier since messages only flow in one direction.

The non-TM implementation of the ring bus can process two types of requests. *READ_REQ*, *WRITE_REQ* are simple memory requests, and the backwards invalidation channel deals with *SNOOP_INV*.

CPU requests move counterclockwise; they go from the cores to the bus controller, eg. CPU_i - CPU_{i-1} - ... - CPU_0 - *BusCtrl*. Requests may be in form of read or write, 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. Memory responses also move in the same direction; from the bus controller to the cores, eg. *BusCtrl* - CPU_n - CPU_{n-1} - ... - CPU_{i+1} - CPU_i. They use the same channel as requests, carrying responses to the read requests that are served by the DDR Ctrl.

On the other hand, moving clockwise are backwards invalidations caused by the writes to memory, which move from the Bus Ctrl towards the cores in the opposite direction, eg. *BusCtrl* - CPU_0 - ... - CPU_{i-1} - CPU_i. These carry only a 32-bit address and a CPU ID field. When a write request meets an invalidation to the same address on any node, it gets cancelled. Moreover, the caches on each core snoop and discard the lines corresponding to the invalidation address, providing system-wide cache coherency. We detail how we extend this protocol for supporting Hybrid Transactional Memory with new messages in the next section.

As an example to demonstrate how the backwards invalidations work in TMbox, Figure 3.2 contains 4 steps:

1. CPU 1 sends *WRITE_REQ* and CPU 2 sends *WRITE_REQ*, with both addresses equal.

2. *WRITE_REQ* (from CPU 1) reaches the DDR controller and a memory request is issued. Right away, a backwards *SNOOP_INV* is sent. Meanwhile, *WRITE_REQ* (from CPU 2) continues its travel on the ring bus.

3. At some point in the bus, possibly at the ring node in CPU 0, *SNOOP_INV* and *WRITE_REQ* (from CPU 2) will collide, and the write request will be destroyed since the addresses are the same.

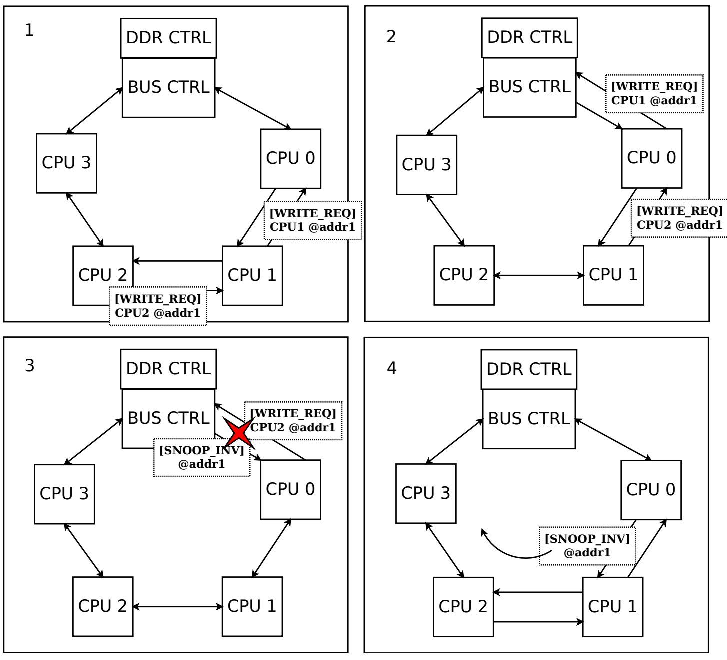


Figure 3.2: An invalidation example on 4 cores.

4. The backwards invalidation message will continue on its way, destroying messages and invalidating cache lines on all cores that belong to that address. When it reaches the other end of the ring bus (again in the DDR controller), it will be discarded.

Finally, CPU 2 will have to retry its write request, but will first have a miss in the cache, which implies that the updated cache line must be brought from memory. This is a typical issue of a write-through protocol, suggesting that using write-back caches might provide better performance.

3.3 Hybrid TM Support in TMbox

TinySTM-ASF is a hybrid port that enables TinySTM to be used with AMD's HTM proposal, ASF [24], which we modified to work with TMbox. Our hardware design closely follows the ASF proposal with the exception of nesting support.

Instruction	Description
`XBEGIN (addr)`	Starts a transaction and saves the abort handler routine (addr) in TM register $TM0. Also 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 handler.
`XABORT (20-bit)`	Used by software to explicitly abort the transaction. Sets TM register $TM2 to ABORT_SOFTWARE, restores the $sp register and jumps to the abort handler. The 20-bit code is stored in the TM register $TM3.
`XLW, XSW`	Transactional load/store of words (4 bytes).
`XLH, XSH`	Transactional load/store of halfwords (2 bytes).
`XLB, XSB`	Transactional load/store of bytes.
`MFTM (reg, TM_reg)`	Move From TM: Reads from a TM register and writes to a general purpose register.

Table 3.1: HTM instructions for TMbox

This version starts the transactions in hardware mode and jumps to software if (i) hardware capacity of the TM Unit is exceeded, (ii) there is too much contention, causing many aborts, or (iii) the application explicitly requires it, e.g. in case of a system call or I/O inside of a transaction.

To enable hardware transactions, we extended our design with a per-core TM Unit that contains a transactional cache which only admits transactional loads and stores. By default, it has a capacity of 16 data lines (256 bytes). If the TM cache capacity is exceeded, the transaction aborts and sets the TM register $TM2 to ABORT_FULL (explained in the next section), after which the transaction reverts to software mode and restarts.

The transactions are first started in hardware mode with a *X*BEGIN instruction. A transactional load/store causes a cache line to be written to the special TM Cache. *X*COMMIT 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. Modifications made to the transactional lines are not sent to memory until the whole transaction successfully commits. The TM

Unit provides single-cycle operations on the transactional read/writeset stored inside. A Content Addressable Memory (CAM) is built using LUTs both 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. Additionally, the TM Unit can serve LD/ST requests on an L1 miss if the line is found on the TM cache.

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.

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 committing and locking schemes with software transactions.

3.3.1 Instruction Set Architecture Extensions

To support HTM, we augmented the MIPS R3000 ISA with the new transactional instructions listed in Table 3.1 and in Appendix A. 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, eg. 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. Load Linked and

```
    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     //increment 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 word
    ADDi  $8, $8, 1      //increment a
    XSW   $8, 0($a0)   //transactional ST word
    XCOMMIT          //if abort go to $ABORT
```

Figure 3.3: TMbox MIPS assembly for `atomic{a++}` (NOPs and branch delay slots are not included for simplicity).

Store Conditional instructions can be used simultaneously with TM instructions, provided that they do not access the same address.

Figure 3.3 shows a transactional atomic increment operation 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).

3.3.2 Bus Extensions

To support HTM, two new bus messages were introduced. A new type of request, namely *COMMIT_REQ*, and a new response type, *LOCK_RING* were added to the TMbox. When doing a commit, the CPU sends a *COMMIT_REQ* request. When it reaches the DDR controller, it generates a backwards *LOCK_RING* mes-

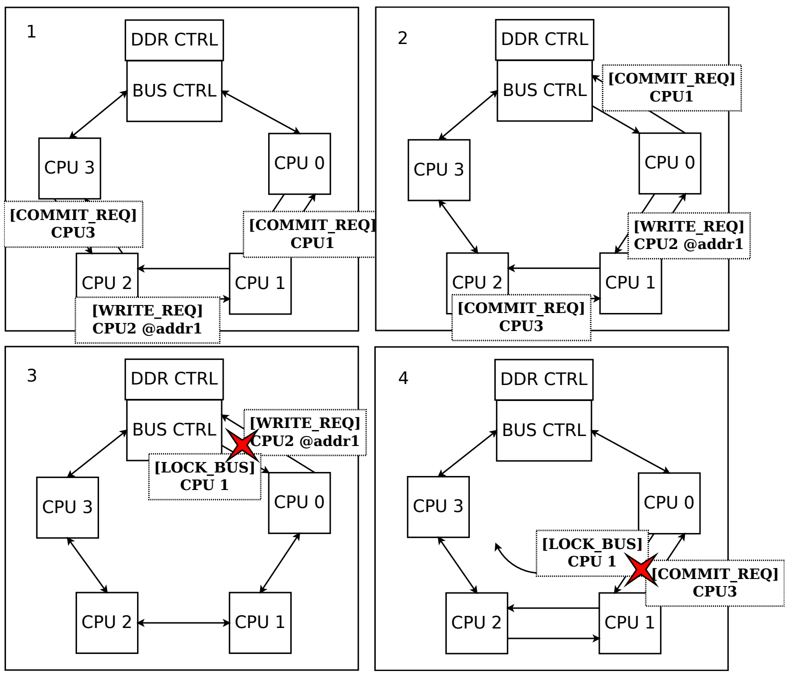


Figure 3.4: A commit example on 4 cores.

sage that will prevent any other CPU to send any write until a new *LOCK_RING* is received. Thus, a committing CPU can send *COMMIT_REQ*, receive a backwards *LOCK_RING* and be sure it has a safe "channel" to send all the transactional writes to memory, and finally send the "closing" *COMMIT_REQ* to trigger a second *LOCK_RING* message, which would inform all CPUs that the commit is completed and they are free to resume normal operation and send write requests (Between two backwards *LOCK_RING* messages, *READ_REQ* messages can still be sent). As with regular writes, *LOCK_RING* will destroy any *COMMIT_REQ* or *WRITE_REQ* request it founds in the bus, to prevent other CPUs to write or start commits during commit time. More efficient schemes can be supported in the future to enable parallel commits [20]. A TM example, as in Figure 3.4:

1. After some transactional accesses, CPU 1 sends *COMMIT_REQ*, while and CPU 2 sends *WRITE_REQ* and CPU 3 sends another *COMMIT_REQ*.

2. *COMMIT_REQ* (from CPU 1) will reach the DDR controller, creating a backwards *LOCK_RING* message. *WRITE_REQ* (from CPU 2) and *COMMIT_REQ* (from CPU 3) will continue their way through the bus.

3. The backwards *LOCK_RING* message will destroy *WRITE_REQ* (from CPU 2) and *COMMIT_REQ* (from CPU 3). The CPUs will now avoid sending new *WRITE_REQ* messages.

4. When CPU 1 receives the *LOCK_RING* message (with its ID attached), it will have made sure that it can send its writes safely. In order to commit now, it will send bursts of *WRITE_REQ* to the DDR.

In the end, CPU 1 will send a second *COMMIT_REQ*, which will generate a second backwards *LOCK_RING* that will re-enable writes and commits by all CPUs.

3.3.3 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 LD/ST, uncached accesses are also supported, as shown in Figure 3.5. Cache misses first make a read request to memory to bring the line and to wait in *WaitMemRD* state. In case of a store, 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 operation 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.

While processing a transactional store inside of an atomic block, an incoming invalidation to the same address causes 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. On the other hand, data inconsistencies in TMbox are detected only during the transactional

Component	6-LUTs	Component	6-LUTs
PC_next	138	Mem_ctrl	156
Control	139	Reg_File	147
Bus_mux	155	ALU	157
Shifter	201	MULT	497
Pipeline	112	Cache	1985
TLB	202	TM_Unit	759
Bus_node	619	DDR_ctrl	1119
UART+Misc.	560	**TOTAL**	**6946**

Table 3.2: LUT occupation of components

execution, between *XBEGIN* and *XCOMMIT/XABORT* (eager conflict detection).

When the speculative data is being committed to memory, each transactional write committed causes an invalidation signal, which traverses the ring, aborting the transactions that already have those lines in the TM cache. So, once a transaction gets to the commit phase, it will certainly commit its speculative changes to memory. However, all other transactions are stalled in the current implementation. 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 and granting the exclusive access of the bus to the processor. Another duty is performing burst writes during 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 those TM Cache entries that are also found in L1 cache after a commit/abort. To enable this, in preparation for a new transaction, address entries that are read from the TM Unit are sent to L1 cache as invalidation requests, after which the TM cache is cleared.

