Overlay Architectures for FPGA-Based Software Packet Processing

by

Martin Labrecque

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Your file Votre référence
ISBN: 978-0-494-78251-4

Our file Notre référence
ISBN: 978-0-494-78251-4

Abstract

Overlay Architectures for FPGA-Based Software Packet Processing

Martin Labrecque

Doctor of Philosophy

Graduate Department of Electrical and Computer Engineering

University of Toronto

2011

Packet processing is the enabling technology of networked information systems such as the Internet and is usually performed with fixed-function custom-made ASIC chips. As communication protocols evolve rapidly, there is increasing interest in adapting features of the processing over time and, since software is the preferred way of expressing complex computation, we are interested in finding a platform to execute packet processing software with the best possible throughput. Because FPGAs are widely used in network equipment and they can implement processors, we are motivated to investigate executing software directly on the FPGAs. Off-the-shelf soft processors on FPGA fabric are currently geared towards performing embedded sequential tasks and, in contrast, network processing is most often inherently parallel between packet flows, if not between each individual packet.

Our goal is to allow multiple threads of execution in an FPGA to reach a higher aggregate throughput than commercially available shared-memory soft multi-processors via improvements to the underlying soft processor architecture. We study a number of processor pipeline organizations to identify which ones can scale to a larger number of execution threads and find that tuning multithreaded pipelines can provide compact cores with high throughput. We then perform a design space exploration of multicore soft systems, compare single-threaded and multithreaded designs to identify scalability limits and develop processor architectures allowing threads to execute with as little architectural stalls as possible: in particular with instruction replay and static hazard detection mechanisms. To further reduce the wait times, we allow threads to speculatively execute by leveraging transactional memory. Our multithreaded multiprocessor along with our compilation and simulation framework makes the FPGA easy to use for an average programmer who can write an application as a single thread of computation with coarse-grained synchronization around shared data structures. Comparing with multithreaded processors using

lock-based synchronization, we measure up to 57% additional throughput with the use of transactional-memory-based synchronization. Given our applications, gigabit interfaces and 125 MHz system clock rate, our results suggest that soft processors can process packets in software at high throughput and low latency, while capitalizing on the FPGAs already available in network equipment.

Acknowledgements

First, I would like to thank my advisor, Gregory Steffan, for his constructive comments throughout this project and especially for his patience in showing me how to improve my writing.

I also aknowledge my labmates and groupmates for their camaradery. I also thank the UTKC, my sempais and Tominaga Sensei for introducing me to the 'basics'.

Special thanks go to my relatives for their encouragements and support. Most of the credit goes to my parents, who are coaches and unconditional fans of mine. An important mention goes to my brother for helping me move in Toronto. My beloved Mayrose also deserves a special place on this page for making the sun shine even on rainy days.

Finally, the financial support from the "Fonds québécois de la recherche sur la nature et les technologies" made this work possible.

Contents

List of Figures

List of Tables

Glossary

ASIC	Application Specific Integrated Circuit
BRAM	Block Random Access Memory
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DDR	Double data rate
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
Gbps	Gigabits per second
HTM	Hardware Transactional Memory
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
ISP	Internet Service Provider
LAN	Local Area Network
LE	Logic Element
LUT	Lookup-Table
MAC	Media Access Control
MESI	Modified/Exclusive/Shared/Invalid
NAT	Network Address Translation
NP	Network Processor
PCI	Peripheral Component Interconnect
PE	Processing Engine/Element
QoS	Quality of Service
RAM	Random Access Memory
RISC	Reduced Instruction Set Computing/Computer
RTL	Register-Transfer Level/Language
SDRAM	Synchronous dynamic random access memory
SIMD	Single Instruction Multiple Data
SRAM	Static random access memory
STM	Software Transactional Memory
TCP	Transmission Control Protocol
TM	Transactional Memory
VLAN	Virtual Local Area Network

Chapter 1

Introduction

Packet processing is a key enabling technology of the modern information age and is at the foundation of digital telephony and TV, the web, emails and social networking. Because of the rapidly evolving standards for hardware (e.g. from 10 to 1000 Mbits/sec over copper wires), protocols (e.g. from CDMA to LTE wireless protocols) and applications (from static web content to streaming video), the bandwidth requirements are increasing exponentially and the tasks that define packet processing for a given application and transport medium must be updated periodically. In the mean time, accommodating those needs by designing and fabricating processors with an application-specific integrated circuit (*ASIC*) approach has progressively become more difficult because of the increasing complexity of the circuits and processes, the high initial cost and the long time to market. Therefore processing packets in software at high throughput is increasingly desirable. To execute software, a designer can choose network processor ASICs which have a fixed organization, or FPGAs (Field-Programmable Gate Arrays) which are configurable chips that can implement any kind of digital circuit and do not require a large initial capital investment.

1.1 FPGAs in Packet Processing

Other than being able to implement a considerable number of equivalent logic gates, FPGAs present advantages specifically in the context of packet processing. First, FPGAs can be connected with a very low latency to a network link and can be reprogrammed easily, thus providing a way for the networking

hardware equipment to keep up with the constantly changing demands of the Internet. Also, FPGAs can partition their resources to exploit the parallelism available in packet streams, rather than processing packets in sequence as single-core systems do. FPGAs provide several other advantages for computing: they allow for application-specific hardware acceleration, their code can be reused in several designs, and they can be integrated with almost any memory or peripheral technology, thus reducing a board's device count and power requirements.

FPGAs are already widely adopted by network appliance vendors such as Cisco Systems and Huawei and there exists a number of commercial reference designs for high-speed edge and metro network nodes consisting of Altera and Xilinx devices [10, 60]. Overall, sales in the communications industry represent more than 40%[1] of the market for Altera and Xilinx (who, in turn, account for 87% of the total PLD market [11]). In most of these use-cases however, FPGAs serve as data paths for passing packets between other chips in charge of the bulk of the packet processing. The challenge to the widespread use of FPGAs for computing is providing a programming model for the application functions that need to be updated frequently. The current design flow of FPGAs usually involves converting a full application into a hardware circuit—a process that is too cumbersome for the needs of many applications and design teams. To make configurable chips easy to program for a software developer, FPGA-based designs increasingly implement soft processors in a portion of their configurable fabric. While software packet processing on FPGAs is only in its infancy, the processor-on-FPGA execution paradigm has gained considerable traction in the rest of the embedded community. There remains several challenges in making a system of these processors efficient, and this thesis aims at addressing some of them.

1.2 Soft Processors in Packet Processing

Although there exist tools to compile a high-level program down to a logic circuit, the quality of the circuit often suffers because of the difficulty of analyzing pointer references. Since converting complex applications into state machines leads to the creation of a very large number of states, current synthesis tools often produce bulky circuits that are not always desirable from a frequency or area standpoint. Reprogramming the configurable fabric of an FPGA in the field may also require board-level

[1] Telecom and wireless represent 44% of Altera's sales [11]. Communications account for 47% of Xilinx's sales [156].

circuitry that adds to the cost and complexity of the final system. Software programmable processor systems, i.e. programmable cores that potentially have accelerators sharing a bus, are therefore very desirable, assuming that they can deliver the performance required for given clock frequency and area specifications.

While data intensive manipulations can be performed in parallel on a wide buffer of data either with wide-issue or vector-processing elements, control intensive applications are best described in a sequential manner because of an abundance of dependent conditional statements. To support arbitrary software compiled from a high-level programming language, we focus on the architecture of general-purpose processors on FPGA platforms, which could later be augmented with application-specific accelerators to execute the data-intensive portions of an application. We next highlight areas where currently available commercial soft processors need improvement for packet processing, and how we overcome those difficulties in this thesis.

1.3 Research Goals

The goal of this research is to allow multiple threads of execution, such as in packet processing applications, to reach a higher aggregate throughput via improvements to the underlying soft processor architecture. We demonstrate that our resulting soft processors can provide an efficient packet processing platform while being easy to use for software programmers, by setting the following goals.

1. **To build soft processors with an improved area efficiency with the intent of instantiating a plurality of them.** To this end, we augment the single-threaded soft processor architectures that are currently commercially available with support for customized code generation and multithreading and study their area and performance trade-offs.

2. **To explore the memory system and processor organization trade-off space for scaling up to larger numbers of processors, while minimizing contention and maximizing locality.** To reach this goal, we assemble instances of these processors into a multiprocessor infrastructure that requires the design of a memory hierarchy. Studying different cache architectures to hide the latency of off-chip memory accesses and evaluating the scalability limits of such designs gives us insight on how to build an efficient packet processing system.

3. **To build a real packet processing system on an FPGA to validate our architectural exploration with real applications.** For this purpose, we integrate our soft multiprocessor on a platform with high-speed packet interfaces, a process requiring the support for fast packet input and off-chip memory storage.

4. **To characterize and tackle the primary bottleneck in the system, which we identify as inter-thread synchronization in our packet processing applications.** As a first remedy, we introduce hardware thread scheduling which is both a compiler and architectural technique in our solution. While this technique improves throughput significantly, significant synchronization stalls remain. To utilize the available hardware thread contexts more efficiently, we introduce optimistic concurrency through transactional memory. We first study in isolation and on single-threaded processors, methods to track memory accesses to enable optimistic parallelism. We further refine this mechanism to incorporate it in our multithreaded multiprocessor framework. Finally, to measure the benefits of our programming model, we compare our system with speculative execution against alternate programming models, including flow-affinity scheduling where each thread of an application accesses different data.

1.4 Thesis Organization

This thesis is organized as follows: in Chapter 2, we provide some background information on FPGAs, network processing and soft processors. In Chapter 3, we identify representative packet processing applications and investigate how to execute them efficiently. In particular, we quantify performance problems in the pipeline programming model for packet processing applications and determine that the run-to-completion approach (Section 3.1.2) is easier to program and more apt at fully utilizing multiple concurrent processing threads. In consequence, we determine early on that the synchronization bottleneck must be addressed for the simpler run-to-completion approach to be competitive in terms of performance; this objective is the driving force for the rest of this thesis. In Chapter 4, we study multithreaded soft processor variations: those cores act as a building block for the remainder of the thesis. Our target for this chapter is a Stratix FPGA with 41,250 logic elements, a mid-range device. In Chapter 5, we replicate our soft cores and study their scalability when using conventional single-

threaded soft processors as a baseline. As we are interested in maximizing the use of larger FPGAs, we leverage the Transmogrifier 4 infrastructure with a Stratix FPGA containing 79,040 logic elements (the largest chip in its family). Encouraged by the results, we select an FPGA board for packet processing with 4 gigabit Ethernet ports, which represents even today a considerable bandwidth at the non-ISP level. We migrate our processor infrastructure to this new board and build the rest of our experimentation on the board's Virtex-II Pro FPGA with 53,136 logic cells : we dedicate Chapter 6 to describe this platform and our baseline processor organization. As we determine that our packet processing applications are hindered by the use of mutexes, in Chapter 7, we present a hardware thread scheduler to address the synchronization bottleneck. In Chapter 8, we present our full system integration of speculative execution with multithreaded multiprocessors and we are able to validate that a programming model with speculative execution on a pool-of-threads provides an improved throughput compared to a programming model where threads have a specialized behavior. Finally, in Chapter 9, we state our conclusions and directions for future work.

Chapter 2

Background

This chapter presents some background information regarding software packet processing and its implementation in network processor ASICs. We then provide some context information on FPGAs, on their programming model, and how they can be used to process packets via software-defined applications. For reading convenience, we place the related work specific to the processor architecture components that we study in their respective later thesis chapters. We start by giving some high-level information on packet processing and its application types.

2.1 Software Packet Processing

With close to 2 billion users in 2010 [104] and growing needs for interactive multimedia and online services, the computing needs of the Internet are expanding rapidly. The types of interconnect devices range from traditional computers, to emerging markets such as sensors, cell phones and miniaturized computers the size of a wall adapter called "plug computers" [26]. Ericsson predicts that there will be 50 billion network connections between devices by 2020 [40]. As an increasing number of businesses depend on network services, packet processing has broad financial and legal impacts [45, 46].

Many packet processing applications must process packets at line rate, and to do so, they must scale to make full use of a system composed of multiple processors and accelerators cores and of the available bandwidth to and from packet-buffers and memory channels. As network users are moving towards networks requiring higher-level application processing, flexible software is best suited to adapt

to fast changing requirements. Given the broad and varied use of packet processing, in this section we clarify the application-types and bandwidths for which software packet processing is suitable, and what challenges emerge when programming these applications in a multicore environment.

2.1.1 Application Types

We divide network processing applications into three categories:

1) Basic Common packet processing tasks performed in small office/home office network equipment include learning MAC addresses, switching and routing packets, and performing port forwarding, port and IP filtering, basic QoS, and VLAN tagging. These functions are typically limited to a predefined number of values (e.g. 10 port forwarding entries) such that they can be implemented in an ASIC switch controller chip, without the need for software programmability.

2) Byte-Manipulation A number of network applications, in particular cryptography and compression routines, apply a regular transformation to most of the bytes of a packet. Because these workloads often require several iterations of specialized bit-wise operations, they benefit from hardware acceleration such as the specialized engines present in network processors [56], modern processors (e.g. Intel AES instruction extensions), and off-the-shelf network cards; they also generally do not require the programmability of software.

3) Control-Flow Intensive Network packet processing is no longer limited solely to routing, with many applications that require deep packet inspection becoming increasingly common. Some applications, such as storage virtualization and server load balancing, are variations on the theme of routing that reach deeper into the payload of the packets to perform content-based routing, access control, and bandwidth allocation. Other applications have entirely different computing needs such as the increasingly complex firewall, intrusion detection and bandwidth management systems that must recognize applications, scan for known malicious patterns, and recognize new attacks among a sea of innocuous packets. Furthermore, with the increasing use of application protocols built on HTTP and XML, the distinction between payload and header processing is slowly disappearing. Hence in this thesis we focus on such control-flow intensive applications. We focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets.

2.1.2 Network Processors

Until recently, the machines that process packets were exclusively made out of fixed ASICs performing increasingly complex tasks. To keep up with these changing requirements, computer architects have devised a new family of chips called *network processors*. They are software programmable chips designed to process packets at line speed: because the processing latency usually exceeds the packet inter-arrival time, multiple packets must be processed concurrently. For this reason, network processors (NPs) usually consist of multithreaded multiprocessors. Multithreading has been also used extensively in ASIC network processors to hide pipeline stalls [57].

The factors limiting the widespread adoption of network processors are (i) the disagreement on what is a good architecture for them; and (ii) their programming complexity often related to their complex ISAs and architectures. Network processor system integrators therefore avoid the risk of being locked-in a particular ASIC vendor's solutions. We next give an overview of the literature on network processor architecture research and how it relates to our work.

StepNP [114] provides a framework for simulating multiprocessor networks and allows for multi-threading in the processor cores. It has even been used to explore the effect of multithreading in the face of increasing latencies for packet forwarding [115]. In our work, we explore similar issues but do so directly in FPGA hardware, and focus on how performance can be scaled given an FPGA implementation and single-channel memory interface.

Ravindran et al. [122] created hand-tuned and automatically generated multiprocessor systems for packet-forwarding on FPGA hardware. However, they limited the scope of their work to routing tables which can fit in the on-chip FPGA Block RAMs. Larger networks will demand larger routing tables hence necessitating the use of off-chip RAM. Our work differs in that we heavily focus on the effect of the on-board off-chip DDR(2)-SDRAM, and do so over a range of benchmarks rather than for a specific application.

There exists a large body of work focusing on architecture exploration for NPs [31, 32, 49, 114, 130, 135, 149], however none of them has seriously investigated nor argued against the run-to-completion programming model on which we focus (see Section 3) and that is becoming increasingly common in commercial products such as the PowerNP/Hifn 5NP4G [5, 36], Mindspeed M27483 TSP3 [68],

Broadcom BCM1480 [20], AMCC nP7510 [29] and Vitesse IQ2200 [118].

2.1.3 Fast- vs Slow-Path Processing

Network equipment typically connects multiple network ports on links with speeds that span multiple orders of magnitude: 10Mbps to 10Gbps are common physical layer data rates. While local area networks can normally achieve a high link utilization, typical transfer speeds to and from the Internet are on the order of megabits per second [27] as determined by the network utilization and organization between the Internet service providers.

The amount of processing performed on each packet will directly affect the latency introduced on each packet and the maximum allowable sustained packet rate. The amount of buffering available on the network node will also help mitigate bursts of traffic and/or variability in the amount of processing. For current network appliances that process an aggregate multi-gigabit data stream across many ports, there is typically a division of the processing in *data plane* (a.k.a. fast path) and *control plane* (a.k.a. slow path) operations. The data plane takes care of forwarding packets at full speed based on rules defined by the control plane which only processes a fraction of the traffic (e.g. routing protocols). Data plane processing is therefore very regular from packet to packet and deterministic in the number of cycles per packet. Data plane operations are typically implemented in ASICs on linecards; control plane operations are typically implemented on a centralized supervisor card. The control plane, often software programmable and performing complex control-flow intensive tasks, still has to be provisioned to handle high data rates. For example, the Cisco SPP network processor in the CRS-1 router is designed to handle 40Gbps [24]. On smaller scale network equipment (eg., a commodity desktop-based router at the extreme end of the spectrum), the two planes are frequently implemented on a single printed circuit board either with ASICs or programmable network processors or a combination of both. In that case, the amount of computation per packet has a high variance, as the boundary between the fast and slow path is often blurred.

In this thesis, we focus on complex packet processing tasks that are best suited to a software implementation, since a complete hardware implementation would be impractical. Our benchmark applications therefore target the control plane, rather than the data plane of multi-gigabit machines. We now introduce FPGAs, in contrast with ASIC network processors, and explain how FPGAs can

implement packet processing.

2.2 FPGAs and their Programming Model

An FPGA is a semiconductor device with programmable lookup-tables (*LUTs*) that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6 typically). Thousands of these building blocks are connected with a programmable interconnect to implement larger-scale circuits. FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (on the order of a few kilobits), that together provide a small storage capacity but a large bandwidth for circuits in the FPGA.

FPGAs have been used extensively for packet processing [15,55,92,98,102,110,129] due to several advantages that they provide: (i) ease of design and fast time-to-market; (ii) the ability to connect to a number of memory channels and network interfaces, possibly of varying technologies; (iii) the ability to fully exploit parallelism and custom accelerators; and (iv) the ability to field-upgrade the hardware design.

Other than being used in commercial end-products, FPGAs also provide an opportunity for prototyping and high-speed design space exploration. While the Internet infrastructure is dominated by vendors with proprietary technologies, there is a push to democratize the hardware, to allow researchers to revisit some low-level network protocols that have not evolved in more than a decade. This desire to add programmability in the network is formally embraced by large scale projects such as CleanSlate [41], RouteBricks [37] and GENI [137], in turn supported by massive infrastructure projects such as Internet2 [6] and CANARIE [131]. FPGAs are a possible solution to fulfill such a need as they allow one to rapidly develop low-level packet processing applications. As an example of FPGA-based board, the NetFPGA development platform [91] (see Chapter 6) allows networking researchers to create custom hardware designs affordably, and to test new theories, algorithms, and applications at line-speeds much closer to current state-of-the-art. The challenge is that many networking researchers are not necessarily trained in hardware design; and even for those that are, composing packet processing hardware in a *hardware-description language* is time consuming and error prone.

FPGA network processing with software programmable processors has been explored mostly in the

perspective of one control processor with FPGA acceleration [3, 4, 38, 64, 89, 140]. A pipeline-based network processor has been proposed where an ILP solver finds the best processor count per pipeline stage [67, 165]. The most relevant work to our research is found in Kachris and Vassiliadis [65] where two Microblaze soft-processors with hardware assists are evaluated with a focus on the importance of the load balance between the processors and the hardware assists. The amount of parallelism they exploit is limited to two threads programmed by hand and they do not make use of external memory. FPL-3 [30] is another project suggesting compilation from a high-level language of packet processing; however, we prefer to use a well-accepted language for our experiments.

2.2.1 Hard Processors on FPGA

While the FPGA on the current NetFPGA board on which we perform our measurements has two embedded PowerPC hard cores, the next generation of NetFPGA will not (assuming a Virtex-5 XCV5TX240T-2 FPGA device [90]), making an alternative way of implementing processors—via the reconfigurable fabric—invaluable to software programmers. At the moment of this writing, Altera abandoned hard cores on FPGAs with the Excalibur device family that included an ARM922T processor, the Xilinx Virtex-5 family of FPGAs provides a limited number of devices with hard PowerPC cores, and the Virtex-6 family does not offer any hard processor cores. Interestingly, because processors are common and useful, the latest Xilinx 7-Series introduced ARM-based processors [157]. So far, hard processors FPGA cannot reach the gigahertz speed of operation of modern processors for single-threaded workloads: the PowerPC cores in the FXT family of Virtex-5 FPGAs can reach at most 550MHz [155] and the state-of-the-art Xilinx 7-Series Zynq cores are reported to operate at 800MHz [157]. In retrospect, while the two major FPGA vendors, Xilinx and Altera, formerly had hard processors in some of their devices, it seems that they have, up until recently, slowed their investment in hard processor cores, possibly given the emergence of soft processors.

2.2.2 Soft Processors

Improving logic density and maximum clock rates of FPGAs have led to an increasing number of FPGA-based system-on-chip (i.e. single-chip) designs, which in turn increasingly contain one or more *soft processors*—processors composed of programmable logic on the FPGA. Despite the raw performance

drawbacks, a soft processor has several advantages compared to creating custom logic in a hardware-description language: it is easier to program (e.g., using C), portable to different FPGAs, flexible (i.e., can be customized), and can be used to control or communicate with other components/custom accelerators in the design. Soft processors provide a familiar programming environment allowing non-hardware experts to target FPGAs and can provide means to identify and accelerate bottleneck computations through additional custom hardware [8, 95, 153].

The traditional network packet forwarding and routing are now well understood problems that can be accomplished at line speed by FPGAs but more complex applications are best described in a high-level software executing on a processor. Soft processors are very well suited to packet processing applications that have irregular data access and control flow, and hence unpredictable processing times. As FPGA-based systems including one or more soft processors become increasingly common, we are motivated to better understand the architectural trade-offs and improve the efficiency of these systems.

FPGAs are now used in numerous packet processing tasks and even if many research projects have demonstrated working systems using exclusively soft processors on FPGAs [64, 67, 108], the bulk of the processing is often however assumed by another on-board ASIC processor. Our proposed soft processor system improves on commercially available soft processors [16, 152] by: (i) specifically taking advantage of the features of FPGAs; and (ii) incorporating some benefits from ASIC network processor architectures such as multithreaded in-order single issue cores, which can be found in the Intel IXP processor family [57] [1] and the QuantumFlow processor [25]. Our final system targets the NetFPGA [91] card, which is unique with its four Gigabit Ethernet ports; we envision however that our design could be adapted for system with a different number of Ethernet ports such as RiceNIC [132].

Because most soft processors perform control-intensive tasks (the bulk of the reconfigurable fabric being reserved for data intensive tasks), commercial SPs (in particular NIOS-II [16] and Microblaze [152]) issue instructions one at a time and in order. There exists a large number of open-source soft processors with instruction sets as varied as ARM, AVR, SPARC, MIPS and full-custom [1, 117]. Vector soft processors [71, 164] offer instructions for array-based operations, which relate to applications domains such as graphics and media processing, which are not our focus in this thesis. The most related soft processors to our investigations are the PicaRISC processor from

[1] Now owned by Netronome Systems Inc.

Altera [69] which has not been publicly documented yet and the multithreaded UT-II processor [43], which we describe in Chapter 4. SPREE [161] gives an overview of the performance and area consumption of soft processors. As a reference point, on a Stratix EP1S40F780C5 device with the fastest speed grade, a platform that we use in Chapter 4, the NiosII-fast (the variation with the fastest clock rate [16]) reaches 135 MHz (but instructions are not retired on every cycle). We next review how packet processing can be expressed, i.e. how will the soft-processors be programmed inside an FPGA.

Chapter 3

Choosing a Programming Model

While nearly all modern packet processing is done on multicores, the mapping of the application to those cores is often specific to the underlying processor architecture and is also a trade-off between performance and ease-of-programming. In this chapter, we first explore the multiple ways of managing parallelism in packet processing and focus on the most versatile and convenient approach. Then, after defining a set of representative software packet processing applications, we quantitatively justify our choice of programming model for extracting parallelism to be able to scale performance.

3.1 Programming Models Overview

In this section, we examine the three main programming models illustrated in Figure 3.1.

3.1.1 Pipelining

To program the multiple processing elements (PEs) of a network processor, most research focuses on breaking the program into one or several parallel pipelines of tasks that map to an equal number of processor pipelines (as shown in Figure 3.1 (a) and (c)). In this simplest single-pipeline model, while inter-task communication is allowed, sharing of data between tasks is usually not supported. The pipeline model rather focuses on exploiting data locality, to limit the instruction storage per PE and to regroup accesses to shared resources for efficient instruction scheduling. Pipelining is widely used as the underlying parallelization method [33, 59, 145, 148], most often to avoid the difficulty of managing

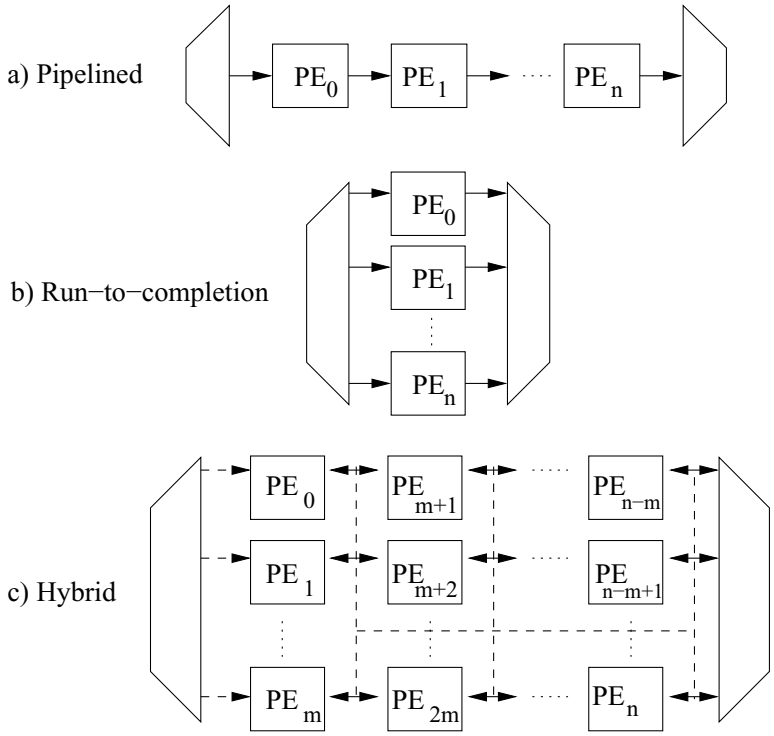


Figure 3.1: Parallelization models.

locks: there is no need to synchronize if writes to shared data structures are done in a single pipeline stage[1], even if there are multiple readers.

The pipeline programming model is well suited for heterogeneous PEs arranged in a one-dimensional array. Conversely, programs are best suited for pipelining if they are composed of data-independent and self-contained kernels executing in a stable computation pattern with communication at the boundaries [136]. To obtain the best performance on such a pipeline architecture, all the processing stages have to be of similar latency so that one stage is ready to accept work at the same time as the previous stage is ready to hand off some work. However, efficiently balancing the pipeline stages and the amount of communication between them to maximize multiprocessor utilization is complicated and often not possible for complex applications. Even if an acceptable result is obtained, this difficult process must be repeated when the application is updated or the code is ported to a different processor architecture. Furthermore, handling packets that require varying amounts of computation or breaking down frequent accesses to large stateful data structures (such as routing tables) to ensure lock-free

[1] Assuming the writes can be flushed atomically.

operation is impractical in a pipeline with many stages. We quantify those difficulties when pipelining existing applications in Chapter 8.1.

In earlier work [74], we have demonstrated the difficulties of the pipeline model. We have investigated the compilation issues in compiling a high-level description of an application to a network processor. Our applications were defined as task graphs that we attempted to transform to execute efficiently on the underlying hardware. We created a parametric model of a network processor that allowed us to approximate the performance of different execution schemes. We observed that the decomposition of the application into a graph of tasks resulted in a significant amount of idle cycles in processing elements, a result that was also observed in NepSim [93]. The system remained complex to analyze, debug and to target by a compiler, because of problems such as load imbalance and the competing effects of task scheduling and mapping. We also identified that not all workloads can easily be expressed as task graphs: the bulk of numerous network processor applications is contained in a single loop nest.

Earlier publications [85, 97] have asserted that network processing is a form of streaming [138], a computing model related to pipelining. In stream programming, the programmer uses a custom language to describe regular computations and their parallelism such that the compiler can schedule precisely concurrent SIMD operations in automatically balanced pipeline stages. We argue that complex control flow in advanced network processing is too unpredictable to be best described as a stream program. For example, a network application that interprets TCP packets to provide data caching for a database server will have widely varying amounts of processing to apply on packets: describing the work as a pipeline of tightly scheduled parallel instructions is impractical and inefficient.

3.1.2 Run-to-Completion

The model in Figure 3.1(b) refers to the method of writing a program where different processors process packets from beginning-to-end by executing the same program. Different paths in the program will exercise different parts of the application on different threads, which do not execute in lock-step. The programming is therefore intuitive but typically requires the addition of locks to protect shared data structures and coherence mechanisms when shared data can be held in multiple locations.

Nearly all modern multicore processors are logically organized in a grid to which the programmer

can map an execution model of his choice. The major *architectural* factors that would push a programmer away from the intuitive run-to-completion model are: (i) a high cache coherence penalty across private data caches, or (ii) a reduced instruction and data cache storage compared to a task decomposition in pipeline(s). Most network processors do not implement cache coherence (except commodity multicore machines) and for our experiments, our data cache is shared and therefore has roughly the same hit rate with or without task pipelining. For network processors with no special communication channel between the cores for pipeline operations, such as the 100Gbps-rated 160-threads Cisco QuantumFlow Processor [25], run-to-completion is the natural programming model.

3.1.3 Pipeline Hybrid

The grid of processors or *hybrid* scheme (see Figure 3.1 (c)) also requires a task decomposition and presents, to some extent, the same difficulty as the pipelined model. While packets can flow across different pipelines, the assumption is that a specialized engine at the input would dispatch a packet to a given pipeline, which would function generally independently. Enforcing this independence to minimize lock contention across pipelines is actually application specific and can lead to severe load-imbalance. The number of processors assigned to each pipeline must also be sized according to the number of network interfaces to provide a uniform response time. In the presence of programs with synchronized sections, abundant control flow and variable memory access latencies, achieving a good load balance is often infeasible. The hybrid model is also inflexible: the entire code must be re-organized as a whole if the architecture or the software is changed. A variation on the hybrid model consists of using the processors as run-to-completion but delegating atomic operations to specialized processors [143]. That model also removes the need for locks (assuming point-to-point lock-free communication channels) but poses the same problems in terms of ease of programming and load imbalance as the pipeline model.

Because the synchronization around shared data structures in stateful applications makes it impractical to extract parallelism otherwise (e.g., with a pipeline of balanced execution stages), we adopt the *run-to-completion*/*pool-of-threads* model, where each thread performs the processing of a packet from beginning-to-end, and where all threads essentially execute the same program code. Consequently our work can be interpreted as an evaluation of a run-to-completion or of an hybrid model where we focus on a single replicated pipeline stage. We next present our applications and quantify in Section 3.3

what would be the impact of pipelining them.

3.2 Benchmark Applications

To measure packet throughput, we need to define the processing performed on each packet. Network packet processing is no longer limited solely to routing, with many applications that require deeper packet inspection becoming increasingly common and desired. Most NP architecture evaluations to date have been based on typical packet processing tasks taken individually: *microbenchmarks*. NetBench [101], NPBench [86] and CommBench [150] provide test programs ranging from MD5 message digest to media transcoding. Stateless kernels that emulate isolated packet processing routines fall into the first two categories in Section 2.1.1 which are not our focus. For those kernels that are limited to packet header processing, the amount of instruction-level parallelism (ILP) can exceed several thousand instructions [86]. Because such tasks are best addressed by SIMD processors or custom ASICs, in this thesis we instead focus on control-flow intensive applications where the average ILP is only five (a number in agreement with other studies on control-flow intensive benchmarks [144]). While microbenchmarks are useful when designing an individual PE or examining memory behavior, they are not representative of the orchestration of an entire NP application. There is little consensus in the research community on an appropriate suite of benchmarks for advanced packet processing, at which current fixed-function ASICs perform poorly and network processors should excel in the future.

To take full advantage of the software programmability of our processors, our focus is on control-flow intensive applications performing deep packet inspection (i.e., deeper than the IP header). In addition, and in contrast with prior work [86, 101, 150], we focus on *stateful* applications—i.e., applications in which shared, persistent data structures are modified during the processing of most packets. Since there is a lack of packet processing benchmark suites representing applications that are threaded and synchronized, we have developed the four control-flow intensive applications detailed in Table 3.1. Except for `Intruder` [22] and its variation `Intruder2`, the benchmarks are the result of discussions with experts in the networking community, in particular at the University of Toronto and at Cisco Systems Inc., with a focus on realistic applications.

Table 3.1 also describes the nature of the parallelism in each application. Given that the applications

Table 3.1: Benchmark Applications

	Description	Critical Sections	Input packet trace
`Classifier`	Performs a regular expression matching on TCP packets, collects statistics on the number of bytes transferred and monitors the packet rate for classified flows to exemplify network-based application recognition. In the absence of a match, the payloads of packets are reassembled and tested up to 500 bytes before a flow is marked as non-matching. As a use case, we configure the widely used PCRE matching library [53] (same library that the popular Snort [124] intrusion detection/prevention system uses) with the HTTP regular expression from the "Linux layer 7 packet classifier" [87].	Has long transactions when regular expressions are evaluated; exploits parallelism across flows stored in a global synchronized hash-table.	Publicly available packet trace from 2007 on a 150Mbps trans-Pacific link (the link was upgraded from 100Mbps to 150Mbps on June 1 2007) [28] HTTP server replies are added to all packets presumably coming from an HTTP server to trigger the classification.
`NAT`	Exemplifies network address translation by rewriting packets from one network as if originating from one machine, and appropriately rewriting the packets flowing in the other direction. As an extension, `NAT` collects flow statistics and monitors packet rates.	Exhibits short transactions that encompass most of the processing; exploits parallelism across flows stored in a global synchronized hash-table.	Same packet trace as `Classifier`.
`UDHCP`	Derived from the widely-used open-source DHCP server. As in the original code, leases are stored in a linearly traversed array and IP addresses are leased after a ping request for them expires, to ensure that they are unused.	Periodic polling on databases for time expired records results in many read-dominated transactions as seen in Table 6.2. Has high contention on shared lease and awaiting-for-ping array data structures.	Packet trace modeling the expected DHCP message distribution of a network of 20000 hosts [14].
`Intruder`	Network intrusion detection [22] modified for packetized input. Extensive use of queues and lists, reducing the effectiveness of signatures due to random memory accesses [75]; mostly CPU-bound with bursts of synchronized computation on highly- contended data structures.	Packets are stored in a synchronized associative array until complete messages are fully reassembled. They are then checked against a dictionary before being removed from the synchronized data structures and sent over the network. Multiple lists and associative array make extensive use of the memory allocation routines.	256 flows sending random messages of at most 128 bytes, broken randomly in at most 4 fragments, containing 10% of 'known attacks'. The fragments are shuffled with a sliding window of 16 and encapsulated in IP packets.
`Intruder2`	Network intrusion detection [22] modified for packetized input and re-written to have array-based reassembly buffers to avoid the overhead of queues, lists and maps that also reduced the effectiveness of signatures due to the large amount of `malloc()`/`free()` calls [75].	Similar to above, with significantly lighter data strutures because of statically allocated memory. Has two synchronization phases: first a per-flow lock is acquired and released to allow processing each packet individually, then most of the computation is performed on reassembled messages before the per-flow variables are modified again under synchronization.	Same as row above.

initially exist as sequential programs where an infinite loop processes one packet on each iteration, the parallelism discussed is across the iterations of the main infinite loop. We only utilize the `Intruder` benchmarks starting in Chapter 8 because they present a different behavior with regards to speculative execution, which is the focus of the last part of this thesis. Since quantitative metrics about our benchmarks have evolved slightly as we edited the benchmarks, we report updated numbers in the appropriate sections of the thesis. The baseline measurements in Table 6.2 (co-located with the experimental results) reports statistics on the dynamic accesses per critical section for each application. Note that the critical sections comprise significant numbers of loads and stores with a high disparity between the average and maximum values, showing that our applications are stateful and irregular in terms of computations per packet. We next analyze the representative traits of each application and generalize them to other control-flow intensive network applications, particularly with respect to packet ordering, data parallelism, and synchronization.

Packet Ordering In a network device, there is typically no requirement to preserve the packet ordering across flows from the same or different senders: they are interpreted as unrelated. For a given flow, one source of synchronization is often to preserve packet ordering, which can mean: i) that packets must be processed in the order that they arrived; and/or ii) that packets must be sent out on the network in the order that they arrived. The first criteria is often relaxed because it is well known that packets can be reordered in a network [146], which means that the enforced order is optimistically the order in which the original sender created the packets. The second criteria can be managed at the output queues and does not usually affect the core of packet processing. For our benchmark applications in Table 3.1, while we could enforce ordering in software, we allow packets to be processed out-of-order because our application semantics allow it.

