APPLICATION AND SYSTEM SUPPORT FOR RECONFIGURABLE COPROCESSORS IN MULTICORE DEVICES

by

Philip C. Garcia

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

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 8/4/2011

This dissertation is approved by the following members of the Final Oral Committee:

Katherine Compton, Associate Professor, Electrical and Computer Engineering
Mikko Lipasti, Professor, Electrical and Computer Engineering
Nam Kim, Assistant Professor, Electrical and Computer Engineering
Michael Schulte, Associate Professor, Electrical and Computer Engineering
David Wood, Professor, Computer Science

To my family and friends, without whom, this thesis would not have been possible

ACKNOWLEDGMENTS

There are many people I'd like to thank for helping me survive my many years in college. Most importantly, I'd like to thank my parents for always being there for me, and being understanding when I repeatedly told them I had no clue how long I had left in school. Their support throughout the years has been remarkable, and I cannot stress how important their influence on me has been. I would also like to thank the rest of my family for being so patient with me. I know everyone was wondering when I was going to graduate, and when I would get to see everyone again, and I hope to soon. I love all of you very much, and cannot thank you enough for your support.

I'd also like to thank all of the friends I've made over the years. Your companionship, and help throughout the years have allowed me to continue on, and yet somehow, those of you who took real jobs after graduating did not convince me to drop out and get a job. My friends have taught me more than any book I have read or class I have taken, for this I am extremely grateful. I will not name names, for fear of leaving many many people out, but you are all important to me, and I cannot stress enough how much you have done for me. I'd especially like to thank my "irc" friends in #acm for . . . being there. The channel's stories of the grasshopper and the octopus, bad jokes, name calling, and links to questionable websites have provided hours of entertainment, and a wonderful diversion from my research.

Surviving years of graduate school is not an easy ordeal, and the pains of undertaking a PhD are quite immense. Therefore I would like to thank Jorge Cham for his wonderful comic *Piled Higher and Deeper*, which represented both the painful truths and the more whimsical aspects of graduate life. Additionally, no PhD would be possible without the help of my labmates and colleagues. Countless discussions have helped shape the way I approach my research, and the direction I chose to take it. The years spent working in an academic lab are rather magical, and allow one to absorb more knowledge than I ever thought possible.

I'd also like to thank the many graduate students in vastly different areas that I have gotten the fortune to know. It is always wonderful to talk to someone doing research on a field completely apart from your own, and to hear their views on the world. They also have let me realize that I do not struggle alone, and that the problems of PhD students are relatively universal in nature.

The many days (and more nights) I spent in lab were not spent in silence. As a lover of music, I have spent countless hours listening to many different bands. Listening to music allows me to both relax and better concentrate. I can also relate to many of my favorite bands. I'd therefore like to thank my favorite musicians for the countless hours of entertainment you've provided me. In particular, I'd like to thank some of my favorite bands including: Soul

Asylum, The Replacements, The Jayhawks, Two Cow Garage, Paul Westerberg, Slobberbone, Grand Champeen, Big Star, Wilco, and the Kinks. Your music has helped me to maintain my sanity. Additionally the many live concerts I managed to see during graduate school have helped me to maintain a "real" perspective of the world outside of academia. They have been a great chance to relax, and have allowed me to become friends with people from all walks of life, bonding over our shared love of music. Additionally, no mention of music in graduate school would be complete without thanking Strictly Discs, my neighborhood record store that I spent way too much of my stipend at.

Finally, I'd like to thank my thesis committee for their advice, and guidance during my PhD. I want to thank you all for taking the time to help me grow as a researcher, and I am proud to have gotten to know each one of you. This thesis would have been impossible without your input.

This material is based upon work supported by the National Science Foundation under Grant No. 0702605. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABSTRACT

Embedded multicore devices often require high performance with minimal power consumption; many systems use dedicated hardware units to meet these constraints. However, embedded systems have also become increasingly multi-purpose and must be able to execute a wide range of applications – some of which might not yet be known at design time. It is therefore difficult to choose an appropriate mix of dedicated hardware that meets a device's size, cost, and capability constraints. A reconfigurable hardware (RH) coprocessor is a potential solution, as it is highly effective at accelerating a variety of different tasks (which need not necessarily be known in advance), and does so using less energy than general-purpose processors.