3.3.4 Area and Compilation Overheads

Figure 3.2 shows the LUT usage of each component of the TMbox. The new TM Unit of 16 entries contributes to a 11% increase in core area. Setting the

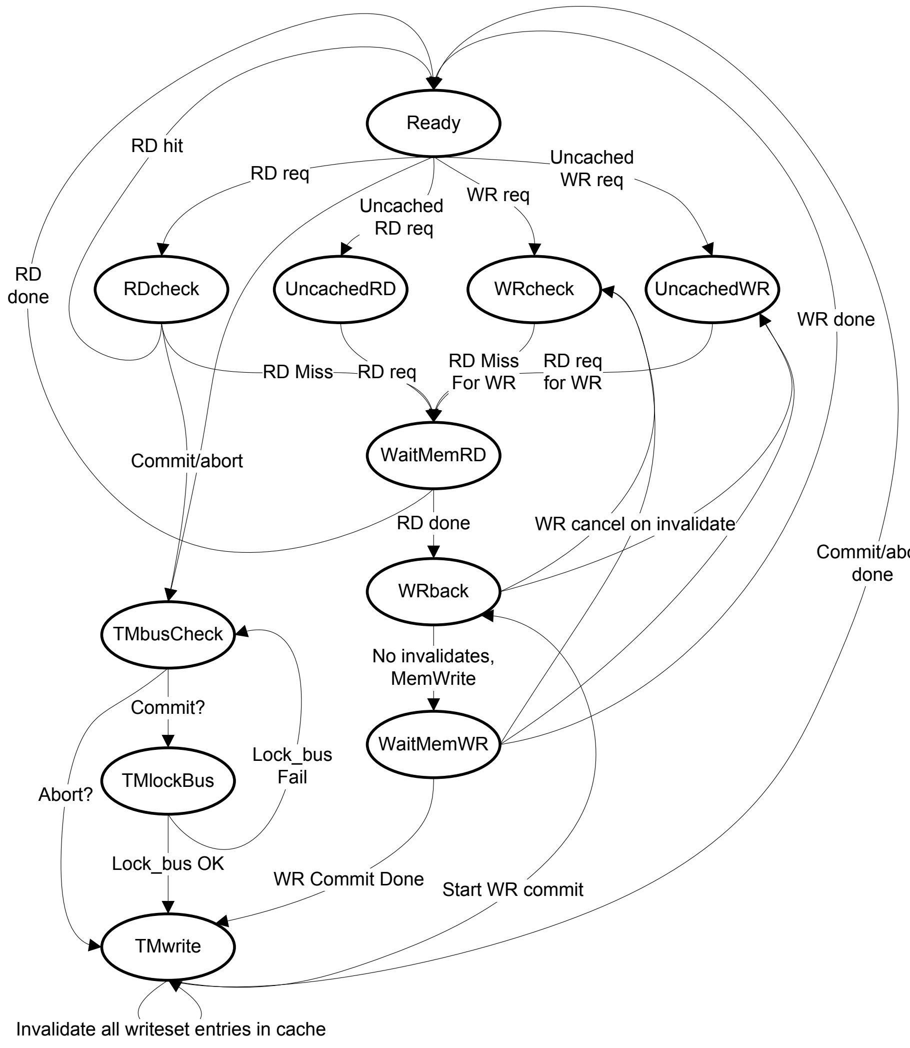


Figure 3.5: Cache state diagram. Some transitions (LL/SC) are not shown for visibility.

TMU Size	Slice Regs	Slice LUTs	BRAMs
16 entries	484	759	2
32 entries	964	1539	2
64 entries	1924	3593	2

Table 3.3: Area occupation of different TM Unit sizes

Design	Freq.(MHz)	Slice LUTs	Slice regs	#BRAMs
2-core BeeFarm	31.25	15,628	5,527	24
4-core BeeFarm	31.25	28,962	9,970	38
8-core BeeFarm	31.25	57,575	19,655	66
8-core TMBox	50	44,584	18,376	58
16-core TMBox	50	86,893	35,368	104

Table 3.4: BeeFarm and TMbox FPGA real estate usage

TM Cache size to 64 lines per core causes an extra usage of 2800 LUTs per core on average. Although bigger TM capacity is desirable, we opted for fitting more cores on the FPGA and kept the TM Cache size at 16 for our experiments. If the TM cache capacity is exceeded, the hardware transaction aborts since there is no way of completing in HW, it reverts to software and restarts there. Figure 3.3 describes the area occupation for different sizes of TM Unit, and how the LUT usage increases with more entries.

TMbox can fit 16 cores in a Virtex-5 FPGA, occupying 86,797 LUTs (95% of total slices) and 105 BRAMs (49%). Table 3.4 shows the FPGA resource usage for BeeFarm and TMbox designs, both without FPUs. We have significantly reduced the total LUT usage and increased the frequency of our soft core by switching to a ring network from a shared bus that did not map well on FPGA fabric. The difference of area also corresponds to a simplified per-CPU memory controller and a PC unit, and the TLB being turned off in the TMbox, all for being able to fit 16 cores in a synthesis-friendly way. Table 3.5 compares the compilation times of the BeeFarm and the TMbox. Profitable reductions in runtimes for synthesis and map considerably bring down the time-to-bitstream for the TMbox.

Design	**Synthesis**	**Map**	**PnR**	**#Routed signals**
2-core BeeFarm	6:44	8:30	4:09	93331
4-core BeeFarm	10:11	13:42	9:02	174677
8-core BeeFarm	44:33	28:32	20:09	348788
8-core TMBox	14:43	18:08	17:09	265055
16-core TMBox	29:35	41:44	44:45	518944

Table 3.5: BeeFarm and TMbox Compilation Times

3.4 Experimental Evaluation

In this section, we first examine the trade-offs of our implementation, and then discuss the results of executing three TM benchmarks. We used Xilinx ISIM and ModelSim for offline functional simulation and Chipscope Pro for real-time debugging, for which we apply various triggers and hardware debug registers. All results were obtained using 64-bit Xilinx ISE 12.2 running on RHEL5.

3.4.1 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 chapter we chose to use a small, 16-entry TM cache, larger caches can be supported on the TMbox on larger FPGAs, to accommodate for the extra area overheads 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 run to completion if there are larger transactions that do not fit in the TM cache, since there is no hardware or software mechanism to decide what to do in this case. The largest overhead related to STMs is due to keeping track of each transactional load/store 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, as a requirement of hybrid execution, TinySTM-ASF additionally keeps the lock variables inside the TM cache. This results in allowing a maximum of 8 variables in the read/write-sets 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 the same cache line. Demonstrating this 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, whereas it took more than two hours using a shared crossbar for interconnection and we could not fit more than 8 cores [123].

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 and to improve system performance.

On the processor side, the shallow pipeline negatively affects the operating frequency of the CPU. Furthermore, larger L1 caches that can not fit on our Virtex5 FPGA could be supported on larger, newer generation FPGAs, which would help the system to better exploit data locality. Having separate caches for instructions and data might also be a profitable enhancement.

3.4.2 Experimental Results

Eigenbench [65] is a synthetic benchmark for TM mimicry that can be tuned to discover TM bottlenecks, with the ability to imitate transactional behavior in terms of number and size of transactions, read/write sets and many other orthogonal characteristics. As Figure 3.6 shows, the transactions in EigenBench with 2 read – 8 write 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 version with 4 read and 4 write variables fits in the TM cache and shows a clear improvement over STM.

TM Benchmark	Description
`Eigenbench [65]`	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 128 attacks.
`SSCA2 [93]`	An efficient and scalable graph kernel construction algorithm. We have used problem scale value 12.

Table 3.6: TM Benchmarks Used

In the SSCA2 results presented in Figure 3.7, we get an 1-8% improvement over STM because this benchmark contains small transactions that fit in the transactional cache. Although Intruder (Figure 3.8) is a benchmark that is frequently used for TM, it is not a particularly 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 a direct-update STM which has to undo all changes on each abort. This benchmark can not be run on pure HTM because it contains memory operations like malloc/free inside transactions that are complex to run under HTM and are currently not supported on TMbox. These 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 up to 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, the system performance is benchmark-dependent: Compared to the sequential versions, the TM versions can perform in the range of 0.2x (for Intruder) to 2.4x (for SSCA2). As for the scalability of the ring network, SSCA2 results suggest

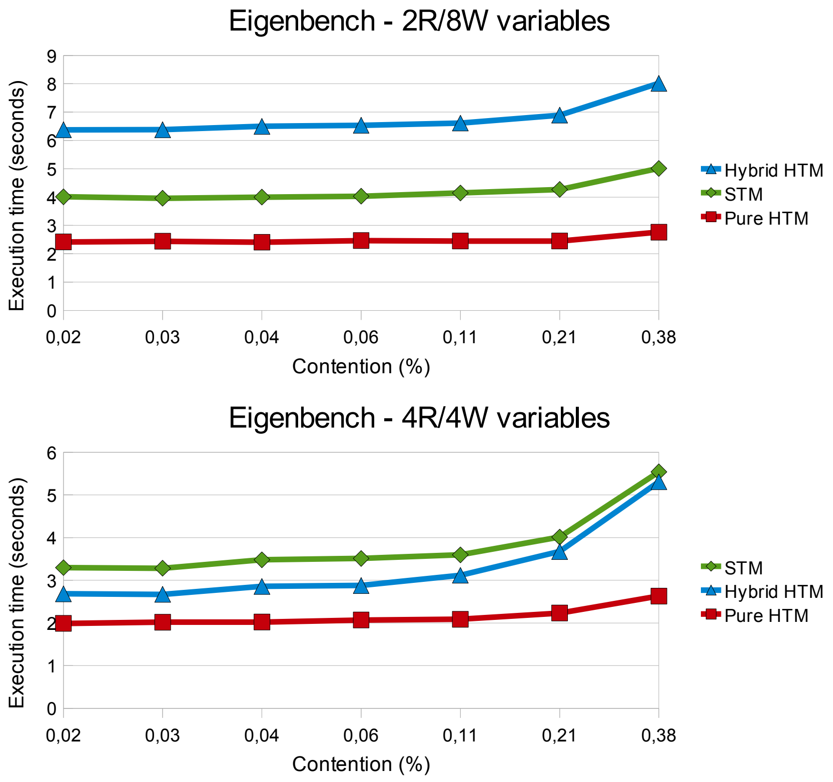


Figure 3.6: Eigenbench performance on a 16 core TMbox.

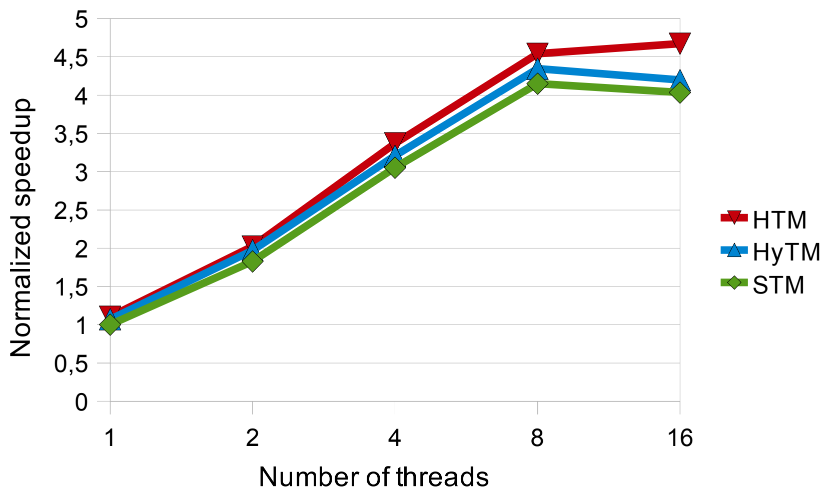


Figure 3.7: SSCA2 benchmark performance on 1–16 cores.

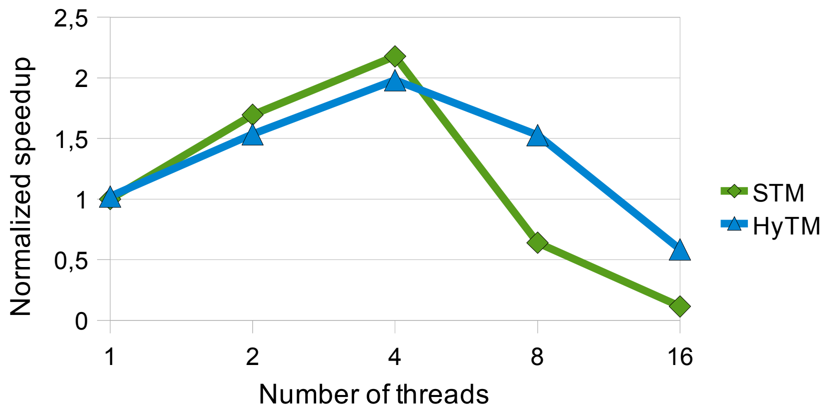


Figure 3.8: Intruder benchmark performance on 1–16 cores.

that up until 8 cores, the TMbox nicely scales[1]. However for the 16 core setup, there is a decrease in scalability, therefore with these frequencies, an even larger ring is probably not scalable and not advisable. In the next section, we sketch a multi-ring setup to be able to fit tens of Honeycomb cores inside the 4 FPGAs of the BEE3 platform.

Various design choices made can alter conclusions, especially for TM. The cache sizes, memory latencies, using write-through caches versus write-back caches, TM unit sizes, effects of the ring bus are all factors that have an influence on overall performance. For this reason, without an accurate timing model in place, we do not make conclusions about TM performance in this thesis. We do, however, present a complete infrastructure that enables running, examining, profiling and visualizing STM, HTM and Hybrid TM benchmarks with high detail. The profiling and visualization infrastructure for TMbox is presented in the next chapter.

3.5 Fitting inside the BEE3

To use all four FPGAs on the BEE3 infrastructure and to be able to use tens of cache coherent Honeycomb cores, a 4-FPGA version has been designed and implemented. As Figure 3.9 shows, in this design, there is a single DDR channel located on FPGA 0, which is accessed by an arbiter (similar to BeeFarm, Chapter

[1]Thanks to having faster memory than processors.