Data Parallelism Packet processing typically implies tracking flows (or clients for `UDHCP`) in a database, commonly implemented as a hash-table or direct-mapped array. The size of the database is bounded by the size of the main memory—typically larger than what can be contained in any single data cache—and there is usually little or a very short-term reuse of incoming packets. Because a network device executes continuously, a mechanism for removing flows from the database after some elapsed time is also required. In *stateful* applications, i.e. applications where shared, persistent data structures

are modified during the processing of most packets, there may be variables that do not relate directly to flows (e.g. a packet counter). Therefore, it is possible that the processing of packets from different flows access the same shared data and therefore the processing of those packets in parallel may conflict. Also, for certain applications, it may possible to extract intra-packet parallelism (e.g. parallelization of a loop), however those cases are rare because they are likely to leave some processors underutilized so we do not consider them further. Whenever shared data is accessed by concurrent threads, those accesses must be synchronized to prevent data corruption.

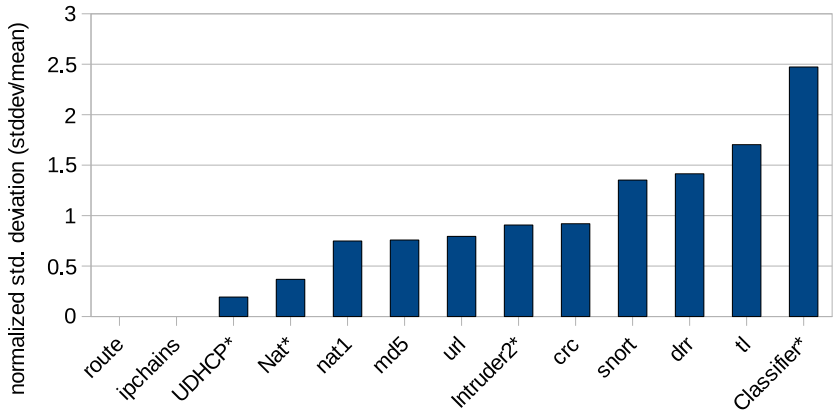
Synchronization To increase parallelism, implementing finer-grain synchronization is not always feasible since repeatedly entering and exiting critical sections will likely add significant overhead. For example, `NAT` and `Classifier` have a significant fraction of their code synchronized because there is an interaction between the hash table lock and the per-flow lock (see Table 3.1): a thread cannot release the lock on the hash table prior to acquiring a lock on a flow descriptor to ensure that the flow is not removed in the mean time. Mechanisms for allowing coarser-grained sections while preserving performance are therefore very desirable for packet processing.

3.3 Motivating Run-to-Completion

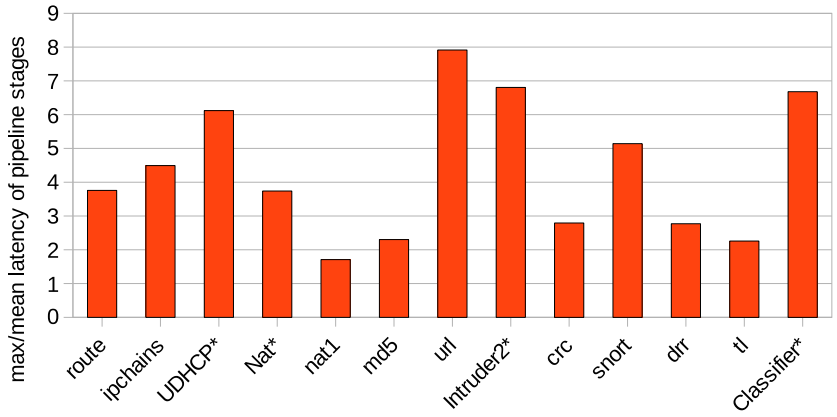
Because a number of network processors implement the pipeline model [88] with the promise of extracting parallelism while being lock-free, we must justify our choice of a different model (run-to-completion). For this purpose, we use benchmarks from NetBench [100][2], and our stateful benchmarks running on a single thread. As Netbench's applications require a number of system and library calls, they cannot be ported easily to our NetThreads embedded target (Chapter 6), so we instead record execution traces using the PIN tool [123]. We only monitor the processing for each packet and ignore the packet and console I/O routines.

As seen in Figure 3.2(a), our four applications span the spectrum of latency variability (i.e. jitter) per packet that is represented by the NetBench benchmarks. `Route` and `ipchain` have completely deterministic behavior (no variability), while table lookup `tl` and the regular expression matching `Classifier` have the most variation across packets. Considering that for those applications the amount

[2]Except `dh` which is not packet based nor `ssl` because of its inlined console output.



(a) Normalized variability in packet processing latency with a single thread (stddev/mean latency).



(b) Imbalance in pipeline stages with an emulated pipeline of 8 threads (max stage latency divided by mean stage latency).

Figure 3.2: Single-threaded computational (a) variability and (b) load imbalance in an emulated 8-stage pipeline based on instruction clustering for NetBench [100] benchmarks and our benchmarks (marked with a *).

of computation can be more than doubled depending on the packet, we conclude that they are less amenable to pipelining. Even if re-circulating a packet from the end to the beginning of a pipeline were effective at mitigating this huge variation in latency [88], we would also have to effectively divide the packet processing into pipeline stages.

To emulate pipelining, we employ a previously-proposed graph-clustering method that greedily clusters instructions with the highest control and data flow affinity [121] to eliminate cyclic dependences and minimize communication between pipeline stages. Since NetTM has 8 threads, we cluster the instructions into 8 pipeline stages based on the profile information[3]. In all benchmarks in Figure 3.2(b), clustering the code into pipeline stages leads to significant load imbalance. `url` has the largest pipeline imbalance (i.e. the rate of the pipeline is 7.9 times slower than the average rate of all the stages) because of the clustering of the Boyer-Moore string search function in a single pipeline stage. Even `route` which has a deterministic execution (Figure 3.2(a)) has load imbalance because of the clustering of the checksum routine in a single longer-latency pipeline stage, and `ipchains` has similar problems. While hardware accelerators could be used to accelerate checksum operations, a programmer cannot rely on them to balance the latency of arbitrary code in stages. To get a better load balance, a programmer would replicate the slowest stages and move to the hybrid or run-to-completion model, and add synchronization around stateful data structures.

3.4 Summary

Modern network appliance programmers are faced with larger and more complex systems-on-chip composed of multiple processor and acceleration cores and must meet the expectation that performance should scale with the number of compute threads [112, 145]. When the application is composed of parallel threads, accesses to shared data structures must be synchronized. These dependences make it difficult to pipeline the code into balanced stages of execution to extract parallelism. We demonstrated in this chapter that such applications are more suitable to a *run-to-completion* model of execution, where a single thread performs the complete processing of a packet from start to finish. The main advantages of this model is that there is a unique program, it can be reused on new architectures and its scalability

[3] Control-flow and data dependences are based on the profiled basic-blocks.

is more predictable. The programmer only writes one program in the most natural way possible and the compilation infrastructure and the processor architecture ease the parallelization problem. For performance, the system must be able to execute multiple instances of the program in parallel. We next investigate ways of improving the efficiency of soft processor cores to later build multicores.

Chapter 4

Improving Soft Processor Area Efficiency with Multithreading

In an FPGA packet processor, each processing element must be compact and optimized to deliver the maximum performance, as we want to replicate those processing elements to take advantage of all the available memory bandwidth. For soft processors in general, especially multicore systems, raw single-threaded performance is often not as crucial as the aggregate performance of all the cores—otherwise an off-chip or on-chip (if available) single hard processor would be preferable. Other metrics or their combination may also be of importance, such as minimizing the area of the processor, matching the clock frequency of another key component in the same clock domain, handling requests or interrupts within specified time constraints (i.e., real-time), or processing a stream of requests or data at a sufficient rate. Flexibility and control over performance/area trade-offs in the soft processor design space are key, and hence, for comparing soft processor designs, a summarizing metric that combines area, frequency, and cycle count such as *area efficiency* is most relevant.

In this chapter, we first summarize our work on custom code generation for processors to improve area efficiency. After, we demonstrate how to make 3, 5, and 7-stage pipelined multithreaded soft processors 33%, 77%, and 106% more area efficient than their single-threaded counterparts, the result of careful tuning of the architecture, ISA, and number of threads.

4.1 Custom Code Generation for Soft Uniprocessors

Since soft processors may be easily modified to match application requirements, it is compelling to go beyond default compilation (e.g., default `gcc`), and customize compilation aspects such as the code generation. A first step in our research is to determine the extent to which we can customize the soft processors to (i) improve the performance of the existing processors and (ii) save area to accommodate more processors on a chip. As well, we are interested in using a compiler to enable code transformations that can potentially save reconfigurable hardware resources. We perform our experiments on the processors generated by the SPREE infrastructure [161, 162]. SPREE takes as input an architectural description of a processor and generates an RTL implementation of it, based on a library of pre-defined modules. The RTL currently targets hardware blocks of the Altera FPGAs so the Quartus tools are integrated in the SPREE infrastructure to characterize the area, frequency and power of the resulting processors.

In a recent publication [83], we summarize our findings on soft processor customization, notably: (i) we can improve area efficiency by replacing a variable-amount shifter with two fixed-amount shifters; (ii) hazard detection logic is a determining factor in the processor's area and operating frequency; (iii) we can eliminate load delay slots in most cases; (iv) branch delay slots can be removed in a 7-stage pipeline even with no branch prediction; (v) 3-operand multiplies are only justified for a 3-stage processor (and otherwise `Hi`/`Lo` registers are best); (vi) unaligned memory loads and stores do not provide a significant performance benefit for our benchmarks; (vii) we are able to remove one forwarding line with simple operand scheduling and improve area efficiency; and (viii) we can limit the compiler's use of a significant fraction of the 32 architected registers for many benchmarks without degrading performance. To maximize the efficiency of the customized architecture of our soft processors, we combined several of these optimizations and obtained a 12% additional area efficiency increase on average (and up to 47% in the best case). By including instruction subsetting [83] in the processors and our optimizations, the mean improvement is 13% but the maximum is 51%. For the remainder of this thesis, we will implement 3-operand multiplies and the removal of hazard detection, delay slots and unaligned access instructions; all four techniques become even more beneficial in the processors that we propose in the next chapter. Since subsetting the architecture actually requires recompilation

of the FPGA design, we next investigate another architectural technique that may be more rewarding for software-only programmers that do not have the option of re-customizing the processor architecture when they update their application.

4.2 Multithreaded Soft Processors

A promising way to improve area efficiency is through the use of multithreading. Fort *et al.* [43] present a 4-way multithreaded soft processor design, and demonstrate that it provides significant area savings over having four soft processors—but with a moderate cost in performance. In particular, they show that a multithreaded soft processor need not have hazard detection logic nor forwarding lines, so long as the number of threads matches the number of pipeline stages such that all instructions concurrently executing in different stages are independent. Researchers in the CUSTARD project [35] have also developed a similar pipelined 4-stage 4-way multithreaded soft processor.

In this chapter, rather than comparing with multiple soft processors, we show that a multithreaded soft processor can be better than one that is single-threaded. In particular, we demonstrate: (i) that multithreaded soft processors are more area-efficient and are capable of a better sustained instructions-per-cycle (IPC) than single-threaded soft processors; (ii) that these benefits increase with the number of pipeline stages (at least up to and including 7-stage pipelines); (iii) that careful optimization of any unpipelined multi-cycle paths in the original soft processor is important, and (iv) that careful selection of certain ISA features, the number of registers, and the number of threads are key to maximizing area-efficiency.

4.3 Soft Processor Infrastructure

In this section we briefly describe our infrastructure for designing and measuring soft processors, including the SPREE system for generating single-threaded pipelined soft processors, our methodology for comparing soft processor designs, our compilation infrastructure, and the benchmark applications we study.

SPREE: We use the SPREE system [163] to generate a wide range of soft processor architectures. SPREE takes as input ISA and datapath descriptions and produces RTL which is synthesized, mapped,

placed, and routed by `Quartus 5.0` [9] using the default optimization settings. The generated processors target Altera Stratix FPGAs, and we synthesize for a `EP1S40F780C5` device—a mid-sized device in the family with the fastest speed grade. We determine the area and clock frequency of each soft processor design using the arithmetic mean across 10 seeds (which produce different initial placements before placement and routing) to improve our approximation of the true mean. For each benchmark, the soft processor RTL design is simulated using `Modelsim 6.0b` [103] to (i) obtain the total number of execution cycles, and (ii) to generate a trace which is validated for correctness against the corresponding execution by an emulator (MINT [141]).

Measurement: For Altera Stratix FPGAs, the basic logic element (LE) is a 4-input lookup table plus a flip-flop—hence we report the area of these processors in *equivalent LEs*, a number that additionally accounts for the consumed silicon area of any hardware blocks (e.g. multiplication or block-memory units). For the processor clock rate, we report the maximum frequency supported by the critical path of the processor design. To combine area, frequency, and cycle count to evaluate an optimization, we use a metric of *area efficiency*, in million instructions per second (MIPS) per thousand equivalent LEs. It is important to have such a summarizing metric since a system designer may be most concerned with soft processor area in some cases, or frequency or wallclock-time performance in others. Finally, we obtain dynamic power metrics for our benchmarks using Quartus' Power Play tool [9]. The measurement is based on the switching activities of post-placed-and-routed nodes determined by simulating benchmark applications on a post-placed-and-routed netlist of a processor in Modelsim [103]: we divide the energy consumed in nano-Joules by the number of instructions executed (nJ/instr), discounting the power consumed by I/O pins.

Single-Threaded Processors: The single-threaded processors that we compare with are pipelined with 3 stages (`pipe3`), 5 stages (`pipe5`), and 7 stages (`pipe7`). The 2 and 6 stage pipelines were previously found to be uninteresting [163], hence we study the 3, 5, and 7 stage pipelines for even spacing. All three processors have hazard detection logic and forwarding lines for both operands. The 3-stage pipeline implements shift operations using the multiplier, and is the most area-efficient processor generated by SPREE [163] (at 1256 equiv. LEs, 78.3 MHz). The 5-stage pipeline also has a multiplier-based shifter, and implements a compromise between area efficiency and maximum operating frequency (at 1365 equiv. LEs, 86.8 MHz). The 7-stage pipeline has a barrel shifter, leads to the largest processor,

Table 4.1: Benchmark applications evaluated.

Source	Benchmark	Modified	Dyn. Instr. Counts	Category
MiBench [50]	BITCNTS	di	26,175	L
XiRisc [21]	BUBBLE_SORT	-	1,824	L
	CRC	-	14,353	S
	DES	-	1,516	S
	FFT*	-	1,901	M
	FIR*	-	822	M
	QUANT*	-	2,342	S
	IQUANT*	-	1,896	M
	VLC	-	17,860	L
RATES [133]	GOL	di	129,750	O

* Contains multiply

di Reduced data input set and number of iterations

Categories: dominated by (L)oads, (S)hifts, (M)ultiplies, (O)ther

Table 4.2: Benchmark applications mixes evaluated.

Mix	T0	T1	T2	T3	T4	T5	T6
1	FFT	QUANT	BUBBLE_SORT	GOL	FIR	CRC	VLC
2	FIR	CRC	VLC	GOL	IQUANT	DES	BITCNTS
3	IQUANT	DES	BITCNTS	GOL	FFT	QUANT	BUBBLE_SORT

and has the highest frequency (at 1639 equiv. LEs, 100.6 MHz). `Pipe3` and `pipe5` both take one extra cycle for shift and multiply instructions, and `pipe3` requires an extra cycle for loads from memory.

Compilation: Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 that target variations of the 32-bits MIPS I [66] ISA; for example, we can trade support for Hi/Lo registers with 3-operand multiplies, enable or disable branch delay slots, and vary the number of architected registers used. Integer division is implemented in software.

Benchmarking: We evaluate our soft processors using the 10 embedded benchmark applications described in Table 4.1, which are divided into 4 categories: dominated by loads (`L`), shifts (`S`), multiplies (`M`) or by none of the above (other, `O`).[1] By selecting benchmarks from each category, we also create

[1]The benchmarks chosen are a subset of those used in previous work [83] since for now we require both data and instructions to each fit in separate single MegaRAMs in the FPGA.

3 multiprogrammed mixes (rows of Table 4.2) that each execute until the completion of the shortest application in the mix. To use these mixes of up to 6 applications, threads available on a processor are filled by choosing one application per thread from the left to right of the mix along a row of Table 4.2. We measure multithreaded processors using (i) multiple copies of the same program executing as separate threads (with separate data memory), and (ii) using the multiprogrammed mixes.

4.4 Multithreading a Soft Processor

Commercially-available soft processors such as Altera's NIOS II and Xilinx's Microblaze are both single-threaded, in-order, and pipelined, as are our SPREE processors (described in the previous section). Such processors require hazard detection logic and forwarding lines for correctness and good performance. These processors can be multithreaded with minimal extra complexity by adding support for instructions from multiple independent threads to be executing in each of the pipeline stages of the processor—an easy way to do this is to have as many threads as there are pipeline stages. This approach is known as *Fine-Grained Multithreading* (FGMT [113]), and is also the approach adopted by Fort *et al.* [43] and the CUSTARD project [35].[2]

In this section we evaluate several SPREE processors of varying pipeline depth that support fine-grained multithreading. Since each pipe stage executes an instruction from an independent thread, these processors no longer require hazard detection logic nor forwarding lines—which as we show can provide improvements in both area and frequency. Focusing on our base ISA (MIPS), we also found that load and branch delay slots are undesirable, which makes intuitive sense since the dependences they hide are already hidden by instructions from other threads—hence we have removed them from the modified version of the ISA that we evaluate.

To support multithreading, the main challenge is to replicate the hardware that stores state for a thread: in particular, each thread needs access to independent architected registers and memory. In contrast with ASICs, for FPGAs the relative latency of on-chip memory vs logic latency grows very slowly with the size of the memory—hence FPGAs are amenable to implementing the replicated storage required for multithreading. We provide replicated program counters that are selected in a round-robin

[2]The CUSTARD group also investigated *Block Multi-Threading* where threads are switched only at long-latency events, but found this approach to be inferior to FGMT.

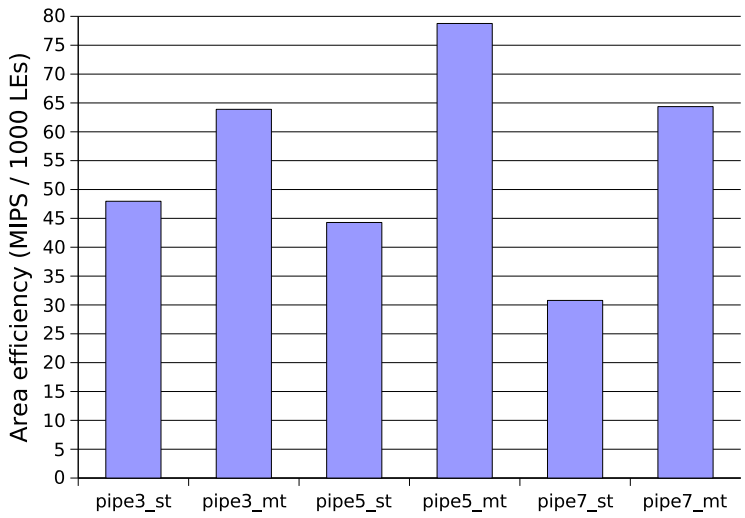


Figure 4.1: Area efficiency of single-threaded (`st`) and multithreaded (`mt`) processors with varying pipeline depths, where the number of threads is equal to the number of stages for the multithreaded processors. Results are the mean across all single benchmarks (i.e., not the mixes).

fashion each cycle. Rather than physically replicating the register file, which would require the addition of costly multiplexers, we index different ranges of a shared physical register file by appropriately shifting the register numbers. We implement this register file using block memories available on Altera Stratix devices; in particular, we could use either `M4K`s (4096 bits capacity, 32 bits width) or `M512`s (512 bits capacity, 16 bits width). We choose `M4K`s because (i) they more naturally support the required 32-bit register width; and (ii) we can implement the desired register file using a smaller total number of block memories, which minimizes the amount of costly multiplexing logic required.

We must also carefully provide separation of instruction and data memory as needed. For the processors in this chapter, we support only on-chip memory—we support caches and off-chip memory in Chapter 5. Similar to the register file, we provide only one physical instruction memory and one physical data memory, but map to different ranges of those memories as needed. In particular, every thread is always allocated a unique range of data memory. When we execute multiple copies of a single program, then threads share a range of instruction memory,[3] otherwise instruction memory ranges are unique as well.

Figure 4.1 shows the mean area efficiency, in MIPS per 1000 equivalent LEs, across all single

[3]Since each thread needs to initialize its global pointer and stack pointer differently (in software), we create a unique initialization routine for each thread, but otherwise they share the same instruction memory range.

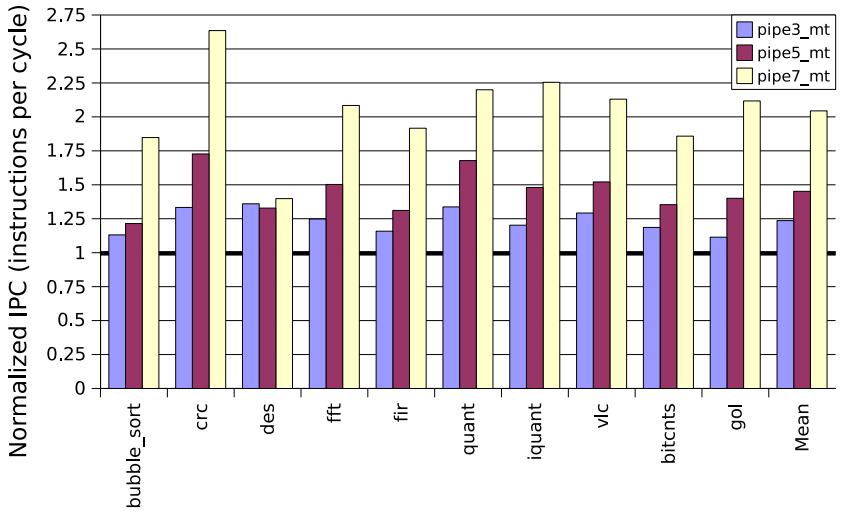


Figure 4.2: IPC gain of multithreaded over single-threaded processors.

benchmarks from Table 4.1 (i.e., we do not yet consider the multiprogrammed mixes). We measure single-threaded (`st`) and multithreaded (`mt`) processors with varying pipeline depths, and for the multithreaded processors the number of threads is equal to the number of pipeline stages. The area efficiency of the 3, 5, and 7-stage pipelined multithreaded processors is respectively 33%, 77% and 106% greater than each of their single-threaded counterparts. The 5-stage pipeline has the maximum area efficiency, as it benefits the most from the combination of optimizations we describe next. The 3 and 7-stage pipeline have similar area efficiencies but offer different trade-offs in IPC, thread count, area, frequency, and power.

Figure 4.2 shows the improvement in instructions-per-cycle (IPC) of multithreaded processors over single-threaded processors. For the 3, 5, and 7-stage pipelines IPC improves by 24%, 45% and 104% respectively. These benefits are partly due to the interleaving of independent instructions in the multithreaded processors which reduce or eliminate the inefficiencies of the single-threaded processors such as unused delay slots, data hazards, and mispredicted branches (our single-threaded processors predict branches are “not-taken”). We have shown that multithreading offers compelling improvements in area-efficiency and IPC over single-threaded processors. In the sections that follow we describe the techniques we used to achieve these gains.

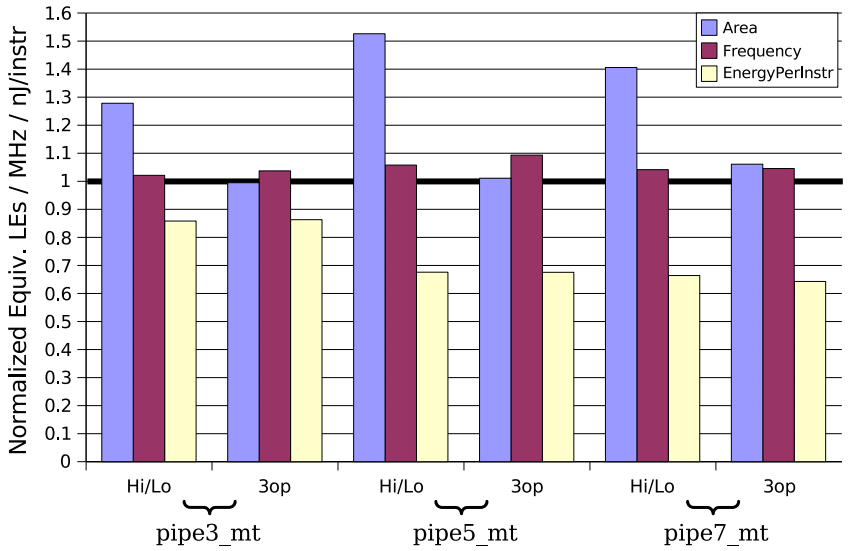


Figure 4.3: Hi/Lo registers vs 3-operand multiplies for various pipeline depths, normalized to the corresponding single-threaded processor.

4.5 Tuning the Architecture

In this section, we identify two architectural features of multithreaded processors that differ significantly from their single-threaded version and hence must be carefully tuned: the choice of Hi/Lo registers versus 3-operand multiplies, and the organization of multicycle paths.

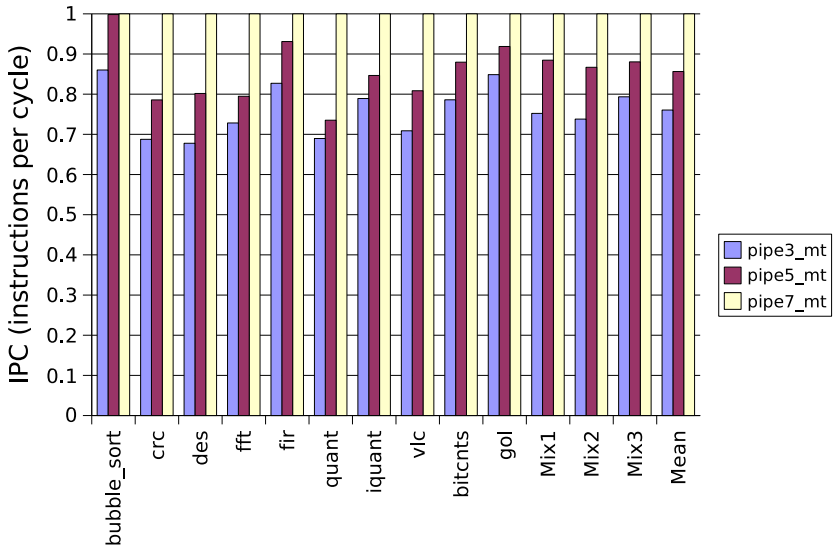
Optimizing the Support for Multiplication: By default, in the MIPS ISA the 64-bit result of 32-bit multiplication is stored into two special 32-bit registers called Hi and Lo—the benefit of these being that multicycle multiplication need not have a write-back path into the regular register file, allowing higher-frequency designs. Hence for a multithreaded implementation of a MIPS processor we must also replicate the Hi and Lo registers. Another alternative is to modify MIPS to support two 3-operand multiply instructions, which target the regular register file and compute the upper or lower 32-bit result independently. We previously demonstrated that Hi/Lo registers result in better frequency than 3-operand multiplies but at the cost of extra area and instruction count, and are a better choice for more deeply-pipelined single-threaded processors [83]. In this chapter we re-investigate this option in the context of multithreaded soft processors. Figure 4.3 shows the impact on area, frequency, and energy-per-instruction with Hi/Lo registers or 3-operand multiplies, for multithreaded processors of varying pipeline stages each relative to the corresponding single-threaded processor. We observe that Hi/Lo

registers require significantly more area than 3-operand multiplies due to their replicated storage but more importantly to the increased multiplexing required to route them.

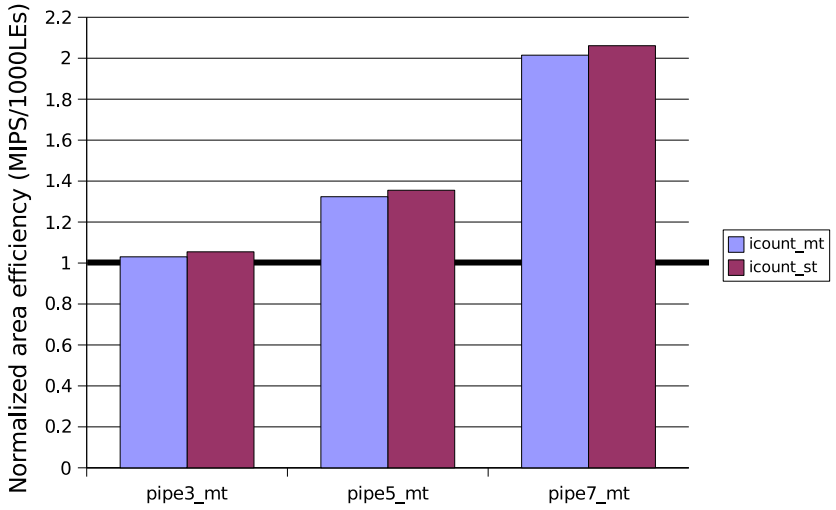
Since the frequency benefits of Hi/Lo registers are no longer significant, and 3-operand multiplies also have significantly reduced energy-per-instruction, we chose support for 3-operand multiplies as our default in all multithreaded experiments in this chapter. In Figure 4.4(a) we show the raw IPC of the multithreaded processors on all benchmarks as well as the multiprogrammed mixes, demonstrating two key things: (i) that our results for replicated copies of the individual benchmarks are similar to those of multiprogrammed mixes; and (ii) that stalls for the multithreaded 7-stage pipeline have been completely eliminated (since it achieves an IPC of 1). For the three-stage pipeline, the most area efficient for single-threaded processors, the baseline multithreaded processor is only 5% more area efficient than the single-threaded processor (see Figure 4.4(b)). Area and frequency of our multithreaded processors are similar to those of the single-threaded processors, hence the majority of these gains (36% and 106% for the 5 and 7-stage pipelines) are related to reduction in stalls due to various hazards in the single-threaded designs. Figure 4.4(b) shows that the reduction of instructions due to the removal of delay slots and 3-operand multiplies also contributes by 3% on average to the final area efficiency that utilizes the scaled instruction count of single-threaded processors to compare a constant amount of work. Comparing with Figure 4.1, we see that single-threaded soft processors favor short pipelines while multithreaded soft processors favor deep pipelines.

Optimizing Multicycle Paths: Our 3 and 5-stage processors must both stall for certain instructions (such as shifts and multiplies), which we call *unpipelined multicycle paths* [163]. It is important to optimize these paths, since otherwise such stalls will impact all other threads in a multithreaded processor. Fort et al. [43] address this challenge by queuing requests that stall in a secondary pipeline as deep as the original, allowing other threads to proceed. We instead attempt to eliminate such stalls by modifying the existing processor architecture.

For the single-threaded 3-stage pipeline, multicycle paths were created by inserting registers to divide critical paths, improving frequency by 58% [163]; this division could also have been used to create two pipeline stages such that shifts and multiplies would be pipelined, but this would have created new potential data hazards (see Figure 4.5), increasing the complexity of hazard detection logic—hence this option was avoided for the single-threaded implementation. In contrast, for a multithreaded



(a) Raw IPC. The 7-stage pipeline has an IPC of 1 because it never stalls.



(b) Area efficiency normalized to the single-threaded processors computed with the instruction count on the multithreaded-processors (`icount_mt`) and with the scaled instruction count on the single-threaded processor (`icount_st`).

Figure 4.4: IPC and area efficiency for the baseline multithreaded processors.

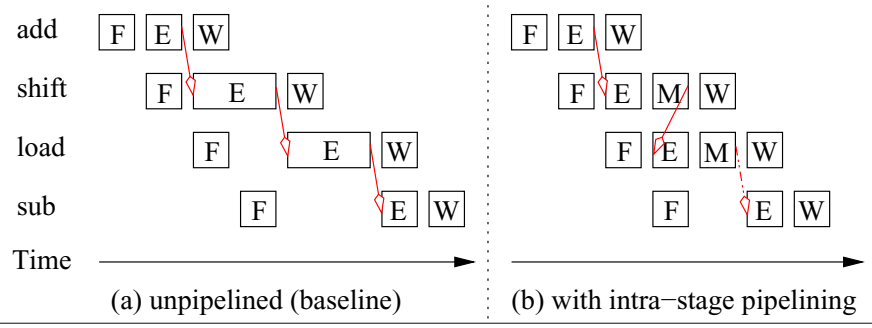


Figure 4.5: Example execution showing multicycle paths in the 3-stage pipeline where: (i) shifts and loads require two cycles in the execute stage; and (ii) we assume each instruction has a register dependence on the previous. Assuming a single-threaded processor, forward arrows represent forwarding lines required, while the backward arrow indicates that a stall would be required. The pipeline stages are: `F` for fetch, `E` for execute, `M` for memory, and `W` for write-back.

processor we can pipeline such paths without concern for data hazards (since consecutive instructions are from independent threads)—hence we do so for the 3-stage processor. The single-threaded 5-stage pipeline also contains multicycle paths. However, we found that for a multithreaded processor that pipelining these paths was not worth the cost, and instead opted to revert the multicycle paths to be single-cycle at a cost of reduced frequency but improved instruction rate. Consequently, eliminating these multicycle paths results in an IPC of 1 for the 5-stage multithreaded processor. The 7-stage processor has no such paths, hence we do not consider it further here.

Figure 4.6 shows the impact on both cycle count and area-efficiency of optimizing multicycle paths for the 3 and 5-stage pipeline multithreaded processors, relative to the corresponding baseline multithreaded processors. First, our optimizations reduced area for both processors (by 1% for the 3-stage, and by 4% for the 5-stage); however, frequency is also reduced for both processors (by 3% for the 3-stage and by 5% for the 5-stage). Fortunately, in all cases cycle count is reduced significantly, improving the IPC by 24% and 45% for the 3-stage and 5-stage processors over the corresponding single-threaded processors. Overall, this technique alone improves area-efficiency by 18% and 15% for the 3 and 5-stage processors over their multithreaded baseline.

Focusing on the multiprogrammed mixes, we see that the cycle count savings is less pronounced: when executing multiple copies of a single program, it is much more likely that consecutive instructions

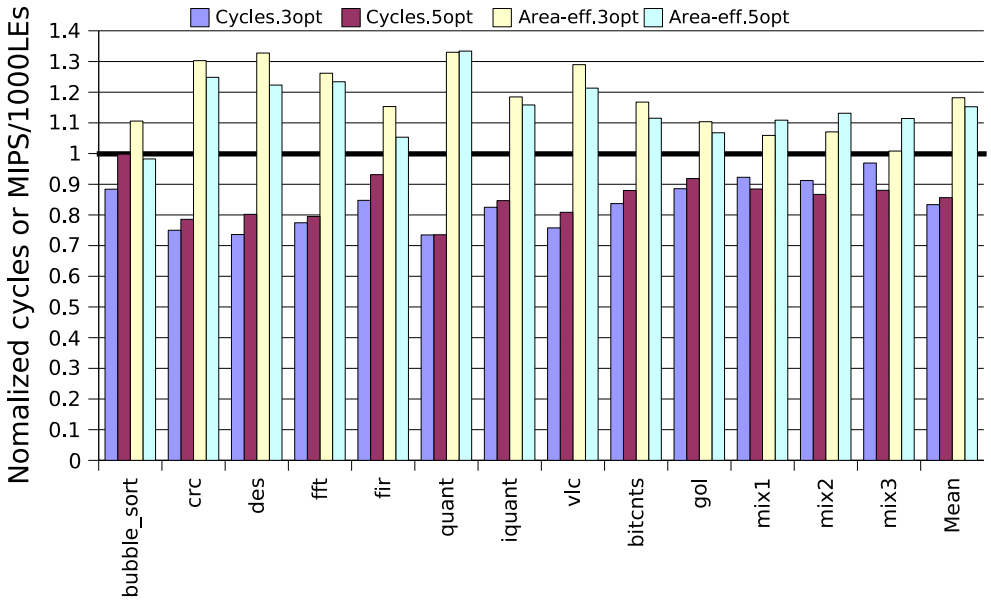


Figure 4.6: Impact on both cycle count and area-efficiency of optimizing multicycle paths for the 3 and 5-stage pipeline multithreaded processors, relative to the corresponding baseline multithreaded processors.

will require the same multicycle path, resulting in avoided stalls with the use of pipelining[4]; for multiprogrammed workloads such consecutive instructions are less likely. Hence for multiprogrammed mixes we achieve only 5% and 12% improvements in area-efficiency for the 3 and 5-stage pipelines.

4.6 Reducing Thread State

In this section, we investigate two techniques for improving the area-efficiency of multithreaded soft processors by reducing thread state.

Reducing the Register File: In previous work [83] we demonstrated that 8 of the 32 architected registers (`s0-s7`) could be avoided by the compiler (such that programs do not target them at all) with only a minor impact on performance for most applications. Since our multithreaded processors have a single physical register file, we can potentially significantly reduce the total size of the register file by similarly removing these registers for each thread. Since our register file is composed of `M4K` block memories, we found that this optimization only makes sense for our 5-stage pipeline: only for that

[4]As shown in Figure 4.5(b), the transition from an instruction that uses a multicycle path to an instruction that doesn't creates a pipeline stall.