In this thesis, I propose a reconfigurable computing system-on-chip that combines general-purpose processor core(s) with a reconfigurable coprocessor. Applications executing on this system use the RH to accelerate commonly-executed functions. In this thesis, I first describe the communication model used between the processor(s) and RH coprocessor. I then describe the programming interface applications use to access the RH, and show that my model allows applications to securely access the RH coprocessor without requiring operating system intervention – greatly reducing the overhead of using the coprocessor. Because of this, my RH coprocessor can even accelerate tasks (or kernels of an application) whose execution time (when running in software) is measured in hundreds of cycles.

After establishing the platform, I examine how my proposed system performs, and propose extensions to the system to further improve system performance. In this thesis, I will demonstrate that, when using my coprocessor memory interface, workloads executing across eight processor cores and the shared RH fabric perform $\sim 95\%$ as well as they would on an idealized system where the coprocessor has zero-cycle access to shared memory. Additionally, I examine the impact hybrid RH/software applications have on software-only applications, and propose a mechanism that prevents streaming RH applications from polluting shared levels of the system's cache; this simple modification improved the performance of software-only applications by up to 32%. I also examine the behavior of software-only applications coscheduled alongside hybrid RH/software applications on simultaneous multithreaded processors, showing that they perform up to $\sim 95\%$ as fast as they do when a multicore system executes the two applications. This is much faster than two software applications can run when coscheduled together, but not as fast as a multicore machine

because the hybrid application still requires CPU resources to execute, slowing down the coscheduled software-only application

Finally, I examine methods that allow multicore RH systems to better utilize RH resources, allowing systems with limited RH resources to perform nearly as well as systems containing more RH resources. I first show that hybrid applications that call the same RH kernel can better utilize the RH by sharing the configured resources. On eight-processor systems executing eight copies of the same applications, workloads that shared configured RH kernels performed 97.4% as well as systems that did not, despite the fact that shared systems required $\sim \frac{1}{8}th$ of the RH resources. I also examined a modified RH kernel scheduling algorithm that periodically determines which RH kernels should be loaded on the RH at any given time. This new scheduling algorithm could better select which RH kernels should be configured on multicore systems. I show that this new scheduler always performs as good, or better than the previous scheduler, and in extreme cases can result in RH allocations that improve system performance by over 2x.

In this thesis, I examine many of the design choices involved in creating a multicore RH computing system, and examine how a modern operating system should present the RH resources to user applications. I then demonstrate that such a system provides the performance required in next-generation computing application, while providing the programmability and flexibility to accelerate many different application domains, and even offer performance improvements to applications not considered when the chip was first fabricated. By doing this, embedded systems manufacturers can make faster, more capable products that consume less energy. Additionally, the hardware in these new devices will be able to adapt to new applications that are created *a*fter the device has shipped, allowing all applications to be accelerated by the processor, and not just the applications that the processor was optimized for.

Chapter 1

Introduction

The ever-increasing demand for faster and more capable embedded computing systems presents a continual challenge to computer architects. Although process technology improvements increase the number of transistors that can fit on a single chip, it is not always clear how those transistors can best improve system performance. Previously, computer architects used aggressive pipelining, out-of-order processing, and superscalar designs to improve the performance of individual processor cores [98, 59]. However, these techniques are no longer sufficient causing many computer architects to examine multicore and multithreaded design approaches [98, 78, 62, 49] to improve system throughput for parallel workloads.

Although these new multicore systems can improve the performance of parallel applications, or execute multiple applications simultaneously, this is often done at the expense of single-threaded performance. Even when these systems execute multithreaded workloads, single-threaded performance can be a limiting factor due to an imbalance in how much work each thread must compute [78]. This often results in processor cores sitting idle waiting for another core to finish executing a long-running thread. Amdahl's law limits the execution time of a multithreaded application to that of its slowest thread [62], further motivating the need to accelerate single threads of execution. Increasing the performance of all processor cores uniformly will cause every thread to execute faster, but may not be the best solution because doing this increases the power consumption of every core.