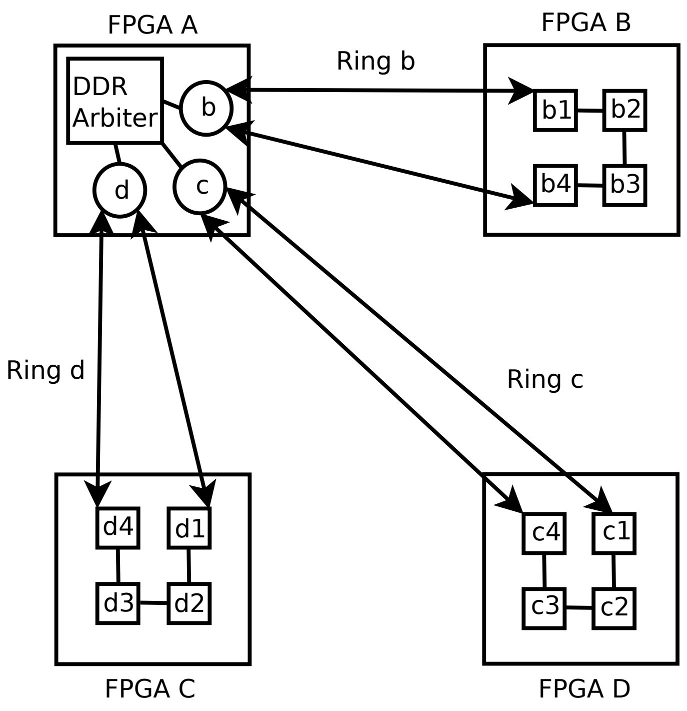


Figure 3.9: A 4-FPGA setup for the BEE3.

6.3 of [9]) that selects which ring to serve among the 4 rings that make memory requests. One of the rings (not shown in figure) resides on the same chip as the DDR controller, and the other three rings are on remote FPGAs. By using such a tree network, we can distribute the latency impact that a large a 40-node ring would bring, onto smaller rings of 10 nodes each. If necessary, the designer can slow down the local ring to match the latencies of the remote rings, or a scheduler could be devised to intelligently decide which cores to engage at runtime.

This version simulates correctly but fails to run properly on the actual chip. This is due to the actual wires between the FPGAs and their latencies not being simulated accurately. Therefore, we have to resort to online debugging using Chipscope [69]. This is when things worsen: For online debugging, various triggers must be engaged on different FPGAs at the same time, hence a graduate student's nightmare! Eventually, since we can only simulate, we only briefly discuss this implementation.

Figure 3.10 depicts the arbiter picking requests between two rings. Ring0 resides on the local FPGA, FPGA A, and Ring1 is on a remote FPGA B. When the

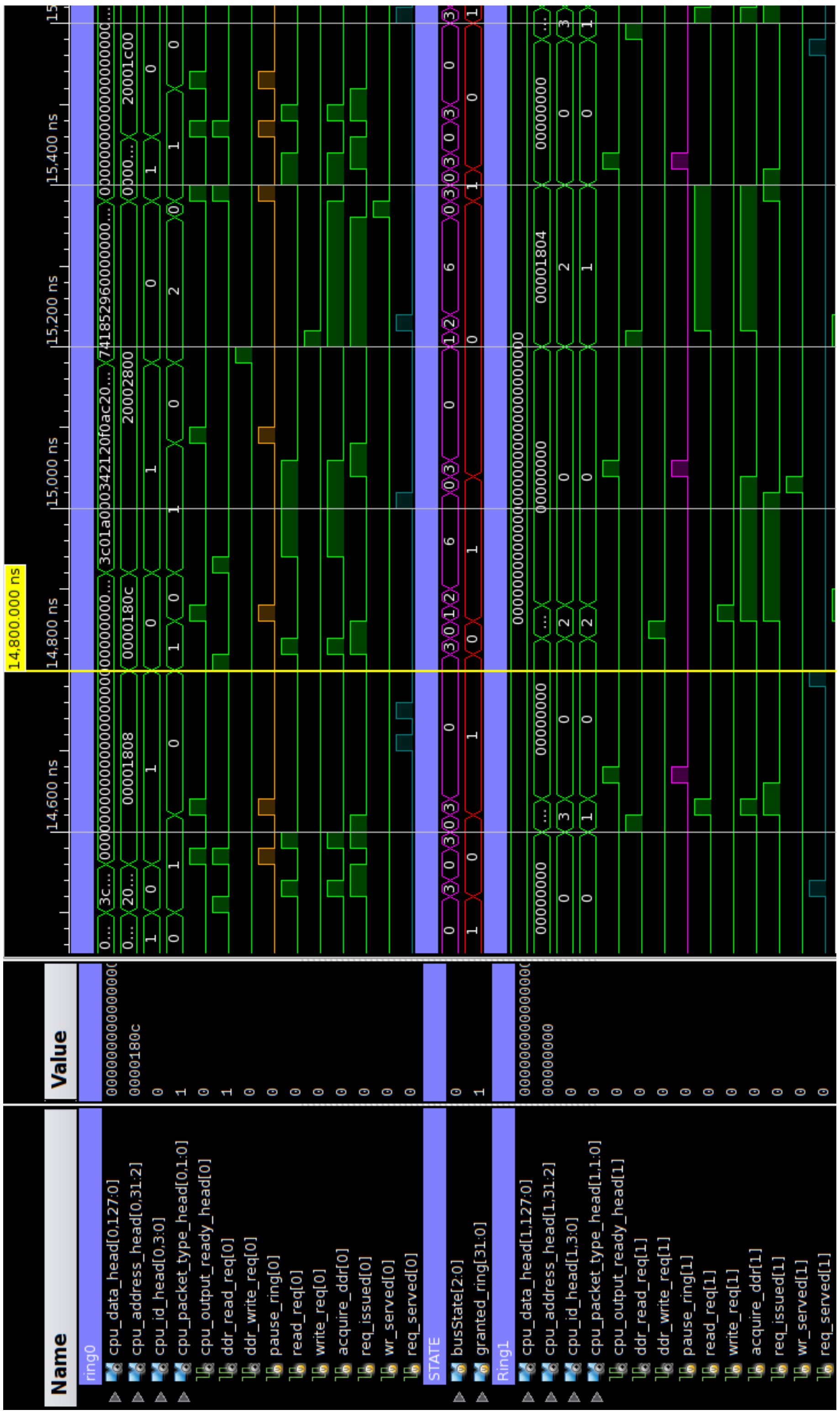


Figure 3.10: A waveform simulation of a 2-ring arbiter.

cpu_packet_type_head[x] has a positive value, there is a new packet arriving in the queue, representing a memory request from FPGAx (shown by the *ddr_read_req* and *ddr_write_req* signals). The *busState* FSM (under the sub-group *STATE* in Figure 3.10) selects the input source (dictated by the *granted_ring signal*) and moves to a corresponding state, depending if it's a DDR read/write (states 3 and 6) or a *BootMem* read. Accordingly, *acquire_ddr*, *req_issued* and finally *req_served* signals are set. Finally, when the *pause_ring* signal is set, the packet is dequeued. In this version, state 6 takes long because it is also responsible for first sending the invalidation signals to remote FPGAs, then getting an acknowledgement, at each write to the DDR. This was necessary for the invalidations and the writes not to cross from one FPGA to another at the same time, since invalidations have to cancel all write messages that they find on the way.

Another possibility of using 4 FPGAs would be to design and implement a distributed shared memory multiprocessor with a directory scheme that will provide system-wide memory coherency. Each FPGA would have a quarter of the memory space and a directory that is responsible for local addresses and a proxy directory with information about the lines from the local memory that are cached on remote nodes. On a miss or write, all node-local directories would be consulted and updated. If the status of a local-line changes, a message will be sent around the 4-node ring network that covers the BEE3, so that all remote proxy directories receive a notification.

3.6 Related Work

Some mostly initial work has 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, buffers for read/write sets and per-CPU caches augmented with transactional read-write bits and TCC-type HTM support, with a ninth core for running Linux and serving OS requests from other cores [98].

Kachris and Kulkarni describe a TM implementation for embedded systems which can work without caches, using a central transactional controller on four Microblaze cores [75]. TM is used as a simple synchronization mechanism that can be used with higher level CAD tools like EDK for non-cache coherent embedded MPSoC. The proposal occupies a small area on chip, but it is a centralized solution that would not scale as we move up to tens of cores. Similarly, the compact TM proposal, composed by off-the-shelf cores with a software API managing transactions, can be useful for early validation of programs to TM [111].

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 [84]. Application-specific signatures are compared to detect conflicts in a single pipeline stage. The design takes little area, reducing false conflicts. The underlying bit-level parallelism used for signatures makes this approach a good match for FPGAs. This proposal was the first soft core prototype with HTM, albeit only with 2 cores; it is not clear what is done in case of overflow or how the design would scale. Another approach is TMACC, which accelerates STMs on commodity machines and uses Bloom filters implemented on FPGAs to do so [18].

Ferri et al. proposed an energy-efficient HTM on a cycle-accurate SW simulator, where transactions can overflow to a nearby victim cache [45]. It is a realistic system with cache coherence, and non-centralized TM support, running a wide range of benchmarks on various configurations, however bus-based snoopy protocol would not scale with more cores, the simulator is not scalable and would suffer from modelling larger numbers of processors, and no ISA changes are possible to the ARM-based hard CPU core.

Recently, an HTM was proposed by C. Thacker for the Beehive system [128]. In case of overflow, the entire transaction is run under a commit lock without using the transactional hardware. We believe that software transactions might have more to offer. The Beehive design also uses a uni-directional ring where messages are added to the head of a train with the locomotive at the end [128].

Ring networks are suggested as a better architecture for shared memory multiprocessors by Barroso et al. [11] and a cache coherent bi-directional ring was presented by Oi et al. [101], but as far as we know, using backwards-propagating write-destructive invalidations is a novel approach. Unlike some of the proposals above, our system features a large number of processors and is completely modifiable. This enables investigating different interconnects, ISA extensions or coherency mechanisms.

3.7 TMbox Conclusions

We have presented a 16-core Hybrid TM design on an FPGA, providing hardware support and accelerating a modern TM implementation and running benchmarks that are widely used in TM research.

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. We believe that this is a good direction for further research.

We have also shown that a ring network fits well on FPGA fabric and using smaller cores can help building larger prototypes. Newer generations of FPGAs will continue to present multi-core researchers with interesting possibilities, having become so mature, as to permit investigating credible large-scale systems architecture. We are looking forward to extending the TMbox with a memory directory to utilize all four FPGAs on the BEE3 board.

TMbox enables to study many other topics such as shared memory vs distributed memory, message passing, homogeneous vs heterogeneous processing on different memory models, utilizing various interconnect architectures or ISA extensions. The TMbox is available at http://www.velox-project.eu/releases.

3.7.1 Publications

Published work related to the material in this chapter can be found in:

- Nehir Sönmez, Oriol Arcas, Otto Pflucker, Adrian Cristal, Osman Unsal, Ibrahim Hur, Satnam Singh and Mateo Valero, "TMbox: A Flexible and Reconfigurable 16-core Hybrid Transactional Memory System", In Proc.

19th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2011), Salt Lake City, USA, May 2011.

- Oriol Arcas, Nehir Sönmez, Osman Unsal, Adrian Cristal and Mateo Valero, “A Flexible Hybrid Transactional Memory Multicore on FPGA”, Jornadas de Paralelismo 2011, La Laguna, Tenerife (Spain), Sep 2011.
- Nehir Sönmez, Oriol Arcas, Osman S. Unsal, Adrian Cristal and Satnam Singh, “TMbox: A Flexible and Reconfigurable Hybrid Transactional Memory System”, book chapter to appear in Multicore Technology Architecture, Reconfiguration, and Modeling”, eds. S. Sangwine, M. Y. Qadri, CRC Press, 2012.

Chapter 4

Profiling and Visualization for TMbox

Abstract

Multi-core prototyping presents a good opportunity for establishing low overhead and detailed profiling and visualization in order to study new research topics. In this chapter, we design and implement a low execution, low area overhead profiling mechanism and a visualization tool for observing Transactional Memory (TM) behaviors on FPGA. To achieve this, we non-disruptively create and bring out events on the fly and process them offline on a host. There, our tool regenerates the execution from the collected events and produces traces for comprehensively inspecting the behavior of interacting multithreaded programs. With zero execution overhead for hardware TM events, single-instruction overhead for software TM events, and utilizing a low logic area of 2.3% per processor core, we run TM benchmarks to evaluate various different levels of profiling detail with an average runtime overhead of 6%. We demonstrate the usefulness of such detailed examination of SW/HW transactional behavior in two parts: (i) we port the STAMP application Intruder to Hybrid TM to speed it up by 24.1%, and (ii) we closely inspect transactions to point out pathologies such as repetitive aborts, killer transactions and starvation. The SW/HW profiling and event visualization infrastructure that we present offers possibilities of extension to many other directions.

4.1 Introduction

TM is attracting a lot of recent attention both in academia and in industry. Performance and scalability are both important for a successful 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.

The purpose of this chapter 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.

4.1.1 Contributions

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 [21]. 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# [144]. 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.

For gathering online profiling information, first we describe (i) the profiling hardware that supports generating TM-specific hardware events with zero execution overhead, and (ii) a key 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 [19] is engaged. The resulting profiling framework facilitates to visualize, identify and quantify TM bottlenecks: It allows getting insights into 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 TMbox system used, a Hybrid TM implementation on FPGA. Section 4.3 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 4.4 presents overhead results running TM applications and example traces illustrating the features of our profiling mechanism. Section 4.6 concludes the chapter.

4.2 Profiling 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. A design decision was to try to still support as many cores as possible on the FPGA by creating a system with a low overhead in terms of look-up tables and flip-flops used. The extensibility of the designed system should enable future projects to easily implement additional monitoring techniques.

Since the profiling infrastructure will be designed on actual hardware, we can not implement unrealistic behavior, and the new circuitry has to map well on the reconfigurable fabric, with minimal area and routing 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, utilize the idle cycles on an already-existing network. To be absolutely non-intrusive, give higher priority to real packets, buffer the event packets and send them only when there is no traffic.