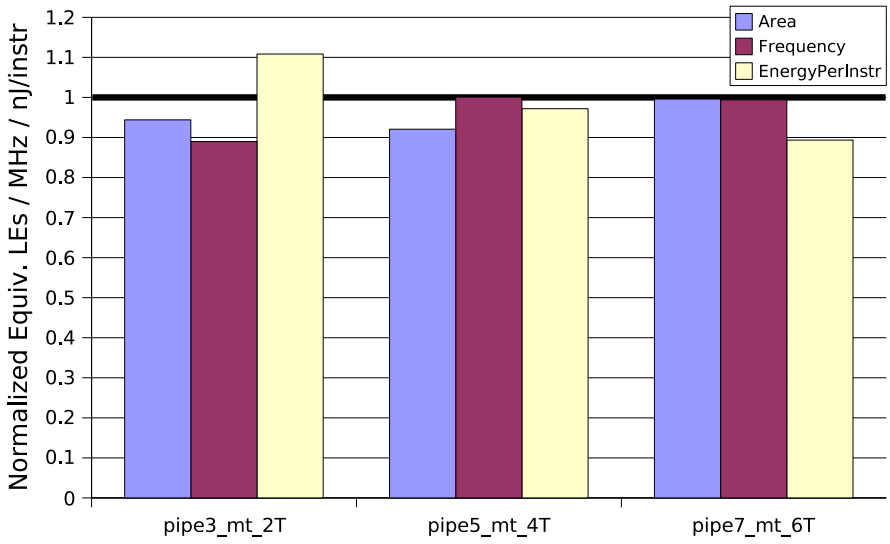


Figure 4.7: Impact of having one thread less than the pipeline depth, normalized to processors having the number of threads equal to the pipeline depth.

processor does the storage saved by removing registers allow us to save entire M4K blocks. In particular, if we remove 7 of 32 registers per thread then the entire resulting 25-register register file fits in a single M4K block (since 25 registers $\times$ 5 threads $\times$ 32 bits $<$ 4096 bits).[5] In fact, since our register file is actually duplicated to provide enough read and write ports, this optimization allows us to use two M4Ks instead of four. For our 5-stage multithreaded processor this optimization allows us to save 5% area and improve frequency by 3%, but increases cycle count by 3% on average due to increased register pressure.

Reducing the Number of Threads: Multithreaded soft processors proposed to date have supported a number of threads equal to the number of pipeline stages [35, 43]. For systems where long multicycle stalls are possible (such as with high latency off-chip memory), supporting a larger number of threads than pipeline stages may be interesting. However, for our work which so far assumes on-chip memory, it may also be beneficial to have fewer threads than pipeline stages: we actually require a minimum number of threads such that the longest possible dependence between stages is hidden, which for the processors we study in this chapter requires one less thread than there are pipeline stages. This reduction by one thread may be beneficial since it will reduce the latency of individual tasks, result in the same

[5] However, rather than simply shifting, we must now add an offset to register numbers to properly index the physical register file.

overall IPC, and reduce the area by the support for one thread context. Figure 4.7 shows the impact on our CAD metrics of subtracting one thread from the baseline multithreaded implementation. Area is reduced for the 3-stage pipeline, but frequency also drops significantly because the computation of branch targets becomes a critical path. In contrast, for the 5 and 7-stage pipelines we achieve area and power savings respectively, while frequency is nearly unchanged. Overall this possibility gives more flexibility to designers in choosing the number of threads to support for a given pipeline, with potential area or power benefits. When combined with eliminating multicycle paths, reducing the number of threads by one for the 5-stage pipeline improves area-efficiency by 25%, making it 77% more area-efficient than its single-threaded counterpart with 78 MIPS/1000 LEs.

4.7 Summary

We have shown that, relative to single-threaded 3, 5, and 7-stage pipelined processors, multithreading can improve overall IPC by 24%, 45%, and 104% respectively, and area-efficiency by 33%, 77%, and 106%. In particular, we demonstrated that (i) intra-stage pipelining is undesirable for single threaded processors but can provide significant increases in area-efficiency for multithreaded processors; (ii) optimizing unpipelined multicycle paths is key to gaining area-efficiency; (iii) for multithreaded soft processors that 3-operand multiplies are preferable over Hi/Lo registers such as in MIPS; (iv) reducing the registers used can potentially reduce the number of memory blocks used and save area; (v) having one thread less than the number of pipeline stages can give more flexibility to designers while potentially saving area or power. Other than removing registers, which can be detrimental to performance, we will incorporate all the above optimizations in the multithreaded designs that we evaluate in the remainder of this thesis. In summary, this chapter shows that there are significant benefits to multithreaded soft processor designs over single-threaded ones, and gives system designers a strong motivation to program with independent threads. In the next chapter, we investigate the impact of broadening the scope of our experiments from on-chip to off-chip memory.

Chapter 5

Understanding Scaling Trade-offs in Soft Processor Systems

Based on encouraging performance and area-efficiency results in the previous chapter, we are motivated to better understand ways to scale the performance of such multithreaded systems and multicores composed of them. In this chapter, we explore the organization of processors and caches connected to a single off-chip memory channel, for workloads composed of many independent threads. A typical performance goal for the construction of such a system is to fully-utilize a given memory channel. For example, in the field of packet processing the goal is often to process packets at line rate, scaling up a system composed of processors and accelerators to make full use of the available bandwidth to and from a given packet-buffer (i.e., memory channel). In this chapter, we design and evaluate real FPGA-based single-threaded processors, multithreaded processors, and multiprocessor systems connected to DDR SDRAM on EEMBC benchmarks—investigating different approaches to scaling caches, processors, and thread contexts to maximize throughput while minimizing area. For now, we consider systems similar to packet processing where there are many independent tasks/threads to execute, and maximizing system throughput is the over-arching performance goal. Our main finding is that while a single multithreaded processor offers improved performance over a single-threaded processor, multiprocessors composed of single-threaded processors scale better than those composed of multithreaded processors. We next present the two scaling axes that we explore in this chapter:

Scaling Soft Multithreaded Processors In the previous chapter, we demonstrated that a soft multi-

threaded processor can eliminate nearly all of the stall-cycles observed by a comparable single-threaded processor by executing an independent instruction in every stage of the processor pipeline [76]. In this chapter, we extend our multithreaded processors to interface with off-chip DDR memory through caches, which in contrast with uniprocessors presents some interesting challenges and design options. In particular, we present an approach called instruction *replay* to handle cache misses without stalling other threads, and a mechanism for handling the potential live-locks resulting from collisions in the cache from the replayed instructions. We also evaluate several cache organization alternatives for soft multithreaded processors, namely shared, private, and partitioned caches, as well as support for different numbers of hardware contexts. We finally investigate issues related to sharing and conflicts between threads for soft multithreaded processors. In contrast with previous studies of systems with on-chip memory [43, 76], we find with off-chip memory that single-threaded processors are generally more area-efficient than multithreaded processors.

Scaling Soft Multiprocessors We also evaluate multiprocessors composed of uniprocessors or multithreaded processors. We find that, for a system of given area, multiprocessors composed of multithreaded processors provide a much larger number of thread contexts, but that uniprocessor-based multiprocessors provide the best overall throughput.

In this chapter, we are not proposing new architectural enhancements to soft processors: we are rather trying to understand the trade-offs to give us direction when later building soft multiprocessors for packet processing.

5.1 Related Work

Caches built in the FPGA fabric [160] are routinely utilized to improve the performance of systems with off-chip memory. The commercial off-the-shelf soft processors Nios-II [7] and Microblaze [154] both support optional direct-mapped instruction and data caches with configurable cache line sizes. Both processors allocate a cache line upon write misses (allocate-on-write) but the Microblaze uses a write-through policy while NIOS-II uses write-back. Both vendors have extended their instruction set to accomplish tasks such as cache flushing, invalidating and bypassing. Our implementation is comparable to those designs (other than we do not allocate cache lines on writes) and we did not require

ISA extensions: this renders our conclusions widely applicable.

There is little prior work which has studied soft multiprocessor organization with off-chip memory in a real FPGA system, with commercial benchmarks. Fort et al. [43] compared their soft multithreaded processor design to having multiple uniprocessors, using the Mibench benchmarks [50] and a system with on-chip instruction memory and off-chip shared data storage (with no data cache). They conclude that multithreaded soft processors are more area efficient than multiple uniprocessors. We further demonstrated that a soft multithreaded processor can be more area efficient than a single soft uniprocessor with on-chip memory [76].

The addition of off-chip memory and caches introduces variable-latency stalls to the processor pipeline. Handling such stalls in a multithreaded processor without stalling all threads is a challenge. Fort *et. al.* [43] use a FIFO queue of loads and stores, while Moussali *et. al.* [107] use an instruction scheduler to issue ready instructions from a pool of threads. In contrast with either of these approaches, our instruction *replay* approach requires little additional hardware support.

5.2 Experimental Framework

In this section we briefly describe our infrastructure for designing and measuring soft processors, our methodology for comparing soft processor designs, our compilation infrastructure, and the benchmark applications that we study.

Caches The Altera Stratix FPGA that we target provides three sizes of block-memory: M512 (512bits), M4K (4Kbits) and M-RAM (512Kbits). We use M512s to implement register files. In contrast with M-RAM blocks, M4K blocks can be configured to be read and written at the same time (using two ports), such that the read will return the previous value—hence, despite their smaller size, caches built with M4Ks typically out-perform those composed of M-RAMs, and we choose M4K-based caches for our processors.

Platform Our RTL is synthesized, mapped, placed, and routed by `Quartus 7.2` [9] using the default optimization settings. The resulting soft processors are measured on the Transmogrifier platform [42], where we utilize one Altera Stratix FPGA `EP1S80F1508C6` device to (i) obtain the total number of execution cycles, and (ii) to generate a trace which is validated for correctness against the corresponding

execution by an emulator (the MINT MIPS emulator [141]). Our memory controller connects a 64-bit-wide data bus to a 1Gbyte DDR SDRAM DIMM clocked at 133 MHz, and configured to transfer two 64-bit words (i.e., one cache line) on each memory access.

Measurement For Altera Stratix FPGAs, the basic logic element (LE) is a 4-input lookup table plus a flip-flop—hence we report the area of our soft processors in *equivalent LEs*, a number that additionally accounts for the consumed silicon area of any hardware blocks (e.g. multiplication or block-memory units). Even if a memory block is partially utilized by the design, the area of the whole block is nonetheless added to the total area required. For consistency, all our soft processors are clocked at 50 MHz and the DDR remains clocked at 133 MHz. The exact number of cycles for a given experiment is non-deterministic because of the phase relation between the two clock domains, a difficulty that is amplified when cache hit/miss behavior is affected. However, we have verified that the variations are not large enough to significantly impact our measurements.

Compilation and Benchmarks Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 that target variations of the 32-bit MIPS I [66] ISA; for example, for multithreaded processors we implement 3-operand multiplies (rather than MIPS Hi/Lo registers [76, 83]), and eliminate branch and load delay slots. Integer division is implemented in software. Table 5.1 shows the selected benchmarks from the EEMBC suite [39], avoiding benchmarks with significant file I/O that we do not yet support, along with the benchmarks dynamic instruction counts as impacted by different compiler settings. For systems with multiple threads and/or processors, we run multiple simultaneous copies of a given benchmark (i.e., similar to packet processing), measuring the time from the start of execution for the first copy until the end of the last copy.[1]

The processor and caches are clocked together at 50 MHz while the DDR controller is clocked at 133 MHz. There are three main reasons for the reduced clock speed of the processor and caches: i) the original 3-stage pipelined processor with on-chip memory could only be clocked at 72 MHz on the slower speed grade Stratix FPGAs on the TM4; ii) adding the caches and bus handshaking further reduced the clock frequency to 64 MHz; and iii) to relax the timing constraints when arbitrating signals crossing clock domains, we chose a 20 ns clock period which can be obtained via a multiplication of

[1] We verified that in most cases no thread gets significantly ahead of the others.

Table 5.1: EEMBC benchmark applications evaluated. `ST` stands for single-threaded and `MT` stands for multithreaded.

Category	Benchmark	Dyn. Instr. Counts (x10^6)	
		ST	MT
Automotive	a2time01	374	356
	aifirf01	33	31
	basefp01	555	638
	bitmnp01	114	97
	cacheb01	16	15
	canrdr01	38	35
	idctrn01	62	57
	iirflt01	88	84
	puwmod01	17	14
	rspeed01	23	21
	tblook01	149	140
Telecom	autcor00data_2	814	733
	conven00data_1	471	451
	fbital00data_2	2558	2480
	fft00data_3	61	51
	viterb00data_2	765	750
Networking	ip_pktcheckb4m	42	38
	ip_reassembly	385	324
	ospfv2	49	33
	qos	981	732

a rational number with the 133 MHz DDR clock period (7.5 ns). In our evaluation in Section 5.4, we estimate the impact of higher processor clock frequencies that match the actual critical paths of the underlying circuits, and find that the results do not alter our conclusions.

Our system has a load-miss latency of only 8 processor cycles and our DDR controller uses a closed-page policy so that every request opens a DRAM row and then closes it [62]. Furthermore, our current memory controller implementation has room for improvement such as by: (i) setting the column access latency to 2 instead of 3; (ii) tracking open DRAM pages and saving unnecessary row access latency; (iii) fusing the single edge conversion, phase re-alignment, and clock crossing which together amount to a single clock crossing.

5.3 Integrating Multithreaded Processors with Off-Chip Memory

As prior work has shown [43, 76], multithreaded soft processors can hide processor stalls while saving area, resulting in more area-efficient soft systems than those composed of uniprocessors or multiprocessors. Our multithreaded soft processors support fine-grained multithreading, where an instruction for a different thread context is fetched each clock cycle in a round-robin fashion. Such a processor requires the register file and program counter to be logically replicated per thread context. However, since consecutive instructions in the pipeline are from independent threads, we eliminate the need for data hazard detection logic and forwarding lines—assuming that there are at least $N-1$ threads for an N-stage pipelined processor [43, 76]. Our multithreaded processors have a 5-stage pipeline that never stalls: this pipeline depth was found to be the most area-efficient for multithreading in the previous chapter [76]. In this section we describe the challenges in connecting a multithreaded soft processor to off-chip memory through caches, and our respective solutions.

5.3.1 Reducing Cache Conflicts

The workload we assume for this study is comprised of multiple copies of a single task (i.e., similar to packet processing), hence instructions and an instruction cache are easily shared between threads without conflicts. However, since the data caches we study are direct-mapped, when all the threads access the same location in their respective data sections, these locations will all map to the same cache

entry, resulting in pathologically bad cache behavior. As a simple remedy to this problem we pad the data sections for each thread such that they are staggered evenly across the data cache, in particular by inserting multiples of padding equal to the cache size divided by the number of thread contexts sharing the cache. However, doing so makes it more complicated to share instruction memory between threads: since data can be addressed relative to the global pointer, we introduce a short thread-specific initialization routine that adjusts the global pointer by the padding amount; there can also be static pointers and offsets in the program, that we must adjust to reflect the padding. We find that applying this padding increases the throughput of our base multithreaded processor by 24%, hence we apply this optimization for all of our experiments.

5.3.2 Tolerating Miss Latency via Replay

When connected to an off-chip memory through caches, a multithreaded processor will ideally not stall other threads when a given thread suffers a multiple-cycle cache miss. In prior work, Fort *et. al.* [43] use a FIFO queue of loads and stores, while Moussali *et. al.* [107] use an instruction scheduler to issue ready instructions from a pool of threads. For both instruction and data cache misses, we implement a simpler method requiring little additional hardware that we call instruction *replay*. The basic idea is as follows: whenever a memory reference instruction suffers a cache miss, that instruction *fails*—i.e., the program counter for that thread is not incremented. Hence that instruction will execute again (i.e., replay) when it is that thread context's turn again, and the cache miss is serviced while the instruction is replaying. Other threads continue to make progress, while the thread that suffered the miss fails and replays until the memory reference is a cache hit. However, since our processors can handle only a single outstanding memory reference, if a second thread suffers a cache miss it will itself fail and replay until its miss is serviced.

To safely implement the instruction replay technique we must consider how cache misses from different threads might interfere. First, it is possible that one thread can load a cache block into the cache, and then another thread replaces that block before the original thread is able to use it. Such interference between two threads can potentially lead to live-lock. However, we do not have to provide a solution to this problem in our processors because misses are serviced in order and the miss latency is guaranteed to be greater than the latency of a full round-robin of thread contexts—hence a memory

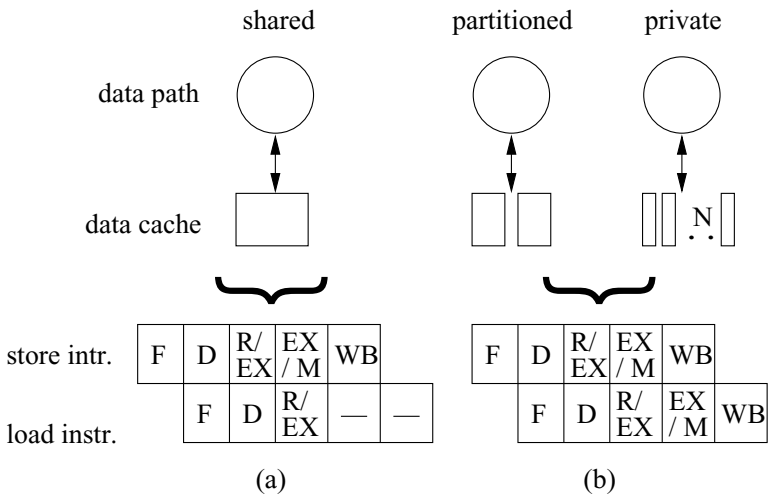


Figure 5.1: Cache organizations and the corresponding impact on the execution of a write hit from one thread followed by a load from a consecutive thread: (a) a shared data cache, for which the load is aborted and later replays (the hazard is indicated by the squashed pipeline slots marked with "—"); (b) partitioned and private caches, for which the load succeeds and there is no hazard. The pipeline logic is divided into fetch `F`, decode `D`, register `R`, execute `EX`, memory `M`, and write-back `WB`.

reference suffering a miss is guaranteed to succeed before the cache line is replaced. However, a second possibility is one that we must handle: the case of a memory reference that suffers a data cache miss, for which the corresponding instruction cache block is replaced before the memory reference instruction can replay. This subtle pathology can indeed result in live-lock in our processors, so we prevent it by saving a copy of the last successfully fetched instruction for each thread context.

5.3.3 Cache Organization

Each thread has its own data section, hence despite our padding efforts (Section 5.3.1), a shared data cache can still result in conflicts. A simple solution to this problem is to increase the size of the shared data cache to accommodate the aggregate data set of the multiple threads, although this reduces the area-saving benefits of the multithreaded design. Furthermore, since our caches are composed of FPGA memory blocks which have only two ports (one connected to the processor, one connected to the DRAM channel), writes take two cycles: one cycle to lookup and compare with the tag, and another cycle to perform the write (on a hit). As illustrated in Figure 5.1(a), this can lead to further contention between consecutive threads in a multithreaded processor that share a cache: if a second consecutive thread

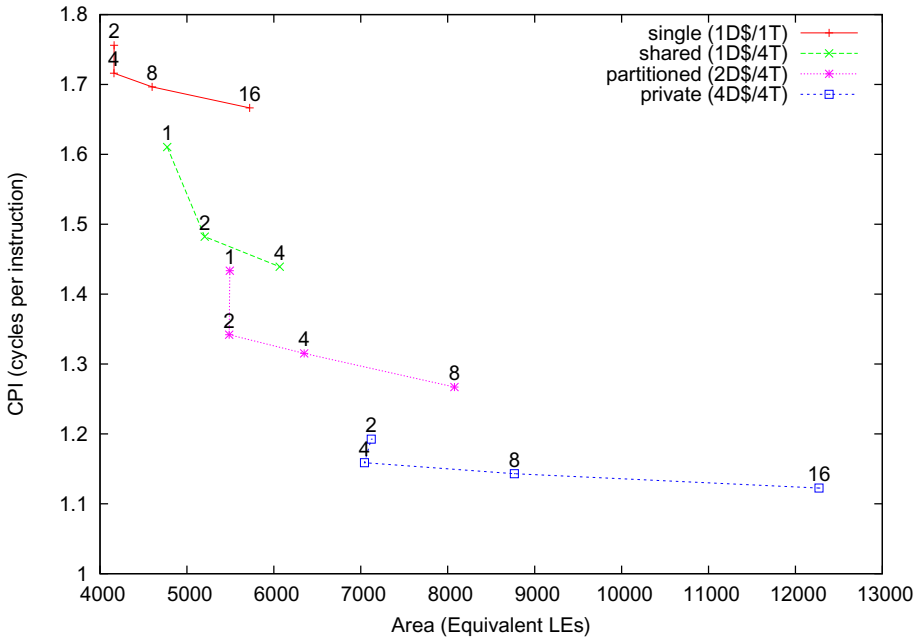


Figure 5.2: CPI versus area for the *single* threaded processor and for the multithreaded processors with *shared*, *partitioned*, and *private* caches. 1D$/4T means there is one data cache and 4 threads total. Each point is labeled with the total cache capacity in KB available per thread.

attempts a memory reference directly after a write-hit, we apply our failure/replay technique for the second thread, rather than stall that thread and subsequent threads.

Rather than increasing the size of the shared cache, we consider two alternatives. The first is to have *private caches* such that each thread context in the multithreaded processor accesses a dedicated data cache. The second, if the number of threads is even, is to have *partitioned caches* such that non-consecutive threads share a data cache—for example, if there are four threads, threads 1 and 3 would share a cache and threads 2 and 4 would share a second cache. As shown in Figure 5.1(b), both of these organizations eliminate port contention between consecutive threads, and reduce (partitioned) or eliminate (private) cache block conflicts between threads.

5.4 Scaling Multithreaded Processor Caches

In this section we compare single-threaded and multithreaded soft processors, and study the impact of cache organization and thread count on multithreaded processor performance and area efficiency.

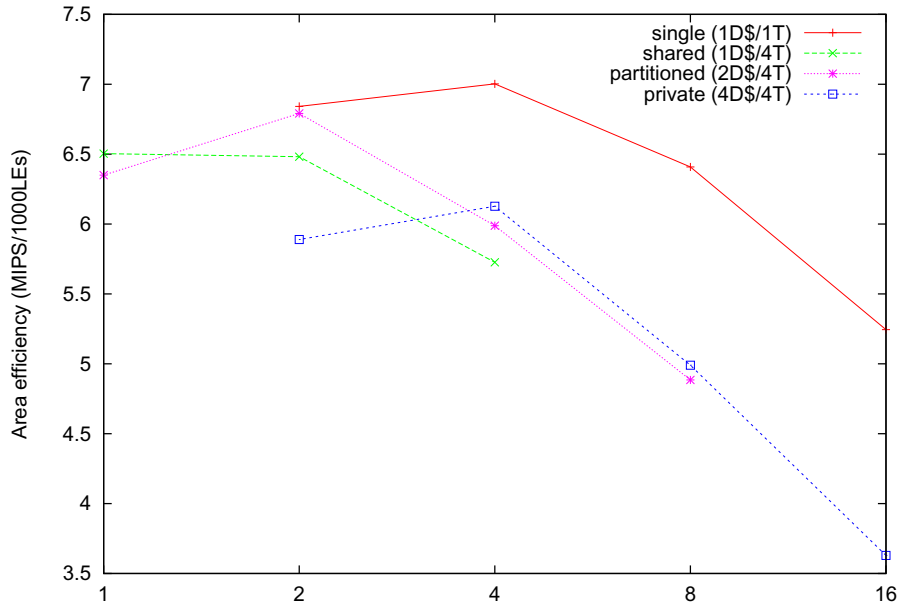
In Figure 5.2 we plot performance versus area for the single threaded processor and the three

possible cache organizations for multithreaded processors (shared, partitioned, and private), and for each we vary the sizes of their caches. For performance, we plot cycles-per-instruction (*CPI*), which is computed as the total number of cycles divided by the total number of instructions executed; we use this metric as opposed to simply execution time because the single-threaded and multithreaded processors run different numbers of threads, and because the compilation of benchmarks for the single-threaded and multithreaded processors differ (as shown in Table 5.1). CPI is essentially the inverse of throughput for the system, and this is plotted versus the area in equivalent LEs for each processor design—hence the most desirable designs minimize both area and CPI.

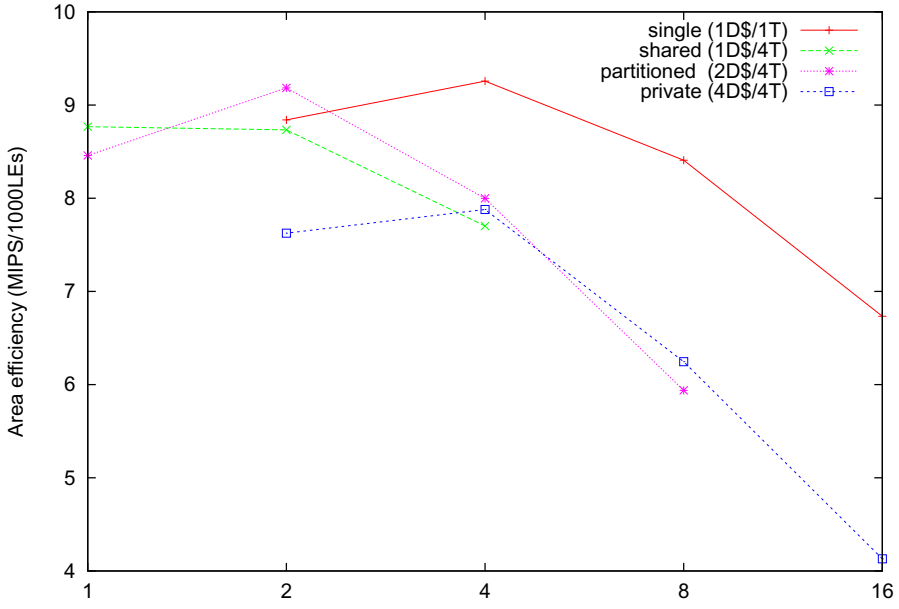
We first observe that the single-threaded and different multithreaded processor designs with various cache sizes allow us to span a broad range of the performance/area space, giving a system designer interested in supporting only a small number of threads the ability to scale performance by investing more resources. The single-threaded processor is the smallest but provides the worst CPI, and this is improved only slightly when the cache size is doubled (from 2KB to 4KB). Of the multithreaded processors, the shared, partitioned, and private cache designs provide increasing improvements in CPI at the cost of corresponding increases in area. The shared designs outperform the single-threaded processor because of the reduced stalls enjoyed by the multithreaded architecture. The partitioned designs outperform the shared designs as they eliminate replays due to contention. The private cache designs provide the best performance as they eliminate replays due to both conflicts and contention, but for these designs performance improves very slowly as cache size increases.

In Figure 5.2, there are several instances where increasing available cache appears to cost no additional area: a similar behavior is seen for the single-threaded processor moving from 2KB to 4KB of cache and for the partitioned multithreaded processor moving from 1KB to 2KB of cache per thread. This is because the smaller designs partially utilize M4K memories, while in the larger designs they are more fully utilized—hence the increase appears to be free since we account for the entire area of an M4K regardless of whether it is fully utilized. For the private-cache multithreaded designs, moving from 2KB to 4KB of cache per thread actually saves a small amount of area, for similar reasons plus additional savings in LEs due to fortuitous mapping behavior.

To better understand the trade-off between performance and area for different designs, it is instructive to plot their area efficiency as shown in Figure 5.3(a). We measure area efficiency as millions



(a) 50MHz processor clock frequency.



(b) Max allowable processor clock frequency.

Figure 5.3: Area efficiency versus total cache capacity per thread for the single-threaded processor and the multithreaded processors, reported using (a) the implemented clock frequency of 50MHz, and (b) the maximum allowable clock frequency per processor design.

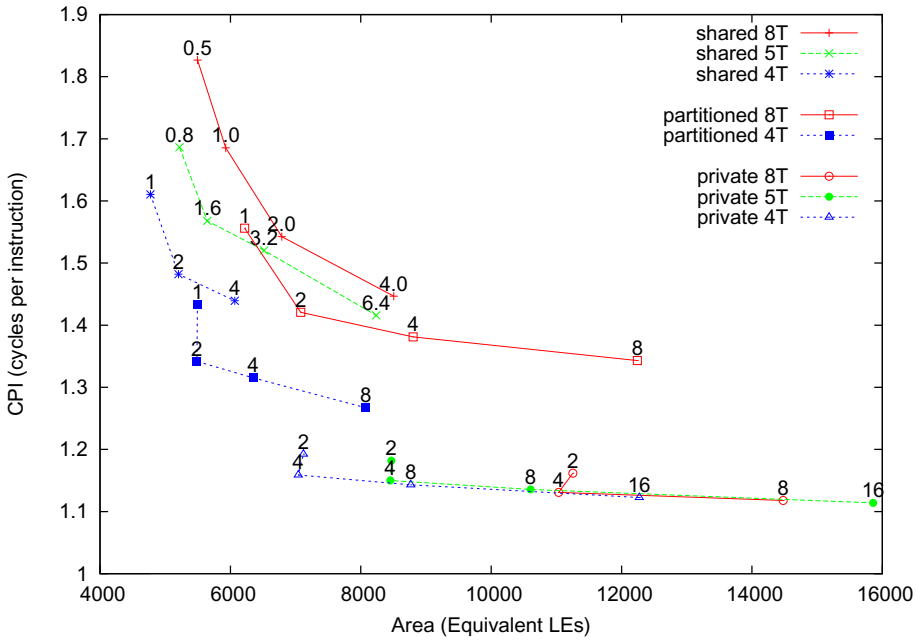


Figure 5.4: CPI versus area for multithreaded designs supporting varying numbers of thread contexts. Each point is labeled with the total available cache capacity per thread.

of instructions per second (MIPS) per 1000 LEs. The single-threaded processors are the most area-efficient, in contrast with previous work comparing similar processors with on-chip memory and without caches [76], as we provide a private data cache storage for each thread in the multithreaded core. The partitioned design with 2KB of cache per thread is nearly as area-efficient as the corresponding single-threaded processor. The shared-cache designs with 1KB and 2KB of cache per thread are the next most area-efficient, with the private-cache designs being the least area-efficient. These results tell us to expect the single-threaded and partitioned-cache multithreaded designs to scale well, as they provide the best performance per area invested.

Due to limitations of the Transmogrifier platform, all of our processors are actually clocked at 50MHz, while their maximum possible frequencies (i.e., f_{max}) are on average 65MHz. To investigate whether systems measured using the true maximum possible frequencies for the processors would lead to different conclusions, we estimate this scenario in Figure 5.3(b). We observe that the relative trends are very similar, with the exception of the single-threaded processor with 2KB of cache for which the area efficiency drops below that of the corresponding partitioned design to close to that of the shared cache design.

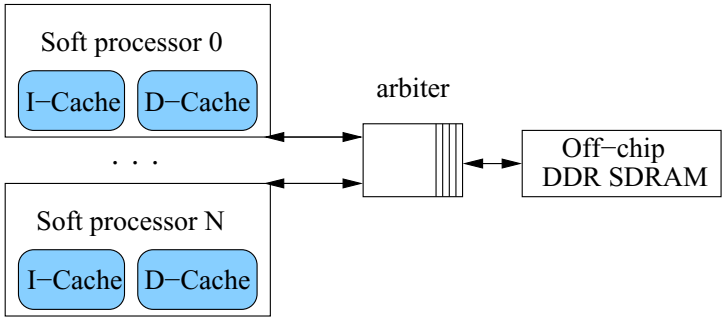


Figure 5.5: Diagram showing an arbiter connecting multiple processor cores in a multiprocessor.

Impact of Increasing Thread Contexts Our multithreaded processors evaluated so far have all implemented a minimal number of threads contexts: four thread contexts for five pipeline stages. To justify this choice, we evaluated multithreaded processors with larger numbers of threads for the different cache designs and for varying amounts of available cache per thread (shown in Figure 5.4). For shared and partitioned designs we found that increasing the number of thread contexts (i) increases the CPI, due to increased contention and conflicts, and (ii) increases area, due to hardware support for the additional contexts. Since the private cache designs eliminate all contention and conflicts, there is a slight CPI improvement as area increases significantly with additional thread contexts. These results confirmed that the four-thread multithreaded designs are the most desirable.

5.5 Scaling Multiprocessors

For systems with larger numbers of threads available, another alternative for scaling performance is to instantiate multiple soft processors. In this section we explore the design space of soft multiprocessors, with the goals of maximizing (i) performance, (ii) utilization of the memory channel, and (iii) utilization of the resources of the underlying FPGA. To support multiple processors we augment our DRAM controller with an arbiter that serializes requests by queuing up to one request per processor (shown in Figure 5.5); note that this simple interconnect architecture does not impact the clock frequencies of our processors.

In Figure 5.6 we plot CPI versus area for multiprocessors composed of single-threaded or multithreaded processors; we replicate the processor designs that were found to be the most area-efficient according to Figure 5.3(a). For each multiprocessor design, each design point has double

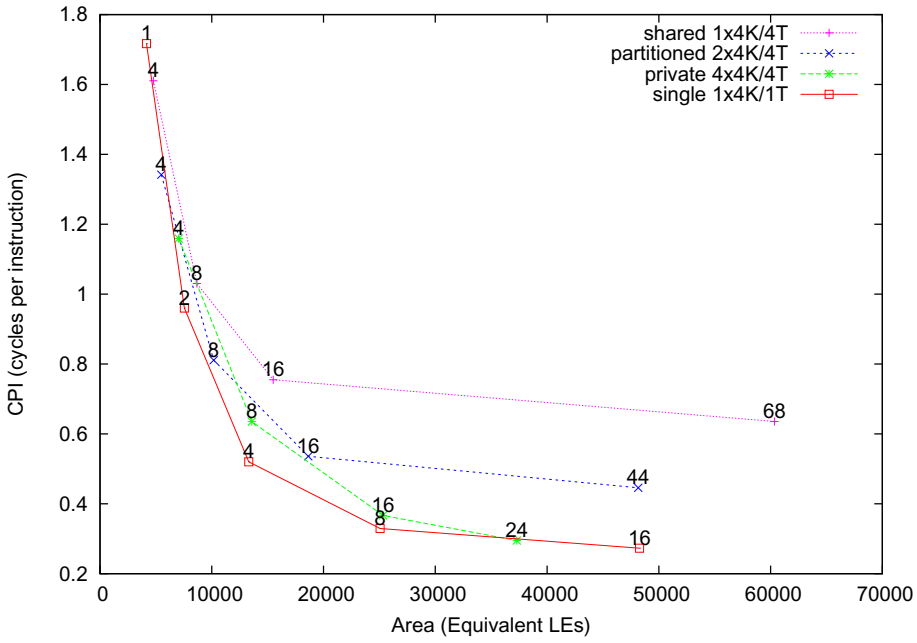


Figure 5.6: CPI versus area for multiprocessors composed of single-threaded and multithreaded processors. 1x4K/4T means that each processor has one 4KB data cache and 4 thread contexts, and each point is labeled with the total number of thread contexts supported.

the number of processors as the previous, with the exception of the largest (rightmost) for which we plot the largest design supported by the FPGA—in this case the design that has exhausted the M4K block memories.

Our first and most surprising observation is that the Pareto frontier (the set of designs that minimize CPI and area) is mostly comprised of single-threaded multiprocessor designs, many of which outperform multithreaded designs that support more than twice the number of thread contexts. For example, the 16-processor single-threaded multiprocessor has a lower CPI than the 44-thread-context partitioned-cache multithreaded multiprocessor of the same area. We will pursue further insight into this result later in this section. For the largest designs, the private-cache multithreaded multiprocessor provides the second-best overall CPI with 24 thread contexts (over 6 processors), while the partitioned and shared-cache multithreaded multiprocessor designs perform significantly worse at a greater area cost.

Sensitivity to Cache Size Since the shared and partitioned-cache multithreaded designs have less cache per thread than the corresponding private-cache multithreaded or single-threaded designs, in Figure 5.7 we investigate the impact of doubling the size of the data caches (from 4KB to 8KB per cache) for those

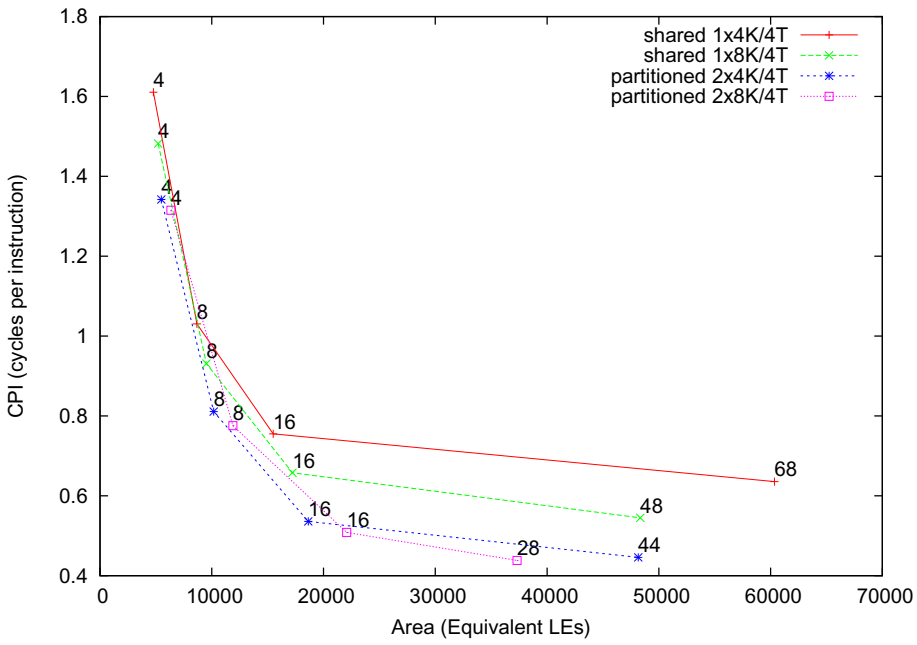
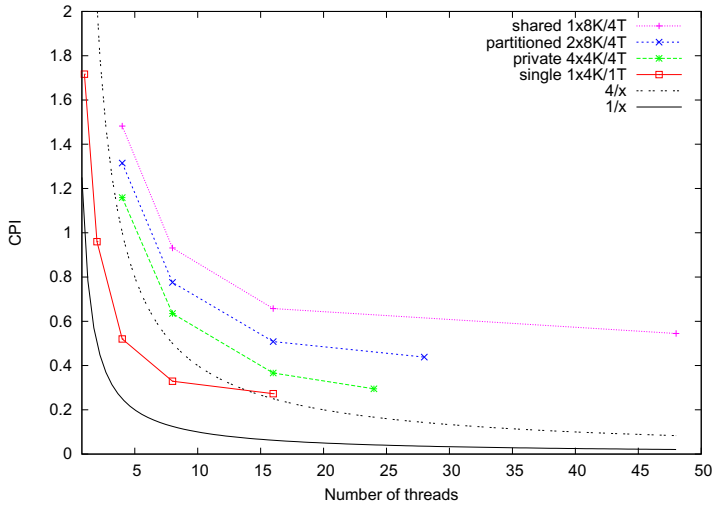


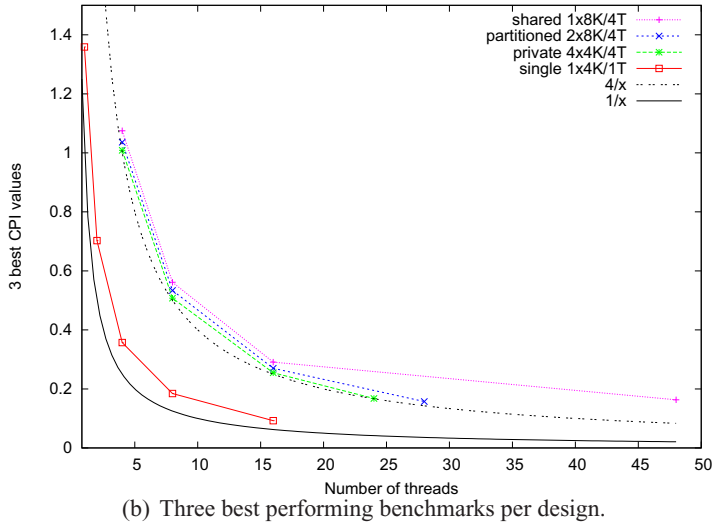
Figure 5.7: CPI versus area for the shared and partitioned designs as we increase the size of caches from 4KB to 8KB each. Each point is labeled with the total number of thread contexts supported.

two designs. We observe that this improves CPI significantly for the shared-cache designs and more modestly for the partitioned cache design, despite the fact that the 4KB designs were the most area-efficient (according to Figure 5.3(a)). Hence we evaluate the 8KB-cache shared and partitioned-cache designs in subsequent experiments.