This thesis focuses instead on the use of a reconfigurable coprocessor to accelerate individual threads within a workload. Applications on these systems execute primarily on a general-purpose processor; however the most compute-intensive sections of an application (referred to as *kernels* in this thesis) can execute on the coprocessor. Offloading this computation to a coprocessor can result in reduced energy consumption and/or faster execution.

The use of reconfigurable hardware as a coprocessor provides the flexibility to create new accelerators after the system has been developed and the chips have been fabricated. The development of heterogeneous systems-on-chip often starts years before the chip is used in actual products. Because of this, the full set of applications that might execute on the processor may not be known at design time. In this thesis, I present a system model for integrating an on-chip reconfigurable hardware (RH) coprocessor for multicore and multithreaded systems, and examine the related system infrastructure necessary to support the RH coprocessor.

1.1 Contributions

In this thesis I make the following contributions:

- Design and evaluation of an on-chip RH coprocessor interface that can access the processor's cache-coherent memory hierarchy using virtual memory addresses.
- Implementation of a mechanism allowing applications to directly access and use configured RH kernels without OS intervention or violating process isolation.
- Reduction of RH kernel overhead such that an application can efficiently query, execute, and poll RH kernels, which allows short-running kernels (executing under 100 processor cycles) to still obtain speedups.
- Determined that:
  - RH coprocessors and general-purpose processors should share at least one level of the cache hierarchy to facilitate the transfer of data between the two.
  - An L1 cache attached to the RH coprocessor does not greatly improve coprocessor performance, but does significantly reduce the dynamic energy consumption of the RH's memory accesses.
  - Multicore RH computing systems need a hardware TLB miss handler to improve memory performance.
  - A shared memory interface between the RH coprocessor and the cache hierarchy is scalable to at least eight cores, with eight-core workloads obtaining $\sim 95\%$ of their performance when using a zero-latency RH coprocessor memory.
- Implemention of a mechanism to prevent RH kernels whose memory request exhibit streaming behavior, and have little cache locality, from polluting shared cache levels. This improves performance of coscheduled SW-only threads by up to 32%.
- Creation of SMT processor extensions that allow RH kernels to execute at almost full speed while allowing coscheduled threads to execute at $\sim 95\%$ of their single-core performance when coscheduled alongside an application that continuously executes long-running RH kernels.
- Creation of methods for sharing configured RH kernels between multiple applications to allow systems with limited RH resources to better allocate the RH fabric. Eight-core systems that shared RH kernels were 97.4% as fast as systems with unlimited hardware, while using only one-eighth of the RH fabric resources.
- Modified a reconfigurable hardware kernel scheduling algorithm to better allocate kernels when reconfigurable resources are constrained. This scheduler always performs as good, or better than scheduling algorithms created for related projects when used on multicore processors.

1.2 Thesis Organization

My thesis can be broken down into roughly three parts. The first part of the thesis focuses on my system model, and how hybrid RH/software applications perform on single processor systems. In Chapter 2 I examine some of the prior work that led to the research I performed in this thesis, including an overview of heterogeneous systems, how these systems can use RH, and prior RH systems. Chapter 3 describes the reconfigurable computing system proposed in this thesis. Chapter 4 examines the programming model I developed to allow communication between the OS, applications and RH kernels, as well as the methods I developed to eliminate much of the latency inherent in accessing hardware coprocessors on systems maintaining process-isolation. Chapter 5 examines the benchmarks and methodology used to evaluate the proposed platform. In Chapter 6 I examine the execution of RH kernels, comparing the memory access patterns of the RH kernels with the same kernels, when executing on traditional CPUs. I then apply this knowledge to develop communication mechanisms that allow RH kernels to transfer data with the general-purpose processors on a single core system.

The second part of this thesis focuses on the scalability and performance of the reconfigurable coprocessor in multicore and multithreaded systems. Chapter 7 examines the performance of this new multicore reconfigurable computing platform, and examines the performance when the OS coschedules software-only applications alongside hybrid RH/software applications. Chapter 8 examines the impact of coupling an RH coprocessor with a simultaneous multithreaded processor, focusing on the impact of coscheduling hybrid RH/software applications alongside software-only applications. In both of these chapters, I present platform extensions that improve overall system performance.