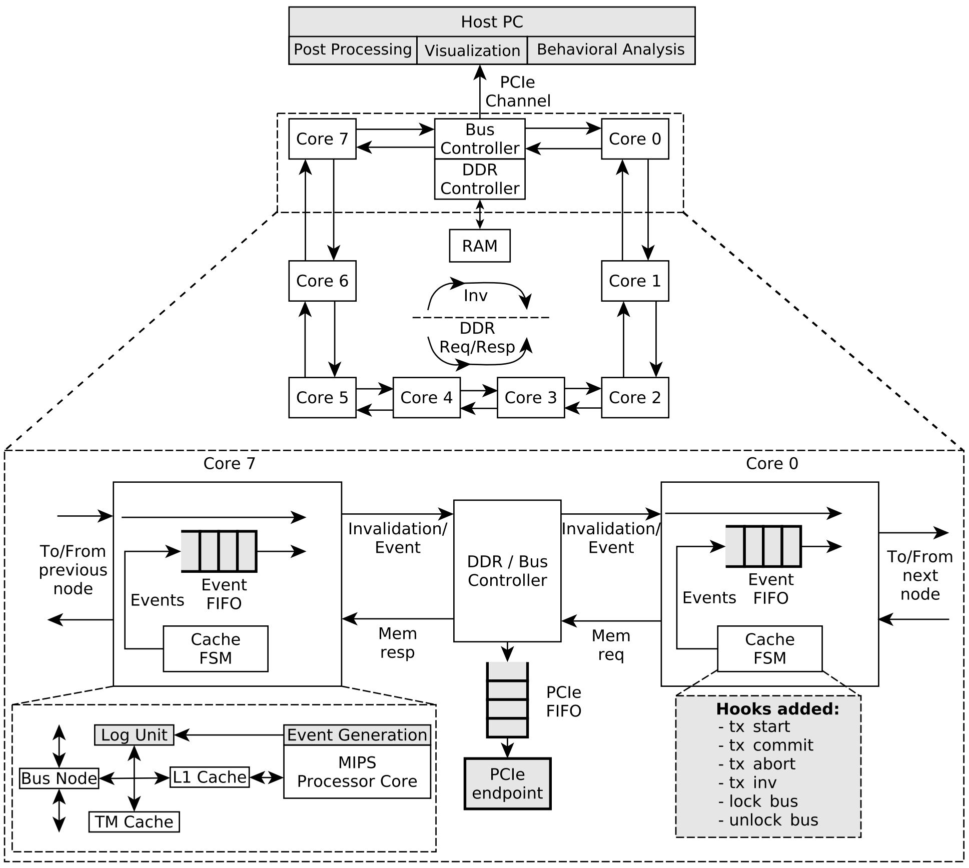


Figure 4.1: An 8-core TMbox system block diagram and modifications made to enable profiling (shaded).

4.2.1 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,

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 4.2: Event format for the profiling packets.

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 new message format size to support event propagation is bounded by the fixed message type of the invalidation ring bus. Another drawback is watching out for event 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.

4.3 The Profiling and Visualization Infrastructure for TMbox

The profiling and visualization framework developed 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 Bus Controller 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 recomposes them back to meaningful, threaded events with absolute timestamps, and creates the Paraver trace of the complete application, ready for visualization.

4.3.1 Event specification and generation

4.3.1.1 HTM events

The event generation unit (Figure 4.1) 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 4.2 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, 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.

4.3.1.2 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.

4.3.2 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 4.1.

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 (as shown in Figure 3.7 in [79]). This is a 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.

4.3.3 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.

To accommodate for the possible increases in profiling data bandwidth, we placed a large FIFO on the output of the PCIe. The limitations of this buffer can also be examined, since the size depends on maximum traffic, however this is left as future work, as in our experiments we didn't detect any FIFO overflows. Furthermore, using faster connections to the host could alleviate this problem.

4.3.4 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 [19]. 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.

4.4 Experimental Evaluation

In order to demonstrate the low overhead benefits of the monitoring framework proposed, we ran STAMP [93] applications using Eigenbench [65], 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 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

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

Profiling Type	**Area Overhead (per CPU core)**	**Actions Tracked**
STM-only (x86 host)	NONE	SW start tx, SW commit tx, SW abort tx
STM-only	32 6-LUTs + 1 BRAM	SW start tx, SW commit tx, SW abort tx
HTM-only	129 6-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 6-LUTs + 1 BRAM	HTM-only + STM-only
Hybrid TM (FG1)	129 6-LUTs + 1 BRAM	Hybrid TM (CG) + SW tx r/w + tx ID
Hybrid TM (FG2)	129 6-LUTs + 1 BRAM	Hybrid TM (FG1) + SW inv + SW PC

Table 4.1: Area overhead per processor core and the tracked events in different profiling options

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 4.1). 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

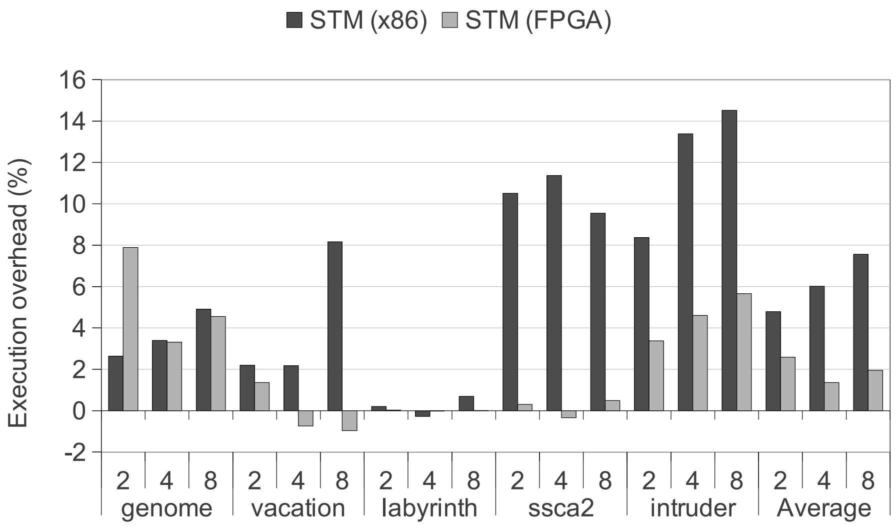


Figure 4.3: Runtime overhead (%) for STM (x86) vs. STM (FPGA), in different core counts and applications. (avg. 20 runs)

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) 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.

4.4.1 Runtime and area overhead

In Figure 4.3, 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).

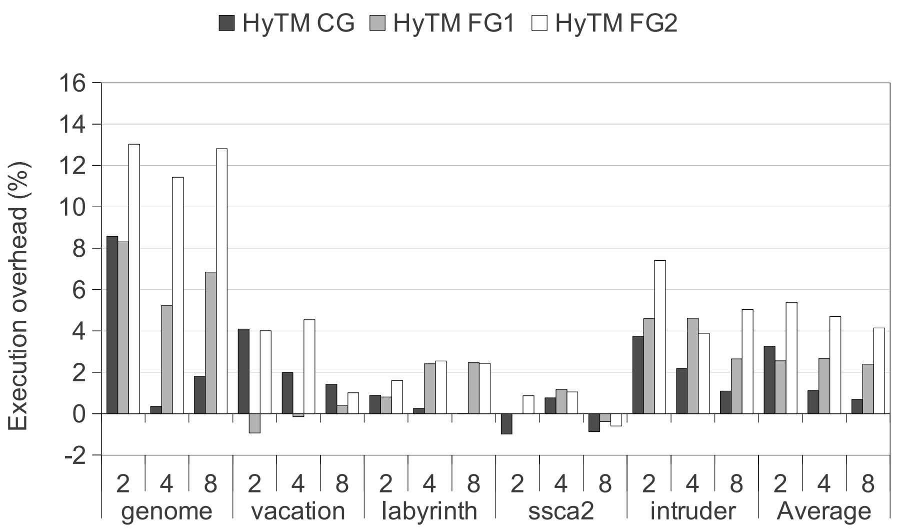


Figure 4.4: Runtime overhead (%) for different Hybrid TM profiling levels, core counts and applications. (avg. 20 runs)

Figure 4.4 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, the Genome benchmarks has many transactions that can never fit the dedicated TM hardware and exhibit higher overheads.

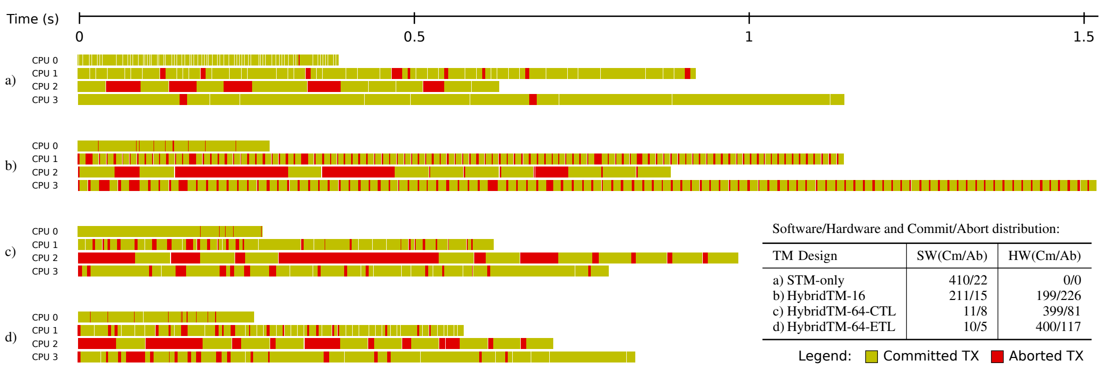
Interesting cases of low Hybrid TM CG overheads appear when running Genome with 4 threads and SSCA2 with 2 and 8 threads. This behavior is due to the specific Eigenbench parameters for STAMP benchmarks, eg. Genome’s parameter values for CPU 3 are huge and cause the application to behave much differently 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. The fixed area overhead of the PCIe endpoint plus the PCI_FIFO occupies 3978 LUTs and 30 BRAMs. For future work, it would be interesting 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 as negative values in the case of Vacation.

At first thought, the cross effects of profiling on TM might seem to cause an increased number of aborts. Since profiling delays execution, making the critical sections larger, the possibilities of conflict and aborts therefore are increased. However, since we are dealing with numerous interacting threads, it is not possible to make general statements on faster/slower execution or more/less aborts when the interaction of atomic blocks is altered. For example, with profiling enabled, two transactions that used to conflicting might correspond and interfere in a completely different way as to cease to conflict. Although using low overhead software events minimizes this effect, using too many events inside transactions might cause a larger deviation on the execution. Generally, it would be advisable to keep the software events at a minimum, and to omit having superfluous events.

4.4.2 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.



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 4.5: Improving Intruder-Eigenbench step by step from an STM-only version to utilize Hybrid TM appropriately.

4.4.2.1 Intruder-Eigenbench

Figure 4.5 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.

STM-only: Here, the TM application is profiled with FG1 STM 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. The transaction size depends on its read/write set and can be obtained simply by counting the event flags in Paraver or by using the post processing application. In FG1 level profiling we keep a count of the number of read/write events created inside each transaction.

Hybrid-TM-16: Introduces HTM with a 16-entry TM Cache per processor core, so this trace depicts both STM and HTM events. CPU 1 seems to have benefited from using HTM as shown in the table in Figure 4.5, 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 2 and 4 also start utilizing HTM efficiently, causing the software transactions to reduce in number. However, CPU 3 suffers from long aborting transactions and a huge 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, an 24.1% improvement in execution time when moving from STM-only to Hybrid TM-64-ETL was observed. This final version of Intruder is able to utilize both the TM hardware and the software TM options better. Although there are only 3 fewer aborts in software now, they cause much less wasted work [109], helping the application to run faster to completion.

Other than the Intruder benchmark, we also tried these experiments with a TM Cache that was 4x as large (64 entries) on the rest of the benchmarks. In general, we observed only a slight overall improvement. Eigenbench is not made with smaller Hybrid TM systems in mind (such as FPGA implementations); the small transactions fit well in the transactional hardware, but the larger ones are usually still too large. On our FPGA, it is not possible to have larger TM Caches (which grow up exponentially in size) with an interestingly high number of cores.

4.4.2.2 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.