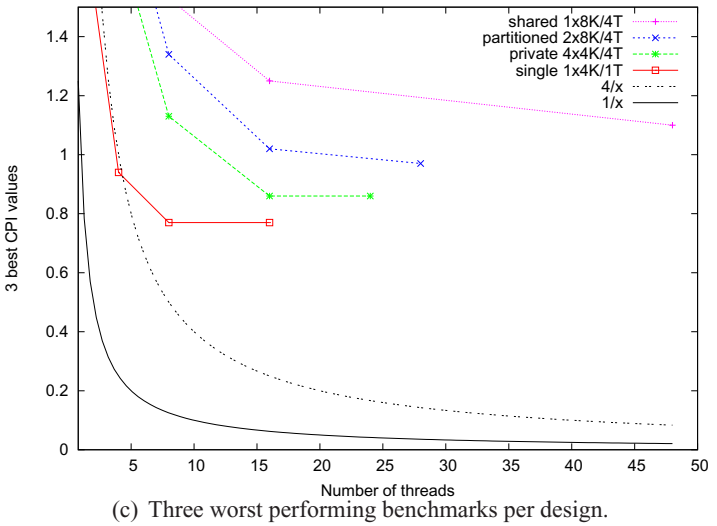
Per-Thread Efficiency In this section, we try to gain an understanding of how close to optimality each of our architectures performs—i.e., how close to a system that experiences no stalls. The optimal CPI is 1 for our single-threaded processor, and hence $1/X$ for a multiprocessor composed of X single-threaded processors. For one of our multithreaded processors the optimal CPI is also 1, but since there are four thread contexts per processor, the optimal CPI for X multithreaded processors is $4/X$. In Figure 5.8(a) we plot CPI versus total number of thread contexts for our single and multithreaded designs, as well as the two ideal curves (as averaged across all of our benchmarks). As expected, for a given number of threads the single-threaded processors exhibit better CPI than the corresponding multithreaded designs. However, it is interesting to note that the private-cache multithreaded designs perform closer to optimally than the single-threaded designs. For example, with 16 threads (the largest design), the single-threaded multiprocessor has a CPI that is more than 4x greater than optimal, but



(a) All benchmarks.



(b) Three best performing benchmarks per design.



(c) Three worst performing benchmarks per design.

Figure 5.8: CPI versus total thread contexts across all benchmarks (a), and the three best (b) and worst (c) performing benchmarks per design.

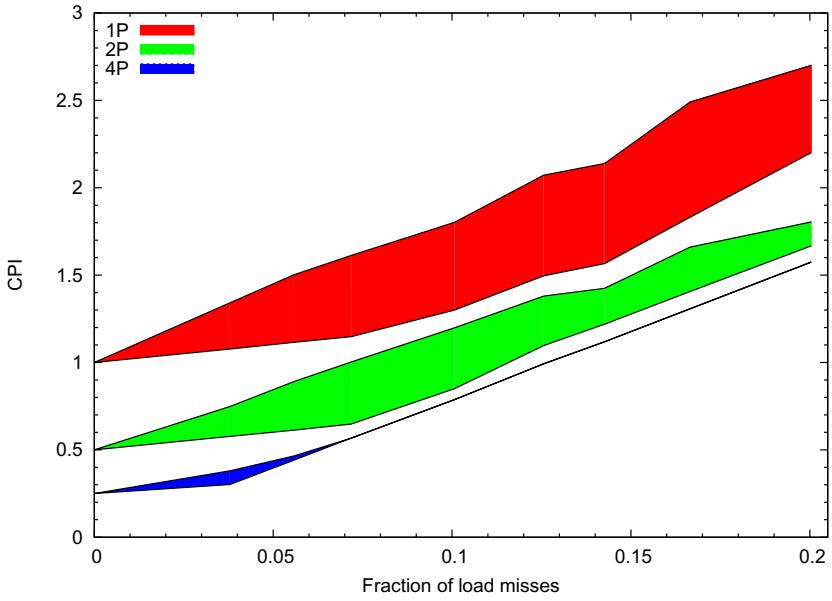


Figure 5.9: CPI versus fraction of load misses. For each surface the upper edge plots the single-threaded multiprocessor designs (4KB data cache per processor) and the lower edge plots the private-cache multithreaded multiprocessor designs (4KB data cache per thread context).

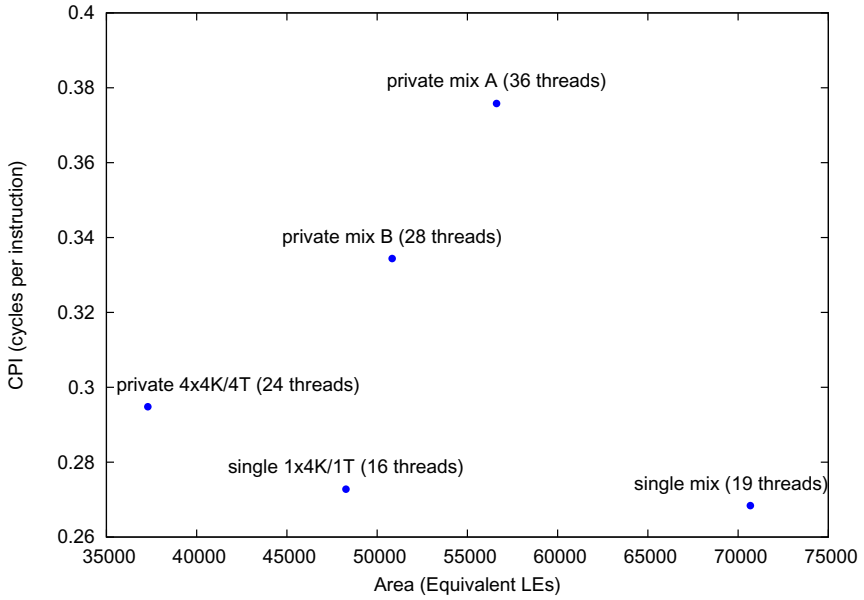
regardless, this design provides the lowest CPI across any design. This graph also illustrates that private-cache designs outperform partitioned-cache designs which in turn outperform shared-cache designs.

Potential for Customization A major advantage of soft systems is the ability to customize the hardware to match the requirements of the application—hence we are motivated to investigate whether the multithreaded designs might dominate the single-threaded design for certain applications. However, we find that this is not the case. To summarize, in Figures 5.8(b) and 5.8(c) we plot CPI versus total number of thread contexts for the three best performing and three worst performing benchmarks per design, respectively. For neither extremity do the multithreaded designs outperform the single-threaded designs. Looking at Figure 5.8(b), we see that for the best performing benchmarks the private-cache multithreaded designs perform nearly optimally. In the worst cases, the single-threaded designs maintain their dominance, despite the 16-processor design performing slightly worse than the 8-processor design.

Understanding the Single-Threaded Advantage To clarify the advantage of the single-threaded multiprocessor designs, we use a synthetic benchmark that allows us to vary the density of load misses. In particular, this benchmark consists of a thousand-instruction loop comprised of loads and no-ops,

and the loads are designed to always miss in the data cache. This benchmark allows us to isolate the impact of load misses since they can be the only cause of stalls in our processors. In Fig 5.9 we compare the CPI for single-threaded designs with the private-cache multithreaded designs as the fraction of load misses increases. In particular, for a certain number of processors we plot a surface such that the top edge of the surface is the single-threaded design and the bottom edge is the corresponding private-cache multithreaded design. Hence the three surfaces show this comparison for designs composed of 1, 2, and 4 processors. Looking at the surface for a single processor, as the fraction of load misses increases, the multithreaded processor (bottom edge) gains a somewhat consistent CPI advantage (of about 0.5 at 10% misses) over the single-threaded processor (top edge). However, this advantage narrows as the number of processors–and the resulting pressure on the memory channel–increases, and for four processors, the multithreaded designs have only a negligible advantage over the single-threaded designs—and the multithreaded processors require marginally greater area than their single-threaded counterparts.

Exploiting Heterogeneity In contrast with ASIC designs, FPGAs provide limited numbers of certain resources, for example block memories. This leads to an interesting difference when targeting an FPGA as opposed to an ASIC: replicating only the same design will eventually exhaust a certain resource while under-utilizing others. For example, our largest multiprocessor design (the 68-thread shared-cache multithreaded design) uses 99% of the M4K block memories but only 43% of the available LEs. Hence we are motivated to exploit heterogeneity in our multiprocessor design to more fully utilize all of the resources of the FPGA—in this case we consider adding processors with M-RAM based caches despite the fact that individually they are less efficient than their M4K-based counterparts. In particular, we extend our two best-performing multiprocessor designs with processors having M-RAM-based caches as shown in Figure 5.10. In this case, extending the multithreaded processor with further processors does not improve CPI but only increases total area, i.e. potentially taking away resources available for other logic in the FPGA. For the single-threaded case, the heterogeneous design improves CPI slightly but at a significant cost in area—however, this technique does allow us to go beyond the previously maximal design.



Composition of Heterogeneous Multiprocessors			
	Private mix A	Private mix B	Single mix
M4K-based	mt_private x 5	mt_private x 5	st x 15
M-RAM-based	mt_shared x 4	mt_partitioned x 2	st x 4
Total threads	36	28	19

Figure 5.10: CPI versus area for our two best-performing maximal designs (the 16-thread-context single-threaded design and the 24-thread-context private-cache multithreaded design), and for those designs extended with processors with M-RAM-based caches.

5.6 Summary

In this chapter we explored architectural options for scaling the performance of soft systems, focussing on the organization of processors and caches connected to a single off-chip memory channel, for workloads composed of many independent threads. For soft multithreaded processors, we present the technique of instruction *replay* to handle cache misses processors without stalling other threads. Our investigation of real FPGA-based processor, multithreaded processor, and multiprocessor systems has led to a number of interesting conclusions. First, we showed that multithreaded designs help span the throughput/area design space, and that private-cache based multithreaded processors offer the best performance. Looking at multiprocessors, we found that designs based on single-threaded processors perform the best for a given total area, followed closely by private-cache multithreaded multiprocessor designs. We demonstrated that as the number of processors increases, multithreaded processors lose their latency-hiding advantage over single-threaded processors, as both designs become bottlenecked on the memory channel. Finally, we showed the importance of exploiting heterogeneity in FPGA-based multiprocessors to fully utilize a diversity of FPGA resources when scaling-up a design. Armed with this knowledge of multiprocessors, we attempt to build our packet processing system on the circuit board with gigabit network interfaces that we describe in the next chapter.

Chapter 6

NetThreads: A Multithreaded Soft Multiprocessor

To avoid the tedious and complicated process of implementing a networking application in low-level hardware-description language (which is how FPGAs are normally programmed), we instead propose to run the application on *soft processors* – processors composed of programmable logic on the FPGA. To build such a system, we leverage the compiler optimizations, the instruction replay mechanism and the multithreading with off-chip memory from Chapters 4 and 5, while still aiming to execute applications with the run-to-completion paradigm discussed in Chapter 3. In this chapter, we briefly describe the baseline NetThreads [81] multithreaded multiprocessor system that allows us to program the NetFPGA platform [91] in software using shared memory and conventional lock-based synchronization; we also motivate our choice of the NetFPGA board as our experimentation platform for packet processing. Next, we describe our benchmarks and show the baseline performance of our system. This architecture and platform serves as a starting point for evaluating new soft systems architectures in the subsequent chapters. We conclude this section by describing how this infrastructure was also successfully used as the enabling platform for two other projects led by software programmers in the network research community.

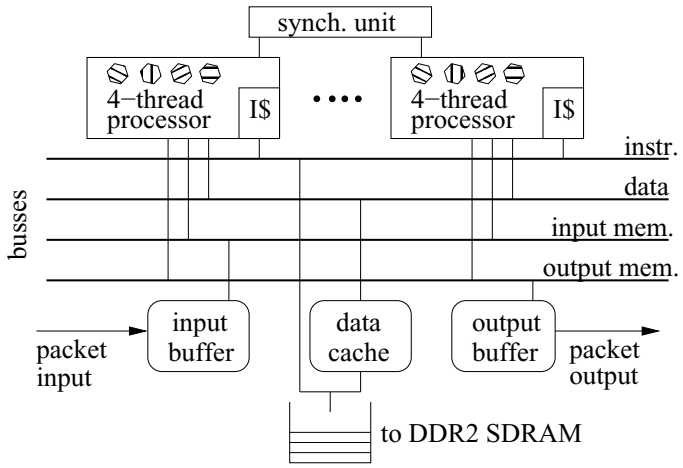


Figure 6.1: The architecture of a 2-processor soft packet multiprocessor. The suspension dots indicate that the architecture can allow for more cores (see Appendix B).

6.1 Multithreaded Soft Multiprocessor Architecture

Our base processor is a single-issue, in-order, 5-stage, 4-way multithreaded processor, shown to be the most area efficient compared to a 3- and 7-stage pipeline in Chapter 5. While the previous chapter also showed us that single-threaded processors could implement more cores in a fixed amount of area, the situation is different here. Because we are now using a Xilinx FPGA with coarser 18kbits block RAMs, and we assign all our thread contexts to perform the same packet processing in a *shared*—rather than private—data memory space, the multithreaded processors actually do not require more block RAMs than single-threaded cores (in this baseline architecture), and block RAMs were the limiting factor to adding more cores in Chapter 5. Since the FPGA logic elements are not a critical limiting factor either (see Section 6.2), we are in a situation where it is most advantageous to capitalize on the better CPI of multithreaded processors (as demonstrated in Chapter 4). Also, in Chapter 5 we assumed independent application threads, whereas from this chapter onwards we use real packet processing applications with threads that share data and synchronize, so we expect more stalls related to synchronization that multithreading can help tolerate.

As shown in Figure 6.1 and summarized in Table 6.1, the memory system of our packet processing design is composed of a private instruction cache for each processor, and three data memories that are shared by all processors; this organization is sensitive to the two-port limitation of block RAMs available on FPGAs. The first memory is an input buffer that receives packets on one port and services processor

Table 6.1: On-chip memories.

Memory	Description
Input buffer	Receives packets on one port and services processor requests on the other port, read-only, logically divided into ten fixed-sized packet slots.
Output buffer	Sends packets to the NetFPGA MAC controllers on one port, connected to the processors via its second port.
Data cache	Connected to the processors on one port, 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [134]) on the other port.
Input/Output buffers and Data cache	16KB, single-cycle random access, arbitrated across processors, 32 bits bus.
Instruction caches	16KB, single-cycle random access, private per processor, 32 bits bus.

requests on the other port via a 32-bit bus, arbitrated across processors. The second is an output memory buffer that sends packets to the NetFPGA output-queues on one port, and is connected to the processors via a second 32-bit arbitrated bus on the second port. Both input and output memories are 16KB, allow single-cycle random access and are controlled through memory-mapped registers; the input memory is read-only and is logically divided into ten fixed-sized packet slots able to hold at most one packet each. The third is a shared memory managed as a cache, connected to the processors via a third arbitrated 32-bit bus on one port, and to a DDR2 SDRAM controller on the other port. For simplicity, the shared cache performs 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [134]), which is clocked at 200MHz. The SDRAM controller services a merged load/store queue in-order; since this queue is shared by all processors it serves as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores (as opposed to the increased complexity of having private data caches). In Chapter 7, the queue has 16 entries but using the techniques proposed in that chapter, we are able to increase the size of the queue to 64 for the remaining chapters. Finally, each processor has a dedicated connection to a synchronization unit that implements 16 mutexes. In our current instantiation of NetThreads, 16 mutexes is the maximum number that we can support while meeting the 125MHz target clock speed. In the NetThreads ISA, each lock/unlock operation specifies a unique identifier, indicating one of these 16 mutexes.

Similar to other network processors [25, 56], our packet input/output and queuing in the input and output buffers are hardware-managed. In addition to the input buffer in Figure 6.1, the NetFPGA framework can buffer incoming packets for up to 6100 bytes (4 maximally sized packets) but the small

overall input storage, while consistent with recent findings that network buffers should be small [17], is very challenging for irregular applications with high computational variance and conservatively caps the maximum steady-state packet rate sustainable via packets dropped at the input of the system.

Since our multiprocessor architecture is bus-based, in its current form it will not easily scale to a large number of processors. However, as we demonstrate later in Section 7.3, our applications are mostly limited by synchronization and critical sections, and not contention on the shared buses; in other words, the synchronization inherent in the applications is the primary roadblock to scalability.

6.2 NetThreads Infrastructure

This section describes our evaluation infrastructure, including compilation, our evaluation platform, and how we do timing, validation, and measurement.

Compilation: Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 that target variations of the 32-bit MIPS-I [66] ISA. We modify MIPS to eliminate branch and load delay slots (as described in Section 4.1 [83]). Integer division and multiplication, which are not heavily used by our applications, are both implemented in software. To minimize cache line conflicts in our direct-mapped data cache, we align the top of the stack of each software thread to map to equally-spaced blocks in the data cache. The processor is big-endian which avoids the need to perform network-to-host byte ordering transformations (IP information is stored in packet headers using the big-endian format). Network processing software is normally closely-integrated with operating system networking constructs; because our system does not have an operating system, we instead inline all low-level protocol-handling directly into our programs. To implement time-stamps and time-outs we require the FPGA hardware to implement a device that can act as the system clock using a counter.

Platform: Our processor designs are inserted inside the NetFPGA 2.1 `Verilog` infrastructure [91] that manages four 1GigE Media Access Controllers (MACs), which offer a considerable bandwidth (see Section 2.1.3). We added to this base framework a memory controller configured through the Xilinx Memory Interface Generator to access the 64 Mbytes of on-board DDR2 SDRAM clocked at 200MHz. The system is synthesized, mapped, placed, and routed under high effort to meet timing constraints

by Xilinx ISE 10.1.03 and targets a Virtex II Pro 50 (speed grade 7ns). Other than using the network interfaces, our soft processors can communicate through DMA on a PCI interface to a host computer when/if necessary. This configuration is particularly well suited for many packet processing applications because: (i) the load of the soft processors is isolated from the load on the host processor, (ii) the soft processors suffer no operating system overheads, (iii) they can receive and process packets in parallel, and (iv) they have access to a high-resolution system timer (much higher than that of the host timer). As another advantage, this platform is well supported and widely-used by researchers worldwide [90], thus offering a potential group of early adopters for our soft systems.

FPGA resource utilization Our two-CPU full system hardware implementation consumes 165 block RAMs (out of 232; i.e., 71% of the total capacity). The design occupies 15,671 slices (66% of the total capacity) and more specifically, 23158 4-input LUTs when optimized with high-effort for speed. Considering only a single CPU, the post-place & route timing results give an upper bound frequency of 129MHz.

Timing: Our processors run at the clock frequency of the Ethernet MACs (125MHz) because there are no free PLLs (Xilinx DCMs) after merging-in the NetFPGA support components. Due to these stringent timing requirements, and despite some available area on the FPGA, (i) the private instruction caches and the shared data write-back cache are both limited to a maximum of 16KB, and (ii) we are also limited to a maximum of two processors. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA.

Validation: At runtime in debug mode and in RTL simulation (using `Modelsim` 6.3c), the processors generate an execution trace that has been validated for correctness against the corresponding execution by a simulator built on MINT [141]. We validated the simulator for timing accuracy against the RTL simulation.

Measurement: We drive our design with a modified `Tcpreplay` 3.4.0 that sends packet traces from a Linux 2.6.18 Dell PowerEdge 2950 system, configured with two quad-core 2GHz Xeon processors and a Broadcom NetXtreme II GigE NIC connecting to a port of the NetFPGA used for input and a NetXtreme GigE NIC connecting to another NetFPGA port used for output. We characterize the throughput of the system as being the maximum sustainable input packet rate obtained by finding, through a bisection search, the smallest fixed packet inter-arrival time where the system does not drop

any packet when monitored for five seconds—a duration empirically found long enough to predict the absence of future packet drops at that input rate.

To put the performance of the soft processors in perspective, handling a 10^9 bps stream (with a standard inter-frame gap equivalent to the delay of transmitting 12 bytes at 1Gbps), i.e. a maximized 1Gbps link, with 2 processors running at 125 MHz, implies a maximum of 152 cycles per packet for minimally-sized 64B packets; and 3060 cycles per packet for maximally-sized 1518B packets.

6.3 Baseline Performance

In this section, we present our system performance in terms of latency and throughput, showing the potential and limitations of our implementation. We then focus on describing where the performance bottlenecks are located.

6.3.1 Latency

When replying to ICMP ping packets (an example of minimum useful computation) with only 1 thread out of 4 executing in round-robin, we measured a latency of 5.1μs with a standard deviation of 44ns. By comparison, when using an HP DL320 G5p server (Quad Xeon 2.13GHz) running Linux 2.6.26 and equipped with an HP NC326i PCIe dual-port gigabit network card as the host replying to the ping requests, we measured an average round-trip time of 48.9μs with a standard deviation of 17.5μs. NetThreads therefore offers a latency on average 9.6 times shorter, with a standard deviation 397 times smaller.

6.3.2 Throughput

We begin by evaluating the raw performance that our system is capable of, when performing minimal packet processing for tasks that are completely independent (i.e., unsynchronized). We estimate this upper-bound by implementing a packet echo application that simply copies packets from an input port to an output port. With minimum-sized packets of 64B, the echo program executes 300$\pm$10 dynamic instructions per packet, and a single round-robin CPU can echo 124 thousand packets/sec (i.e., 0.07 Gbps). With 1518B packets, the maximum packet size allowable by Ethernet, each echo task requires 1300$\pm$10 dynamic instructions per packet. In this case, the generator software on the host server (see

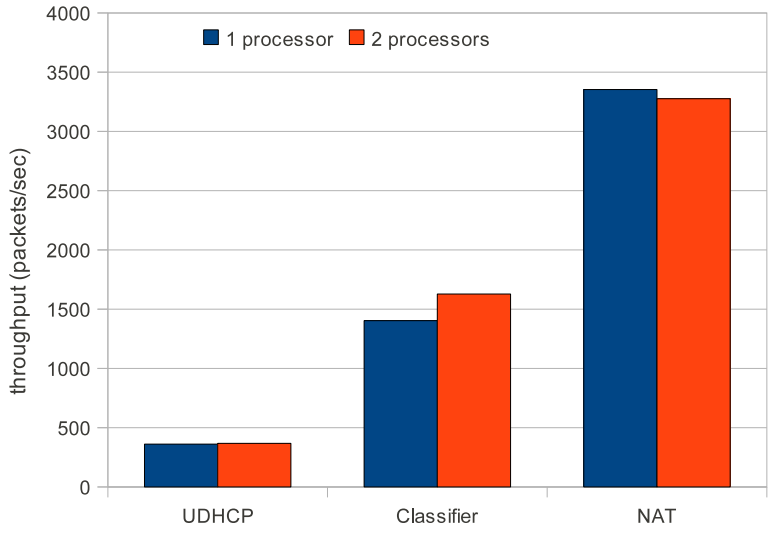


Figure 6.2: Throughput (in packets per second) measured on the NetFPGA with either `1` or `2` CPUs.

Section 6.2) sends copies of the same preallocated ping request packet through `Libnet` 1.4. With two CPUs and 64B packets, or either one or two CPUs and 1518B packets, our PC-based packet generator cannot generate packets fast enough to saturate our system (i.e., cannot cause packets to be dropped). This amounts to more than 58 thousand packets/sec (>0.7 Gbps). Hence the scalability of our system will ultimately be limited by the amount of computation per packet/task and the amount of parallelism across tasks, rather than the packet input/output capabilities of our system.

Figure 6.2 shows the maximum packet throughput of our (real) hardware system with thread scheduling. We find that our applications do not benefit significantly from the addition of a second CPU due to increased lock and bus contention and cache conflicts: the second CPU either slightly improves or degrades performance, motivating us to determine the performance-limiting factors.

6.3.3 Identifying the Bottlenecks

To reduce the number of designs that we would pursue in real hardware, and to gain greater insight into the bottlenecks of our system, we developed a simulation infrastructure. While verified for timing accuracy, our simulator cannot reproduce the exact order of events that occurs in hardware, hence there is some discrepancy in the reported throughput. For example, `Classifier` has an abundance of control paths and events that are sensitive to ordering such as routines for allocating memory, hash table access, and assignment of mutexes to flow records. We depend on the simulator only for an approximation of

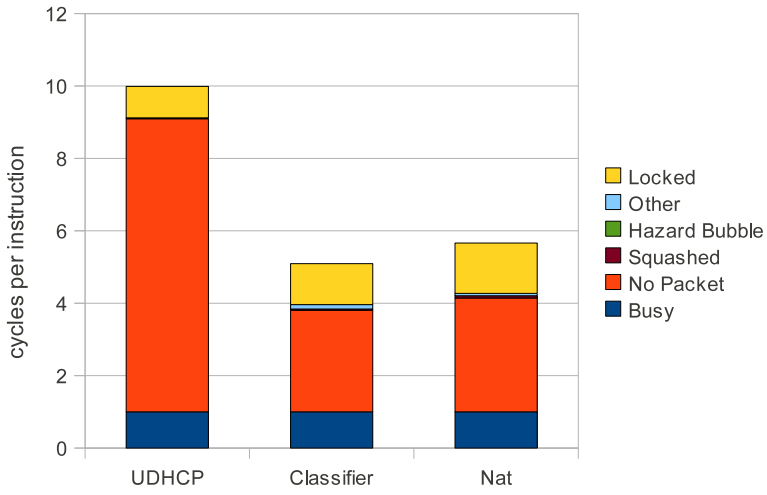


Figure 6.3: Breakdown of how cycles are spent for each instruction (on average) in simulation.

the relative performance and behavior of applications on variations of our system.

To obtain a deeper understanding of the bottlenecks of our system, we use our simulator to obtain a breakdown of how cycles are spent for each instruction, as shown in Figure 6.3. In the breakdown, a given cycle can be spent executing an instruction (`busy`), awaiting a new packet to process (`no packet`), awaiting a lock owned by another thread (`locked`), squashed due to a mispredicted branch or a preceding instruction having a memory miss (`squashed`), awaiting a pipeline hazard (`hazard bubble`), or aborted for another reason (`other`, memory misses or bus contention). The fraction of time spent waiting for packets (`no packet`) is significant and we verified in simulation that it is a result of the worst-case processing latency of a small fraction of packets, since the packet rate is held constant at the maximum sustainable rate.

In Table 6.2, we measure several properties of the computation done per packet in our system. First, we observe that task size (measured in dynamic instructions per second) has an extremely large variance (the standard deviation is larger than the mean itself for all three applications). This high variance is partly due to our applications being best-effort unpipelined `C` code implementations, rather than finely hand-tuned in assembly code as packet processing applications often are. We also note that the applications spend over 90% of the packet processing time either awaiting synchronization or within critical sections (dynamic synchronized instructions), which limits the amount of parallelism and the overall scalability of any implementation, and in particular explains why our two CPU implementation

Benchmark	Dyn. Instr. ×1000 /packet	Dyn. Sync. Instr. %/packet	Sync. Uniq. Addr. /packet	
			Reads	Writes
UDHCP	34.9±36.4	90±105	5000±6300	150±60
Classifier	12.5±35.0	94±100	150±260	110±200
NAT	6.0±7.1	97±118	420±570	60±60
Intruder	12527±18839	10.7±6.3	37±55	23±20
Intruder2	12391±18331	4.9±3.4	61±10	11±14

Table 6.2: Application statistics (mean±standard-deviation): dynamic instructions per packet, dynamic synchronized instructions per packet (i.e., in a critical section) and number of unique synchronized memory read and write accesses.

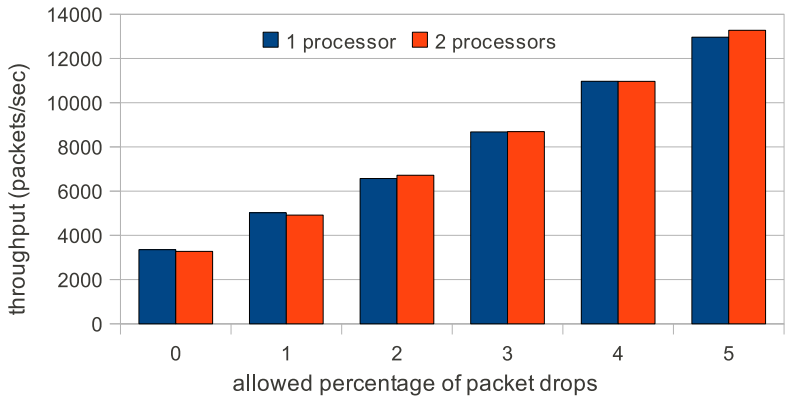


Figure 6.4: Throughput in packets per second for NAT as we increase the tolerance for dropping packets from 0 to 5%, with either 1 or 2 CPUs.

provides little additional benefit over a single CPU in Figure 6.2. These results motivate future work to reduce the impact of synchronization.

Our results so far have focused on measuring throughput when zero packet drops are tolerated (over a five second measurement). However, we would expect performance to improve significantly for measurements when packet drops are tolerated. In Figure 6.4, we plot throughput for NAT as we increase the tolerance for dropping packets from 0 to 5%, and find that this results in dramatic performance improvements—confirming our hypothesis that task-size variance is undermining performance.

In summary, because of our run-to-completion programming model, multiple threads/processors collaborate to perform the same application and synchronization has been inserted in our benchmarks to keep shared data coherent. With multiple packets serviced at the same time and multiple packet

flows tracked inside a processor, the shared data accessed by all threads is not necessarily the same, and can sometimes be exclusively read by some threads. In those cases, critical sections may be overly conservative by preventively reducing the number of threads allowed in a critical section. In our baseline measurements, we observe a large fraction of cycles spent waiting on locks and on packets to arrive in Figure 6.3, i.e. a low processor utilization, along with the large throughput sensitivity to tolerating momentary packet contention on the input packet buffer. These factors are indicative that the threads are often blocked on synchronization, given that we measured that our system can have a CPI of 1 without memory misses (Section 4.5) and that it can sustain an almost full link utilization of network traffic (Section 6.3.2). We therefore find that in our benchmarks, and we expect this to be the case for applications with a pool of threads executing the same program, synchronization is a severe bottleneck, and it is the focus of a number of chapters in this thesis.

6.4 Successful uses of NetThreads

NetThreads is available online [81, 82]. NetThreads has been downloaded 392 times at the time of this writing and the authors have supported users from sites in Canada, India, and the United States. In a tutorial in our university, computer science students exposed for the first time to NetThreads were able to successfully run regression tests within 5 minutes by compiling a `C` program and uploading it along with the NetThreads bitfile (no CAD tools required, and other executables are supplied).

With the participation of academic partners, in particular Professor Ganjali's group and Professor Jacobsen's group and at the University of Toronto, we have developed proof-of-concept applications that use our FPGA-based multiprocessor system.

1. **A publish/subscribe application developed in collaboration with Professor Arno Jacobsen's group in Computer and Electrical Engineering at the University of Toronto** [126]. Publish/subscribe applications are part of an emergent technology that enables among others the ubiquitous news feeds on the Internet, the timely dissemination of strategic information for rescue missions and the broadcasting of financial data to trading agents. Network intrusion detection (computer security) applications can also be expressed as publish/subscribe use cases.

2. **A packet generator application on FPGA developed in collaboration with Professor Yashar**

Ganjali's group in Computer Science at the University of Toronto [44, 127]. This hardware accelerated application enables precise traffic flow modeling that is vital to creating accurate network simulations and for network equipment buyers to make cost-effective decisions. The bandwidth and precision required for this system exceed what a conventional computer can deliver.

Chapter 7

Fast Critical Sections via Thread Scheduling

Chapters 4 and 5 have demonstrated that support for multithreading in soft processors can tolerate pipeline and I/O latencies as well as improve overall system throughput—however this earlier work assumes an abundance of completely independent threads to execute. In this chapter, we show that for real workloads, in particular packet processing applications, there is a large fraction of processor cycles wasted while awaiting the synchronization of shared data structures, limiting the benefits of a multithreaded design. We address this challenge by proposing a method of scheduling threads in hardware that allows the multithreaded pipeline to be more fully utilized without significant costs in area or frequency. We evaluate our technique relative to conventional multithreading using both simulation and a real implementation on a *NetFPGA* board, evaluating three deep-packet inspection applications that are threaded, synchronize, and share data structures, and show that overall packet throughput can be increased by 63%, 31%, and 41% for our three applications compared to the baseline system presented in the previous chapter.

7.1 Multithreading and Synchronization

Prior work [35, 43, 106] and Chapters 4 and 5 have demonstrated that supporting *multithreading* can be very effective for soft processors. In particular, by adding hardware support for multiple thread contexts (i.e., by having multiple program counters and logical register files) and issuing an instruction from

a different thread every cycle in a round-robin manner, a soft processor can avoid stalls and pipeline bubbles without the need for hazard detection logic [43, 76]: a pipeline with N stages that supports $N-1$ threads can be fully utilized without hazard detection logic [76]. Such designs are particularly well-suited to FPGA-based processors because (i) hazard detection logic can often be on the critical path and can require significant area [83], and (ii) using the block RAMs provided in an FPGA to implement multiple logical register files is comparatively fast and area-efficient.

A multithreaded soft processor with an abundance of independent threads to execute is also compelling because it can tolerate memory and I/O latency [84], as well as the compute latency of custom hardware accelerators [106]. In prior work and Chapters 4 and 5, it is generally assumed that such an abundance of completely independent threads is available—modeled as a collection of independent benchmark kernels [35, 43, 76, 84, 106]. However, in real systems, threads will likely share memory and communicate, requiring (i) synchronization between threads, resulting in synchronization latency (while waiting to acquire a lock) and (ii) *critical sections* (while holding a lock).

While a multithreaded processor provides an excellent opportunity to tolerate the resulting synchronization latency, the simple round-robin thread-issue schemes used previously fall short for two reasons: (i) issuing instructions from a thread that is blocked on synchronization (e.g., spin-loop instructions or a synchronization instruction that repeatedly fails) wastes pipeline resources; and (ii) a thread that currently owns a lock and is hence in a critical section only issues once every $N-1$ cycles (assuming support for $N-1$ thread contexts), exacerbating the synchronization bottleneck for the whole system.

The closest work on thread scheduling for soft processors that we are aware of is by Moussali *et. al.* [106] who use a table of pre-determined worst-case instruction latencies to avoid pipeline stalls. Our technique can handle the same cases while additionally prioritizing critical threads and handling unpredictable latencies. In the ASIC world, thread scheduling is an essential part of multithreading with synchronized threads [139]. The IXP [57] family of network processors use non-preemptive thread scheduling where threads exclusively occupy the pipeline until they voluntarily de-schedule themselves when awaiting an event. Other examples of in-order multithreaded processors include the Niagara [73] and the MIPS 34K [72] processors where instructions from each thread wait to be issued in a dedicated pipeline stage. While thread scheduling and hazard detection are well studied in general (operating

systems provide thread management primitives [109] and EPIC architectures, such as IA-64 [58], bundle independent instructions to maximize instruction-level parallelism), our goal is to achieve thread scheduling efficiently in the presence of synchronization at the fine grain required to tolerate pipeline hazards.

7.2 Implementing Thread Scheduling

A multithreaded processor has the advantage of being able to fully utilize the processor pipeline by issuing instructions from different threads in a simple round-robin manner to avoid stalls and hazards. However, for real workloads with shared data and synchronization, one or more threads may often spin awaiting a lock, and issuing instructions from such threads is hence a waste of pipeline resources. Also, a thread which holds a lock (i.e., is in a critical section) can potentially be the most important, since other threads are likely waiting for that lock; ideally we would allocate a greater share of pipeline resources to such threads. Hence in this section we consider methods for *scheduling* threads that are more sophisticated than round-robin but do not significantly increase the complexity nor area of our soft multithreaded processor.