Next, this thesis examines how to efficiently allocate limited RH computing resources on multicore systems. In Chapter 9 I propose extensions to the system model that allow one more more copies of an RH kernel to be (safely) shared between simultaneously executing threads. I then compare the performance of systems using the sharing mechanism both with systems containing "sufficient" reconfigurable resources, as well as those with limited resources that do not permit sharing. In Chapter 10 I examine alternative methods for allocating RH resources, addressing shortcomings of prior scheduling algorithms used on multiprocessor reconfigurable systems. Finally, in Chapter 11 I illustrate the performance benefits of using a single RH fabric instead of a partitioned one.

The remaining chapters further analyze the results from this thesis. Chapter 12 compares the performance of the proposed reconfigurable computing system with that of a system containing a SIMD processor. Chapter 13 discusses the impact of this research on other coprocessor architectures and how techniques developed for this thesis can both increase the performance of other heterogeneous system-on-chip architectures, as well as how to adapt the proposed programming interface to these architectures. Finally, Chapter 14 summarizes the contributions made in this thesis.

Chapter 2

Background and Prior Work

Reconfigurable computing systems combine one or more general-purpose processors with a reconfigurable hardware (RH) fabric. Embedded systems designers use RH for many different purposes: interfacing processors with external hardware peripherals, creating dedicated coprocessors that accelerate a specific function or task, etc [30, 47]. This thesis examines systems that use RH to accelerate various tasks or functions.

2.1 Heterogeneous Computing

The idea of using heterogeneous compute resources in a computing platform is not new [75]. Computing systems always contained some heterogeneous resources; however, in the past these resources provided "distinct" functionality often associated with I/O (network cards, sound cards, graphics adapters, modems, etc). Modern embedded systems use heterogeneous processing elements much more extensively, often to accelerate computations [75].

Many previous heterogeneous embedded systems implemented a single application, or a specific set of applications, and had strict power and cost constraints. Because of this, system designers attempt to maximize application performance and minimize the system's energy consumption by tailoring the system to the algorithms within the application. In some of these systems, entire applications execute on custom hardware ASIC (Application Specific Integrated Circuit) coprocessors. Other heterogeneous systems different algorithms within an application or set of application to a mixture of digital signal processors, vector processors, general-purpose processors, and/or other specialized processors. By executing computations on the resource optimized for the computation, heterogeneous systems can obtain higher performance while consuming less energy.

Modern computing systems often use high-performance graphic processing units (GPUs) as a form of heterogeneous computing. GPUs are highly parallel devices that perform graphics calculations orders of magnitude faster than a general-purpose CPU. More recently, researchers have used GPUs to perform general-purpose computations [112, 31]. In these systems, programmers use new programming languages such as CUDA [31] and OpenCL [112] to design accelerators that use the SIMD (single instruction multiple data) processors within the GPU to accelerate different algorithms [100, 1].

Although the offloading of computation onto GPUs is an important example of heterogeneous computing, modern embedded systems (and future "general-purpose" systems are likely to) take advantage of many other types of heterogeneous resources. For instance, many smart phones and other high performance mobile devices contain tens of heterogeneous resources on them, including video encoder/decoders, support for various wireless standards (3G, 802.11, etc), mp3 and speech encoder/decoders, etc. In the past, general-purpose computing systems often relegated these heterogeneous coprocessors to I/O tasks, with most computation occurring on the CPU. However, general-purpose systems are also starting to use heterogeneous coprocessors as computational accelerators. More recently, researchers have examined the impact that the increasing number of transistors on a chip has on power consumption [38], and predict that future chips will not be able to power all of their transistors at the same time, resulting in "dark silicon". Single-chip heterogeneous systems help solve the problems of dark silicon by only powering CPU cores and/or compute accelerators needed to perform a given task. In my proposed system, RH kernels and/or CPUs are often idle, allowing them to easily be put into low-power modes.