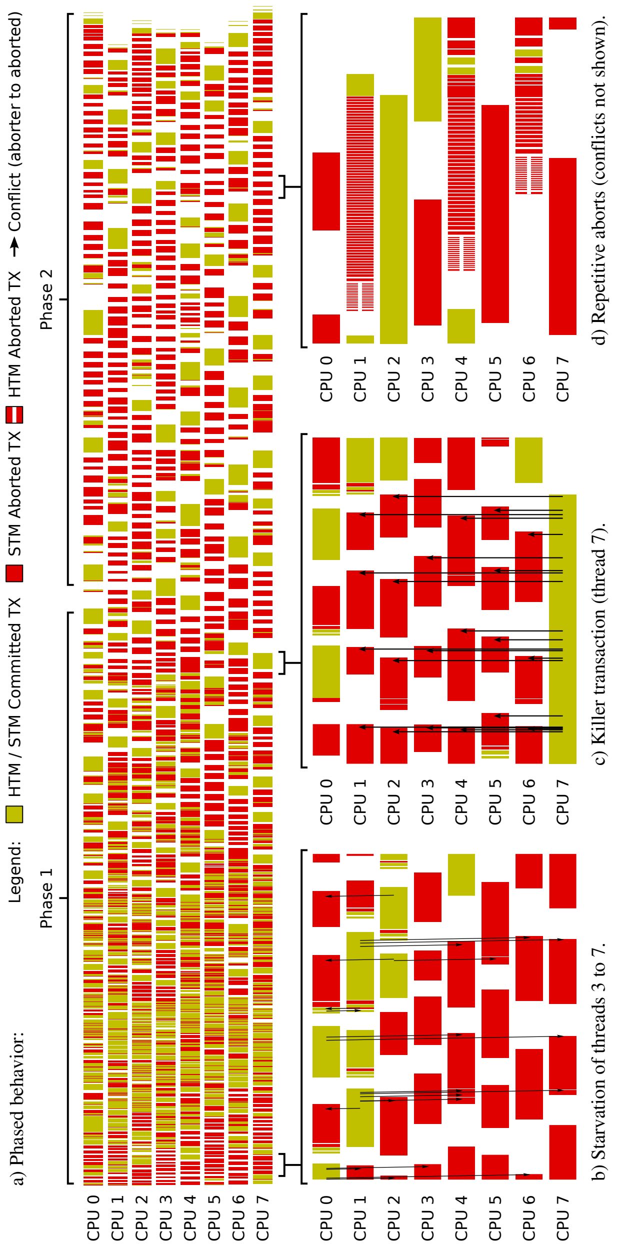


Figure 4.6: Example traces showing phased behavior and transactional pathologies in Intruder, a network intrusion detection algorithm that scans network packets for matches against a known set of intrusion signatures. The benchmark consists of 3 steps: capture, re-assembly, and detection. The main data structure in the capture phase is a simple queue, and the re-assembly phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session. It has high levels of contention due to the re-assembly phase rebalancing its tree.

Figure 4.6 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 [124] or guiding a contention manager that can take appropriate decisions:

- **Phased behavior:** An inherent property of real world benchmarks is phased behaviour. 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. Figure 4.6 shows an example where there are more transactions and parallelism in the first half, when 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, there are enough complete packets for the detector function, which generates less (but larger) transactions in number, which results in more aborts among them. Dynamic switching mechanisms would be suitable for treating adequately phased behavior in transactions.

- **Starvation:** A clear example of starvation on CPUs 3, 5, 6 and 7 (towards the beginning of the benchmark) is shown.

- **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.

- **Repetitive aborts:** Demonstrates the pathology of repetitive aborts and its effect on the execution, as in [15]. Finding the optimal abort threshold (to switch to STM mode) could be important in such cases.

Additionally, our infrastructure could also perform the following actions automatically:

Suggest HW/SW partitioning for transactions: Some transactions are more suitable to run in software (eg. those that include syscalls) and others in hardware (short transactions, or those that always fit the dedicated resources). By partitioning all transactions into HW and SW, we can avoid the wasted work caused by the transactions that are sure to abort in HTM mode (because of overflow, I/O, etc.).

Propose which locking/versioning strategy to use: CTL vs. ETL, or lazy versioning vs. eager versioning could be dynamically switched in flexible STM schemes. The application or the TM infrastructure can choose to use one mechanism over another, either statically or dynamically, by looking at how early the aborts happen, transaction sizes and other relevant data.

Debugging: By using software programmable `events`, the register values in hardware can quickly be brought up to the software layer and analyzed there. When these debugging events are enabled, they have to be lightweight and as non-disruptive as possible, which our low-overhead events accomplish.

Some of the suspected bottlenecks of the TMbox system were visualized and quantified with these analysis capabilities. Although the ring network is very suitable to map on FPGA fabric, it has some drawbacks, e.g. the cores closer to the main memory execute complete their writes more quickly, whereas the cores on the other end might be bound to finish executing last, an inherent attribute of a ring bus type core interconnect. An architecture-aware scheduling scheme for transactions might be beneficial for NUMA (Non-Uniform Memory Access) systems such as the TMbox.

Additionally, by modifying the application software and the post-processing application, and adding new 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 gives a better idea of which transactions are frequent aborters of which other transactions, which could help contention management schemes. Recently such advanced profiling was studied in the context of STMs in Java [7] and Haskell [109; 121].

As we have shown, an implementation with very low overhead on application runtime can be achieved, which causes minimal changes in application timing characteristics. With this new infrastructure in place, a complete cycle-accurate overview of the TM behavior of an application running on several processor cores can be obtained and analyzed [79]. A visual depiction of the runtime behavior of a TM program enables the viewer to easily identify application parts with different characteristics and to detect parts with sub-optimal behavior [8]. This can allow for specifically optimizing the poorly performing parts of the underlying TM system.

4.5 Related Work

FPGAs are excellent tools for the acceleration of full-system multiprocessor simulations [26; 34; 35]. The flexibility of FPGAs is beneficial for architectural exploration. In our approach, we have exploited the FPGA properties to design and implement a low overhead monitoring infrastructure for a hybrid TM platform. 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 [98]. It features buffers for read/write sets and per-CPU caches that are augmented with transactional read-write bits and. A ninth core runs Linux and serves OS requests from other cores. It does not have any profiling support. There also exist work on TM for embedded systems [75], and Bloom filter implementations to accelerate transactional reads and writes [84; 128]. They also lack support for profiling and visualization.

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 [75]. Pusceddu et al. present a single FPGA with support for Software Transactional Memory [111]. Ferri et al. propose an energy-efficient HTM on a cycle-accurate SW simulator, where transactions can overflow to a nearby victim cache [45].

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 [84]. TMACC [18] proposes the acceleration of transactional memory for commodity cores. The conflict detection uses Bloom filters implemented on an FPGA, which accelerates the conflict detection of the STM. Moderate-length transactions benefit from the scheme whereas smaller transactions do not.

The TM support for Beehive stores transactional data in a direct-mapped data cache and overflows to a victim buffer [128]. Bloom filters are also used for conflict detection. Damron et al. present Hybrid Transactional Memory (HyTM) [33], 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. The aforementioned systems lack a comprehensive support for the profiling of transactions. The existing work on TM profiling are discussed next.

Chung et al. gather statistics on the behavior of TM programs in hardware and describe the common case behavior and performance pathologies [15]. Ansari et al. present a framework to profile and metrics to understand Software TM applications [7]. 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. Zyulkyarov et al. offered an extensive profiling environment for Bartok STM in C# [143; 144]. Most of these papers are about STM. The general ideas in these papers are transferable from STM systems, but the specific implementation is quite different on HTM systems and therefore is not directly applicable.

The programming model of the underlying TMbox system [122] is comparable to the TCC model [54]. The monitoring techniques used in this work are in some parts similar to the TAPE [21] 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 [10]. SW techniques for profiling, targeting low overhead, have been researched [46; 97], alongside of OS support [141], and HW support for profiling [37; 142]. Further, techniques to profile parallel programs using message passing communication have been developed [120]. as well as an event-based distributed monitoring system for software and hardware malfunction detection, as described by Faure et al. [43].

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 can not 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. We have presented the first comprehensive environment to profile hybrid TM systems, with HW-only profiling through zero overhead profiling. Later, we have introduced Hardware-assisted profiling of SW executions and then merged it all into an analysis and visualization tool that enables profiling and optimization of TM programs.

4.6 TMbox Profiling 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. Furthermore, if a software simulator was used instead of FPGA emulation, the overheads to reach this much detail would have been high when simulating full programs.

Our infrastructure can be very useful to port applications to appropriately use Hybrid TM, as demonstrated with the Intruder benchmark to get a speedup of 24.1% compared to the STM-only version.

With the profiling and visualization infrastructure presented, it was possible to pinpoint many bottlenecks and pathological transactional behaviors that were previously published, with even lower overhead. Some of these include serialization, killer transactions and repetitive aborts. Depicting full multi-threaded executions, we can also examine phased behavior in transactional programs, portions where there is a high number of aborts, or series of long transactions. This information would be very useful in guiding a dynamically adaptable contention manager, so that the system could self-optimize depending on each workload or each phase.

We support almost all types of transactional profiling mentioned previously in the literature: per-atomic block profiling [121] or source code identification [144] using our flexible software events, as well as useful/wasted work analysis [109] or constructing conflict tables [121] in post-processing phase.

Furthermore, a single-cycle debugging instruction to read out any machine register or an internal value was provided. This can help to propagate very precise hardware information during the executions up to the software layer to be analyzed there. Disruptiveness is an advantage here, the program has to be altered as little as possible, which our low-overhead events accomplish.

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 [112], or speculative multithreading [82]. The event-based framework created in this chapter can be easily extended to enable the analysis of various processor core functionalities such as ALU, TLB and cache operations, locking behavior or memory access patterns, which can be useful for the construction of adaptive and self-optimizing systems.

4.6.1 Publications

Published work related to the material in this chapter can be found in:

- Nehir Sönmez, Adrian Cristal, Osman S. Unsal, Tim Harris, Mateo Valero "Why you should profile Transactional Memory Applications on an Atomic Block basis: A Haskell Case Study", Second Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG), Jan 2009, Paphos, Cyprus.

- Oriol Arcas, Philipp Kirchhofer, Nehir Sönmez, Martin Schindewolf, Osman S. Unsal, Wolfgang Karl and Adrian Cristal, "A low-overhead profiling and visualization framework for Hybrid Transactional Memory", In Proc. 20th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2012), Toronto, Canada, May 2012.

Chapter 5

Thesis Conclusions: Experience, Trade-offs and Future Trends

5.1 Experience and Trade-offs in Emulating Future Multicores

In this section, based on our experience of designing and building a complete FPGA-based multiprocessor emulation system that supports run-time and compiler infrastructure and on the actual executions of our experiments running Transactional Memory benchmarks, we comment on the advantages, drawbacks and future trends of using hardware-based emulation for research.

Firstly, we have observed several challenges that still face the computer architecture researcher that adopts FPGA-based emulation. These can be grouped in three: debugging FPGA designs, programmability and tools for FPGAs and limitations for computer architecture research.

- **Debugging FPGA designs:** Waveform simulators such as Modelsim are of indispensable help when designing and testing a circuit. However, in the example of working with a DDR controller that resides outside of the FPGA, a precise simulation model of the controller should be developed in HDL. In its absence, a fully-working simulation of a design can fail to run when loaded onto the actual FPGA. For such cases, certain alternative versions could also be developed, eg. one that substitutes the DDR with an on-chip memory of BRAMs to look for possible problems interfacing the DDR controller. Such 'extra' designs prove to be extremely useful for debugging off-the-chip interactions. Another important 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 debugging framework that is tailored to capturing information about multiprocessor systems.

Modifying slightly the online debugging core [69] ie. adding it a new signal to be inspected online also requires a complete re-do of Translate, Map and Place and Route, which can be very inconvenient. One solution to this problem might be to try to exploit floor planning tools (e.g. PlanAhead) or the explicit use of layout constraints (e.g. Xilinx's RLOCs) to place the processor cores and other components more carefully, which could significantly reduce the time-to-bitstream. Since actual hardware with area and timing constraints has to be designed when using FPGAs, mapping and placement issues are a lot more relevant compared to using software simulators.

- **Programmability and tools for FPGAs:** One of the biggest problems with employing large designs on FPGAs is that place and route times can be prohibitively long. Recently, synthesis tools have started to make use of the host multithreading capabilities, and to execute multiple hardware compilations in parallel for designs with tough timing constraints [72]. In the case of adding a simple counter to the design for observing the occurrence of an event, the re-synthesis, mapping, placing and routing and preparing the bitstream of an 8-core BeeFarm design takes more than an hour on our 8-core Intel server. The speed advantages of FPGAs should not get hindered by the slowness of the tools to program FPGAs.

  Other researchers have advocated the use of higher level HDLs to improve programmer productivity. We undertook our design in VHDL/Verilog and based on our experience, we would also consider moving to a higher level representation because the design effort and debug difficulty of working with a lower level language when producing a system for emulation, rather than production, is probably not worthwhile compared to a system that offers rapid design space exploration, e.g. Bluespec [67]. Most of the available designs ready for reuse at the time of writing are either in Verilog or VHDL.

- **Limitations for computer architecture research:** 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. Multi-FPGA usage can also bring extra complications. However, new FPGAs on 28 nm technology double the total number of gates available on chip, while allowing the designer to have access to megabytes of Block RAM capacity and thousands of DSP units. Such abundance of resources will be more suitable for building larger and faster prototypes on reconfigurable infrastructures.

  We also observe an impedance mismatch between the speed of the off-chip DDR memory, which runs much faster than the internal processing elements. Other than implementing an accurate timing partition, such as the pipelined and partitioned timing approaches [34; 125], this mismatch could also be exploited by using the fast external memory to model multiple independent smaller memories. This would better support architecture research where each processor core has its own local memory. Alternatively, one memory controller on each FPGA can be dedicated to model secondary-level caches, again subject to area constraints. Multi-ported memory research for FPGAs is another branch of investigation that can be very useful to overcome input-output port limitations [85].

5.2 Future Trends

- **FPGAs:** In the near future, FPGAs are expected to steadily grow larger in size and capacity, and faster in frequency, continuing the trends set by Moore's Law. We also expect and hope for faster connections with higher bandwidth, better interfaces and of course, better and faster tools. Furthermore, we expect more appropriate languages for representing architectures at a higher level than RTL descriptions, which are too detailed and tedious to deal with, especially when complex multiprocessors are designed.