The most sophisticated possibility would be to give priority to any thread that holds a critical lock, and otherwise to schedule a thread having an instruction that has no hazards with current instructions in the pipeline. However, this method is more complex than it sounds due to the possibility of nested critical sections: since a thread may hold multiple locks simultaneously, and more than one thread may hold different locks, scheduling such threads with priorities is very difficult and could even result in deadlock. A correct implementation of this aggressive scheduling would likely also be slow and expensive in terms of hardware resources.

Instead of giving important threads priority, in our approach we propose to only *de-schedule* any thread that is awaiting a lock. In particular, any such thread will no longer have instructions issued until any lock is released in the system—at which point the thread may spin once attempting to acquire the lock and if unsuccessful it is blocked again.[1] Otherwise, for simplicity we would like to issue instructions from the unblocked threads in round-robin order.

[1] A more sophisticated approach that we leave for future work would only unblock threads waiting on the particular lock that was released.

To implement this method of scheduling we must first overcome two challenges. The first is relatively minor: to eliminate the need to track long latency instructions, our processors *replay* instructions that miss in the cache rather than stalling (Chapter 5 [84]). With non-round-robin thread scheduling, it is possible to have multiple instructions from the same thread in the pipeline at once—hence to replay an instruction, all of the instructions for that thread following the replayed instruction must be squashed to preserve the program order of instructions execution.

The second challenge is greater: to support any thread schedule other than round-robin means that there is a possibility that two instructions from the same thread might issue with an unsafe distance between them in the pipeline, potentially violating a data or control hazard. We consider several methods for re-introducing hazard detection. First, we could simply add hazard detection logic to the pipeline—but this would increase area and reduce clock frequency, and would also lead to stalls and bubbles in the pipeline. Second, we could consult hazard detection logic to find a hazard-free instruction to issue from any ready thread—but this more complex approach requires the addition of an extra pipeline stage, and we demonstrate that it does not perform well. A third solution, which we advocate in this chapter, is to perform *static hazard detection* by identifying hazards at compile time and encoding hazard information into the instructions themselves. This approach capitalizes on unused bits in block RAM words on FPGAs[2] to store these *hazard bits*, allowing us to fully benefit from more sophisticated thread scheduling.

Static Hazard Detection

With the ability to issue from any thread not waiting for a lock, the thread scheduler must ensure that dependences between the instructions from the same thread are enforced, either from the branch target calculation to the fetch stage, or from the register writeback to the register read. The easiest way to avoid such hazards is to support *forwarding lines*. By supporting forwarding paths between the writeback and register-read stages of our pipeline, we can limit the maximum hazard distance between instructions from the same thread to two stages.

Our scheduling technique consists of determining hazards at compile-time and inserting *hazard distances* as part of the instruction stream. Because our instructions are fetched from off-chip DDR2

[2]Extra bits in block RAMs are available across FPGA vendors: block RAMs of almost all granularities are configured in widths that are multiples of nine bits, while processors normally have busses multiples of eight bits wide.

Program Disassembly	Hazard distance	Min. issue cycle
addi r1,r1,r4	0	0
addi r2,r2,r5	1	1
or r1,r1,r8	0	3
or r2,r2,r9	0	4

Figure 7.1: Example insertion of hazard distance values. The 3 register operands for the instructions are, from left to right, a destination and two source registers. Arrows indicate register dependences, implying that the corresponding instructions must be issued with at least two pipeline stages between them. A hazard distance of one encoded into the second instruction commands the processor to ensure that the third instruction does not issue until cycle 3, and hence the fourth instruction cannot issue until cycle 4.

memory into our instruction cache, it is impractical to have instructions wider than 32 bits. We therefore compress instructions to accommodate the hazard distance bits in the program executable, and decompress them as they are loaded into the instruction cache. We capitalize on the unused capacity of block RAMs, which have a width multiple of 9 bits—to support 32-bit instructions requires four 9-bit block RAMs, hence there are 4 spare bits for this purpose.

To represent instructions in off-chip memory in fewer than 32 bits, we compress them according to the three MIPS instruction types [19]: for the `R`-type, we merge the function bit field into the opcode field and discard the original function bit field; for the `J`-type instructions, we truncate the target bit field to use fewer than 26 bits; and for the `I`-type, we replace the immediate values by their index in a lookup table that we insert into our FPGA design. To size this lookup table, we found that there are usually more than 1024 unique 16-bit immediates to track, but that 2048 entries is sufficient to accommodate the union of the immediates of all our benchmarks. Therefore, the instruction decompression in the processor incurs a cost of some logic and 2 additional block RAMs[3], but not on the critical path of the processor pipeline. After compression, we can easily reclaim 4 bits per instruction: 2 bits are used to encode the maximum hazard distance, and 2 bits are used to identify lock request and release operations. Our compiler automatically sets these bits accordingly when it recognizes memory-mapped accesses for the locks.

[3]For ease of use, the immediate lookup table could be initialized as part of the loaded program executable, instead of currently, the FPGA bit file.

An example of code with hazard distances is shown in Figure 7.1: the compiler must account for the distances inserted between the previous instructions to avoid inserting superfluous hazard distances. It first evaluates hazards with no regard to control flow: if a branch is not-taken, the hazards will be enforced by the hazard distance bits; if the branch is taken, the pipeline will be flushed from instructions on the mispredicted path and hazards will be automatically avoided. If we extended our processors to be able to predict taken branches, we would have to recode the hazard distance bits to account for both execution paths. As a performance optimization, we insert a hazard distance of 2 for unconditional jumps to prevent the processor from fetching on the wrong path, as it takes 2 cycles to compute the branch target (indicated by the arrows from the `E` to `F` pipeline stages in Figure 7.2). In our measurements, we found it best not to insert any hazard distance on conditional branches; an improvement would consist of using profile feedback to selectively insert hazard distances on mostly taken branches. With more timing margin in our FPGA design, we could explore other possible refinements in the thread scheduler to make it aware of load misses and potentially instantiate lock priorities.

At runtime upon instruction fetch, the hazard distances are loaded into counters that inform the scheduler about hazards between instructions in unblocked threads as illustrated in Figure 7.2. When no hazard-free instruction is available for issue (Figure 7.2c), the scheduler inserts a pipeline bubble. In our processors with thread scheduling, when two memory instructions follow each other and the first one misses, we found that the memory miss signal does not have enough timing slack to disable the second memory access while meeting our frequency requirements. Our solution is to take advantage of our hazard distance bits to make sure that consecutive memory instructions from the same thread are spaced apart by at least one instruction.

Note that in off-the-shelf soft processors, the generic hazard detection circuitry identifies hazards at runtime (potentially with a negative impact on the processor frequency) and inserts bubbles in the pipeline as necessary. To avoid such bubbles in multithreaded processors, ASIC implementations [72, 73] normally add an additional pipeline stage for the thread scheduler to select hazard-free instructions. We evaluate the performance of this approach along with ours in the next section.

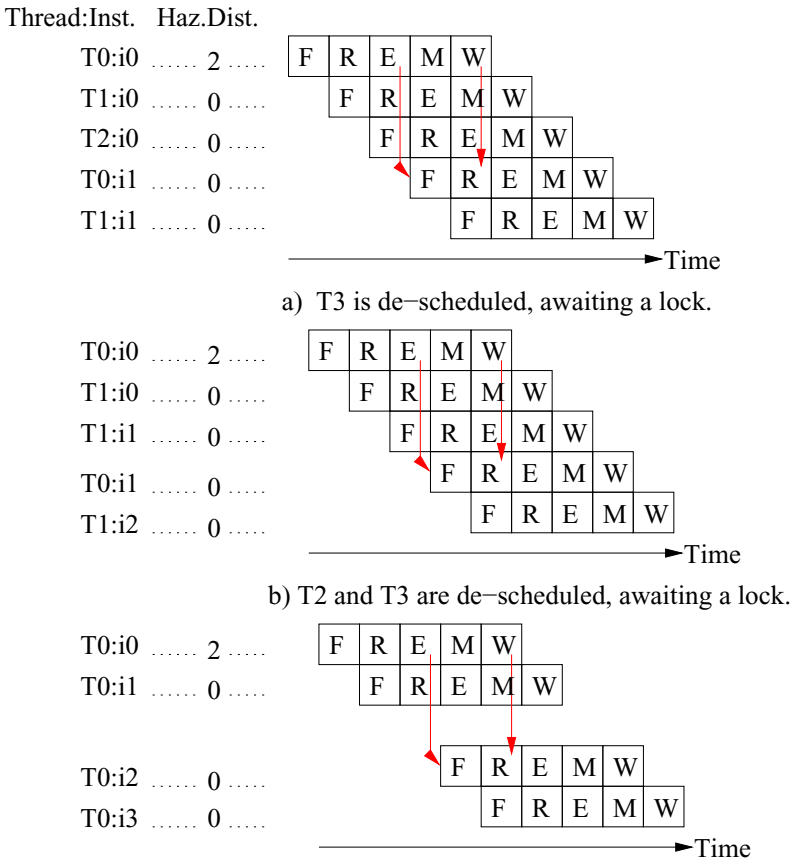


Figure 7.2: Examples using hazard distance to schedule threads. The pipeline stages are: `F` for fetch, `R` for register, `E` for execute, `M` for memory, and `W` for write-back. The arrows indicate potential branch target (`E` to `F`) or register dependences (`W` to `R`).

7.3 Experimental results

In this chapter, our application characteristics are unchanged from Table 6.2 and Figure 7.3(a) shows the maximum packet throughput of our simulated system (Section 6.3.3), normalized to that of a single round-robin CPU; these results estimate speedups for our scheduling on a single CPU (`S1`) of 61%, 57% and 47% for `UDHCP`, `Classifier` and `NAT` respectively.[4] We also used the simulator to estimate the performance of an extra pipeline stage for scheduling (`E1` and `E2`, as described in Section 7.2), but find that our technique dominates in every case: the cost of extra squashed instructions for memory misses and mispredicted branches for the longer pipeline overwhelms any scheduling benefit—hence we do not pursue that design in hardware.

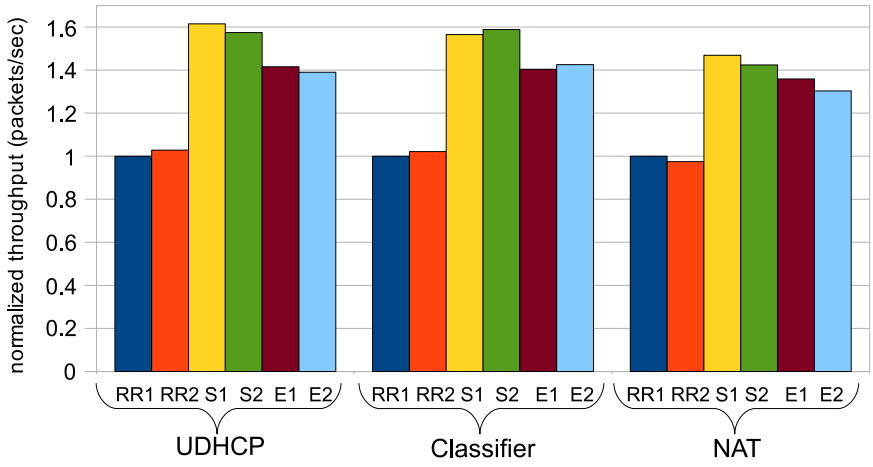
Figure 7.3(b) shows the maximum packet throughput of our (real) hardware system, normalized to that of a single round-robin CPU. We see that with a single CPU our scheduling technique (`S1`) significantly out-performs round-robin scheduling (`RR1`) by 63%, 31%, and 41% across the three applications. However, we also find that our applications do not benefit significantly from the addition of a second CPU due to increased lock and bus contention, and reduced cache locality: for `Classifier` two round-robin CPUs (`RR2`) is 16% better, but otherwise the second CPU either very slightly improves or degrades performance, regardless of the scheduling used. We also observe that our simulator (Figure 7.3(a)) indeed captures the correct relative behaviour of the applications and our system.

Comparing two-CPU full system hardware designs, the round-robin implementation consumes 163 block RAMs (out of 232, i.e., 70% of the total capacity) compared to 165 blocks (71%) with scheduling: two additional blocks are used to hold the lookup table for instruction decoding (as explained in Section 7.2). The designs occupy respectively 15,891 and 15,963 slices (both 67% of the total capacity) when optimized with high-effort for speed. Considering only a single CPU, the post-place & route timing results give an upper bound frequency of 136MHz for the round-robin CPU and 129MHz for scheduling. Hence the overall overhead costs of our proposed scheduling technique are low, with a measured area increase of 0.5% and an estimated frequency decrease of 5%.

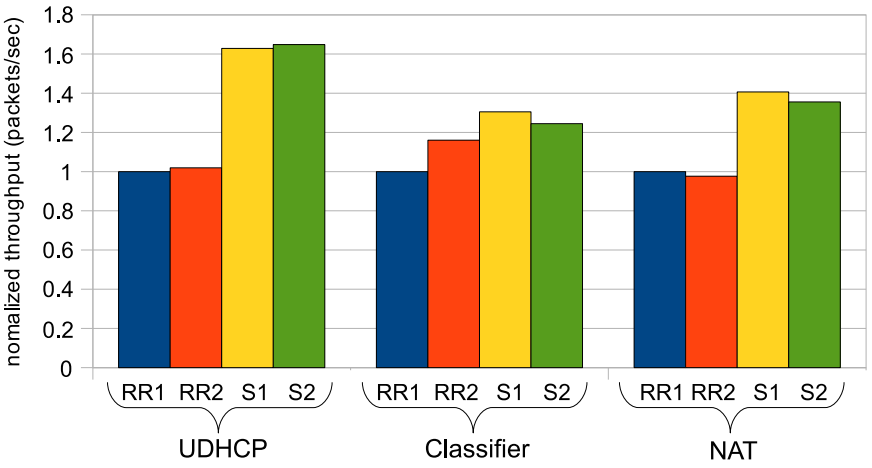
Identifying the Bottlenecks

Much like in Section 6.3.3, to obtain a deeper understanding of the bottlenecks of our system,

[4] We will report benchmark statistics in this order from this point on.



(a) Simulation results.



(b) Hardware results.

Figure 7.3: Throughput (in packets per second) normalized to that of a single round-robin CPU. Each design has either round-robin scheduling (RR), our proposed scheduling (S), or scheduling via an extra pipeline stage (E), and has either 1 or 2 CPUs.

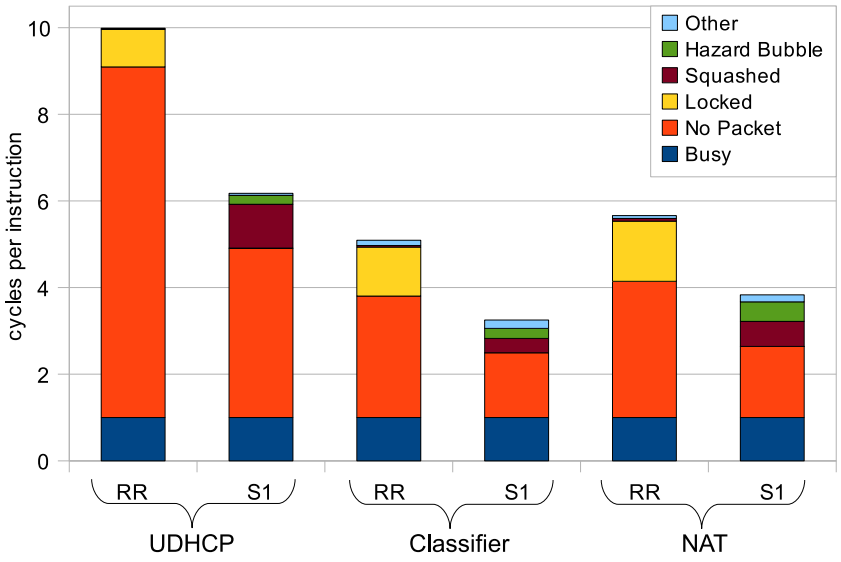


Figure 7.4: Average cycles breakdown for each instruction at the respective maximum packet rates from Figure 7.3(a).

we use our simulator to obtain a breakdown of how cycles are spent for each instruction, as shown in Figure 7.4. In the breakdown, a given cycle can be spent executing an instruction (`busy`), awaiting a new packet to process (`no packet`), awaiting a lock owned by another thread (`locked`), squashed due to a mispredicted branch or a preceding instruction having a memory miss (`squashed`), awaiting a pipeline hazard (`hazard bubble`), or aborted for another reason (`other`, memory misses or bus contention). Figure 7.4 shows that our thread scheduling is effective at tolerating almost all cycles spent spinning for locks. The fraction of time spent waiting for packets (`no packet`) is reduced by 52%, 47%, and 48%, a result of reducing the worst-case processing latency of packets: our simulator reports that the task latency standard deviation decreases by 34%, 33%, and 32%. The fraction of cycles spent on squashed instructions (`squashed`) becomes significant with our proposed scheduling: recall that if one instruction must replay that we must also squash and replay any instruction from that thread that has already issued. The fraction of cycles spent on bubbles (`hazard bubble`) also becomes significant: this indicates that the CPU is frequently executing instructions from only one thread, with the other threads blocked awaiting locks.

While our results in this chapter have focused on measuring throughput when zero packet drops are tolerated (over a five second measurement), we are interested in re-doing the experiment from Figure 6.4. Again, we would expect performance to improve significantly for measurements when packet drops are

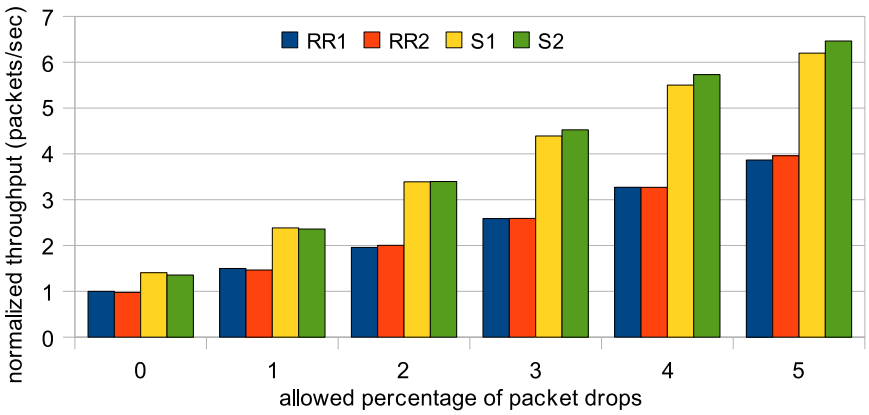


Figure 7.5: Throughput in packets per second for `NAT` as we increase the tolerance for dropping packets from 0 to 5%. Each design has either round-robin scheduling (`RR`) or our proposed scheduling (`S`) and has either `1` or `2` CPUs.

tolerated. In Figure 7.5, we plot throughput for `NAT` as we increase the tolerance for dropping packets from 0 to 5%, and find that this results in dramatic performance improvements for both fixed round-robin (previously shown in Figure 6.4) and our more flexible thread scheduling—again confirming our hypothesis that task-size variance is still undermining performance.

7.4 Summary

In this chapter, we show that previously studied multithreaded soft processors with fixed round-robin thread interleaving can spend a significant amount of cycles spinning for locks when all the threads contribute to the same application and have synchronization around data dependences. We thus demonstrate that thread scheduling is crucial for multithreaded soft processors executing synchronized workloads. We present a technique to implement a more advanced thread scheduling that has minimal area and frequency overhead, because it capitalizes on features of the FPGA fabric. Our scheme builds on static hazard detection and performs better than the scheme used in ASIC processors with hazard detection logic because it avoids the need for an additional pipeline stage. Our improved handling of critical sections with thread scheduling improves the instruction throughput which results in reduced processing latency average and variability. Using a real FPGA-based network interface, we measured packet throughput improvements of 63%, 31% and 41% for our three applications.

For the remainder if this thesis, to eliminate the critical path for hazard detection logic, we employ

the static hazard detection scheme presented in this chapter [77] both in our architecture and compiler. While we were able to reclaim an important fraction of processor cycles, Figure 7.4 shows that our processors are still very under-utilized, motivating more aggressive synchronization techniques to increase the concurrency across the threads.

Chapter 8

NetTM: Improving NetThreads with Hardware Transactional Memory

As reconfigurable computing hardware and in particular FPGA-based systems-on-chip comprise an increasing number of processor and accelerator cores, supporting sharing and synchronization in a way that is scalable and easy to program becomes a challenge. As we first discuss in this chapter, *Transactional memory* (TM) is a potential solution to this problem, and an FPGA-based system provides the opportunity to support TM in hardware (HTM). Although there are many proposed approaches to support HTM for ASIC multicores, these do not necessarily map well to FPGA-based soft multicores.

We propose NetTM: support for HTM in an FPGA-based soft multithreaded multicore that matches the strengths of FPGAs—in particular by careful selection of TM features such as version and contention management, and with conflict detection via support for application-specific signatures. We evaluate our system using the NetFPGA [91] platform and four network packet processing applications that are threaded and shared-memory. Relative to NetThreads [81], an existing two-processor four-way-multithreaded system with conventional lock-based synchronization, we find that adding HTM support (i) maintains a reasonable operating frequency of 125MHz with an area overhead of 20%, (ii) can "transactionally" execute lock-based critical sections with no software modification, and (iii) achieves 6%, 54% and 57% increases in packet throughput for three of four packet processing applications studied, due to reduced false synchronization.

8.1 The Potential for Improving Synchronization with Hardware Transactional Memory

In this section, we motivate both quantitatively and qualitatively the need for having parallel programs that can be synchronized efficiently without sacrificing performance, i.e. in which TM could play a critical role.

8.1.1 Motivating Programmer-Friendly Parallelism

Conventional lock-based synchronization is both difficult to use and also often results in frequent conservative serialization of critical sections, as demonstrated by underutilized processors in Section 7.3. While systems based on shared memory can ease the orchestration of sharing and communication between cores, they require the careful use of synchronization (i.e., lock and unlock operations). Consequently, threads executing in parallel wanting to enter the same *critical section* (i.e., a portion of code that accesses shared data delimited by synchronization) will be serialized, thus losing the parallel advantage of such a system. Hence designers face two important challenges: (i) multiple processors need to share memory, communicate, and synchronize without serializing the execution, and (ii) writing parallel programs with manually inserted lock-based synchronization is error-prone and difficult to debug.

Alternatively, we propose that *Transactional memory* (TM) is a good match to software packet processing: it both (i) can allow the system to optimistically exploit parallelism between the processing of packets and reduce false contention on critical sections whenever it is safe to do so, and (ii) offers an easier programming model for synchronization. A TM system optimistically allows multiple threads inside a critical section—hence TM can improve performance when the parallel critical sections access independent data locations. With transactional execution, a programmer is free to employ coarser critical sections, spend less effort minimizing them, and not worry about deadlocks since a properly implemented TM system does not suffer from them. To guarantee correctness, the underlying system dynamically monitors the memory access locations of each transaction (the *read set* and *write set*) and detects *conflicts* between them. While TM can be implemented purely in software (STM), a hardware implementation (HTM) offers significantly lower performance overhead. The key question is: how

amenable to optimistic transactional execution is packet processing on a multicore—i.e., on a platform with interconnected processor or accelerator cores that synchronize and share memory?

In NetThreads, each processor has support for thread scheduling (see Chapter 7): when a thread cannot immediately acquire a lock, its slot in the round-robin order can be used by other threads until an unlock operation occurs—this leads to better pipeline utilization by minimizing the execution of instructions that implement spin-waiting. To implement lock-based synchronization, NetThreads provides a synchronization unit containing 16 hardware mutexes; in our ISA, each lock/unlock operation specifies a unique identifier, indicating one of these 16 mutexes. In Section 8.7.5, through simulation, we find that supporting an unlimited number of mutexes would improve the performance of our applications by less than 2%, except for Classifier which would improve by 12%. In the rest of this section, we explain that synchronization itself can in fact have a much larger impact on throughput for our benchmarks and we explain how is this also a challenge for many emerging other packet processing applications.

8.1.2 The Potential for Optimistic Parallelism

Although it depends on the application, in many cases, only the processing of packets belonging to the same *flow* (i.e., a stream of packets with common application-specific attributes, such as addresses, protocols, or ports) results in accesses to the same shared state. In other words, there is often parallelism available in the processing of packets belonging to independent flows. Melvin et al. [99] show that for two NLANR [116] packet traces the probability of having at least two packets from the same flow in a window of 100 consecutive preceding packets is approximately 20% and 40%. Verdú et al. [142] further show that the distance between packets from the same flow increases with the amount of traffic aggregation on a link, and therefore generally with the link bandwidth in the case of wide area networks.

To demonstrate the potential for optimistic parallelism in our benchmark applications, we profiled them using our full-system simulator (Section 6.3.3). In particular, what we are interested in is how often the processing of packets has a conflict—i.e. for two threads each processing a packet, their write sets intersect or a write set intersects a read set. In Figure 8.1, we show the average fraction of packets for which their processing conflicts for varying windows of 2 to 16 packets: while the number of dependent packets increases with the window size, the number increases very slowly because of

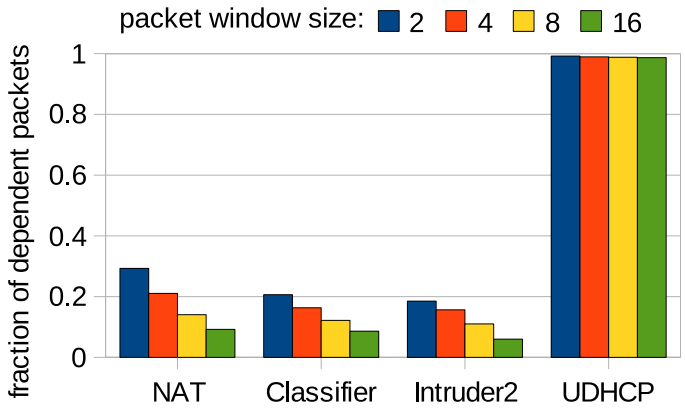


Figure 8.1: Average fraction of conflicting packet executions for windows of 2 to 16 consecutive packets.

the mixing of many packet flows in the packet stream so the fraction (or ratio) of dependent packets does diminish with the increasing window size. For three of our applications, `NAT`, `Classifier`, and `Intruder2`, the fraction of conflicting packet-executions varies from around 20% to less than 10% as the window considered increases from 2 to 16 packets, indicating two important things: first, that conventional synchronization for critical sections in these applications would be overly-conservative 80% to 90% of the time, and second that there is hence a strong potential for optimistic synchronization for these applications. For `UDHCP`, our profile indicates that nearly all packet-executions conflict. In reality, `UDHCP` contains several critical sections, some that do nearly always conflict, but many others that do not conflict—hence the potential for optimistic parallelism exists even for `UDHCP`. Now with the knowledge that synchronization is generally conservative for our packet processing applications, we are interested in knowing if synchronization would also be a problem when parallelizing other, more advanced applications that are representative of future use cases.

8.1.3 The Need for Simpler Synchronization

The Internet has witnessed a transformation from static web pages to social networking and peer-to-peer data transfers. This transformation of user behavior patterns requires network connectivity providers to constantly and proactively adapt their services to adequately provision and secure their infrastructure. As networking requirements evolve constantly, many network equipment vendors opt for network processors to implement functions that are likely to change over time. Because many network protocols are now considered outdated, there is even a desire to have vendors open the hardware to accept

user/researcher code [41]. However, once faced with board-specific reference code, programmers are often hesitant to edit it, in part due to the challenge of modifying the synchronization and parallelization mechanisms in those carefully tuned multicore programs.

With multicore processors becoming commodity, general-purpose processors are closing the performance gap with network processors for network-edge applications [59] fueling an increased industry use of open-source software that than can turn an off-the-shelf computer into a network device; examples include the Click Modular router [105], Snort [124], Quagga [61], and the XORP project [52]. Those applications are often coded as a single thread of computation with many global data structures that are unsynchronized—hence porting them to multicore is a substantial challenge when performance depends on constructing finer-grained synchronized sections. There is therefore a need for simpler synchronization mechanisms to support control-flow intensive programs.

In this section, we show that the processing of different packets is not always independent, and the application must maintain some shared state. We also show that protecting this shared state with traditional synchronization leads to conservative serializations of parallel threads in most instances and, beyond this quantitative argument, there is a demand for easier parallelism to scale the performance of packet processing applications to multicores. Because transactional memory can improve both performance and ease of programming, we are motivated to find ways to enable it in our soft processor-based platform. First, we review previous implementations of TM on FPGAs.

8.2 Prior Work in FPGA Hardware Transactional Memory

There is a large body of work studying the design space for HTM in ASIC multicores [18, 94]. Earlier proposed transactional memory implementations [51] buffer transactional writes and make them visible only at the end of a transaction. They augment their caches with speculatively read and written bits and rely on iterating in the cache for commits and instantaneous clear of speculative cache tag bits. Later extensions [12, 120] allow caches to overflow speculative data to a specified location in main memory. In these proposals, long transactions were required to amortize commit overheads, but were in turn more likely to conflict. For the Rock processor [34] (a commercial ASIC implementation of HTM), it was found that most of these proposals were too complex to implement given the other

types of speculation already present in the processor and a limited version of TM was implemented that doesn't support function calls in transactions. LOGTM-SE [158], the most similar approach to ours, decouples conflict detection from the caches to make commits faster. In contrast with LOGTM-SE, which handles transaction conflicts in software and is evaluated with only small transactions, we alleviate the programmer from having to modify lock-based code to support transactions, and we consider much longer transactions.

Most previous FPGA implementations of HTM were used to prototype an ASIC design [48, 134, 147]—as opposed to targeting the strengths of an FPGA as a final product. To provide a low-overhead implementation, our work also distinguishes itself in the type of TM that we implement and in the way that we perform conflict detection. To track transactional speculative state, prior FPGA implementations [48, 63, 147] use (i) extra bits per line in a private cache per thread or in a shared cache, and (ii) *lazy version management* (i.e., regular memory is modified only upon commit), and (iii) *lazy conflict detection* (i.e., validation is only performed at commit time). These approaches are not a good match for product-oriented FPGA-based systems because of the significant cache storage overhead required, as we explain later in Section 8.4.

The most closely related work to ours is the CTM system [63] which employs a shared-cache and per-transaction buffers of addresses to track accesses. However, the evaluation of CTM is limited to at-most five modified memory locations per transaction—in contrast, NetTM supports more than a thousand per transaction. Rather than using off-the-shelf soft cores as in CTM and other work [48, 134, 147] and thus requiring the programmer to explicitly mark each transactional access in the code, in NetTM we integrate TM with each soft processor pipeline and automatically handle loads and stores within transactions appropriately. This also allows NetTM to transactionally execute lock-based critical sections, assuming that the programmer has followed simple rules (described later in Section 8.3).

8.3 Programming NetTM

Specifying Transactions TM semantics [54] imply that any transaction will appear to have executed atomically with respect to any other transaction. For NetTM, as in most TM systems, a transactional

critical section is specified by denoting the start and end of a transaction, using the same instruction API as the lock-based synchronization for NetThreads—i.e., `lock(ID)` can mean "start transaction" and `unlock(ID)` can mean "end transaction". Hence existing programs need not be modified, since NetTM can use existing synchronization in the code and simply interpret critical sections as transactional. We next describe how the lock identifier, `ID` in the previous example, is interpreted by NetTM.

Locks vs Transactions NetTM supports both lock-based and TM-based synchronization, since a code region's access patterns can favor one approach over the other. For example, lock-based critical sections are necessary for I/O operations since they cannot be undone in the event of an aborted transaction: specifically, for processor initialization, to protect the sequence of memory-mapped commands leading to sending a packet, and to protect the allocation of output packet memory (see later in this section). We use the identifier associated with lock/unlock operations to distinguish whether a given critical section should be executed via a transaction or via traditional lock-based synchronization: this mapping of the identifiers is provided by the designer as a parameter to the hardware synchronization unit (Figure 8.4). When writing a deadlock-free application using locks, a programmer would typically need to examine carefully which identifier is used to enter a critical section protecting accesses to which shared variables. NetTM simplifies the programmer's task when using transactions: NetTM enforces the atomicity of all transactions regardless of the lock identifier value. Therefore only one identifier can be designated to be of the transaction type, and doing so frees the remaining identifiers/mutexes to be used as unique locks. However, to support legacy software, a designer is also free to designate multiple identifiers to be of the transaction type.

Composing Locks and Transactions It is desirable for locks and transactions in our system to be composable, meaning they may be nested within each other. For example, to atomically transfer a record between two linked lists, the programmer might nest existing atomic delete and insert operations within some outer critical section. NetTM supports composition as follows. *Lock within lock* is straightforward and supported. *Transaction within transaction* is supported, and the start/end of the inner transaction are ignored. As opposed to lock-based synchronization, a deadlock is therefore not possible across transactions. NetTM uses a per-thread hardware counter to track the nesting level of lock operations to decide when to start/commit a transaction, or acquire/release a lock in the presence of

```
lock(ID_TX);
...
lock(ID_MUTEX);
...
unlock(ID_MUTEX);
...
unlock(ID_TX);
```

(a) Undefined behavior: nesting lock-based synchronization inside transaction.

```
lock(ID_MUTEX);
lock(ID_TX);
...
unlock(ID_MUTEX);
x = val;
...
unlock(ID_TX);
```

(b) Undefined behavior: partial nesting does not provide atomicity.

```
lock(ID_TX);
x = val;
...
unlock(ID_TX);
lock(ID_MUTEX);
y = x;
unlock(ID_MUTEX);
```

(c) Race condition: lock-based critical sections are not atomic with respect to transactions.

```
lock(ID_TX);
x = val1;
x = val2;
unlock(ID_TX);
...
y = x;
```

(d) Race condition: transactions are not atomic with respect to unsynchronized accesses (i.e., to `x`).

Figure 8.2: Example mis-uses of transactions as supported by NetTM. `ID_TX` identifies a critical section as a transaction, and `ID_MUTEX` as a lock. Critical section nesting occurs when a program is inside more than one critical section is at a particular instant.

nesting. *Lock within transaction* is not supported as illustrated in Figure 8.2(a), since code within a lock-based critical section should never be undone, and we do not support making transactions irrevocable [47]. *Transaction within lock* is supported, although the transaction must be fully nested within the lock/unlock, and will not be executed atomically—meaning that the transaction start/end are essentially ignored, under the assumption that the enclosing lock properly protects any shared data. Our full-system simulator can assist the programmer by monitoring the dynamic behavior of a program and identifying the potentially unsafe nesting of transactions inside locked-based critical sections, as exemplified in Figure 8.2(b). NetTM implements *weak atomicity* [96], i.e. it guarantees atomicity between transactions or between lock-based critical sections, but not between transactions and non-transactional code. Figures 8.2(c) and (d) shows examples of non-atomic accesses to the variable `x` that could result in race conditions: while those accesses are supported and are useful in some cases (e.g. to get a snapshot of a value at a particular instant), the programmer should be aware that they are not thread safe, just like with traditional synchronization when using locks with different identifiers in Figure 8.2(c) or a lock in Figure 8.2(d).

Memory Semantics NetTM comprises several on-chip memories (Table 6.1) with varying properties, hence it is important to note the interaction of TM with each. The input buffer is not impacted by

Table 8.1: Dynamic Accesses per Transaction

	All accesses				Unique	
	Loads		Stores		Store Locations	
Benchmark	mean	max	mean	max (filtered)	mean	max
`Classifier`	2433	67873	1313	32398 (573)	38	314
`NAT`	114	739	28	104 (98)	19	55
`Intruder2`	152	7765	40	340 (255)	22	111
`UDHCP`	61	3504	4	301 (36)	4	20

TM since it is read-only, and it's memory-mapped control registers should not be accessed within a transaction. The output buffer can be both read and written, however it only contains packets that are each accessed by only one thread at a time (since the allocation of output buffer packets is protected by a regular lock). Hence for simplicity the output buffer does not support an undo log, and the programmer must take into account that it does not roll-back on a transaction abort (i.e., within a transaction, a program should never read an output buffer location before writing it).

Improving Performance via Feedback TM allows the programmer to quickly achieve a correct threaded application. Performance can then be improved by reducing transaction aborts, using feedback that pin-points specific memory accesses and data structures that caused the conflicts. While this feedback could potentially be gathered directly in hardware, for now we use our cycle-accurate simulator of NetTM to do so. For example, we identified memory management functions (`malloc()` and `free()`) as a frequent source of aborts, and instead employed a light-weight per-thread memory allocator that was free of conflicts.

Benchmark Applications Table 3.1 describes the nature of the parallelism in each application, and Table 8.1 shows statistics on the dynamic accesses per transaction for each application (filtering will be described later in Section 8.6). Note that the transactions comprise significant numbers of loads and stores; the actual numbers can differ from the data in Table 6.2 because of the code transformations applied to the original code, as explained in Section 8.3.

8.4 Version Management

There are many options in the design of a TM system. For NetTM our over-arching goals are to match our design to the strengths and constraints of an FPGA-based system by striving for simplicity, and minimizing area and storage overhead. In this section, we focus on options for *version management*, which have a significant impact on the efficiency and overheads of adding support for HTM to an FPGA-based multicore. Version management refers to the method of segregating transactional modifications from other transactions and regular memory. For a simple HTM, the two main options for version management are *lazy* [51] versus *eager* [158].

Lazy Version Management In a lazy approach, write values are saved in a write-buffer until the transaction commits, when they are copied/flushed to regular memory. Any read must first check the write-buffer for a previous write to the same location, which can add latency to read operations. To support writes and fast reads, a write-buffer is often organized as a cache with special support for conflicts (e.g., partial commit or spill to regular memory). CTM [63] (see Section 8.2) minimizes cache line conflicts by indexing the cache via *Cuckoo* hashing, although this increases read latency. Because lazy schemes buffer modifications from regular memory: (i) they can support multiple transactions writing to the same location (without conflict), (ii) conflict detection for a transaction can be deferred until it commits, (iii) aborts are fast because the write-buffer is simply discarded, and (iv) commits are slow because the write-buffer must be flushed/copied to regular memory.