2.2 Coprocessor Models

Cascaval et. al [20] described a taxonomy of different coprocessor accelerator architectures and their programming models, focusing on the latency of the different interconnection strategies they use, as well as how applications executing on the system interact with the coprocessor. This analysis examined architectures ranging from instruction set extensions (such as floating point units) that are directly integrated into a processor's microarchitecture, up to distributed coprocessors that communicate over a packet-switched network such as ethernet.

For example, ISA extensions requiring new functional units on a CPU usually communicate using the processor's register file or shared memory. They also have a very fine granularity, performing a single, or very few operations per call. Custom instructions often perform a very small and specific task, allowing compilers to more easily generate the appropriate instructions.

At the other extreme, a coprocessor communicating over a network, or by I/O interfaces such as PCI express or USB, operates very differently. In attached coprocessor systems, the application must directly request to use the coprocessor, and the coprocessor often runs asynchronously with respect to the host processor. On systems with protected memory, user applications cannot directly access the coprocessor, and the OS is responsible for setting up the hardware, transferring data, and starting the coprocessor's execution. The processor and the coprocessor usually have their own unique physical memory, requiring direct memory access (DMA) transfers to share data with one another. Because of these high setup costs, as well as the relatively large latency between a CPU and an off-chip coprocessor, short-running tasks cannot execute efficiently on the coprocessor, and these latencies can end up *decelerating* an application.

In addition to these two "coprocessor" models, the survey [20] examines a loosely coupled model where the coprocessor(s) use a shared (often cache-coherent) memory and may or may not be directly coupled into the programming

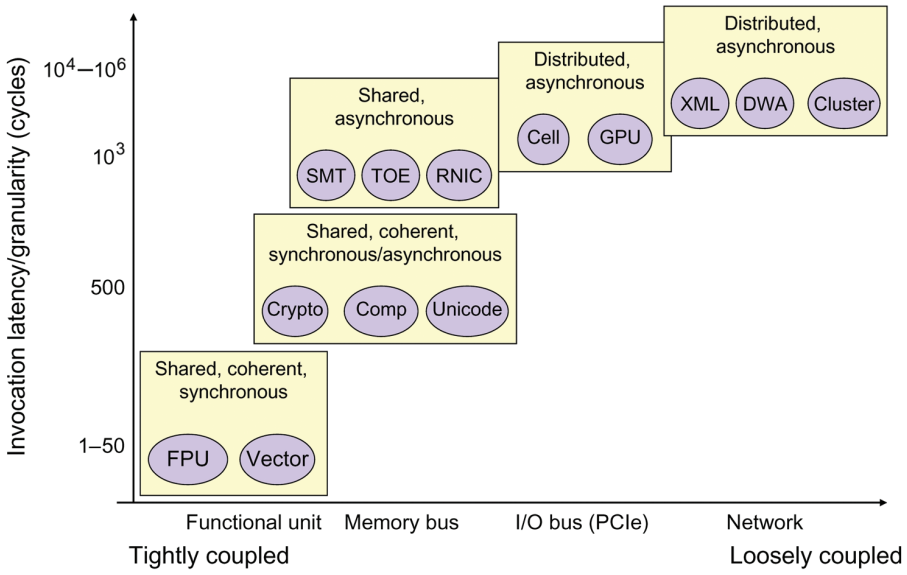


Figure 2.1 Figure from Cascaval et al. [20] illustrating the relationship between the invocation latency and the interface necessary for various types of accelerator architectures (boxes), and the ovals show examples of the different types of accelerators (Comp: compression; TOE: TCP offload engine; RNIC: RDMA network interface controller)

model. Examples of these devices can include on-die acceleration engines for cryptography, compression, and checksums. Figure 2.1 shows the latency and coprocessor interface of several coprocessor accelerators [20].

As more components can be integrated on the same chip, coprocessor communication latency decreases dramatically, altering some of the characteristics of the described coprocessor model. Systems-on-chip (such as those designed for smart phones and tablet computing [97]) fall into this category. In these platforms, communication between the CPU and coprocessor is faster than when the coprocessor is located off-chip, reducing operations that once took thousands of cycles to only tens of cycles – almost approaching the latency of a functional unit within a processor core. Recently, general-purpose processors have also started to resemble systems-on chips. In AMD's Fusion processors [18] and Intel's Sandy Bridge processors [67], the previously off-chip GPU is now located on the same die as the CPU.