New generation FPGAs will also offer a greater number of on-chip hard RAM and DSP blocks, as well as hard processor cores (such as ARM). These processor cores can be very useful for managing the executions or acting as a fast-access host computer that does not use up precious FPGA resources.

- **Open IP and tools:** Opencores [104] and the Open Graphics Project [100] can be very important catalysts for the development of other hardware emulation initiatives in the academia. The OpenFPGA consortium aims to standardize core libraries, APIs and benchmarks for FPGA computing [105]. On the other hand, using proprietary cores and hardware patents (as in MIPS unaligned memory access instructions [56]) can stop the investigators from implementing certain necessary functionalities and/or to look for workarounds. If we had chosen to utilize a different architecture, we could have been subject to such problems.

  While consortia such as RAMP, parameterizable NoCs such as the CONNECT [107] or heterogeneous multicore models such as the FabScalar [23] 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 already available IPs should be structuralized and standardized, so that their reuse can be more straightforward and commonplace.

- **Transactional Memory:** The future of Transactional Memory and of speculation hardware looks promising. Finally, we are seeing HTM to be implemented in more mainstream processors, although the first processor to have support for HTM, the Rock from Sun Microsystems, was cancelled [130]. The 48-core Vega2 chip from Azul systems uses HTM to accelerate Java TM applications. AMD already detailed their Advanced Synchronization Facility (ASF) HTM proposal for the x86 architecture, and more recently Intel described their new upcoming Haswell processor core with HTM support and IBM announced that the upcoming BlueGene/Q supports lazy-lazy TM [28; 29; 30; 57].

These designs can attract even more attention on TM. However, more research might be necessary to make improvements for managing system-wide contention in hardware. Under high contention, taking pessimistic locks [124], guiding a contention manager that can take appropriate decisions [118], or even predicting the outcome of conflicting transactions (similar to branch prediction) could provide with more efficient ways of executing transactions.

While computer architecture is drastically changing, heterogeneous computing promises a greater variety of architectures to be proposed, meanwhile requiring low power consumption. Techniques such as clock gating can also be perfectly employed on FPGAs, as well.

5.3 Thesis Conclusions

This thesis attempted to build an extensive tool to investigate novel multicore designs with Transactional Memory support by developing a flexible prototype called TMbox. The TMbox can fit up to 16 MIPS-compatible cores on a mid-sized Virtex-5 FPGA, supports STM, HTM and Hybrid TM, and has a useful visualization tool to profile entire execution streams and get feedback with low overhead. The FPGA emulator is shown to show better performance than the M5 software simulator, as well as scaling well as the core counts are increased.

Supporting and accessing a modified GCC MIPS toolchain, the MIPS-R3000-compatible TMbox runs TM benchmarks through the TinySTM-ASF Hybrid Transactional Memory infrastructure. However, it is not only a useful tool for studying TM: It can easily be modified to support other state-of-the-art shared memory hardware proposals such as speculative lock elision, runahead execution or speculative multithreading.

There are many ways to keep upgrading and improving the TMbox. A 64-bit datapath, a deeper pipeline, Chip Multi-Threading support, a prefetching mechanism or having larger caches with more levels are some of the most immediate ways of achieving this. Implementing a write-back cache coherence protocol or a directory structure might also improve the performance, although simpler cache coherency mechanisms or involving a greater number of on-chip hard blocks would keep the soft processor cores small in size.

TMbox, inside the guidelines of this thesis, is an open source hardware project. The design of the full HW/SW stack is free software (with small exceptions) and all instructions that are implemented are non-patented or expired. A timing partition that exists in software architectural simulators was not implemented for the TMbox. Various latency numbers can be devised and provided to the system, which can be made to access the memory and the caches and meet certain deadlines in specified amounts of clock cycles. TMbox also doesn't support a full OS, which can prove to be very useful to execute complex programs. This can be achieved by implementing more corner-case instructions required for porting Linux to TMbox-MIPS and supporting all necessary system calls. The TMbox is available online at http://www.velox-project.eu/releases.

We believe that studying computer architecture through FPGA-based hardware prototyping has a long and prosperous way to go. With larger FPGAs that are not affected by the limitations of Moore's Law (yet), many new designs of multi-soft cores can enjoy the assistance of embedded hard processor cores, more capable DSP blocks, larger block RAMs at faster frequencies.

We need more descriptive higher-level languages than VHDL and Verilog for the design of large MpSoCs for architectural prototyping and emulation. Some examples are Bluespec, Lava [13] and SystemC. We also believe that any research in this direction that would increase productivity and ease rapid design space exploration is absolutely worthwhile.

Appendix A

Honeycomb ISA

Table A.1: Honeycomb ISA: Arithmetic Logic Unit

Opcode	**Name**	**Action**	**Opcode bitfields (6+5+5+5+5+6)**					
ADD rd,rs,rt	Add	rd=rs+rt	000000	rs	rt	rd	00000	100000
ADDI rt,rs,imm	Add Immediate	rt=rs+imm	001000	rs	rt			imm
ADDIU rt,rs,imm	Add Immediate Unsigned	rt=rs+imm	001001	rs	rt			imm
ADDU rd,rs,rt	Add Unsigned	rd=rs+rt	000000	rs	rt	rd	00000	100001
AND rd,rs,rt	And	rd=rs&rt	000000	rs	rt	rd	00000	100100
ANDI rt,rs,imm	And Immediate	rt=rs&imm	001100	rs	rt			imm
LUI rt,imm	Load Upper Immediate	rt=imm<<16	001111	rs	rt			imm
NOR rd,rs,rt	Nor	rd=∼(rs—rt)	000000	rs	rt	rd	00000	100111
OR rd,rs,rt	Or	rd=rs|rt	000000	rs	rt	rd	00000	100101
ORI rt,rs,imm	Or Immediate	rt=rs|imm	001101	rs	rt			imm
SLT rd,rs,rt	Set On Less Than	rd=rs<rt	000000	rs	rt	rd	00000	101010
SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	001010	rs	rt			imm
SLTIU rt,rs,imm	Set On Less Than Imm. Uns.	rt=rs<imm	001011	rs	rt			imm
SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt	000000	rs	rt	rd	00000	101011
SUB rd,rs,rt	Subtract	rd=rs-rt	000000	rs	rt	rd	00000	100010
SUBU rd,rs,rt	Subtract Unsigned	rd=rs-rt	000000	rs	rt	rd	00000	100011
XOR rd,rs,rt	Exclusive Or	rd=rs∧ rt	000000	rs	rt	rd	00000	100110
XORI rt,rs,imm	Exclusive Or Immediate	rt=rs∧ imm	001110	rs	rt			imm

The immediate values are normally sign extended.

Table A.2: Honeycomb ISA: Shifter

Opcode	**Name**	**Action**	**Opcode bitfields (6+5+5+5+5+6)**					
SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	000000	rs	rt	rd	sa	000000
SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs	000000	rs	rt	rd	00000	000100
SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	000000	00000	rt	rd	sa	000011
SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000111
SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa	000000	rs	rt	rd	sa	000010
SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000110

The NOP (no operation) instruction is encoded as an opcode with all bits cleared, equivalent to "SLL r0,r0,0", which has no effect.

Table A.3: Honeycomb ISA: Multiply/Divide

Opcode	**Name**	**Action**	**Opcode bitfields (6+5+5+5+5+6)**					
DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt	000000	rs	rt		0	011010
DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt	000000	rs	rt		0	011011
MFHI rd	Move From HI	rd=HI	000000	0000000000		rd	00000	010000
MFLO rd	Move From LO	rd=LO	000000	0000000000		rd	00000	010010
MTHI rs	Move To HI	HI=rs	000000	rs	000000000000000			010001
MTLO rs	Move To LO	LO=rs	000000	rs	000000000000000			010011
MULT rs,rt	Multiply	HI,LO=rs*rt	000000	rs	rt	0000000000		011000
MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt	000000	rs	rt	0000000000		011001

Table A.4: Honeycomb ISA: Branch

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)					
BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	000100	rs	rt			offset
BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4	000001	rs	00001			offset
BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4	000001	rs	10001			offset
BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4	000111	rs	00000			offset
BLEZ rs,offset	Branch On	if(rs<=0) pc+=offset*4	000110	rs	00000			offset
BLTZ rs,offset	Branch On	if(rs<0) pc+=offset*4	000001	rs	00000			offset
BLTZAL rs,offset	Branch On	r31=pc; if(rs<0) pc+=offset*4	000001	rs	10000			offset
BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	000101	rs	rt			offset
BREAK	Breakpoint	epc=pc; pc=0x3c	000000				code	001101
J target	Jump	pc=pc_upper |(target<<2)	000010					target
JAL target	Jump And Link	r31=pc; pc=target<<2	000011					target
JALR rs	Jump And Link Register	rd=pc; pc=rs	000000	rs	00000	rd	00000	001001
JR rs	Jump Register	pc=rs	000000	rs			000000000000000	001000

Table A.5: Honeycomb ISA: Memory Access

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)			
LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)	100000	rs	rt	offset
LBU rt,offset(rs)	Load Byte Unsigned	rt=*(Uchar*)(offset+rs)	100100	rs	rt	offset
LH rt,offset(rs)	Load Halfword	rt=*(short*)(offset+rs)	100001	rs	rt	offset
LBU rt,offset(rs)	Load Halfword Unsigned	rt=*(Ushort*)(offset+rs)	100101	rs	rt	offset
LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)	100011	rs	rt	offset
SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	101000	rs	rt	offset
SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt	101001	rs	rt	offset
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011	rs	rt	offset

Table A.6: Honeycomb ISA: Misc

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)				
MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]	010000	00000	rt	rd	00000000000
MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt	010000	00100	rt	rd	00000000000
SYSCALL	System Call	epc=pc; pc=0x3c	000000	00000000000000000000			001100

Table A.7: Honeycomb ISA: Synchronization

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)					
LL rt,offset(rs)	Load Linked	rt=*(int*)(offset+rs); ll=1	100000	rs	rt			offset
SC rt,offset(rs)	Store Conditional	if(ll=1) *(int*)(offset+rs)=rt	101000	rs	rt			offset

Table A.8: Honeycomb ISA: Transactional Memory

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)					
XLB rt,offset(rs)	Transactional Load Byte	rt=*(char*)(offset+rs)	011000	rs	rt			offset
XLH rt,offset(rs)	Transactional Load Halfword	rt=*(short*)(offset+rs)	011001	rs	rt			offset
XLW rt,offset(rs)	Transactional Load Word	rt=*(int*)(offset+rs)	011010	rs	rt			offset
XSB rt,offset(rs)	Transactional Store Byte	*(char*)(offset+rs)=rt	011011	rs	rt			offset
XSH rt,offset(rs)	Transactional Store Halfword	*(short*)(offset+rs)=rt	011110	rs	rt			offset
XSW rt,offset(rs)	Transactional Store Word	*(int*)(offset+rs)=rt	011111	rs	rt			offset
XBEGIN	Transaction Begin	*(char*)(offset+rs)=rt; TM1=sp	110100	rs	00000	rd	00000	000000
XCOMMIT	Transaction Commit	*(short*)(offset+rs)=rt; sp=TM1	111100	rs	00000	rd	00000	000000
XABORT addr	Transaction Abort	TM3=imm; TM2=1	111100	rs				imm
MFTM rt,TMreg	Move From TMU	rt=TMreg	111100	rs	00000	rd	00000	000010

Table A.9: Honeycomb ISA: Profiling

Opcode	Name	Action	Opcode bitfields (6+5+5+5+5+6)					
EVENT imm	Create Event	*(int*)(offset+rs)=rt	011111	rs	rt			imm
JALL target	Jump And Link And Link	r31=pc; pc=target<<2	000011					target

Appendix B

Honeycomb Registers

Table B.1: Compiler Register Usage

Register	Name	Function
R0	zero	Always contains 0
R1	at	Assembler temporary
R2-R3	v0-v1	Function return value
R4-R7	a0-a3	Function parameters
R8-R15	t0-t7	Function temporary values
R16-R23	s0-s7	Saved registers across function calls
R24-R25	t8-t9	Function temporary values
R26-R27	k0-k1	Reserved for interrupt handler
R28	gp	Global pointer
R29	sp	Stack Pointer
R30	s8	Saved register across function calls
R31	ra	Return address from function call
HI-LO	lo-hi	Multiplication/division results
PC	Program Counter	Points at 8 bytes past current instruction
EPC	epc	Exception program counter return address
TM0		Keeps the abort address
TM1		Keeps the stack pointer(sp)
TM2		Keeps the reason for abort
TM3		Keeps the reason for SW abort

Appendix C

List of Abbreviations

Table C.1: List of Abbreviations (I)

Abbreviation	**Description**
ALU	Arithmetic Logic Unit
AMD	Advanced Micro Devices
API	Application Programming Interface
ARM	Advanced RISC Machines
ASF	Advanced Synchronization Facility
ASIC	Application-Specific Integrated Circuit
BEE3	Berkeley Emulation Engine 3
BRAM	Block Random Access Memory
CAD	Computer-Aided Design
CAM	Content-Addressable Memory
CAS	Compare And Swap
CMP	Chip MultiProcessor
CP	Coprocessor
CPU	Central Processing Unit
DDR	Double Data Rate
DSP	Digital Signal Processor
ECC	Error Correcting Code
EDK	Embedded Development Kit

Table C.2: List of Abbreviations (II)