Eager Version Management In an eager approach, writes modify main memory directly and are not buffered—therefore any conflicts must be detected before a write is performed. To support rollback for aborts, a backup copy of each modified memory location must be saved in an *undo-log*. Hence when a transaction aborts, the undo-log is copied/flushed to regular memory, and when a transaction commits, the undo-log is discarded. A major benefit of an eager approach is that reads proceed unhindered and can directly access main memory, and hence an undo-log is much simpler than a write-buffer since the undo-log need only be read when flushing to regular memory on abort. Because eager schemes modify regular memory directly (without buffering): (i) they cannot support multiple transactions writing to the same location (this results in a conflict), (ii) conflict detection must be performed on every memory access, (iii) aborts are slow because the undo-log must be flushed/copied to regular memory, and (iv)

commits are fast because the undo-log is simply discarded.

Our Decision After serious consideration of both approaches, we concluded that eager version management was the best match to FPGA-based systems such as NetTM for several reasons. First, while similar in required number of storage elements, a write buffer is necessarily significantly more complex than an undo-log since it must support fast reads via indexing and a cache-like organization. Our preliminary efforts found that it was extremely difficult to create a write-buffer with single-cycle read and write access. To avoid replacement from a cache-organized write-buffer (which in turn must result in transaction stall or abort), it must be large or associative or both, and these are both challenging for FPGAs. Second, an eager approach allows spilling transactional modifications from the shared data cache to next level of memory (in this case off-chip), and our benchmarks exhibit large write sets as shown in Table 8.1. Third, via simulation we observed that disallowing multiple writers to the same memory location(s) (a limitation of an eager approach) resulted in only a 1% increase in aborts for our applications in the worst case. Fourth, we found that transactions commit in the common case for our applications,[1] and an eager approach is fastest for commit.

8.5 Conflict Detection

A key consequence of our decision to implement eager version management is that we must be able to detect conflicts with every memory access; hence to avoid undue added sources of stalling in the system, we must be able to do so in a single cycle. This requirement led us to consider implementing conflict detection via *signatures*, which are essentially bit-vectors that track the memory locations accessed by a transaction via hash indexing [23], with each transaction owning two signatures to track its read and write sets. Signatures can represent an unbounded set of addresses, and allow us to decouple conflict detection from version management, and provide an opportunity to capitalize on the bit-level parallelism of FPGAs. In Appendix A [75], we explored the design space of signature implementations for an FPGA-based two-processor system (i.e., for two total threads), and proposed a method for creating application-specific hash functions to reduce signature size without impacting their accuracy. In this

[1] When measured at maximum saturation rate in our TM system with the default contention manager, we measure that 78%, 67%, 92% and 79% of the transactions commit without aborting for `Classifier`, `NAT`, `Intruder`, and `UDHCP` respectively.

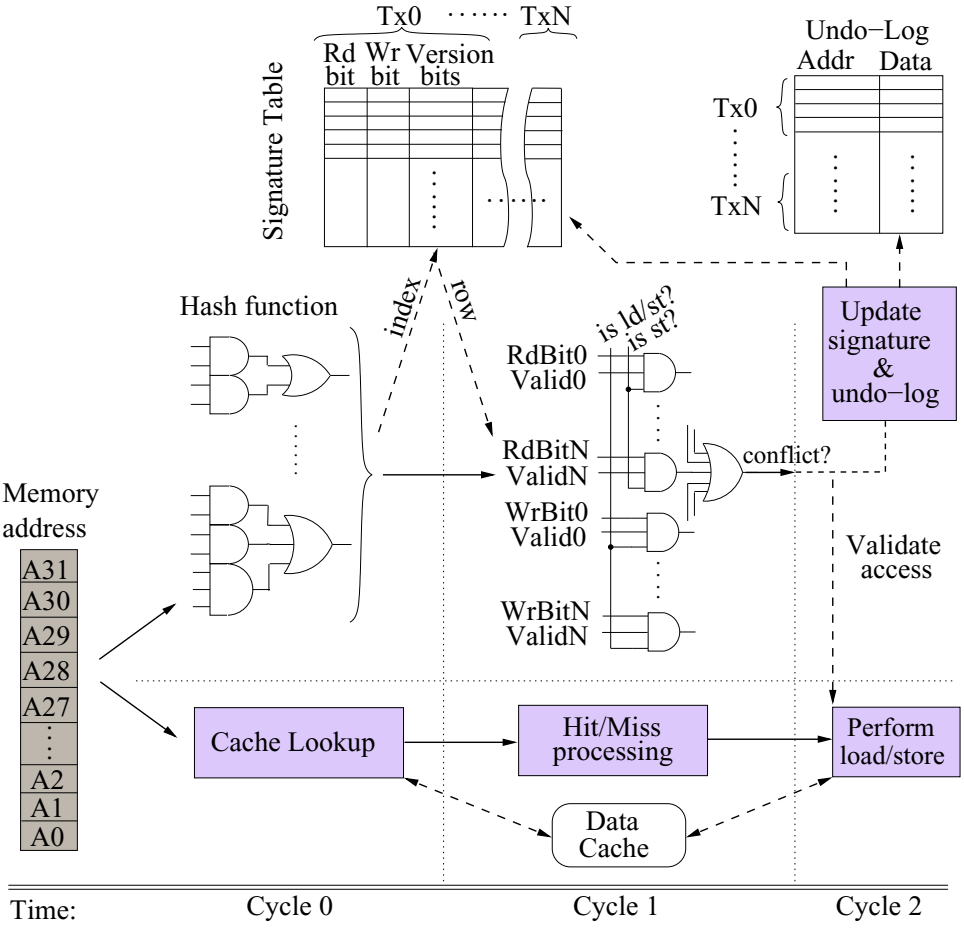


Figure 8.3: Integration of conflict detection hardware with the processor pipeline for a memory access.

section, we briefly summarize our signature framework, and describe how we adapted the previously proposed scheme to support a multithreaded multicore (ie., for eight total threads), as we walk through Figure 8.3.

Application-Specific Hash Functions Signatures normally have many fewer bits than there are memory locations, and comparing two signatures can potentially indicate costly false-positive conflicts between transactions. Hence prior HTMs employ relatively large signatures—thousands of bits long—to avoid such false conflicts. However, it is difficult to implement such large signatures in an FPGA without impacting cycle time or taking multiple cycles to compare signatures. Our key insight was to leverage the customizability of an FPGA-based system to create an application-specific hash function that could minimize false conflicts, by mapping the most contentious memory locations to different signature bits, while minimizing the total number of signature bits. Our approach is based on trie hashing (Appendix A), and we build a trie-based conflict detection unit by (i) profiling the memory addresses accessed by an application, (ii) using this information to build and optimize a *trie* (i.e. a tree based on address prefixes) that allocates more branches to frequently-conflicting address prefixes, and (iii) implementing the trie in a conflict detection unit using simple combinational circuits, as depicted by the "hash function" logic in Figure 8.3. The result of the hash function corresponds to a leaf in the trie, and maps to a particular signature bit. Note that since NetTM implements weak atomicity (see Section 8.3), only transactional memory accesses are recorded in signatures, which means that the signature hash function depends on the memory profile of only the critical sections of our applications, which are the parts of an embedded application which are often the least changing across revisions.

Signature Table Architecture In contrast with prior signature work on FPGAs [75], in NetTM we store signatures in block RAMs. To detect conflicts, we must compare the appropriate signature bits (as selected by the hash index) from every transaction in a single cycle—as shown in Figure 8.3, we decided to have the hash function index the rows of a block RAM (identifying a single signature bit), and to map the corresponding read and write bits for every transaction/thread-context across each row. Therefore with one block RAM access we can read all of the read and write signature bits for a given address for all transactions in parallel. A challenge is that we must clear the signature bits for a given transaction when it commits or aborts, and it would be too costly to visit all of the rows of the block

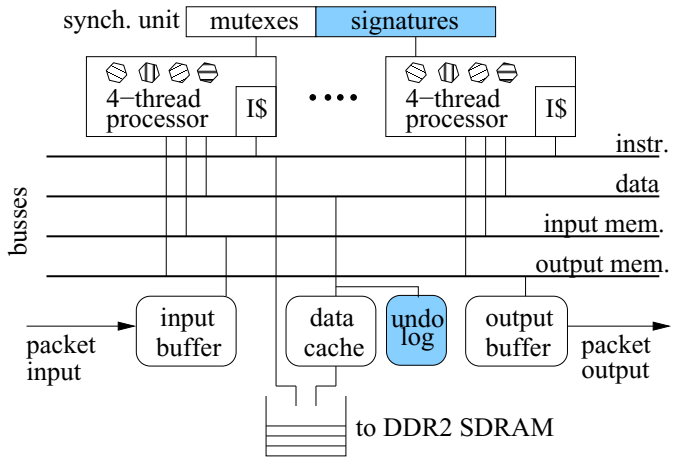


Figure 8.4: The NetThreads architecture, currently with two processors. NetTM supports TM by extending NetThreads, mainly with signatures and an undo-log.

RAM to do so. Instead we add to the signature table a version number per transaction (incremented on commit or rollback), that allows us to compare to a register holding the true version number of the current transaction for that thread context. Comparing version numbers produces a `Valid` signal that is used to ignore the result of comparing signature bits when appropriate. We clear signature bits lazily: for every memory reference a row of the signature table is accessed, and we clear the corresponding signature bits for any transaction with mismatching version numbers. This lazy-clear works well in practice, although it is possible that the version number may completely wrap-around before there is an intervening memory reference to cause a clear, resulting in a false conflict (which hurts performance but not correctness). We are hence motivated to support version numbers that are as large as possible.

8.6 Implementing NetTM

In this section, we describe the key details of the NetTM implementation. As shown earlier in Figure 8.4, the main additions over the NetThreads implementation are the signature table, the undo-log, and support for transactions in the register file and thread scheduler.

Signature Table As shown in Figure 8.3, this data structure of fixed size (one read and one write signature for each hardware thread context) requires concurrent read and write access. For the device we target, the block RAMs are 36bits wide, and we determined experimentally that a signature table composed of at most two block RAMs could be integrated in the TM processor pipeline while preserving

the 125MHz target frequency. We could combine the block RAMs horizontally to allow larger version numbers, or vertically to allow more signature bits; we determined experimentally that the vertical option produced the fewest false conflicts. Hence in NetTM each block RAM row contains a read bit, a write bit, and two version bits (four bits total) for each of eight transactions/thread-contexts, for a total of 32bits (slightly under-utilizing the available 36bit-width). We implement signatures of maximum length ranging from 618 to 904 bits for our applications (limited by the hash function size).

Undo-Log The undo-log is implemented as a single physical structure that is logically partitioned into equal divisions per thread context. On a transaction commit, a per-thread undo-log can be cleared in one cycle by resetting the appropriate write-pointer. On a transaction abort, the undo-log requests exclusive access to the shared data cache, and flushes its contents to the cache in reverse order. This flush is performed atomically with respect to any other memory access, although processors can continue to execute non-memory-reference instructions during an undo-log flush. We buffer data in the undo-log at a word granularity because that matches our conflict detection resolution. In NetTM, the undo-log must be provisioned to be large enough to accommodate the longest transactions. A simple extension to support undo-log overflows could consist of aborting all the other transactions to allow the transaction with a filled undo-log to proceed without conflict.

Undo-Log Filter To minimize the required size of the undo-log as well as its flush latency on aborts, we are motivated to limit the number of superfluous memory locations saved in the undo-log. Rather than resort to a more complex undo-log design capable of indexing and avoiding duplicates, we instead attempt to filter the locations that the undo-log saves. By examining our applications we found that a large source of undo-log pollution was due to writes that need not be backed-up because they belong to the uninitialized portion of stack memory. For example, the recursive regular expression matching code in `Classifier` results in many such writes to the stack. It is straightforward to filter addresses backed-up by the undo-log using the stack pointer of the appropriate thread context; to maintain the clock frequency of our processors, we keep copies of the stack pointer values in a block RAM near the undo-log mechanism. We found that such filtering reduced the required undo-log capacity requirement for `Classifier` from 32k entries to less than 1k entries, as shown in Table 8.1 by comparing the maximum

number of stores per transaction and the filtered number (in parenthesis in the same column).[2]

Pipeline Integration To accommodate the additional signature logic and for NetTM hit/miss processing, a memory access is split into two cycles (as shown in Figure 8.3)—hence NetTM has a 6-stage pipeline, although the results for non-memory instructions are ready in the 5th stage. Because the data cache and the signature table are accessed in two different cycles, NetTM can potentially suffer from additional stalls due to contention on the shared cache lines and signature read-after-write hazards. While NetThreads' non-transactional data cache does not allocate a cache line for write misses, NetTM is forced to perform a load from memory on transactional write misses so that the original value may be saved to the undo-log.

Transactional Registers As with writes to memory, all transactional modifications to the program counter and registers must be undone on a transaction abort. The program counter is easily backed-up in per-thread registers. The register file is actually composed of two copies, where for each register a version bit tracks which register file copy holds the committed version and another bit tracks if the register value was transactionally modified. This way all transactional modifications to the register file can be committed in a single cycle by toggling the version bit without performing any copying.

Thread Scheduling The thread scheduler implemented in NetThreads allows better pipeline utilization: when a thread cannot immediately acquire a lock, its slot in the round-robin order can be used by other threads until an unlock operation occurs [77]. For NetTM we extended the scheduler to similarly de-schedule a thread that is awaiting the contention manager to signal a transaction restart.

8.7 Results on NetFPGA

In this section, we evaluate the benefits of TM for soft multicores by comparing resource utilization and throughput of NetTM (supporting TM and locks) relative to NetThreads (supporting only locks). Because the underlying NetFPGA board is a network card, our application domain is packet processing where threads execute continuously and only have access to a shared DDR2 SDRAM memory as the

[2] A programmer could further reduce the occupation of the undo-log with the `alloca()` function to lower the stack of a transaction (in case the checkpoint event is deeply nested in a function call) and/or defer the allocation of stack variables until after the checkpoint and `alloca()` calls.

last level of storage. We report the maximum sustainable packet rate for a given application as the packet rate with 90% confidence of not dropping any packet over a five-second run—thus our results are conservative given that network appliances are typically allowed to drop a small fraction of packets.[3] We start by describing the TM resource utilization, then show how the baseline TM performance can be improved via tuning and we finally compare our TM implementation with other approaches of exploiting additional parallelism.

8.7.1 Resource Utilization

In total, NetTM consumes 32 more block RAMs than NetThreads, so its block RAM utilization is 71% (161/232) compared to 57% (129/232) for NetThreads. The additional block RAMs are used as follows: 2 for the signature bit vectors, 2 for the log filter (to save last and checkpoint stack pointers) and 26 for the for undo log (1024 words and addresses for 8 threads). NetThreads consumes 18980 LUTs (out of 47232, i.e. 40% of the total LUT capacity) when optimized with high-effort for speed; NetTM design variations range from 3816 to 4097 additional LUTs depending on the application-specific signature size, an overhead of roughly 21% over NetThreads. The additional LUTs are associated with the extra pipeline stage per processor and conflict detection logic.

8.7.2 NetTM Baseline Throughput

In Figure 8.5, the first bar for each application (`CM1`) reports the packet rate for NetTM normalized to that of NetThreads. NetTM improves packet throughput by 49%, 4% and 54% for `Classifier`, `NAT`, and `UDHCP` respectively, by exploiting the optimistic parallelism available in critical sections. The TM speedup is the result of reduced time spent awaiting locks, but moderated by the number of conflicts between transactions and the time to recover from them. `Classifier` has occasional long transactions that do not always conflict, providing an opportunity for reclaiming the corresponding large wait times with locks. Similarly, `UDHCP` has an important fraction of read-only transactions that do not conflict. `NAT` has a less pronounced speedup because of less-contended shorter transactions. Despite having a high average commit rate, for `Intruder` TM results in lower throughput due to bursty periods of large

[3] We use a 10-point FFT smoothing filter to determine the interpolated intercept of our experimental data with the 90% threshold (see Figure 8.6 for an example).

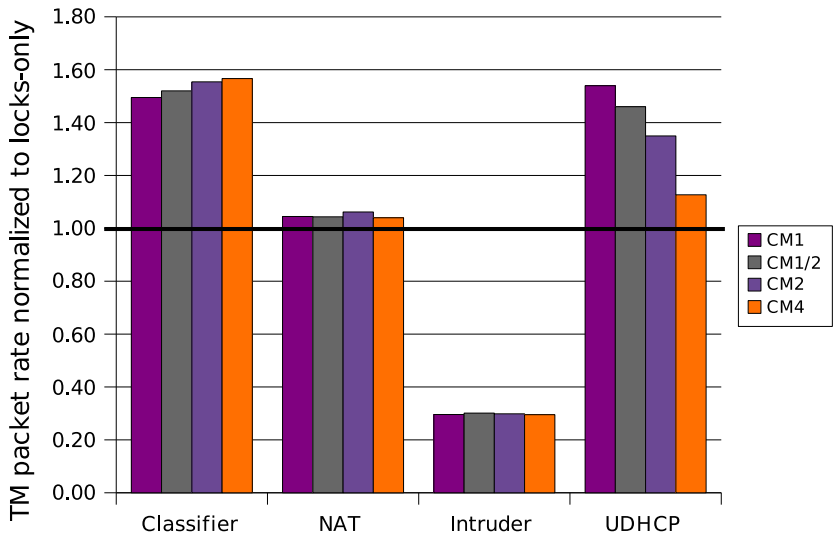


Figure 8.5: Packet throughput of NetTM normalized to NetThreads, with varying contention managers (see Section 8.7.3) that restart: at most one aborted transaction at a time (`CM1`), conditionally one or two (`CM1/2`), at most two (`CM2`), or at most four (`CM4`).

transactions during which there are repeated aborts, leading to limited parallelism and a reduced packet rate. Two indicators could have predicted this behavior: (i) across our benchmarks, `Intruder` has the highest CPU utilization with locks-only, meaning that wait times for synchronization are smaller and true TM conflicts will be harmful when large transactions are aborted; and (ii) `Intruder` has a significant amount of dynamic memory allocation/deallocation leading to more randomized memory accesses, limiting the effectiveness of application-specific signatures and leading to increased false conflicts. Furthermore, the throughput of `Intruder` with locks-only can be improved by 30% by reducing contention through privatizing key data structures: we named this optimized version of the benchmark `Intruder2` (Table 8.1). Despite having a high average commit rate, `Intruder2` has a throughput reduction of 8% (Figure 8.7 in Section 8.7.4): optimistic parallelism is difficult to extract in this case because `Intruder2` has short transactions and an otherwise high CPU utilization such that any transaction abort directly hinders performance. This demonstrates that TM isn't necessarily the best option for every region of code or application. One advantage of NetTM is that the programmer is free to revert to using locks since NetTM integrates support for both transactions and locks.

UDHCP: A Closer Look Figure 8.6 shows for `UDHCP` the probability of not dropping packets as a

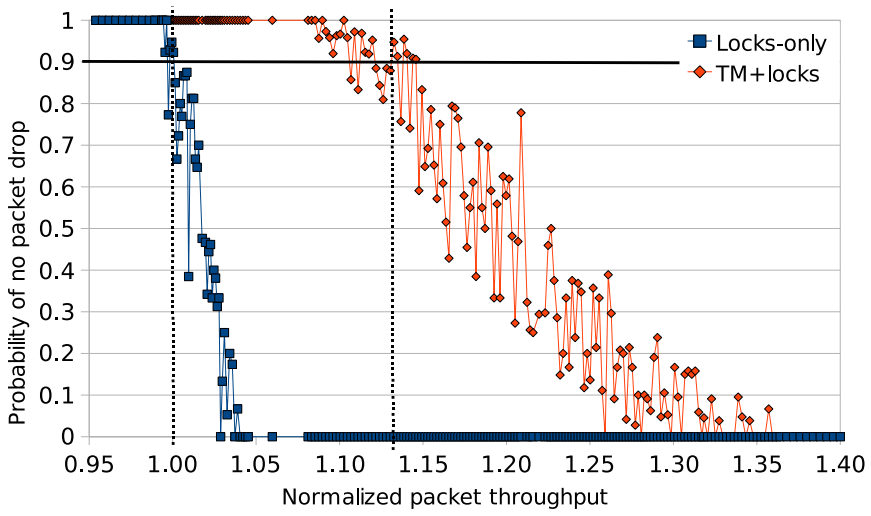


Figure 8.6: For UDHCP, the probability of no packet drops vs packet throughput for both NetThreads and NetTM, normalized to NetThreads. The packet throughput matches the incoming packet rate for a system with no packet drops: as the incoming packet rate is increased the probability of not dropping any packet is reduced. For each implementation, the point of 90% probability is highlighted by a vertical line.

function of the normalized packet throughput (i.e. the inter-arrival packet rate). In addition to providing higher throughput than NetThreads, it is apparent that NetTM also provides a more graceful degradation of packet drops versus throughput, and does so for `Classifier` and `NAT` as well.

8.7.3 Tuning Contention Management

When a conflict between two transactions is detected there are a number of ways to proceed, and the best way is often dependent on the access patterns of the application—and FPGA-based systems such as NetTM provide the opportunity to tune the contention management strategy to match the application. In essence, one transaction must either stall or abort. There are also many options for how long to stall or when to restart after aborting [13]. Stalling approaches require frequent re-validation of the read/write sets of the stalled transaction, and can lead to live-lock (if a stalled transaction causes conflicts with a repeatedly retrying transaction), hence for simplicity we unconditionally abort and restart any transaction that causes a conflict.

The decision of when to restart must be made carefully, and we studied several options as shown in Figure 8.5 with the goals of (i) requiring minimal hardware support, (ii) deciding locally (per-processor,

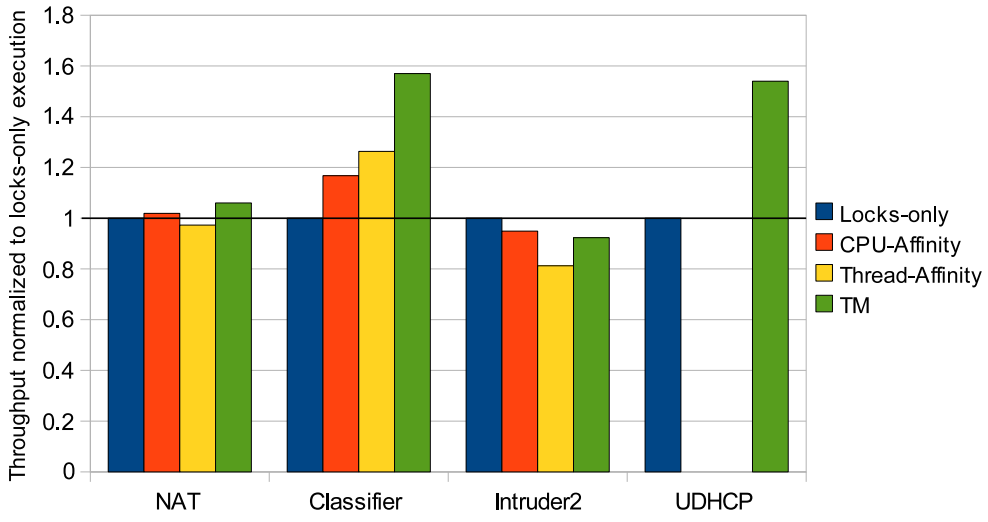


Figure 8.7: Throughput improvement relative to locks-only (NetThreads) for flow-affinity scheduling and TM (NetTM). In its current form, `UDHCP` is unable to exploit flow-affinity scheduling, which explains the missing bars in the graph.

rather than centrally), and (iii) guaranteeing forward progress. On a commit event, `CM1` and `CM2` allow only one or two transactions to restart respectively (other transactions must await a subsequent commit). `CM1/2` adaptively restarts up to two transactions when they are the only aborted transactions in the system, or otherwise restarts only one transaction when there are more than two aborted transactions in the system. `CM4` makes all aborted transactions await some transaction to commit before all restarting simultaneously.

As shown in Figure 8.5, `UDHCP` significantly benefits from the `CM1` contention manager: `UDHCP` has writer transactions that conflict with multiple reader transactions, and `CM1` minimizes repeated aborts in that scenario. In contrast, `Classifier` has more independent transactions and prefers a greater number of transactions restarted concurrently (`CM4`). `NAT` shows a slight preference for `CM2`. In summary, in Figure 8.5, once we have tuned contention management we can achieve speedups of 57% (`CM4`), 6% (`CM2`), and 54% (`CM1`) for `Classifier`, `NAT` and `UDHCP`.

8.7.4 Comparing with Flow-Affinity Scheduling for NetThreads

A way to avoid lock contention that is potentially simpler than TM is attempting to schedule packets from the same flow (i.e., that are likely to contend) to the same hardware context (thread context or CPU). Such an affinity-based scheduling strategy could potentially lower or eliminate the possibility of

lock contention which forces critical sections to be executed serially and threads to wait on each other. We implement flow-affinity scheduling by modifying the source code of our applications such that, after receiving a packet, a thread can either process the packet directly or enqueue (in software with a lock) the packet for processing by other threads. In Figure 8.7 we evaluate two forms of flow-affinity, where packets are either mapped to a specific one of the two CPUs (`CPU-Affinity`), or to a specific one of the eight thread contexts available across both CPUs (`Thread-Affinity`).

Flow-affinity is determined for `NAT` and `Classifier` by hashing the IP header fields, and for `Intruder2` by extracting the flow identifier from the packet payload. We cannot evaluate flow-affinity for `UDHCP` because we did not find a clear identifier for flows that would result in parallel packet processing, since UDHCP has many inter-related critical sections (as shown in Figure 8.1). To implement a reduced lock contention for the flow-affinity approaches we (i) replicated shared data structures when necessary, in particular hash-tables in `NAT` and `Classifier` and (ii) modified the synchronization code such that each CPU operates on a separate subset of the mutexes for `CPU-Affinity`, and uses no mutexes for `Thread-Affinity`.

Figure 8.7 shows that flow-affinity scheduling only improves `Classifier`. `NAT` shows a slight improvement for CPU-based affinity scheduling and otherwise `NAT` and `Intruder2` suffer slowdowns due to load-imbalance: the downside of flow-affinity scheduling is that it reduces flexibility in mapping packets to threads, and hence can result in load-imbalance. The slowdown due to load-imbalance is less pronounced for `CPU-Affinity` because the packet workload is still shared among the threads of each CPU. Overall, Figure 8.7 shows that TM outperforms the best performing flow-affinity approach by 4% for `NAT` and 31% for `Classifier`, while requiring no special code modification.

8.7.5 Additional Mutexes

Other than the true data dependences in Figure 8.1, we are motivated to verify if applications are not serialized because of the round-robin assignment of mutexes to a larger number of shared data structures, i.e. two independent flows that have been assigned the same mutex identifier. While there are alternatives to mutexes in ASIC processors such as atomic operations and load-link/store-conditional, NetThreads, like FPGA multicores made out of off-the-shelf NIOS-II or Microblaze processors, does not yet support them. Figure 8.8 shows that an unlimited number of mutexes improves the performance

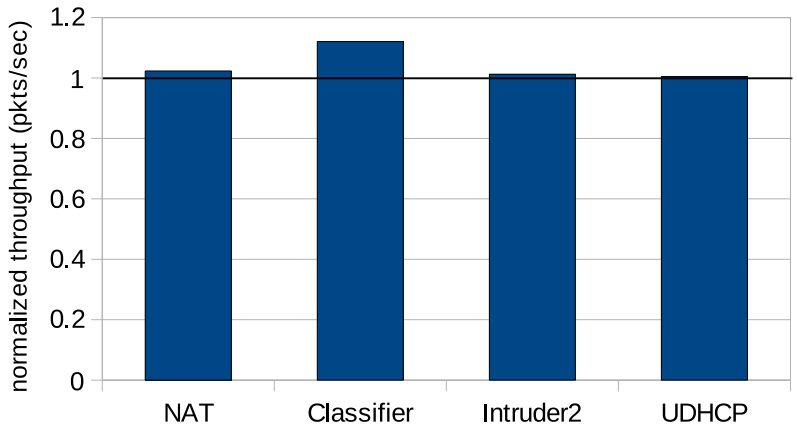


Figure 8.8: Simulated normalized throughput resulting from the use of an unlimited number of mutexes, relative to the existing 16 mutexes.

by less than 2%, except for Classifier (12%). While this is a motivation to implement mechanisms to limit lock aliasing, the performance benefits are marginal overall.

8.8 Summary

In this chapter we describe, implement, and evaluate four threaded, stateful, control-flow-intensive networking applications that share memory and synchronize, a real implementation of an HTM system called NetTM on the NetFPGA [91] platform, and compare with a two-processor, eight-threaded base system that implements only conventional lock-based synchronization (called NetThreads [81]). We describe the selection of TM features that match the strengths of FPGAs, including an eager version management mechanism that allows for longer transactions with more writes and can support both lock-based and transactional synchronization, and an application-specific conflict detection mechanism. Also, we show that an eager TM can be integrated into a multithreaded processor pipeline without impacting clock frequency by adding a stage.

On a NetFPGA board, we measure that NetTM outperforms flow-affinity scheduling, but that NetTM could be extended to exploit a flow affinity approach, assuming the code has such affinity and the programmer is able and willing to replicate and provide scheduling for global data structures as necessary. All in all, we have demonstrated that transactional memory (TM) provides the best overall performance, by exploiting the parallelism available in the processing of packets from independent

flows, while allowing the most flexible packet scheduling and hence load balancing. Our NetTM implementation makes synchronization (i) easier, by allowing more coarse-grained critical sections and eliminating deadlock errors, and (ii) faster, by exploiting the optimistic parallelism available in many concurrent critical sections. For multithreaded applications that share and synchronize, we demonstrate that NetTM can improve throughput by 6%, 55%, and 57% over our NetThreads locks-only system, although TM is inappropriate for one application due to short transactions that frequently conflict. As an extension to this study, in Appendix B, we show that NetTM can scale up to 8 4-threaded cores, and that NetTM performs best for coarse-grained critical sections with low conflict intensity.

Chapter 9

Conclusions

As the capacity of FPGAs continues to grow, they are increasingly used by embedded system designers to implement complex digital systems, especially in telecommunication equipment, the core market of FPGAs. Since software is the method of choice for programming applications with elaborate and periodically changing control-flow, FPGAs require an overlay architecture to execute software. As a possible solution, one can already license off-the-shelf soft processors with the caveat that they are geared towards running sequential control tasks rather then throughput-oriented applications. By improving soft processors, designers will be able to instantiate a number of them for efficiently executing a parallel application with the maximum of programming ease.

In this thesis, we make soft processors more amenable to exploit the parallelism inherent in packet processing applications by (ii) increasing the area efficiency of the cores; (ii) describing the design space of multiprocessor architecture; and (iii) providing techniques to manage the synchronization overheads, in particular using transactional memory. We show that multithreaded multicore organizations can significantly improve the throughput of single-threaded soft processors and that transactional memory can be used to program multiple threads while avoiding the difficulties traditionally associated with parallel programming (Section 8).

This thesis led to the publication of a number of papers [75–78, 80, 83, 84] and the public release of NetThreads [81], NetThreads-RE [82] and NetTM [79] along with a their documentation, which were welcomed by the NetFPGA networking research community. Our experimental approach consists of collaborating with such experts in the networking area to do full-scale studies with real hardware

and packets, as opposed to focusing on micro-benchmarks with simulated inputs and outputs. Our NetThreads infrastructure was used as the enabling technology by non-FPGA adepts to create a precise packet generator [127] that was extended to operate in closed-loop, a system that is also publicly released [44]. Finally, our infrastructure was used to build a flexible system for low-latency processing of stock quotes in algorithmic financial trading [126]. We consider our system architecture successful if it can leverage the configurable fabric of FPGAs to: (i) provide options for application-specific area efficiency tuning; (ii) use knowledge about the application behavior to optimize the synchronization wait times; and (iii) scale the performance of a given (fixed) program to a larger number of threads. To achieve these requirements, we make the following contributions.

9.1 Contributions

1. **Design Space Exploration of Multithreaded and Multicore Soft Systems** To demonstrate that parallel software threads can execute efficiently on FPGAs, we show that for 3, 5, and 7-stage pipelined processors, careful tuning of multithreading can improve overall instructions per cycle by 24%, 45%, and 104% respectively, and area-efficiency by 33%, 77%, and 106%. When considering a system with off-chip memory, the block RAM requirements of a multithreaded system grows much faster per core when each thread requires its own cache space and register file. Therefore we find that single-threaded cores are able to more fully utilize the area of an FPGA and deliver the best performance when the number of block RAMs is the limiting factor. We show that our processor designs can span a broad area versus performance design space, where multithreaded processors dominate core-for-core in terms of performance. In particular, we demonstrate a technique to handle stalls in the multithreaded pipeline that is able to deliver an area-efficiency comparable to a single-threaded core and significantly more throughput, even if the single-threaded core has a larger cache size. We point to a number of designs that can be of interest to a designer when he must trade off area, performance and frequency metrics in a full-system implementation.

2. **Extending Multithreading to Allow for Scheduling a Variable Number of Thread Contexts** Because of their added performance and compact size in a shared memory setting, we elect

multithreaded cores for packet processing. In their most basic form, multithreaded cores are limited to executing instructions from all of their thread contexts in round-robin. Imposing this limitation is what allows baseline multithreaded cores to have no hazard detection and therefore an improved area and clock frequency. We find that in real applications, synchronization across the threads leads to phases where some threads must wait on mutexes for extended periods of time. As this phenomenon was measured to be the leading source of contention, we present a technique that allows to suspend an arbitrary number threads, and recuperate their pipeline slots without re-introducing runtime hazard detection in the pipeline. This technique specifically exploits the multiple-of-9bits width of block RAMs. Using a real FPGA-based network interface, we measure packet throughput improvements of 63%, 31% and 41% for our three applications because of the ability to overlap computations with wait times on mutexes.

3. **Demonstrating Advantages of a Run-to-Completion Programming Model with Transactional Memory for Packet Processing** Because there are many programming models available for packet processing, we evaluate a number of them with our soft core platform to determine if our platform supports the most desirable one. We demonstrate that many complex packet processing applications written in a high-level language, especially those that are stateful, are unsuitable to pipelining. We also show that scheduling packets to different clusters of threads can improve throughput at the expense of code changes that are only possible when such an affinity of packets to threads is possible. We demonstrate that transactional memory (TM) provides the best overall performance in most cases, by exploiting the parallelism available in the processing of packets from independent flows, while allowing the most flexible packet scheduling and hence load balancing. Our implementation of transactional memory also supports locking and can therefore also leverage packet scheduling to threads for added performance.

4. **The First Hardware Transactional Memory Design Integrated with Soft Processor Cores** Our NetTM implementation makes synchronization (i) easier, by allowing more coarse-grained critical sections and eliminating deadlock errors, and (ii) faster, by exploiting the optimistic parallelism available in many concurrent critical sections. Integrating TM directly with the processor cores makes the processor architectural changes seamless for the programmer and allow

for efficient checkpoint, rollback and logging mechanisms. For multithreaded applications that share and synchronize, we demonstrated that a 2-core system with transactional memory can improve throughput by 6%, 55%, and 57% over a similar system with locks only. We also studied aspects that can hinder the benefits of speculative execution and identified that TM is inappropriate for one application due to short transactions that frequently conflict.

5. **NetThreads/NetThreads-RE/NetTM Infrastructures** We provide high-throughput, high utilization and area efficient soft cores for packet processing through releases [79, 81, 82] which have been downloaded 504 times at the time of this writing. These infrastructures can reach close to full utilization of a 1Gbps link for light workloads [127] (see Section 6.3.2)[1], have been used to demonstrate real uses of software packet processing and are also an enabling platform for future research (Section 6.4). Our released FPGA implementation also has a simulator counterpart [82] that can be used to perform extensive application-level debugging and performance tuning.

9.2 Future Work

In this section, we present three avenues to improve on our implementation to further increase its performance.

1. **Custom Accelerators** Because soft processors do not have the high operating frequency of ASIC processors, it is useful for some applications to summarize a block of instructions into a single custom instruction [125]. The processor can interpret that new instruction as a call to a custom logic block (potentially written in a hardware description language or obtained through behavioral synthesis). Hardware accelerators are common in network processing ASICs (e.g. NFP-32xx, Octeon and Advanced PayloadPlus): in the context of FPGAs, accelerators will require special attention to be integrated in a multiprocessor in a scalable fashion. In particular, because of area constraints, threads may have to share accelerators, pipeline them and/or make them multi-purpose to assist in different operations in the packet processing. We envision that this added hardware could also be treated like another processor on chip, with access to the shared memory

[1] The actual bandwidth attained is a function of the amount of computation per packet.

busses and able to synchronize with other processors. Because of the bit-level parallelism of FPGAs, custom instructions can provide significant speedup to some code sections [64, 95].

2. **Maximizing the Use of On-Chip Memory Bandwidth**. Our research on packet processing so far has focused on a particular network card with fast network links, but containing an FPGA that is a few generations old [81, 82]. While we achieved acceptable performance with only two processors, another of our studies has shown that more recent FPGAs could allow scaling to a much larger number of processors and computing threads (Appendix B). In a newer FPGA, memory contention would gradually become a bottleneck as a designer would connect more processors to the shared data cache of our current architecture. While an eager version management scheme scales to multiple data caches [158], a previous implementation of coherent data caches on an FPGA [151] increases considerably the access latency to the data cache, e.g. 18 cycles pure coherence overhead per read miss. Our intuition is thus that a shared data cache performs better for a small number of processors and an interesting future work consists of investigating alternatives in scaling to more processors (possibly with an efficient and coherent system of caches) should focus specifically to match the strengths and weaknesses of FPGAs.