To date, most single-chip heterogeneous architectures have taken a conservative approach to coupling accelerators with the CPU cores, relying on many of the same interfaces used for off-chip accelerators [111]. Although this method simplifies backward compatibility and reduces the communication latency between the coprocessor and processor, it is limited by the protocols established for off-chip accelerators. Accessing these on-chip coprocessors typically requires OS interventions. This often resulting in multiple memory copies to move data between address spaces. These operations can take thousands of cycles, despite the fact that the CPU and coprocessor can exchange data in the tens of cycles [96, 95, 111]. In this thesis I show that these overheads are often unnecessary, and RH kernels can be safely used by user-application with minimal overhead.

Stillwell et al. [111] provide an example of the data flow in a typical coprocessor architecture. In these systems, the OS controls the coprocessor for security purposes. When an application wishes to use an accelerator, it must first copy the data that the coprocessor will use into a buffer, and make an OS system call. Then the OS initiates a DMA transfer to the hardware coprocessor. However, memory pages resident in an application's virtual memory address space are not guaranteed to be available at all times, and the OS must ensure that all of the data transferred by the DMA engine is both resident in physical memory, and will not be swapped out while the hardware is executing. To do this, the OS copies the data from user space into a buffer that exists in kernel space [16, 111]. Alternatively, if a large amount of data must be transferred, the OS instead pins the relevant memory pages in physical memory. However, in this instance the OS must modify multiple data structures within the OS both before and after the DMA transfer, which also adds significant overhead. When an application initiates a task on a coprocessor, it can either continue performing other computations in parallel with the accelerator or block to the OS until the accelerator informs the OS that it is done.

In this traditional coprocessor model, the accelerated algorithm executing on the coprocessor is not "done" when it finishes its computation; it must still transfer the resultant data back to the processor's memory space using a DMA transfer. The destination memory is likely in the OS's address space, so when the DMA transfer is complete, the OS then has to copy this data back into the application's address space. All of these extra memory copies, as well as the OS system calls, impose a significant penalty on accelerator performance [96, 95, 111]. Although many of these tasks can be overlapped to reduce overall latency, this is really only possible for long-running accelerators (executing for millions of processor cycles) [20]. For this class of accelerator, the performance gains from moving an accelerator on-chip are relatively minimal – the latency between the coprocessor and processor, although large when compared to the processor's cycle time, is small in comparison to the total runtime of the coprocessor. Therefore, to maximize the usefulness and performance of on-chip coprocessors, system designers need to provide a new interface to use the coprocessor that eliminates much of the overhead inherent in traditional coprocessor communication.

2.3 Reconfigurable Hardware Overview

In general, RH fabrics can be categorized as either coarse-grained or fine-grained [30]. In fine-grained RH fabrics, such as field programmable gate arrays (FPGAs), operations on single-bit data are very efficient, while operations on larger blocks of data tend to be slower and consume more energy [30, 115, 3]. In contrast, coarse-grained fabrics operate primarily on word-sized blocks of data, and often cannot efficiently handle single-bit data operations [56].

Commercially available RH devices typically contain a fine-grained reconfigurable fabric in the form of an FPGA [30, 131, 5]. In these devices, the hardware is divided into configurable logic blocks containing one or more lookup tables (LUT) and flip-flops. The LUTs within each logic block can be reconfigured to implement any function of its inputs (there are usually between four and six inputs), and the LUT's outputs can optionally be fed through the block's flip-flops [115]. These configurable logic blocks are connected to a highly reconfigurable network that provides

communication between different logic blocks. FPGAs also incorporate more coarse-grained blocks into their network, such as block RAMs and multipliers interspersed throughout the RH fabric [131, 5]. These coarse-grained blocks use less area, less power, and operate at a faster clock frequency than would be achievable if these same structures had been implemented using the FPGA's logic blocks. More recent research has also looked into integrating floating point units into the fabric [11].