Abbreviation	**Description**
FAA	Fetch And Add
FIFO	First-in First-Out
FP	Floating Point
FPU	Floating Point Unit
GAS	GNU Assembler
GCC	GNU Compiler Collection
GNU	GNU's Not Unix
HaSTM	Hardware-Assisted Software Transactional Memory
HTM	Hardware Transactional Memory
HW	Hardware
HyTM	Hybrid Transactional Memory
I/O	Input/Output
IP	Intellectual Property
ISA	Instruction Set Architecture
ISE	Integrated Synthesis Environment
ISIM	ISE Simulator
KB	Kilobyte
LL	Load-Linked
LUT	Look-Up Table
MFTM	Move From TM
MIPS	Microprocessor without Interlocked Pipeline Stages
MPSoC	Multiprocessor System-on-Chip
NOP	No Operation
OS	Operating System
RAMP	Research Accelerator for Multiple Processors
RD	Read
RFE	Return From Exception
RHEL5	Red Hat Enterprise Linux 5
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTL	Register-Transfer Level

Table C.3: List of Abbreviations (III)

Abbreviation	**Description**
SC	Store-Conditional
SoC	System-on-Chip
SPARC	Scalable Processor Architecture
SRAM	Static Random Access Memory
SSCA2	Scalable Synthetic Compact Applications 2
STM	Software Transactional Memory
SW	Software
TCC	Transactional Coherence and Consistency
TL2	Transactional Locking 2
TLB	Translation Lookahead Buffer
TLP	Thread-Level Parallelism
TM	Transactional Memory
TMU	Transactional Memory Unit
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very-high-speed integrated circuits Hardware Description Language
WR	Write
XABORT	Transaction Abort
XBEGIN	Transaction Begin
XCOMMIT	Transaction Commit
XLB	Transactional Load Byte
XLH	Transactional Load Halfword
XLW	Transactional Load Word
XSB	Transactional Store Byte
XSH	Transactional Store Halfword
XSW	Transactional Store Word

References

[1] Aeroflex Gaisler AB, *Leon3 processor*, http://www.gaisler.com/doc/leon3_product_sheet.pdf. 17

[2] ORSoC AB, *Openrisc 1200*, http://opencores.org/openrisc,or1200. 5

[3] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha, *Unlocking concurrency*, ACM Queue (2006), 24–33. iii

[4] 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*, PLDI, 2006. 11, 28

[5] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz, *An evaluation of directory schemes for cache coherence*, 25 years of the international symposia on Computer architecture (selected papers) (New York, NY, USA), ISCA '98, ACM, 1998, pp. 353–362. 37

[6] Hari Angepat, Dam Sunwoo, and Derek Chiou, *RAMP-White: An FPGA-Based Coherent Shared Memory Parallel Computer Emulator*, Austin CAS Conference, March, 2007. 9, 33, 34

[7] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Lujan, Chris Kirkham, and Ian Watson, *Profiling transactional memory applications*, Euromicro'09, 2009, pp. 11–20. 81, 83

[8] Oriol Arcas, Philipp Kirchhofer, Nehir Sonmez, Martin Schindewolf, Wolfgang Karl, Osman S. Unsal, and Adrian Cristal, *A low-overhead profiling and visualization framework for hybrid transactional memory*, Proc. 20th

Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2012) (Toronto, Canada), May 2012, pp. 1–8. 82

[9] Oriol Arcas Abella, *Beehive: an FPGA-based multiprocessor architecture*, Master's thesis, Universitat Politecnica de Catalunya, 2009. 14, 19, 57

[10] Thomas Ball and James R. Larus, *Optimally profiling and tracing programs*, ACM Trans. Program. Lang. Syst. (New York, NY, USA), vol. 16, ACM, July 1994, pp. 1319–1360. 83

[11] Luiz Andre Barroso and Michel Dubois, *Cache coherence on a slotted ring*, International Conference on Parallel Processing, 1991. 60

[12] 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*, MICRO, vol. 26, 2006, pp. 52–60. 6, 15, 29

[13] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh, *Lava: hardware design in haskell*, SIGPLAN Not. **34** (1998), no. 1, 174–184. 91

[14] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Commun. ACM **13** (1970), no. 7, 422–426. 35

[15] 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*, ACM SIGARCH CA. News, vol. 35, June 2007, pp. 81–91. 80, 83

[16] Jean-Louis Brelet, *XAPP201: An Overview of Multiple CAM Designs in Virtex Family Devices*, http://www.xilinx.com/support/documentation/application_notes/xapp201.pdf, 1999. 19

[17] N. L. V. Calazans, F.G. Moraes, K.B. Quintans, and F.B. Neuwald, *Accelerating sorting with reconfigurable hardware*, http://toledo.inf.pucrs.br/~gaph/Projects/Quicksort/public/corequick_ext_abs.pdf. 32

[18] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun, *Hardware acceleration of transactional memory on commodity systems*, ASPLOS '11, 2011, pp. 27–38. 60, 82

[19] Barcelona Supercomputing Center, *Paraver website*, `http://www.bsc.es/paraver`. v, 65, 72

[20] Hassan Chafi, Jared Casper, Brian D. Carlstrom, Austen McDonald, Chi Cao Minh, Woongki Baek, Christos Kozyrakis, and Kunle Olukotun, *A scalable, non-blocking approach to transactional memory*, HPCA '07, 2007, pp. 97–108. 10, 47

[21] Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Jae-Woong Chung, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun, *TAPE: a transactional application profiling environment*, ICS, 2005, pp. 199–208. 65, 83

[22] D. Chiou, H. Sunjeliwala, H. Sunwoo, J. Dam Xu, and N. Patil, *FPGA-based Fast, Cycle-Accurate, Full-System Simulators*, Number UTFAST-2006-01, vol. 15, November Austin, TX, 2006, pp. 795–825. 6, 8, 34

[23] Niket K. Choudhary, Salil V. Wadhavkar, Tanmay A. Shah, Hiran Mayukh, Jayneel Gandhi, Brandon H. Dwiel, Sandeep Navada, Hashem H. Najaf-abadi, and Eric Rotenberg, *Fabscalar: composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template*, Proceedings of the 38th annual international symposium on Computer architecture, ISCA '11, 2011, pp. 11–22. 89

[24] 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*, Proceedings of the 5th European conference on Computer systems, EuroSys '10, 2010, pp. 27–40. 40, 43

[25] Duk Chun and Shabbir Latif, *Mips r4000 synchronization primitives*, www.mips.com/media/files/archives/R4000SynchronizationPrimitives.pdf. 28

[26] Eric S. Chung, Eriko Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai, *A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs*, FPGA '08, 2008, pp. 77–86. 6, 8, 27, 33, 34, 82

[27] 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 Technology Systems (NY, USA), vol. 2, 2009, pp. 1–32. 33

[28] Jaewoong Chung, Luke Yen, Stephan Diestelhorst, Martin Pohlack, Michael Hohmuth, David Christie, and Dan Grossman, *Asf: Amd64 extension for lock-free data structures and transactional memory*, Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, 2010, pp. 39–50. 89

[29] C. Click, *Azuls experiences with hardware transactional memory*, HP Labs - Bay Area Workshop on Transactional Memory, 2009. 89

[30] Intel Corp., *Transactional Synchronization in Haswell*, http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/. 89

[31] Oracle Corp., *OpenSPARC*, http://www.opensparc.net/. 4, 17

[32] William J. Dally and Brian Towles, *Route packets, not wires: on-chip inteconnection networks*, Proceedings of the 38th annual Design Automation Conference (New York, NY, USA), DAC '01, ACM, 2001, pp. 684–689. 3

[33] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum, *Hybrid transactional memory*, ASPLOS '06, 2006. 11, 12, 83

[34] Nirav Dave, Michael Pellauer, and Joel Emer, *Implementing a functional/-timing partitioned microprocessor simulator with an FPGA*, WARFP, 2006. 6, 8, 33, 34, 82, 88

[35] J. Davis, L. Hammond, and K. Olukotun, *A flexible architecture for simulation and testing (FAST) multiprocessor systems*, WARFP, 2005. 82

[36] J. Davis, C. Thacker, and C. Chang, *BEE3: Revitalizing computer architecture research*, Microsoft Research, 2009. iv, 16

[37] 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*, Proc. MICRO, 1997, pp. 292–302. 83

[38] P.G. Del Valle, David Atienza, Ivan Magan, J.G. Flores, E.A. Perez, J.M. Mendias, L. Benini, and G. De Micheli, *A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework*, Proceedings of 14th Annual IFIP/IEEE International Conference on VLSI-SoC, 2006, pp. 140–145. 8

[39] Dave Dice, Ori Shalev, and Nir Shavit, *Transactional locking ii*, In Proc. of the 20th Intl. Symp. on Distributed Computing, 2006. 28

[40] eCos, *ecos*, `http://ecos.sourceware.org`. 28

[41] G. Estrin and C. R. Viswanathan, *Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices*, J. ACM **9** (1962), no. 1, 41–60. 1

[42] B. Fagin and J. Erickson, *DartMIPS: a case study in quantitative analysis of processor design tradeoffs using FPGAs*, More FPGAs. Oxford International Workshop on Field-Programmable Logic and Applications (Oxford, England) (W. Moore and W. Luk, eds.), Abingdon EE&CS Books, August 1993, pp. 353–364. 33

[43] Etienne Faure, Mounir Benabdenbi, and Francois Pecheux, *Distributed online software monitoring of manycore architectures*, IEEE On-Line Testing Symposium, 2010, pp. 56–61. 83

[44] Pascal Felber, Christof Fetzer, and Torvald Riegel, *Dynamic performance tuning of word-based software transactional memory*, PPoPP, 2008, pp. 237–246. 11, 15, 40

[45] 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*, J. Parallel Distrib. Comput., vol. 70, October 2010, pp. 1042–1052. 60, 82

[46] Nathan Froyd, John Mellor-Crummey, and Rob Fowler, *Low-overhead call path profiling of unmodified, optimized code*, ICS '05, 2005, pp. 81–90. 83

[47] Jiri Gaisler, *A portable and fault-tolerant microprocessor based on the sparc v8 architecture*, Dependable Systems and Networks, International Conference on **0** (2002), 409. 5

[48] gem5.org, *gem5 in-order configuration*, `http://gem5.org/InOrder_Pipeline_Stages`. 30

[49] Maya Gokhale, William Holmes, Andrew Kopser, Sara Lucas, Ronald Minnich, Douglas Sweely, and Daniel Lopresti, *Building and using a highly parallel programmable logic array*, Computer **24** (1991), no. 1, 81–89. 1

[50] Ricardo E. Gonzalez, *Xtensa: A configurable and extensible processor*, IEEE Micro **20** (2000), no. 2, 60–70. 1

[51] Jan Gray, *The myriad uses of block RAM*, `http://www.fpgacpu.org/usenet/bb.html`. 7

[52] Jan Gray, *Hands-on computer architecture: teaching processor and integrated systems design with FPGAs*, Proceedings of the 2000 workshop on Computer architecture education (New York, NY, USA), WCAE '00, ACM, 2000. 5, 8

[53] S. Grinberg and S. Weiss, *Investigation of transactional memory using FPGAs*, EEEI, 2006. 35

[54] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun, *Programming with transactional coherence and consistency*, Proceedings of the 11th international conference on Architectural support for programming languages and operating systems (New York, NY, USA), ASPLOS-XI, ACM, 2004, pp. 1–13. 13, 15, 83

[55] Samuel Hangouet, Sebastien Jan, Louis-Marie Mouton, and Olivier Schneider, *Minimips*, `http://opencores.org/project,minimips`. 17

[56] Craig C. Hansen and Thomas J. Riordan, *Risc computer with unaligned reference handling and method for the same*, March 1989. 89

[57] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, P. Boyle, N. Chist, C. Kim, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, and G. Chiu, *The ibm blue gene/q compute chip*, MICRO **99** (2011). 89

[58] Tim Harris, James Larus, and Ravi Rajwar, *Transactional memory, 2nd edition*, 2nd ed., Morgan and Claypool Publishers, 2010. 12

[59] Tim Harris and S. Peyton-Jones, *Transactional memory with data invariants*, 2006. 10

[60] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi, *Optimizing memory transactions*, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI '06, 2006, pp. 14–25. 10

[61] J. R. Hauser and J. Wawrzynek, *Garp: a MIPS processor with a reconfigurable coprocessor*, Proceedings of the 5th IEEE Symposium on FPGA-Based Custom Computing Machines, FCCM '97, 1997, pp. 12–. 33

[62] John Hauser, *SoftFloat*, `http://www.jhauser.us/arithmetic/SoftFloat.html`. 31

[63] John L. Hennessy and David A. Patterson, *Computer architecture: a quantitative approach*, 3rd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. 21

[64] Maurice Herlihy and J. Eliot B. Moss, *Transactional memory: architectural support for lock-free data structures*, Proceedings of the 20th annual international symposium on computer architecture (New York, NY, USA), ISCA '93, ACM, 1993, pp. 289–300. 10, 12

[65] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun, *EigenBench: A simple exploration tool for orthogonal TM characteristics*, IISWC'10, 2010. 53, 54, 72

[66] Altera Inc., *Nios II Processor Reference Handbook*, 2007. 5

[67] Bluespec Inc., *Bluespec system verilog*, `http://www.bluespec.com`. 8, 88

[68] Tabula Inc., *Tabula Spacetime 3D Architecture*, `http://www.tabula.com/technology/TabulaSpacetime_WhitePaper.pdf`. 3