3. **Compiler optimizations** Despite using a number of compiler techniques to improve the area-efficiency of our cores (Section 4.1), we have not tuned `gcc` strictly to deliver more performance. There are many possible automatic compiler optimizations that could, e.g.: (i) schedule instructions based on our pipeline depth and memory latency; (ii) schedule instruction and organize data layout based on the interaction of threads inside a single multi-threaded core; (iii) maximize the benefits of transactional memory by hoisting conflicting memory accesses closer to the beginning of a transaction, optimizing the data layout to ease conflict detection or provide a method for feedback-directed selective parallelization of the code that is least prone to conflict.

Appendix A

Application-Specific Signatures for Transactional Memory

As reconfigurable computing hardware and in particular FPGA-based systems-on-chip comprise an increasing number of processor and accelerator cores, supporting sharing and synchronization in a way that is scalable and easy to program becomes a challenge. As we explained in Chapter 8, *Transactional memory* (TM) is a potential solution to this problem, and an FPGA-based system provides the opportunity to support TM in hardware (HTM). Although there are many proposed approaches to support HTM for ASICs, these do not necessarily map well to FPGAs. In particular, in this work we demonstrate that while *signature*-based conflict detection schemes (essentially bit vectors) should intuitively be a good match to the bit-parallelism of FPGAs, previous schemes result in either unacceptable multicycle stalls, operating frequencies, or false-conflict rates. Capitalizing on the reconfigurable nature of FPGA-based systems, we propose an application-specific signature mechanism for HTM conflict detection. Using both real and projected FPGA-based soft multiprocessor systems that support HTM and implement threaded, shared-memory network packet processing applications, relative to signatures with bit selection, we find that our application-specific approach (i) maintains a reasonable operating frequency of 125MHz, (ii) has an area overhead of only 5%, and (iii) achieves a 9% to 71% increase in packet throughput due to reduced false conflicts. We start off by introducing signatures and how they come into play.

A.1 Transactional Memory on FPGA

In this chapter, we focus on implementing TM for an FPGA-based soft multiprocessor. While TM can be implemented purely in software (STM), an FPGA-based system can be extended to support TM in hardware (HTM) with much lower performance overhead than an STM. There are many known methods for implementing HTM for an ASIC multicore processor, although they do not necessarily map well to an FPGA-based system. Because it is a challenge to implement efficiently for FPGA-based HTM, we focus specifically on the design of the conflict detection mechanism, and find that an approach based on *signatures* [23] is a good match for FPGAs because of the underlying bit-level parallelism. A signature is essentially a bit-vector [128] that tracks the memory locations accessed by a transaction via hash indexing. However, since signatures normally have many fewer bits than there are memory locations, comparing two signatures can potentially indicate costly false-positive conflicts between transactions. Hence prior HTMs employ relatively large signatures—thousands of bits long—to avoid such false conflicts. One important goal for our system is to be able to compare signatures and detect conflicts in a single pipeline stage, otherwise memory accesses would take an increasing number of cycles and degrade performance. However, as we demonstrate in this chapter, implementing previously proposed large signatures in the logic-elements of an FPGA can be detrimental to the processor's operating frequency. Or, as an equally unattractive alternative, one can implement large and sufficiently fast signatures using block RAMs but only if the indexing function is trivial—which can itself exacerbate false-positive conflicts and negate the value of larger signatures.

A.1.1 Signatures for Conflict Detection

To summarize, our goal is to implement a moderately-sized signature mechanism while minimizing the resulting false conflicts. We capitalize on the reconfigurable nature of the underlying FPGA and propose a method for implementing an application-specific signature mechanism that achieves these goals. An application-specific signature is created by (i) profiling the memory addresses accessed by an application, (ii) using this information to build and optimize a *trie* (a tree based on address prefixes) that allocates more branches to frequently-conflicting address prefixes, and (iii) implementing the trie in a conflict detection unit using simple combinational circuits.

As described in Chapter 6, our evaluation system is built on the NetFPGA platform [91], comprising a Virtex II Pro FPGA, 4 × 1GigE MACs, and 200MHz DDR2 SDRAM. On it, we have implemented a dual-core multiprocessor (the most cores that our current platform can accommodate), composed of 125MHz MIPS-based soft processors, that supports an *eager* HTM [158] via a shared data cache. We have programmed our system to implement several threaded, shared-memory network packet processing applications (packet classification, NAT, UDHCP, and intrusion detection).

We use a cycle-accurate simulator to explore the signature design space, and implement and evaluate the best schemes in our real dual-core multiprocessor implementation. We focus on single-threaded processor cores (the updated system diagram can be found in Figure A.2) and, for comparison, we also report the FPGA synthesis results for a conflict detection unit supporting 4 and 8 threads. Relative to signatures with bit selection, the only other signature implementation that can maintain a reasonable operating frequency of 125MHz, we find that our application-specific approach has an area overhead of only 5%, and achieves a 9% to 71% increase in packet throughput due to reduced false conflicts.

A.1.2 Related Work

There is an abundance of prior work on TM and HTM. Most prior FPGA implementations of HTM were intended as fast simulation platforms to study future multicore designs [48, 147], and did not specifically try to provide a solution tuned for FPGAs. Conflict detection has been previously implemented by checking extra bits per line in private [48, 147] or shared [63] caches. In contrast with caches with finite capacity that require complex mechanisms to handle cache line collisions for speculative data, signatures can represent an unbounded set of addresses and thus do not overflow. Signatures can be efficiently cleared in a single cycle and therefore advantageously leverage the bit-level parallelism present in FPGAs. Because previous signature work was geared towards general purpose processors [119, 128, 159], to the best of our knowledge there is no prior art in customizing signatures on a per-application basis.

A.2 Previous Signature Implementations for HTM

A TM system must track read and write accesses for each transaction (the read and write sets), hence an HTM system must track read and write sets for each hardware thread context. The signature method of tracking read and write sets implements Bloom filters [128], where an accessed memory address is represented in the signature by asserting the k bits indexed by the results of k distinct hashes of the address, and a membership test for an address returns true only if all k bits are set. Since false conflicts can have a significant negative impact on performance, the number and type of hash functions used must be chosen carefully. In this chapter, we consider only the case where each one of the k hash functions indexes a different partition of the signature bits—previously shown to be more efficient [128]. The following reviews the four known hash functions that we consider in this chapter.

Bit Selection [128] This scheme directly indexes a signature bit using a subset of address bits. An example 2-bit index for address $a = [a_3 a_2 a_1 a_0]$ could simply be $h = [a_3,\ a_2]$. This is the most simple scheme (i.e., simple circuitry) and hence is important to consider for an FPGA implementation.

$\mathbf{H_3}$ [128] The H_3 class of hash functions is designed to provide a uniformly-distributed hashed index for random addresses. Each bit of the hash result $\mathbf{h} = [h_1,\ h_0]$ consists of a separate XOR ($\oplus$) tree determined by the product of an address $a = [a_3 a_2 a_1 a_0]$ with a fixed random matrix H as in the following example with a 4-bit address and a 2-bit hash [119]:

$$[h_1,\ h_0] = \mathbf{a}H = [a_3 a_2 a_1 a_0] \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} = [a_3 \oplus a_2 \oplus a_0,\ a_2 \oplus a_1] \tag{A.1}$$

Page-Block XOR (PBX) [159] This technique exploits the irregular use of the memory address space to produce hash functions with fewer XOR gates. An address is partitioned into two non-overlapping bit-fields, and selected bits of each field are XOR'ed together with the purpose of XOR'ing high entropy bits (from the low-order bit-field) with lower entropy bits (from the high order bit-field). Modifying the previous example, if the address is partitioned into 2 groups of 2 bits, we could produce the following example 2-bit hash: $[a_2 \oplus a_0,\ a_3 \oplus a_1]$.

Locality-sensitive XOR [119] This scheme attempts to reduce hash collisions and hence the probability of false conflicts by exploiting memory reference spatial locality. The key idea is to make nearby memory locations share some of their k hash indices to delay filling the signature. This scheme produces k H_3 functions that progressively omit a larger number of least significant bits of the address from the computation of the k indices. When represented as H_3 binary matrices, functions require an increasing number of lower rows to be null. Our implementation, called LE-PBX, combines this approach with the reduced XOR'ing of PBX hashing. In LE-PBX, we XOR high-entropy bits with low-entropy bits within a window of the address, then shift the window towards the most significant (low entropy) bits for subsequent hash functions.

A.3 Application-Specific Signatures

All the hashing functions listed in the previous section create a random index that maps to a signature bit range that is a power of two. In Section A.4, we demonstrate that these functions require too many bits to be implemented without dramatically slowing down our processor pipelines. To minimize the hardware resources required, the challenge is to reduce the number of false conflicts per signature bit, motivating us to more efficiently utilize signature bits by creating application-specific hash functions.

Our approach is based on *compact trie hashing* [111]. A trie is a tree where each descendant of a node has in common the prefix of most-significant bits associated with that node. The result of the hash of an address is the leaf position found in the tree, corresponding to exactly one signature bit. Because our benchmarks can access up to 64 Mbytes of storage (16 million words), it is not possible to explicitly represent all possible memory locations as a leaf bit of the trie. The challenge is to minimize false conflicts by mapping the most contentious memory locations to different signature bits, while minimizing the total number of signature bits.

We use a known greedy algorithm to compute an approximate solution to this NP-complete problem [70]. In the first step, we record in our simulator a trace of the read and write sets of a benchmark functioning at its maximum sustainable packet rate. We organize the collected memory addresses in a trie in which every leaf represents a signature bit. This signature is initially too large to be practical (Figure A.1(b)) so we truncate it to an initial trie (Figure A.1(c)), selecting the most frequently

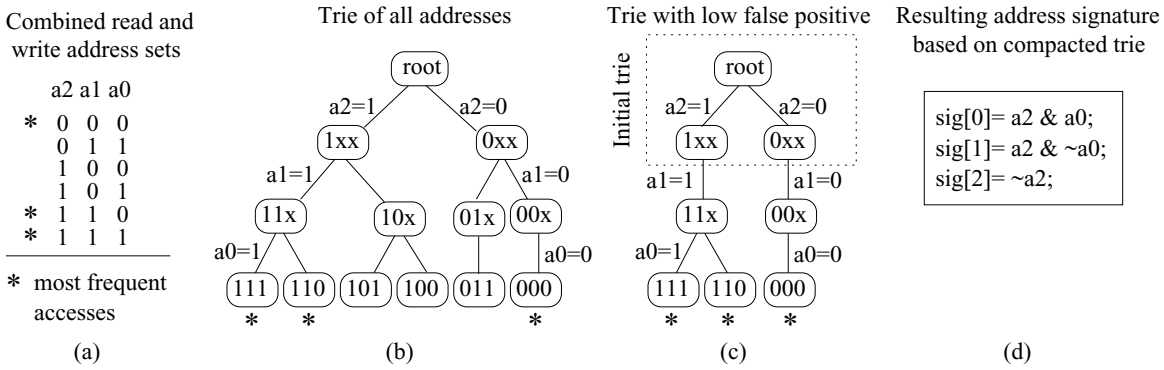


Figure A.1: Example trie-based signature construction for 3-bit addresses. We show (a) a partial address trace, where * highlights frequently accessed addresses, (b) the full trie of all addresses, (c) the initial and final trie after expansion and pruning to minimize false positives, and (d) the logic for computing the signature for a given address (i.e., to be AND'ed with read and write sets to detect a conflict).

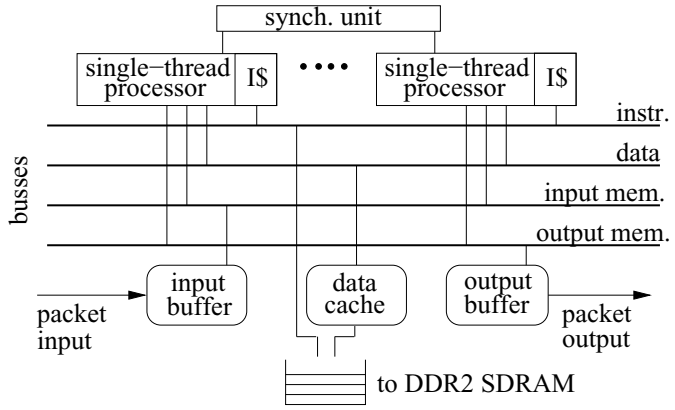


Figure A.2: The architecture of our soft multiprocessor with 2 single-threaded processor cores.

accessed branches. To reduce the hardware logic to map an address to a signature (Figure A.1(d)), only the bits of the address that lead to a branch in the trie are considered. For our signature scheme to be safe, an extra signature bit is added when necessary to handle all addresses not encompassed by the hash function. We then replay the trace of accesses and count false conflicts encountered using our initial hashing function. We iteratively expand the trie with additional branches and leaves to eliminate the most frequently occurring false-positive conflicts (Figure A.1(c)). Once the trie is expanded to a desired false positive rate, we greedily remove signature bits that do not negatively impact the false positive rate (they are undesirable by-products of the expansion). Finally, to further minimize the number of signature bits, we combine signature bits that are likely ($> 80\%$) to be set together in non-aborted transactions.

Table A.1: Applications and their mean statistics.

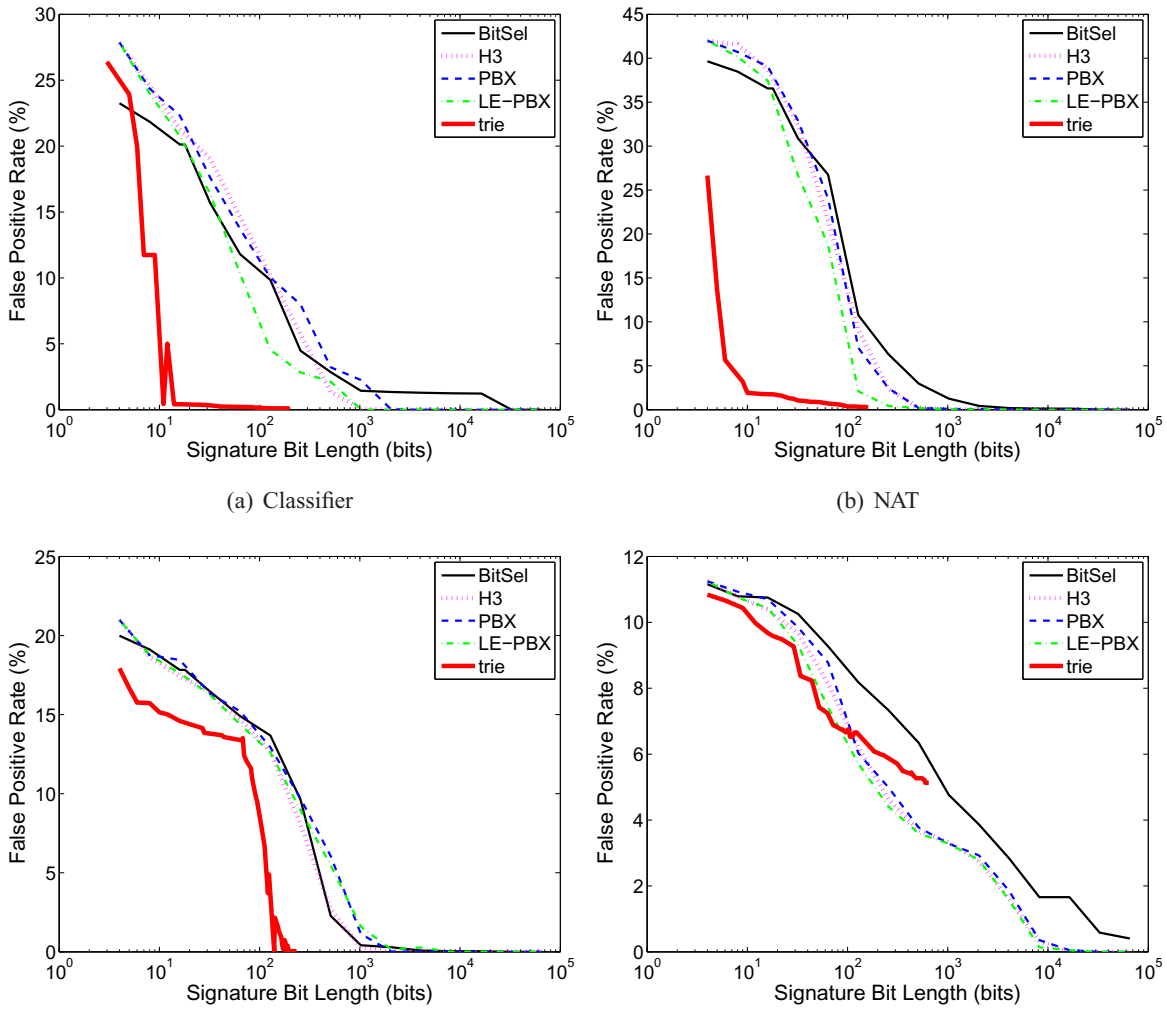
Benchmark	Dyn. Instr. /packet	Dyn. Instr. /transaction	Uniq. Sync. Addr. /transaction	
			Reads	Writes
`Classifier`	2553	1881	67	58
`NAT`	2057	1809	50	41
`UDHCP`	16116	3265	430	20
`Intruder`	12527	399	37	23

Transactional memory support The single port from the processors to the shared cache in Figure A.2 implies that memory accesses undergo conflict detection one by one in transactional execution, therefore a single trie hashing unit suffices for both processors. Our transactional memory processor uses a shadow register file to revert its state upon rollback (versioning [2] avoids the need for register copy). Speculative memory-writes trigger a backup of the overwritten value in an undo-buffer [158] that we over-provision with storage for 2048 values per thread. Each processor has a dedicated connection to a synchronization unit that triggers the beginning and end of speculative executions when synchronization is requested in software.

A.4 Results

In this section, we first evaluate the impact of signature scheme and length on false-positive conflicts, application throughput, and implementation cost. These results guide the implementation and evaluation of our real system. For reference, we report in Table A.1 the statistics of our benchmarks (Table 3.1) that specifically pertain to the synchronization (note that some columns do not have the same units as in Table 6.2, and some numbers reflect code the optimizations explained in Section 8.3).

Resolution of Signature Mechanisms Using a recorded trace of memory accesses obtained from a cycle-accurate simulation of our TM system that models perfect conflict detection, we can determine the false-positive conflicts that would result from a given realistic signature implementation. We use a recorded trace because the false positive rate of a dynamic system cannot be determined without affecting the course of the benchmark execution: a dynamic system cannot distinguish a false-positive conflict from a later true conflict that would have happened in the same transaction, if it was not aborted immediately. We compute the false positive rate as the number of false conflicts divided by the total number of transactions, including repeats due to rollback.



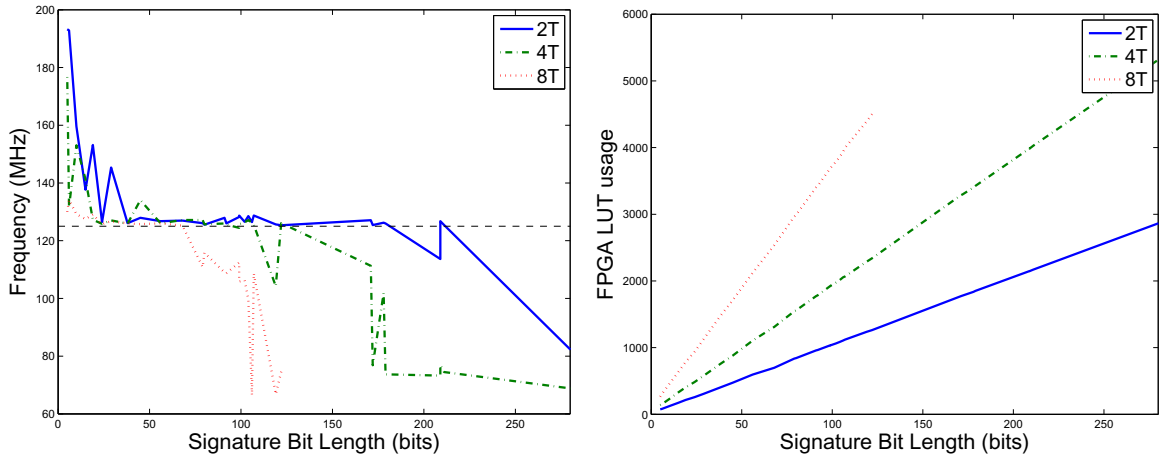
(a) Classifier (b) NAT

(c) UDHCP (d) Intruder

Figure A.3: False positive rate vs signature bit length. Trie-based signatures were extended in length up to the length that provides zero false positives on the training set.

The signatures that we study are configured as follows. The bit selection scheme selects the least significant word-aligned address bits, to capture the most entropy. For H3, PBX and LE-PBX, we found that increasing the number of hash functions caused a slight increase in the false positive rate for short signatures, but helped reduce the number of signature bits required to completely eliminate false positives. We empirically found that using four hash functions is a good trade-off between accuracy and complexity, and hence we do so for all results reported. To train our trie-based hash functions, we use a different but similarly-sized trace of memory accesses as a training set.

Figure A.3 shows the false positive rate for different hash functions (bit selection, H3, PBX, LE-



(a) Frequency (b) LUT usage

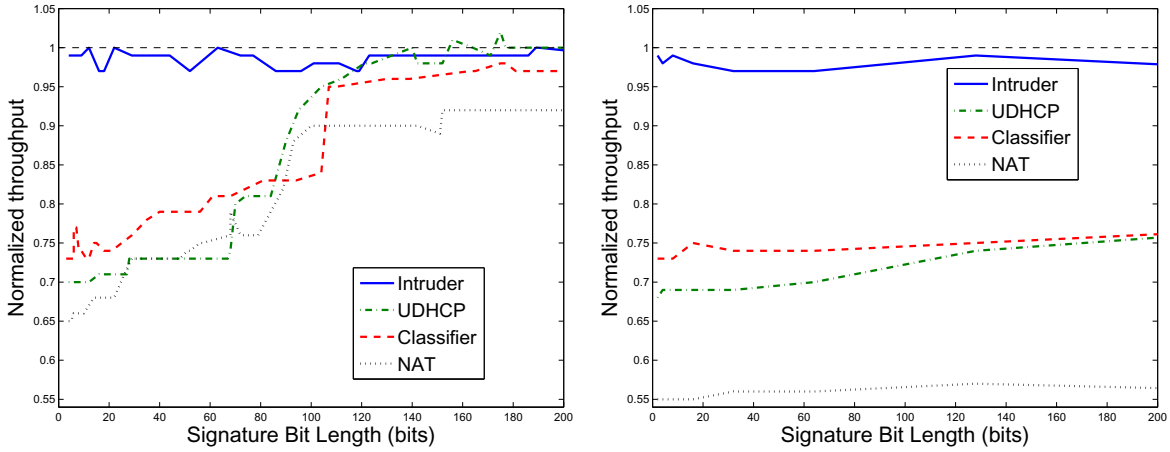
Figure A.4: Impact of increasing the bit length of trie-based signatures on (a) frequency and (b) LUT usage of the conflict detection unit for 2, 4, and 8-thread (2T,4T,8T) systems. The results for H3, PBX and LE-PBX are similar. In (a) we highlight the system operating frequency of 125MHz.

PBX and trie-based) as signature bit length varies. The false positive rate generally decreases with longer signatures because of the reduced number of collisions on any single signature bit—although small fluctuations are possible due to the randomness of the memory accesses. Our results show that LE-PBX has a slightly lower false positive rate than H3 and PBX for an equal number of signature bits. Bit selection generally requires a larger number of signature bits to achieve a low false positive rate, except for UDHCP for which most of the memory accesses point to consecutive statically allocated data. Overall, the trie scheme outperforms the others for CLASSIFIER, NAT and UDHCP by achieving close to zero false positive rate with less than 100 bits, in contrast with several thousand bits. For INTRUDER, the non-trie schemes have a better resolution for signatures longer than 100 bits due to the relatively large amount of dynamic memory used, which makes memory accesses more random. Quantitatively we can compute the entropy of accesses as $\sum_{i=0}^{n-1} -p(x_i)\log_2 p(x_i)$ where $p(x_i)$ is the probability of an address appearing at least once in a transaction—with this methodology INTRUDER has an entropy 1.7 times higher on average than the other benchmarks, thus explaining the difficulty in training its trie-based hash function.

Implementation of a Signature Mechanism Figure A.4 shows the results of implementing a signature-based conflict detection unit using solely the LUTs in the FPGA for a processor system with 2 threads

like the one we implemented (2T) and for hypothetical transactional systems with 4 and 8 threads (4T and 8T). While the plot was made for a trie-based hashing function, we found that H3, PBX and LE-PBX produced similar results. As we will explain later, the bit selection scheme is better suited to a RAM-based implementation. In Figure A.4(a) we observe that the CAD tools make an extra effort to meet our 125 MHz required operating frequency by barely achieving it for many designs. In a 2-thread system, two signatures up to 200 bits will meet our 125MHz timing requirement while a 4-thread system can only accommodate four signatures up to 100 bits long. For 8-threads, the maximum number of signature bits allowed at 125MHz is reduced to 50 bits. Figure A.4(b) shows that the area requirements grow linearly with the number of bits per signature. In practice, for 2-threads at 200 bits, signatures require a considerable amount of resources: approximately 10% of the LUT usage of the total non-transactional system. When the conflict detection unit is incorporated into the system, we found that its area requirements—by putting more pressure on the routing interconnect of the FPGA—lowered the maximum number of bits allowable to less than 100 bits for our 2-thread system (Table A.2). Re-examining Figure A.3, we can see that the trie-based hashing function delivers significantly better performance across all the hashing schemes proposed for less than 100 signature bits.

An alternate method of storing signatures that we evaluate involves mapping an address to a signature bit corresponding to a line in a block RAM. On that line, we store the corresponding read and write signature bit for each thread. To preserve the 125MHz clock rate and our single-cycle conflict detection latency, we found that we could only use one block RAM and that we could only use bit selection to index the block RAM—other hashing schemes could only implement one hash function with one block RAM and would perform worse than bit selection in that configuration. Because the data written is only available on the next clock cycle in a block RAM, we enforce stalls upon read-after-write hazards. Also, to emulate a single-cycle clear operation, we version the read and write sets with a 2-bit counter that is incremented on commit or rollback to distinguish between transactions. If a signature bit remains untouched and therefore preserves its version until a transaction with an aliasing version accesses it (the version wraps over a 2-bit counter), the bit will appear to be set for the current transaction and may lead to more false positives. The version bits are stored on the same block RAM line as their associated signature bits, thus limiting the depth of our 16Kb block RAM to 2048 entries (8-bits wide). Consequently, our best bit selection implementation uses a 11 bit-select of the word-aligned



(a) Trie-based hashing (b) LE-PBX

Figure A.5: Throughput with signatures using trie-based hashing with varying signature sizes normalized to the throughput of an ideal system with perfect conflict detection (obtained using our cycle-accurate simulator).

least-significant address bits.

Impact of False Positives on Performance Figure A.5 shows the impact on performance in a full-system simulation of a varying signature length, when using either a trie-based hashing function or LE-PBX, the scheme with the second-lowest false positive rate. The jitter in the curves is again explained by the unpredictable rollback penalty and rate of occurrence of the false positives, varying the amount contention on the system. Overall, we can see that signatures have a dramatic impact on system throughput, except for INTRUDER for which the false positive rate varies little for this signature size range (Figure A.3(d)). We observe that for CLASSIFIER, UDHCP and NAT, although they achieve a small false positive rate with 10 bits on a static trace of transactional accesses (Figure A.3), their performance increases significantly with longer signatures. We found that our zero-packet drop policy to determine the maximum throughput of our benchmarks is very sensitive to the compute-latency of packets since even a small burst of aborts and retries for a particular transaction directly impacts the size of the input queue which in turn determines packet drops. The performance of NAT plateaus at 161 bits because that is the design that achieves zero false positives in training (Figure A.3(b)). As expected, Figure A.5(b) shows that there is almost no scaling of performance for LE-PBX in the possible signature implementation size range because the false positive rate is very high.

Benchmark	Max. Signature bits	Total LUT usage	LUT overhead	Additional throughput
CLASSIFIER	92	20492	5%	12%
NAT	68	20325	4%	58%
UDHCP	84	20378	4%	9%
INTRUDER	96	20543	5%	71%

Table A.2: Size, LUT usage, LUT overhead and throughput gain of our real system with the best application-specific trie-based hash functions over bit selection.

Measured Performance on the Real System As shown in Table A.2 and contrarily to the other schemes presented, the size of the trie-based signatures can be adjusted to an arbitrary number of bits to maximize the use of the FPGA fabric while respecting our operating frequency. The maximum signature size is noticeably smaller for NAT because more address bits are tested to set signature bits, which requires more levels of logic and reduces the clock speed. In all cases the conflict detection with a customized signature outperforms the general purpose bit selection. This is coherent with the improved false positive rate observed in Figure A.3. We can see that bit selection has the best performance when the data accesses are very regular like in UDHCP, as indicated by the low false positive rate in Figure A.3(c). Trie-based hashing improves the performance of INTRUDER the most because the bit selection scheme suffers from bursts of unnecessary transaction aborts.

CAD Results Comparing two-processor full system hardware designs, the system with trie-based conflict detection implemented in LUTs consumes 161 block RAMs and the application-specific LUT usage reported in Table A.2. Block-RAM-based bit selection requires one additional block RAM (out of 232, i.e., 69% of the total capacity) and consumes 19546 LUTs (out of 47232, i.e. 41% of the total capacity). Since both kinds of designs are limited by the operating frequency, trie-based hashing only has an area overhead of 4.5% on average (Table A.2). Hence the overall overhead costs of our proposed conflict detection scheme are low and enable significant throughput improvements.

A.5 Summary

In this chapter, we describe the first soft processor cores integrated with transactional memory, and evaluated on a real (and simulated) system with threaded applications that share memory. We study

several previously-proposed signature-based conflict detection schemes for TM and demonstrate that previous signature schemes result in implementations with either multicycle stalls, or unacceptable operating frequencies or false conflict rates. Among the schemes proposed in the literature, we find that bit selection provides the best implementation that avoids degrading the operating frequency or stalling the processors for multiple cycles. Improving on this technique, we present a method for implementing more efficient signatures by customizing them to match the access patterns of an application. Our scheme builds on trie-based hashing, and minimizes the number of false conflicts detected, improving the ability of the system to exploit parallelism. We demonstrate that application-specific signatures can allow conflict detection at acceptable operating frequencies (125MHz), single cycle operation, and improved false conflict rates—resulting in significant performance improvements over alternative schemes. On a real FPGA-based packet processor, we measured packet throughput improvements of 12%, 58%, 9% and 71% for four applications, demonstrating that application-specific signatures are a compelling means to facilitate conflict detection for FPGA-based TM systems. With this powerful conflict detection method, we next explore building full multithreaded multicores.

Appendix B

Scaling NetTM to 8 cores

NetFPGA and its VirtexII-Pro FPGA limit the scaling of NetTM to two 4-threaded cores (the most cores that our current platform can accommodate). To evaluate the performance and resource usage of larger-scale designs, we target a newer mid-range Virtex5 FPGA (`XC5VLX110-2`). However, we do not yet have an full system board and hence cannot yet consider the entire NetTM design: hence we elide the support for network and PCI access, and measure only the multiprocessor system. Additional processors are added to the NetTM architecture (Figure 6.1) by simply widening the interconnect arbiters and linearly scaling the load-store queue, the signature tables, and the undo-log. We assume the default `CM4` contention manager (Section 8.7.3), and include the application-specific signature hash function generated for the `Classifier` application.

B.1 CAD Results

Figure B.1 shows the resource usage of transactional and non-transactional multiprocessors with 2, 4 and 8 cores, when synthesized with high effort for speed and with a constraint of 125MHz for the core frequency. The ∼3100 LUT difference between the `TM2P` and `NT2P` systems corresponds roughly to our earlier results in Section 8.7.1, accounting for the change in LUT architecture between the VirtexII-Pro and Virtex5 FPGAs. We can see that the TM processors consume approximately the area of a system with twice the number of non-transactional cores: additional LUTs are associated with the extra pipeline stage per core, conflict detection, the undo-log and pipelining in the interconnect between the cores and the cache. The BRAMs also scale to almost entirely fill the Virtex 5 processor with 8 TM

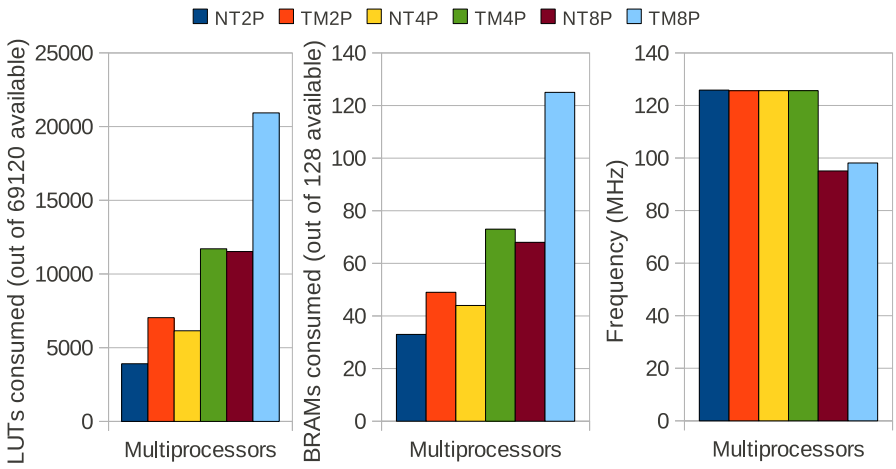


Figure B.1: CAD metrics. Processors are named as follows: *NT* for non-transactional or *TM* for transactional, followed by the number of 4-threaded cores.

```
1:  for 1...N do
2:      for 1...ITER_NON_TM do
3:          work();
4:      end for
5:      acquire_global_lock();
6:      critical_access_at(rand()% RANGE);
7:      for 1..ITER_TM do
8:          work();
9:      end for
10:     release_global_lock();
11: end for
```

Algorithm 1: Parametric benchmark pseudo-code.

cores. However, note that only a small portion of the capacity of the BRAMs that implement the 16 TM log filters is utilized. Lastly, we can see that both 8-core systems fail to meet the 125MHz constraint and have roughly the same achieved frequency: in both cases, the multiplexing in the interconnect becomes significant. However, 125MHz could be achieved by adjusting the architecture to further pipeline cache access (at the cost of greater access latency).

B.2 Performance

Our evaluation so far has focused on realistic workloads for an FPGA embedded in a network card. Since systems with more than two cores cannot be synthesized on the NetFPGA, in this section we explore the performance of these larger systems via cycle-accurate RTL simulation. Since our real

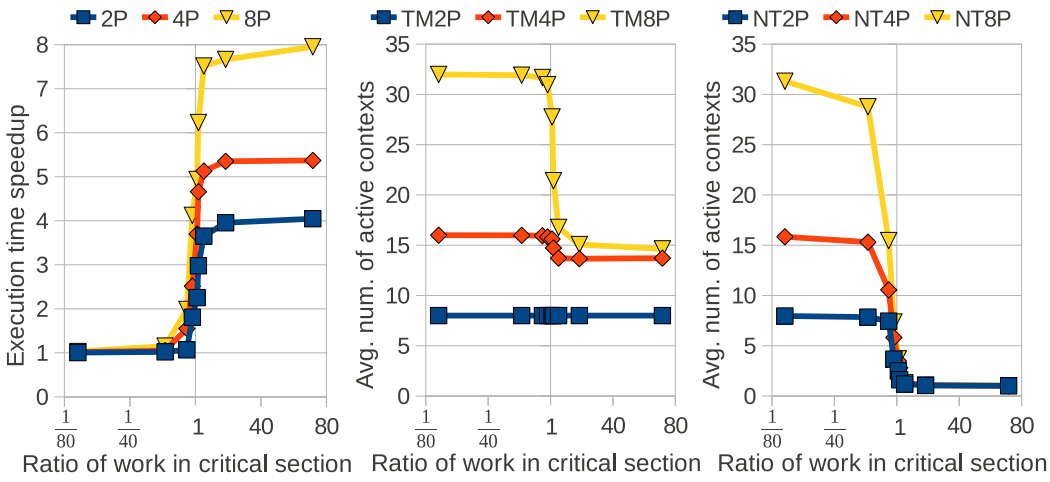


Figure B.2: Left: speedup of TM over non-TM (NT) systems with an equal number of cores (2, 4 or 8) as the ratio of critical work is increased. Middle and right: average number of active contexts for the corresponding processors and application parameters.

applications require a full system including off-chip memory and host interaction, we instead focus on the performance of a parametric application (Algorithm 1) that allows us to vary the relative size of its critical section. For this study we measure N iterations of the benchmark, where N is sufficiently large to minimize the impact of warm-up and tear-down activity. Each thread executes the same code (except for the `rand()` seeding) and is de-scheduled after completing the execution[1]. The `work()` routine only performs computation and stack-based memory accesses that contribute to filling the TM undo-log. Each critical section contains only one memory access that might conflict, at a random location within a parametric range. Each potential location corresponds to one distinct signature bit and by default we use a range of 1024 locations. To minimize the impact of cache contention as we scale the number of processors, we designed the `work()` routine such that the shared-cache bus utilization does not exceed 25% with 8 cores.