[69] Xilinx Inc., *Chipscope pro tool user guide*, `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/chipscope_pro_sw_cores_ug029.pdf`. 30, 57, 87

[70] ———, *Content-addressable memory v6.1*, `http://www.xilinx.com/support/documentation/ip_documentation/cam_ds253.pdf`. 21

[71] ———, *Microblaze processor reference guide*, `www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`. 5, 24, 35

[72] ———, *Xilinx smartxplorer for ISE project navigator users tutorial*, `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/ug689.pdf`. 87

[73] ———, *Virtex-5 family overview*, `http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf`, 2009. 13

[74] R A Hexsel Jorge Tortato Jr, *A minimalist cache coherent MPSoC designed for FPGAs*, Int. J. High Performance Systems Architecture, 2011, pp. 67–76. 35

[75] Chirstoforos Kachris and Chidamber Kulkarni, *Configurable transactional memory*, Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), FCCM '07, 2007, pp. 65–72. 35, 59, 82

[76] Gerry Kane, *MIPS RISC architecture*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. 12

[77] Gokcen Kestor, Srdjan Stipic, Osman Unsal, Adrian Cristal, and Mateo Valero, *RMSTM:a transactional memory benchmark for recognition,mining and synthesis applications*, TRANSACT, 2009. 30

[78] Asif Khan, Muralidaran Vijayaraghavan, Silas Boyd-Wickizer, and Arvind, *Fast and cycle-accurate modeling of a multicore processor*, Proc. ISPASS 2012), April 2012, pp. 1–8. 35

[79] Philipp Kirchhofer, *Enhancing an htm system with hw monitoring capabilities*, Master's thesis, Karlsruhe Institute of Technology, 2012. 14, 71, 81

[80] Thomas F Knight, *An architecture for mostly functional languages*, ACM Lisp and Functional Programming Conference, 1986. 10

[81] Alex Krasnov, Andrew Schultz, John Wawrzynek, Greg Gibeling, and Pierre yves Droz, *RAMP blue: a message-passing manycore system in FPGAs*, FPL 2007, 2007, pp. 27–29. 8, 33, 34, 35

[82] Venkata Krishnan and Josep Torrellas, *A chip-multiprocessor architecture with speculative multithreading*, IEEE Trans. Comput. **48** (1999), no. 9, 866–880. 85

[83] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony D. Nguyen, *Hybrid transactional memory.*, PPOPP'06, 2006, pp. 209–220. 12

[84] Martin Labrecque, Mark Jeffrey, and J. Steffan, *Application-specific signatures for transactional memory in soft processors*, ARC 2010, 2010. 36, 60, 82

[85] Charles Eric LaForest and J. Gregory Steffan, *Efficient multi-ported memories for FPGAs*, Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays, FPGA '10, 2010, pp. 41–50. 88

[86] D. B. Lomet, *Process structuring, synchronization, and recovery using atomic actions*, Proceedings of an ACM conference on Language design for reliable software (New York, NY, USA), ACM, 1977, pp. 128–137. 10

[87] Guangming Lu, Hartej Singh, Nader Bagherzadeh, Fadi Kurdahi, and Eliseu M. C. Filho, *The morphosys parallel reconfigurable system*, In Proc. of the 5th International Euro-Par Conference, 1999, pp. 727–734. 1

[88] Roman Lysecky and Frank Vahid, *A configurable logic architecture for dynamic hardware/software partitioning*, Proceedings of the conference on Design, automation and test in Europe - Volume 1 (Washington, DC, USA), DATE '04, IEEE Computer Society, 2004, pp. 10480–. 1

[89] Vijay K. Madisetti and Lan Shen, *Interface design for core-based systems*, IEEE Design and Test of Computers **14** (1997), 42–51. ix, 4

[90] Philipp Mahr, Alexander Heine, and Christophe Bobda, *On-chip transactional memory system for FPGAs using TCC model*, FPGAworld '09, 2009. 35

[91] Michael Manzke and Ross Brennan, *Extending FPGA based teaching boards into the area of distributed memory multiprocessors*, WCAE '04 (NY, USA), 2004, p. 5. 8

[92] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, *Graphite: A distributed parallel simulator for multicores*, HPCA '10, 2011, pp. 1–12. 6

[93] Chi Cao Minh, Jae Woong Chung, Christos Kozyrakis, and Kunle Olukotun, *STAMP: Stanford transactional applications for multi-processing*, IISWC, 2008. 31, 54, 61, 65, 72, 78

[94] MIPS Technologies, 1225 Charleston Road, Mountain View, CA, *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, 2.5 ed., July 2005. 17

[95] G. E. Moore, *Cramming More Components onto Integrated Circuits*, Electronics **38** (1965), no. 8, 114–117. iii

[96] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood, *LogTM: Log-based transactional memory*, HPCA 2006, 2006, pp. 254–265. 10, 13, 15

[97] Sagnik Nandy, Xiaofeng Gao, and Jeanne Ferrante, *Tfp: Time-sensitive, flow-specific profiling at runtime*, Languages and Compilers for Parallel Computing **2958** (2004), 32–47. 83

[98] Njuguna Njoroge, Jared Casper, Sewook Wee, Yuriy Teslyar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun, *ATLAS:a chip-multiprocessor with tm support*, DATE'07, 2007, pp. 3–8. 8, 34, 35, 59, 82

[99] RTEMS OAR Corporation, *Real-time executive for multiprocessor systems*, `http://www.rtems.com`. 28

[100] OGP, *The Open Graphics Project*, `http://wiki.opengraphics.org/`. 89

[101] Hitoshi Oi and N. Ranganathan, *A cache coherence protocol for the bidirectional ring based multiprocessor*, PDCS'99, 1999, pp. 3–6. 60

[102] Koray Öner, Luiz Andre Barroso, Sasan Iman, Jaeheon Jeong, Krishnan Ramamurthy, and Michel Dubois, *The design of rpm: an fpga-based multiprocessor emulator*, Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays, FPGA '95, 1995, pp. 60–66. 33

[103] Open Virtual Platforms, *Open Virtual Platforms MIPS Malta*, http://www.ovpworld.org. 30

[104] OpenCores.org, *OpenCores Website*, www.opencores.org. 7, 89

[105] openfpga.org, *OpenFPGA General API Specification 0.4*, http://www.openfpga.org/Standards. 89

[106] Yiannacouras P., *The microarchitecture of FPGA-Based soft processors*, Master's thesis, University of Toronto, 2005. 5

[107] Michael K. Papamichael and James C. Hoe, *CONNECT: re-examining conventional wisdom for designing NoCs in the context of fpgas*, Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays, FPGA '12, 2012, pp. 37–46. 89

[108] David A. Penry, Daniel Fay, David Hodgdon, Ryan Wells, Graham Schelle, David I. August, and Dan Connors, *Exploiting parallelism and structure to accelerate the simulation of chip multi-processors*, HPCA, 2006. 8

[109] 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*, Computing Frontiers '08, 2008, pp. 67–78. 78, 81, 85

[110] Christian Pinto, Shivani Raghav, Andrea Marongiu, Martino Ruggiero, David Atienza, and Luca Benini, *Gpgpu-accelerated parallel and fast simulation of thousand-core platforms*, Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '11, 2011, pp. 53–62. 6

[111] Matteo Pusceddu, Simone Ceccolini, Gianluca Palermo, Donatella Sciuto, and Antonino Tumeo, *A compact TM multiprocessor system on FPGA*, FPL'10, 2010, pp. 578–581. 60, 82

[112] Ravi Rajwar and James R. Goodman, *Speculative lock elision: enabling highly concurrent multithreaded execution*, Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (Washington, DC, USA), MICRO 34, IEEE Computer Society, 2001, pp. 294–305. 85

[113] Steve Rhoads, *Plasma soft core*, `http://opencores.org/project,plasma`. iv, 5, 12, 16, 17

[114] Wind River, *Simics 4*, `www.virtutech.com/files/tech.../simics_p4080_hybrid_overview.pdf`. 6

[115] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel, *Is transactional programming actually easier?*, SIGPLAN Not. (New York, NY, USA), vol. 45, ACM, January 2010, pp. 47–56. 10

[116] Gokhan Sayilar, *Implementing an efficient shared fpu to a multicore prototype*. 21

[117] Graham Schelle, Jamison Collins, Ethan Schuchman, Perry 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*, FPGA '10, 2010. 8

[118] William N. Scherer, III and Michael L. Scott, *Advanced contention management for dynamic software transactional memory*, Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing, PODC '05, 2005, pp. 240–248. 12, 90

[119] Nir Shavit and Dan Touitou, *Software transactional memory*, Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA), PODC '95, ACM, 1995, pp. 204–213. 12

[120] Sameer Shende, Allen Malony, Alan Morris, and Felix Wolf, *Performance profiling overhead compensation for MPI programs*, Recent Advances in Parallel Virtual Machine and Message Passing Interface **3666** (2005), 359–367. 83

[121] N. Sonmez, A. Cristal, O. S. Unsal, T. Harris, and M. Valero, *Profiling transactional memory applications on an atomic block basis: A haskell case study*, In Second Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG), 2009. 81, 85

[122] Nehir Sonmez, Oriol Arcas, Otto Pflucker, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero, *TMbox: A flexible and reconfigurable 16-core hybrid transactional memory system*, Proc. FCCM '11, 2011, pp. 146–153. 41, 83

[123] Nehir Sonmez, Oriol Arcas, Gokhan Sayilar, Osman S. Unsal, Adrian Cristal, Ibrahim Hur, Satnam Singh, and Mateo Valero, *From Plasma to BeeFarm: Design experience of an FPGA-based multicore prototype*, ARC'11, March 23-25 2011. 18, 53

[124] Nehir Sonmez, Tim Harris, Adrian Cristal, Osman S. Unsal, and Mateo Valero, *Taking the heat off transactions: Dynamic selection of pessimistic concurrency control*, Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (Washington, DC, USA), IPDPS '09, IEEE Computer Society, 2009, pp. 1–10. 80, 90

[125] Zhangxi Tan, Andrew Waterman, Rimas Avizienis, Yunsup Lee, Henry Cook, David Patterson, and Krste Asanović, *RAMP gold: An FPGA-based architecture simulator for multiprocessors*, DAC '10, 2010, pp. 463 – 468. 6, 8, 27, 34, 35, 88

[126] Chuck Thacker, *A DDR2 controller for BEE3*, 2009. 18, 19

[127] ______, *Beehive: A many-core computer for FPGAs (v5)*, http://projects.csail.mit.edu/ beehive/BeehiveV5.pdf, MSR Silicon Valley, 2010. 8, 19, 33, 34

[128] ______, *Hardware Transactional Memory for Beehive*, http://research.microsoft.com/en-us/um/people/birrell/beehive/hardware transactional memory for beehive3.pdf, MSR Silicon Valley, 2010. 60, 82

[129] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero, *Eazyhtm: eager-lazy hardware transactional memory*, Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, 2009, pp. 145–155. 13, 15

[130] Marc Tremblay and Shailender Chaudhry, *A third-generation 65nm 16-core 32-thread plus 32-scout-thread cmt sparc processor*, IEEE International Solid State Circuits Conference (2008), 82–83. 89

[131] C. Tsen, S. Gonzalez-Navarro, M. Schulte, B. Hickmann, and K. Compton, *A combined decimal and binary floating-point multiplier*, ASAP, 2009. 35

[132] Frank Vahid and Tony D. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, international student ed., Wiley, October 2001. 2

[133] Muralidaran Vijayaraghavan and Arvind, *Bounded dataflow networks and latency-insensitive circuits*, Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign, MEMOCODE'09, 2009, pp. 171–180. 26

[134] John F. Wakerly, *Digital design: Principles and practices*, 4rd ed., Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005. 2

[135] 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*, FPGA, 2009. 8

[136] M. Wazlowski and Others, *PRISM-II compiler and architecture*, Proc. of the 1st IEEE Symposium on Field-Programmable Custom Computing Machines, April 1993, pp. 9–16. 1

[137] Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun, *A practical FPGA-based framework for novel CMP research*, FPGA '07 (Monterey, USA), 2007, pp. 116–125. 33, 34, 35

[138] Reinhold P. Weicker, *Dhrystone: a synthetic systems programming benchmark*, Commun. ACM, vol. 27, 1984, pp. 1013–1030. 30

[139] Timothy Wong, *LEON3 port for BEE2 and ASIC implementation*, `http://cadlab.cs.ucla.edu/software_release/bee2leon3port/`. 35

[140] Bob Zeidman, *Generic FPGA Architecture*, `http://pldworld.kr/html/technote/pldesignline/allaboutfpgas-bobz.htm`. ix, 2

[141] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith, *System support for automatic profiling and optimization*, SIGOPS OS Rev., vol. 31, 1997. 83

[142] Craig B. Zilles and Gurindar S. Sohi, *A programmable co-processor for profiling*, HPCA, 2001. 83

[143] Ferad Zyulkyarov, *Programming, debugging, profiling and optimizing transactional memory programs*, Ph.D. thesis, Universitat Politecnica de Catalunya, 2011. 83

[144] Ferad Zyulkyarov, Srdjan Stipic, Tim Harris, Osman S. Unsal, Adrían Cristal, Ibrahim Hur, and Mateo Valero, *Discovering and understanding performance bottlenecks in transactional applications*, PACT'10, 2010, pp. 285–294. 65, 83, 85