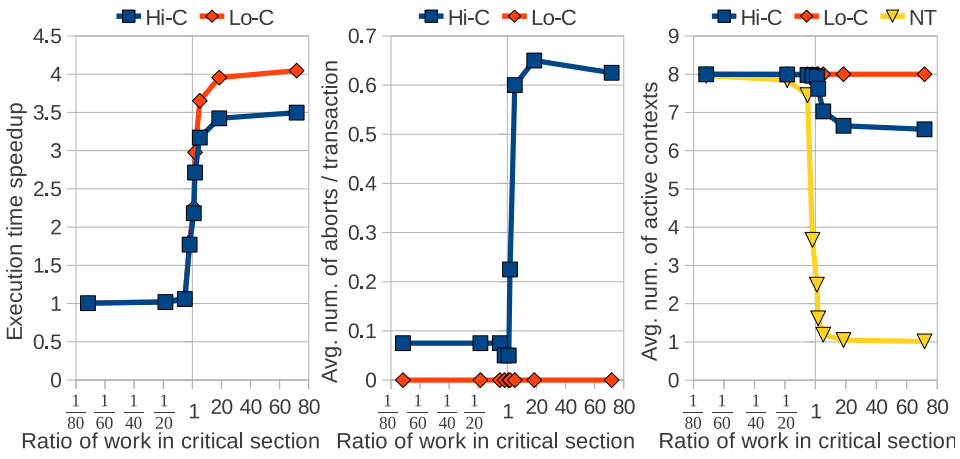
Varying the fraction of critical code Figure B.2 shows the impact of varying the relative size of critical sections compared to the non-critical code ($\frac{\texttt{ITER\_TM}}{\texttt{ITER\_NON\_TM}}$ in Algorithm 1). We observe that the speedup of TM increases with the ratio of critical code. The speedups can be attributed to the average number of active contexts (i.e. not de-scheduled because of pending on a lock or pending on a transaction restart)

[1] This is ensured by forcing each hardware context into a deadlock, which is possible because our TM architecture supports traditional locks.

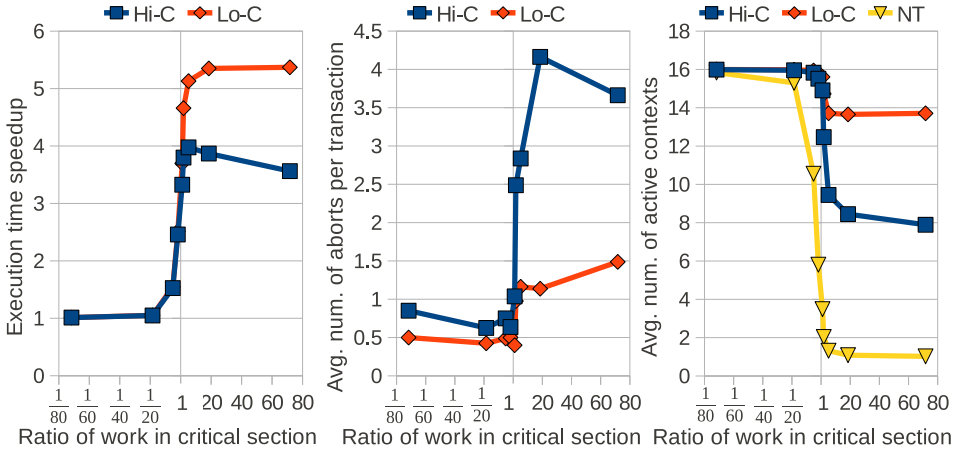
in each processor measured at steady-state, as illustrated in the two right-side graphs in Figure B.2. With mostly non-critical code ($\frac{\texttt{ITER\_TM}}{\texttt{ITER\_NON\_TM}} \ll 1$), almost all the available contexts are active, resulting in a speedup of 1 for TM processors compared to their non-TM counterpart. With mostly critical code ($\frac{\texttt{ITER\_TM}}{\texttt{ITER\_NON\_TM}} \gg 1$), all non-TM processors can only keep 1 thread context active. With multithreading, a core can retire 1 instruction per cycle ($IPC = 1$, other than when there is a cache miss); with a single thread to schedule, a processor core experiences pipeline hazards and can only retire ∼0.5 instructions every cycle for this application. This result explains why the fully utilized TM2P has a speedup of 4.05 over NT2P, and not 2 (i.e. 2 TM cores with IPC of ∼1 each, versus 1 NT context with IPC of ∼0.5). Because the 8-cores TM processor experiences a reduction from 32 to 14.6 active contexts due to aborts, it too experiences pipeline hazards and the speedup is 8.0 (i.e. 8 TM cores with IPC of ∼0.5 versus 1 NT context with IPC of ∼0.5).

Comparing Similar-Area Designs From Figure B.1, we observe that TM support costs roughly 2× LUTs compared to the corresponding non-TM design. This implies that for roughly equivalent area, a designer could choose between TM2P and NT4P, or between TM4P and NT8P—hence we are motivated to compare the performance of these designs. From Figure B.2, we observe that for small critical sections, TM2P and TM4P would each be half the performance of their respective area counterpart (NT4P and NT8P). However, for large critical sections, TM2P would be 4.05× faster than NT4P, and TM4P would be 5.4× faster than NT8P. Large critical sections are inherently easier to program, hence TM provides a trade-off between area-efficiency and programmability.

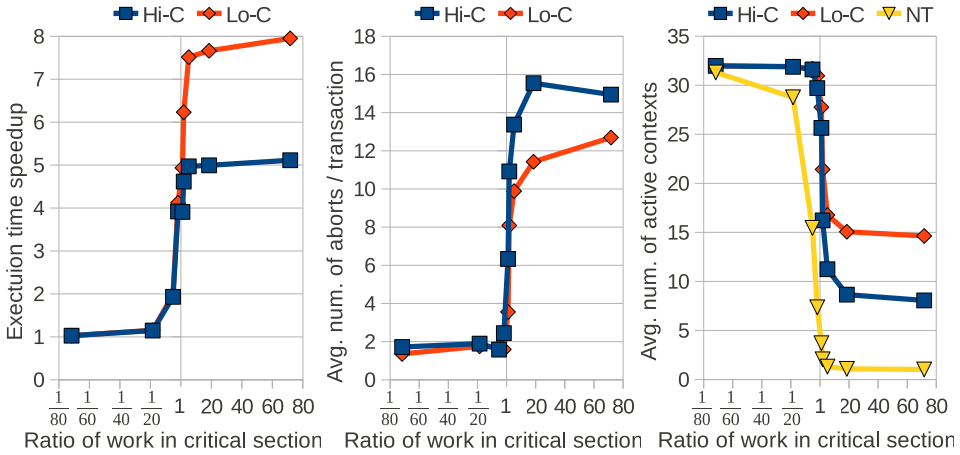
Varying the static conflict intensity In Figure B.3, we control the transactional aborts by varying the range of the random accesses, where each location points to one signature bit. The probability of a conflict when all the contexts are active and have their signature loaded with a potentially conflicting access is given by: $1 - \frac{range!}{(range-\#contexts)!*range^{\#contexts}}$. The probability thus increases with the number of contexts and decreases with a larger range. An 8-core system with 32 contexts is therefore the most impacted system, and the conflict probability is 1 with a memory access range of 8 and 0.39 with a range of 1024 (the default range in Figure B.2). For a 2-core system, those probabilities become 1.00 and 0.27. We respectively label these settings as high-conflict and low-conflict intensity in Figure B.3. With mostly critical code, the 8-core TM system experiences a speedup reduction from 8.0 to 5.1,



(a) 2-core systems



(b) 4-core systems



(c) 8-core systems

Figure B.3: Left: speedup of `TM` over non-TM (`NT`) processors in high (`Hi-C`) and low (`Lo-C`) conflict intensity settings, as the ratio of critical work is increased. Middle: average number of aborts per transaction-instance. Right: average number of active contexts. For reference, the right-most graphs show the appropriate `NT` systems, although `NT` processors are not affected by conflict intensity.

which can be explained by an increased average number of aborts per transaction from 12.7 to 14.9, and a reduction in the average number of active contexts from 14.6 to 8.0: still significantly better than the 1 active context for NT processors, which are not visibly affected by the conflicting range. The 2 and 4-cores systems experience similar but more moderate slowdowns under high-conflict intensity. Interestingly, all systems experience a reduction in the abort rate when the code is mostly transactional. This can be attributed to contention manager de-scheduling aborted threads for a longer period of time, preventing them from repeating aborts. This reduces the potential parallelism and the 4-core system even experiences a speedup reduction when the ratio of critical code is increased.

Bibliography

[1] 1-CORE TECHNOLOGIES . Soft CPU cores for FPGA. http://www.1-core.com, 2010.

[2] AASARAAI, K., AND MOSHOVOS, A. Towards a viable out-of-order soft core: Copy-free, checkpointed register renaming. In *Proc. of FPL* (2009).

[3] ALBRECHT, C., DORING, A., PENCZEK, F., SCHNEIDER, T., AND SCHULZ, H. Impact of coprocessors on a multithreaded processor design using prioritized threads. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Germany, February 2006).

[4] ALBRECHT, C., FOAG, J., KOCH, R., AND MAEHLE, E. DynaCORE : A dynamically reconfigurable coprocessor architecture for network processors. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (Germany, Frebruary 2006), pp. 101–108.

[5] ALLEN, J., BASS, B., BASSO, C., BOIVIE, R., CALVIGNAC, J., DAVIS, G., FRELECHOUX, L., HEDDES, M., HERKERSDORF, A., KIND, A., LOGAN, J., PEYRAVIAN, M., RINALDI, M., SABHIKHI, R., SIEGEL, M., AND WALDVOGEL, M. PowerNP network processor hardware software and applications. *IBM Journal of Research and Development 47*, 2 (2003).

[6] ALMEROTH, K. The evolution of multicast: from the MBone to interdomain multicast to Internet2 deployment. *Network, IEEE 14*, 1 (Jan/Feb 2000), 10 –20.

[7] ALTERA CORP. Nios II Processor Reference Handbook v7.2. http://www.altera.com, 2007.

[8] ALTERA CORPORATION. Nios II C-to-Hardware Acceleration Compiler. http://www.altera.com/c2h.

[9] ALTERA CORPORATION. Quartus II. http://www.altera.com.

[10] ALTERA CORPORATION. Accelerating Nios II Ethernet applications. http://www.altera.com/literature/an/an440.pdf, 2005.

[11] ALTERA CORPORATION. Altera Corportation 2009 annual report (form 10-k). http://investor.altera.com, February 2010.

[12] ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. Unbounded transactional memory. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture (HPCA)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 316–327.

[13] ANSARI, M., KOTSELIDIS, C., LUJÁN, M., KIRKHAM, C., AND WATSON, I. On the performance of contention managers for complex transactional memory benchmarks. In *Proc. of the 8th International Symposium on Parallel and Distributed Computing (ISPDC)* (July 2009).

[14] BAHLMANN, B. DHCP network traffic analysis. *Birds-Eye.Net* (June 2005).

[15] BAKER, Z., AND PRASANNA, V. A methodology for synthesis of efficient intrusion detection systems on FPGAs. In *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 2004), pp. 135 – 144.

[16] BALL, J. The Nios II Family of Configurable Soft-Core Processors. In *Proc. of Hot Chips* (CA, USA, August 2005), Altera.

[17] BEHESHTI, N., GANJALI, Y., GHOBADI, M., MCKEOWN, N., AND SALMON, G. Experimental study of router buffer sizing. In *Proc. of the 8th ACM SIGCOMM conference on Internet measurement (IMC)* (New York, NY, USA, 2008), ACM, pp. 197–210.

[18] BOBBA, J., MOORE, K. E., VOLOS, H., YEN, L., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News 35*, 2 (2007), 81–91.

[19] BONNY, T., AND HENKEL, J. Instruction re-encoding facilitating dense embedded code. In *Proc. of DATE '08* (2008), pp. 770–775.

[20] BROADCOM TECHNICAL PUBLICATIONS. BCM1480 product brief. http://www.broadcom.com/collateral/pb/1480-PB04-R.pdf, April 2007.

[21] CAMPI, F., CANEGALLO, R., AND GUERRIERI, R. IP-reusable 32-bit VLIW RISC core. In *Proc. of the 27th European Solid-State Circuits Conf* (2001), pp. 456–459.

[22] CAO MINH, C., CHUNG, J., KOZYRAKIS, C., AND OLUKOTUN, K. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE International Symposium on Workload Characterization (IISWC)* (Sept. 2008).

[23] CEZE, L., TUCK, J., TORRELLAS, J., AND CASCAVAL, C. Bulk disambiguation of speculative threads in multiprocessors. In *Proc. of the 33rd annual international symposium on Computer Architecture'06* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 227–238.

[24] CISCO SYSTEMS. Cisco carrier routing system. http://www.cisco.com/en/US/prod/collateral/-routers/ps5763/prod_brochure0900aecd800f8118.pdf, 2004.

[25] CISCO SYSTEMS. The Cisco QuantumFlow processor: Cisco's next generation network processor. http://www.cisco.com/en/US/prod/collateral/routers/ps9343/solution_overview_c22-448936.html, 2008.

[26] CLARK, D. Marvell bets on 'plug computers'. *Wall Street Journal February 23* (2009).

[27] COMMUNICATIONS WORKERS OF AMERICA. Report on Internet speeds in all 50 states. http://www.speedmatters.org, August 2009.

[28] COOPERATIVE ASSOCIATION FOR INTERNET DATA ANALYSIS. A day in the life of the Internet. WIDE-TRANSIT link, Jan. 2007.

[29] CORPORATION, A. M. C. nP7510 –10 Gbps Network Processor (OC-192c / 4xOC-48 / 10GE / 10x1GE), 2002.

[30] CRISTEA, M.-L., DE BRUIJN, W., AND BOS, H. FPL-3: Towards language support for distributed packet processing. In *NETWORKING* (2005), pp. 743–755.

[31] CROWLEY, P., AND BAER, J.-L. A modeling framework for network processor systems. *Network Processor Design : Issues and Practices 1* (2002).

[32] CROWLEY, P., FIUCZYNSKI, M., BAER, J.-L., AND BERSHAD, B. Characterizing processor architectures for programmable network interfaces. *Proc. of the 2000 International Conference on Supercomputing* (May 2000).

[33] DAI, J., HUANG, B., LI, L., AND HARRISON, L. Automatically partitioning packet processing applications for pipelined architectures. *SIGPLAN Not. 40*, 6 (2005), 237–248.

[34] DICE, D., LEV, Y., MOIR, M., AND NUSSBAUM, D. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (New York, NY, USA, 2009), ACM, pp. 157–168.

[35] DIMOND, R., MENCER, O., AND LUK, W. Application-specific customisation of multi-threaded soft processors. *IEE Proceedings—Computers and Digital Techniques 153*, 3 (May 2006), 173–180.

[36] DITTMANN, G., AND HERKERSDORF, A. Network processor load balancing for high-speed links. In *International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)* (San Diego, California, July 2002), pp. pp. 727–735.

[37] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. Routebricks: exploiting parallelism to scale software routers. In *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2009), ACM, pp. 15–28.

[38] DORING, A., AND GABRANI, M. On networking multithreaded processor design: hardware thread prioritization. In *46th IEEE International Midwest Symposium on Circuits and Systems* (Switzerland, 2003), vol. 1, pp. 520–523.

[39] EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. EEMBC. http://www.eembc.org.

[40] ERICSSON CORPORATE PUBLIC & MEDIA RELATIONS. CEO to shareholders: 50 billion connections 2020. Press release, April 2010. http://www.ericsson.com/thecompany/press/releases/2010/04/1403231.

[41] FELDMANN, A. Internet clean-slate design: what and why? *SIGCOMM Comput. Commun. Rev. 37*, 3 (2007), 59–64.

[42] FENDER, J., ROSE, J., AND GALLOWAY, D. The Transmogrifier-4: an FPGA-based hardware development system with multi-gigabyte memory capacity and high host and memory bandwidth. In *Proc. of FPT'05* (december 2005), pp. 301– 302.

[43] FORT, B., CAPALIJA, D., VRANESIC, Z. G., AND BROWN, S. D. A multithreaded soft processor for SoPC area reduction. In *Proc. of FCCM '06* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 131–142.

[44] GHOBADI, M., LABRECQUE, M., SALMON, G., AASARAAI, K., YEGANEH, S. H., GANJALI, Y., AND STEFFAN, J. G. Caliper: A tool to generate precise and closed-loop traffic. In *SIGCOMM Conference* (August 2010).

[45] GIOIA, A. FCC jurisdiction over ISPs in protocol-specific bandwidth throttling. 15 Mich. Telecomm. Tech. Law Review 517, 2009.

[46] GOEL, S., AND SHAWKY, H. A. Estimating the market impact of security breach announcements on firm values. *Inf. Manage. 46*, 7 (2009), 404–410.

[47] GOTTSCHLICH, J. E., CONNORS, D., WINKLER, D., SIEK, J. G., AND VACHHARAJANI, M. An intentional library approach to lock-aware transactional memory. Tech. Rep. CU-CS-1048-08, University of Colorado at Boulder, October 2008.

[48] GRINBERG, S., AND WEISS, S. Investigation of transactional memory using FPGAs. In *Proc. of IEEE 24th Convention of Electrical and Electronics Engineers in Israel* (Nov. 2006), pp. 119–122.

[49] GRÜNEWALD, M., NIEMANN, J.-C., PORRMANN, M., , AND RÜCKERT, U. A framework for design space exploration of resource efficient network processing on multiprocessor SoCs. In *Proc. of the 3rd Workshop on Network Processors & Applications* (2004).

[50] GUTHAUS, M., AND ET AL. MiBench: A free, commercially representative embedded benchmark suite. In *In Proc. IEEE 4th Annual Workshop on Workload Characterisation* (December 2001).

[51] HAMMOND, L., WONG, V., CHEN, M., CARLSTROM, B. D., DAVIS, J. D., HERTZBERG, B., PRABHU, M. K., WIJAYA, H., KOZYRAKIS, C., AND OLUKOTUN, K. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News 32*, 2 (2004), 102.

[52] HANDLEY, M., KOHLER, E., GHOSH, A., HODSON, O., AND RADOSLAVOV, P. Designing extensible IP router software. In *Proc. of the 2nd conference on Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2005), USENIX Association, pp. 189–202.

[53] HAZEL, P. PCRE - Perl compatible regular expressions. http://www.pcre.org.

[54] HERLIHY, M., AND MOSS, J. E. B. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News 21*, 2 (1993), 289–300.

[55] HORTA, E. L., LOCKWOOD, J. W., TAYLOR, D. E., AND PARLOUR, D. Dynamic hardware plugins in an FPGA with partial run-time reconfiguration. In *Proc. of the 39th conference on Design automation (DAC)* (New York, NY, USA, 2002), ACM Press, pp. 343–348.

[56] INTEL CORPORATION. *IXP2850 Network Processor: Hardware Reference Manual*, July 2004.

[57] INTEL CORPORATION. *IXP2800 Network Processor: Hardware Reference Manual*, July 2005.

[58] INTEL CORPORATION. *Intel®Itanium®Architecture Software Developer's Manual*, January 2006.

[59] INTEL CORPORATION. Packet processing with Intel multi-core processors. http://download.intel.com/netcomms/solutions/ipservices-wireless/Intel_Holland_Tunnel_-Whitepaper_Final2.pdf, 2008.

[60] IP SEMICONDUCTORS. OC-48 VPN solution using low-power network processors, March 2002.

[61] JAKMA, P., JARDIN, V., OVSIENKO, D., SCHORR, A., TEPPER, H., TROXEL, G., AND YOUNG, D. Quagga routing software suite. http://www.quagga.net.

[62] JEDEC. JESD79: Double data rate (DDR) SDRAM specification. JEDEC Solid State Technology Association, 2003.

[63] KACHRIS, C., AND KULKARNI, C. Configurable transactional memory. In *Proc. of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 65–72.

[64] KACHRIS, C., AND VASSILIADIS, S. Analysis of a reconfigurable network processor. In *Proc. of IPDPS* (Los Alamitos, CA, USA, 2006), IEEE Computer Society, p. 173.

[65] KACHRIS, C., AND VASSILIADIS, S. Performance evaluation of an adaptive FPGA for network applications. In *Proc. of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 54–62.

[66] KANE, G., AND HEINRICH, J. *MIPS RISC Architecture*. Prentice Hall, 1992.

[67] KAUSHIK RAVINDRAN, NADATHUR SATISH, Y. J., AND KEUTZER, K. An FPGA-based soft multiprocessor system for IPv4 packet forwarding. In *15th International Conference on Field Programmable Logic and Applications (FPL-05)* (August 2005), pp. pp 487–492.

[68] KEISTER, S. TSP3 traffic stream processor – M27483. Mindspeed Technologies, Inc., May 2004.

[69] KENNY, J. R., AND KRIKELIS, A. Digital beam former coefficient management using advanced embedded processor technology. In *HPEC* (2007).

[70] KHULLER, S., MOSS, A., AND NAOR, J. S. The budgeted maximum coverage problem. *Inf. Process. Lett. 70*, 1 (1999), 39–45.

[71] KINGYENS, J., AND STEFFAN, J. G. A GPU-inspired soft processor for high-throughput acceleration. In *Reconfigurable Architectures Workshop* (2010).

[72] KISSELL, K. D. MIPS MT: A multithreaded RISC architecture for embedded real-time processing. In *Proc. of HiPEAC '08* (Sweden, January 2008), pp. 9–21.

[73] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE 25*, 2 (March-April 2005), 21–29.

[74] LABRECQUE, M. Towards a compilation infrastructure for network processors. Master's thesis, University of Toronto, 2006.

[75] LABRECQUE, M., JEFFREY, M., AND STEFFAN, J. G. Application-specific signatures for transactional memory in soft processors. In *Proc. of ARC* (2010).

[76] LABRECQUE, M., AND STEFFAN, J. G. Improving pipelined soft processors with multithreading. In *Proc. of FPL '07* (August 2007), pp. 210–215.

[77] LABRECQUE, M., AND STEFFAN, J. G. Fast critical sections via thread scheduling for FPGA-based multithreaded processors. In *Proc. of 19th International Conference on Field Programmable Logic and Applications (FPL)* (Prague, Czech Republic, Sept. 2009).

[78] LABRECQUE, M., AND STEFFAN, J. G. The case for hardware transactional memory in software packet process ing. In *ACM/IEEE Symposium on Architectures for Networking and Communicat ions Systems* (La Jolla, USA, October 2010).

[79] LABRECQUE, M., AND STEFFAN, J. G. NetTM. http://www.netfpga.org/foswiki/bin/view/NetFPGA/OneGig/NetTM, February 2011.

[80] LABRECQUE, M., AND STEFFAN, J. G. NetTM: Faster and easier synchronization for soft multicores via transactional memory. In *International Symposium on Field-Programmable Gate Arrays (FPGA)* (Monterey, Ca, US, February 2011).

[81] LABRECQUE, M., STEFFAN, J. G., SALMON, G., GHOBADI, M., AND GANJALI, Y. NetThreads: Programming NetFPGA with threaded software. In *NetFPGA Developers Workshop* (August 2009).

[82] LABRECQUE, M., STEFFAN, J. G., SALMON, G., GHOBADI, M., AND GANJALI, Y. NetThreads-RE: Netthreads optimized for routing applications. In *NetFPGA Developers Workshop* (August 2010).

[83] LABRECQUE, M., YIANNACOURAS, P., AND STEFFAN, J. G. Custom code generation for soft processors. *SIGARCH Computer Architecture News 35*, 3 (2007), 9–19.

[84] LABRECQUE, M., YIANNACOURAS, P., AND STEFFAN, J. G. Scaling soft processor systems. In *Proc. of FCCM '08* (April 2008), pp. 195–205.

[85] LAI, Y.-K. *Packet Processing on Stream Architecture*. PhD thesis, North Carolina State University, 2006.

[86] LEE, B. K., AND JOHN, L. K. NpBench: A benchmark suite for control plane and data plane applications for network processors. In *21st International Conference on Computer Design* (San Jose, California, October 2001).

[87] LEVANDOSKI, J., SOMMER, E., STRAIT, M., CELIS, S., EXLEY, A., AND KITTREDGE, L. Application layer packet classifier for Linux. http://l7-filter.sourceforge.net.

[88] LIDINGTON, G. Programming a data flow processor. http://www.xelerated.com, 2003.

[89] LIU, Z., ZHENG, K., AND LIU, B. FPGA implementation of hierarchical memory architecture for network processors. In *IEEE International Conference on Field-Programmable Technology* (2004), pp. 295 – 298.

[90] LOCKWOOD, J. W. Experience with the NetFPGA program. In *NetFPGA Developers Workshop* (2009).

[91] LOCKWOOD, J. W., MCKEOWN, N., WATSON, G., GIBB, G., HARTKE, P., NAOUS, J., RAGHURAMAN, R., AND LUO, J. NetFPGA - an open platform for gigabit-rate network switching and routing. In *Proc. of MSE '07* (June 3-4 2007).

[92] LOCKWOOD, J. W., TURNER, J. S., AND TAYLOR, D. E. Field programmable port extender (FPX) for distributed routing and queuing. In *Proc. of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays (FPGA)* (New York, NY, USA, 2000), ACM Press, pp. 137–144.

[93] LUO, Y., YANG, J., BHUYAN, L., AND ZHAO, L. NePSim: A network processor simulator with power evaluation framework. *IEEE Micro Special Issue on Network Processors for Future High-E nd Systems and Applications* (Sept/Oct 2004).

[94] LUPON, M., MAGKLIS, G., AND GONZÁLEZ, A. Version management alternatives for hardware transactional memory. In *Proc. of the 9th workshop on memory performance (MEDEA)* (New York, NY, USA, 2008), ACM, pp. 69–76.

[95] LYSECKY, R., AND VAHID, F. A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning. In *Proc. of the conference on Design, Automation and Test in Europe (DATE)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 18–23.

[96] MARTIN, M., BLUNDELL, C., AND LEWIS, E. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett. 5*, 2 (2006), 17.

[97] MAXIAGUINE, A., UNZLI, S., CHAKRABORTY, S., AND THIELE, L. Rate analysis for streaming applications with on-chip buffer constraints. In *Asia and South Pacific Design Automation Conference (ASP-DAC)* (Japan, January 2004), pp. 131–136.

[98] MCLAUGHLIN, K., O'CONNOR, N., AND SEZER, S. Exploring CAM design for network processing using FPGA technology. In *International Conference on Internet and Web Applications and Services* (February 2006).

[99] MELVIN, S., AND PATT, Y. Handling of packet dependencies: a critical issue for highly parallel network processors. In *Proc. of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems (CASES)* (New York, NY, USA, 2002), ACM, pp. 202–209.

[100] MEMIK, G., AND MANGIONE-SMITH, W. H. Evaluating network processors using NetBench. *ACM Trans. Embed. Comput. Syst. 5*, 2 (2006), 453–471.

[101] MEMIK, G., MANGIONE-SMITH, W. H., AND HU, W. NetBench: A benchmarking suite for network processors. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Jose, CA, November 2001).

[102] MEMIK, G., MEMIK, S. O., AND MANGIONE-SMITH, W. H. Design and analysis of a layer seven network processor accelerator using reconfigurable logic. In *Proc. of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 131.

[103] MENTOR GRAPHICS CORP. Modelsim SE. http://www.model.com, 2004.

[104] MINIWATTS MARKETING GROUP. Internet usage statistics - the big picture. http://www.internetworldstats.com/stats.htm, 2010.

[105] MORRIS, R., KOHLER, E., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *SIGOPS Oper. Syst. Rev. 33*, 5 (1999), 217–231.

[106] MOUSSALI, R., GHANEM, N., AND SAGHIR, M. Microarchitectural enhancements for configurable multi-threaded soft processors. In *Proc. of FPL '07* (Aug. 2007), pp. 782–785.

[107] MOUSSALI, R., GHANEM, N., AND SAGHIR, M. A. R. Supporting multithreading in configurable soft processor cores. In *Proc. of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems (CASES)* (New York, NY, USA, 2007), ACM, pp. 155–159.

[108] MOYER, B. Packet subsystem on a chip. *Xcell Journal* (2006), 10–13.

[109] MUELLER, F. A library implementation of POSIX threads under UNIX. In *Proc. of USENIX* (1993), pp. 29–41.

[110] MUNTEANU, D., AND WILLIAMSON, C. An FPGA-based network processor for IP packet compression. In *SCS SPECTS* (Philadelphia, PA, July 2005), pp. 599–608.

[111] OTOO, E. J., AND EFFAH, S. Red-black trie hashing. Tech. Rep. TR-95-03, Carleton University, 1995.

[112] PASSAS, S., MAGOUTIS, K., AND BILAS, A. Towards 100 gbit/s Ethernet: multicore-based parallel communication protocol design. In *Proc. of the 23rd international conference on Supercomputing (ICS)* (New York, NY, USA, 2009), ACM, pp. 214–224.

[113] PATTERSON, D. A., AND HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2008.

[114] PAULIN, P., PILKINGTON, C., AND BENSOUDANE, E. StepNP: A system-level exploration platform for network processors. *IEEE Design & Test of Computers 19*, 6 (November 2002), 17–26.

[115] PAULIN, P., PILKINGTON, C., BENSOUDANE, E., LANGEVIN, M., AND LYONNARD, D. Application of a multi-processor SoC platform to high-speed packet forwarding. *Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE) 3* (2004).

[116] PITTSBURGH SUPERCOMPUTING CENTER (PSC). National laboratory for applied networking research (NLANR), 1997.

[117] PLAVEC, F. Soft-core processor design. Master's thesis, University of Toronto, 2004.

[118] PREFONTAINE, M. Iq2200 family product brief, 2002.

[119] QUISLANT, R., GUTIERREZ, E., AND PLATA, O. Improving signatures by locality exploitation for transactional memory. In *Proc. of 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (Raleigh, North Carolina, Sept. 2009), pp. 303–312.

[120] RAJWAR, R., HERLIHY, M., AND LAI, K. Virtualizing transactional memory. In *ISCA* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 494–505.

[121] RAMASWAMY, R., WENG, N., AND WOLF, T. Application analysis and resource mapping for heterogeneous network processor architectures. In *Proc. of Third Workshop on Network Processors and Applications (NP-3) in conjunction with Tenth International Symposium on High Performance Computer Architecture (HPCA-10)* (Madrid, Spain, Feb. 2004), pp. 103–119.

[122] RAVINDRAN, K., SATISH, N., JIN, Y., AND KEUTZER, K. An FPGA-Based Soft Multiprocessor System for IPv4 Packet Forwarding. *Proc. of the International Conference on Field Programmable Logic and Applications* (2005), 487–492.

[123] REDDI, V. J., SETTLE, A., CONNORS, D. A., AND COHN, R. S. Pin: a binary instrumentation tool for computer architecture research and education. In *WCAE '04: Proceedings of the 2004 workshop on Computer architecture education* (New York, NY, USA, 2004), ACM, p. 22.

[124] ROESCH, M. Snort - lightweight intrusion detection for networks. In *Proc. of the 13th USENIX conference on System administration (LISA)* (Berkeley, CA, USA, 1999), USENIX Association, pp. 229–238.

[125] ROSINGER, H.-P. Connecting customized IP to the MicroBlaze soft processor using the Fast Simplex Link (FSL) channel. XAPP529, 2004.

[126] SADOGHI, M., LABRECQUE, M., SINGH, H., SHUM, W., AND JACOBSEN, H.-A. Efficient event processing through reconfigurable hardware for algori thmic trading. In *International Conference on Very Large Data Bases (VLDB)* (Singapore, September 2010).

[127] SALMON, G., GHOBADI, M., GANJALI, Y., LABRECQUE, M., AND STEFFAN, J. G. NetFPGA-based precise traffic generation. In *Proc. of NetFPGA Developers Workshop'09* (2009).

[128] SANCHEZ, D., YEN, L., HILL, M. D., AND SANKARALINGAM, K. Implementing signatures for transactional memory. In *Proc. of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 123–133.

[129] SCHELLE, G., AND GRUNWALD, D. CUSP: a modular framework for high speed network applications on FPGAs. In *FPGA* (2005), pp. 246–257.

[130] SEAMANS, E., AND ROSENBLUM, M. Parallel decompositions of a packet-processing workload. In *Advanced Networking and Communications Hardware Workshop* (Germany, June 2004).

[131] SHADE, L. R. Computer networking in canada: From ca*net to canarie. *Canadian Journal of Communication 19*, 1 (1994).

[132] SHAFER, J., AND RIXNER, S. RiceNIC: a reconfigurable network interface for experimental research and education. In *Proc. of the 2007 workshop on Experimental computer science (ExpCS)* (New York, NY, USA, 2007), ACM, p. 21.

[133] SHANNON, L., AND CHOW, P. Standardizing the performance assessment of reconfigurable processor architectures. In *Proc. of FCCM '03* (2003), pp. 282–283.

[134] TEODORESCU, R., AND TORRELLAS, J. Prototyping architectural support for program rollback using FPGAs. In *Proc. of FCCM '05* (April 2005), pp. 23–32.

[135] THIELE, L., CHAKRABORTY, S., GRIES, M., AND KÜNZLI, S. *Network Processor Design: Issues and Practices*. First Workshop on Network Processors at the 8th International Symposium on High-Performance Computer Architecture (HPCA8). Morgan Kaufmann Publishers, Cambridge MA, USA, February 2002, ch. Design Space Exploration of Network Processor Architectures, pp. 30–41.

[136] THIES, W. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2009.

[137] TURNER, J. S. A proposed architecture for the geni backbone platform. In *Proc. of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems* (New York, NY, USA, 2006), ANCS '06, ACM, pp. 1–10.

[138] UL-ABDIN ZAIN-UL ABDIN, Z., AND SVENSSON, B. Compiling stream-language applications to a reconfigurable array processor. In *ERSA* (2005), pp. 274–275.

[139] UNGERER, T., ROBIČ, B., AND ŠILC, J. A survey of processors with explicit multithreading. *ACM Computing Surveys (CSUR) 35*, 1 (March 2003).

[140] VASSILIADIS, S., WONG, S., GAYDADJIEV, G. N., BERTELS, K., KUZMANOV, G., AND PANAINTE, E. M. The molen polymorphic processor. *IEEE Transactions on Computers* (November 2004), 1363– 1375.

[141] VEENSTRA, J., AND FOWLER, R. MINT: a front end for efficient simulation of shared-memory multiprocessors. In *Proc. of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (NC, USA, January 1994), pp. 201–207.

[142] VERDÚ, J., GARCIA, J., NEMIROVSKY, M., AND VALERO, M. Analysis of traffic traces for stateful applications. In *Proc. of the 3rd Workshop on Network Processors and Applications (NP-3)* (2004).

[143] VERDÚ, J., NEMIROVSKY, M., AND VALERO, M. Multilayer processing - an execution model for parallel stateful packet processing. In *Proc. of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (New York, NY, USA, 2008), ACM, pp. 79–88.

[144] WALL, D. W. Limits of instruction-level parallelism. In *Proc. of the fourth international conference on Architectural support for programming languages and operating systems (ASPLOS)* (New York, NY, USA, 1991), ACM, pp. 176–188.

[145] WANG, J., CHENG, H., HUA, B., AND TANG, X. Practice of parallelizing network applications on multi-core architectures. In *Proc. of the 23rd international conference on Supercomputing (ICS)* (New York, NY, USA, 2009), ACM, pp. 204–213.

[146] WANG, Y., LU, G., , AND LI, X. A study of Internet packet reordering. In *ICOIN* (2004).

[147] WEE, S., CASPER, J., NJOROGE, N., TESYLAR, Y., GE, D., KOZYRAKIS, C., AND OLUKOTUN, K. A practical FPGA-based framework for novel CMP research. In *Proc. of the ACM/SIGDA 15th international symposium on Field programmable gate arrays (FPGA)* (New York, NY, USA, 2007), ACM, pp. 116–125.

[148] WENG, N., AND WOLF, T. Pipelining vs. multiprocessors – choosing the right network processor system topology. In *Proc. of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with The 31st Annual International Symposium on Computer Architecture (ISCA 2004)* (Munich, Germany, June 2004).

[149] WENG, N., AND WOLF, T. Pipelining vs. multiprocessors ? Choosing the right network processor system topology. In *Advanced Networking and Communications Hardware Workshop* (Germany, June 2004).

[150] WOLF, T., AND FRANKLIN, M. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (Austin, TX, April 2000), pp. 154–162.

[151] WOODS, D. Coherent shared memories for FPGAs. Master's thesis, University of Toronto, 2009.

[152] XILINX INC. MicroBlaze RISC 32-Bit Soft Processor, August 2001.

[153] XILINX INC. AccelDSP synthesis tool v9.2.01, 2007.

[154] XILINX INC. MicroBlaze Processor Reference Guide v8.0. http://www.xilinx.com, 2007.

[155] XILINX INC. Virtex-5 family overview, June 2009.

[156] XILINX, INC. 2010 form 10-k and proxy. http://investor.xilinx.com, June 2010.

[157] XILINX INC. Zynq-7000 epp product brief - xilinx, February 2011.

[158] YEN, L., BOBBA, J., MARTY, M. R., MOORE, K. E., VOLOS, H., HILL, M. D., SWIFT, M. M., AND WOOD, D. A. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proc. of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 261–272.

[159] YEN, L., DRAPER, S., AND HILL, M. Notary: Hardware techniques to enhance signatures. In *Proc. of Micro'08* (Nov. 2008), pp. 234–245.

[160] YIANNACOURAS, P., AND ROSE, J. A parameterized automatic cache generator for FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT)* (December 2003), pp. 324– 327.

[161] YIANNACOURAS, P., ROSE, J., AND STEFFAN, J. G. The microarchitecture of FPGA-based soft processors. In *Proc. of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (CASES)* (New York, NY, USA, 2005), ACM Press, pp. 202–212.

[162] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. Application-specific customization of soft processor microarchitecture. In *Proc. of the internation symposium on Field programmable gate arrays (FPGA)* (New York, NY, USA, 2006), ACM Press, pp. 201–210.

[163] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. Exploration and customization of FPGA-based soft processors. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* (February 2007).

[164] YIANNACOURAS, P., STEFFAN, J. G., AND ROSE, J. Fine-grain performance scaling of soft vector processors. In *Proc. of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems (CASES)* (New York, NY, USA, 2009), ACM, pp. 97–106.

[165] YUJIA JIN, NADATHUR SATISH, K. R., AND KEUTZER, K. An automated exploration framework for FPGA-based soft multiprocessor systems. In *Proc. of the 2005 International Conference on Hardware/Software Codesign and System Synthesis (CODES)* (September 2005), pp. 273–278.