Coarse-grained RH architectures configure larger more complex computational structures such as entire ALUs [56, 77]. The lower flexibility of a limited set of operations that process word-sized data requires far fewer configuration bits compared to fine-grained architectures. Routing is also performed at a coarser granularity; however, because of this, coarse-grained architectures tend to be inefficient when dealing with single-bit control signals.

The work presented in this thesis does not rely on any particular form of RH, and instead focuses on how the RH communicates with the rest of the system. The RH fabric in this thesis is generic, and not tied to a specific RH architecture. Chapter 3.3 describes the RH fabric that I use to model the RH kernels simulated in this work.

2.4 Reconfigurable Computing

Although there are many types of heterogeneous coprocessors in use today, this thesis focuses on RH coprocessors. Unlike traditional coprocessor architectures, RH coprocessors adapt to the changing demands of the system, only implementing the currently-needed hardware. They can therefore accelerate applications that were not originally envisioned when the system was first designed.

A reconfigurable computing system combines general-purpose processor(s) with an RH fabric. Applications execute as software (SW) on the general-purpose processor(s) and use the RH to accelerate compute-intensive portions (kernels) of their execution. In this thesis, I refer to applications that use RH accelerators as hybrid RH/SW applications. These hybrid applications take advantage of the strengths of both the RH and general-purpose processor(s). The bulk of these applications (overall control flow, exceptional conditions, rare execution paths etc) are written as software, and only the most compute-intensive kernels are also implemented in RH. Hybrid RH/SW applications consume less RH area, and have lower development costs than applications designed to run entirely in RH. In the reconfigurable computing system proposed in this thesis, the OS selects whether an individual kernel executes on the RH or on the general-purpose processor [55]. Figure 2.2 shows an example of the execution of a portion of a hybrid RH/SW application containing three kernels. In this example, the three kernels are all accelerated to varying degrees, and the software's execution time remains constant.

Hybrid RH/SW applications often obtain similar performance to full-RH applications, but with the flexibility of software (more modes of operation, highly configurable execution). Additionally, only implementing smaller and less complex hardware modules in the RH reduces the development time of hybrid applications. This methodology also allows the OS more flexibility when selecting which kernels to configure on the RH (many smaller kernels versus fewer larger kernels).

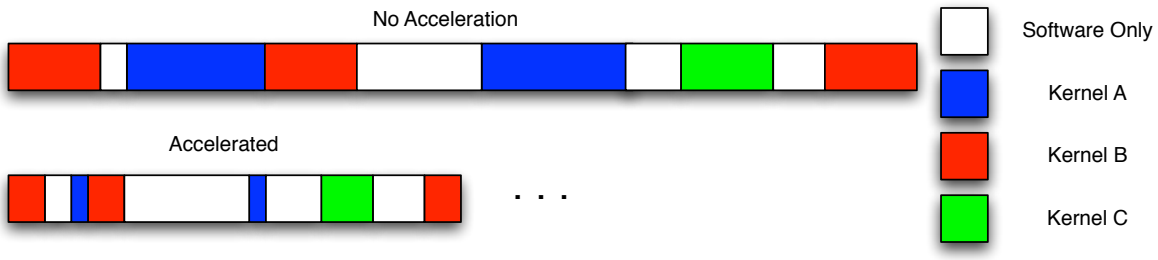


Figure 2.2 Example execution of a hybrid RH/SW application containing three RH kernels.

2.5 Existing Reconfigurable Computing Systems

Researchers have proposed many previous systems that coupled general-purpose processors with a RH fabric [133, 10, 130, 82, 79, 57, 51, 86, 7, 28]. There are two primary types of hybrid RH/SW systems: CPUs containing RH functional units, and CPUs with an attached RH coprocessor [30, 47]. A third type of system implements soft-core processors directly on the RH fabric [108]. Because these systems use the RH as either a coprocessor or as a functional unit, they can be classified accordingly. In this section I examine how processors communicate with the RH on existing reconfigurable computing systems.

Processors with an RH functional unit allow for the creation of custom instructions on the RH [133, 10, 130, 86, 79, 28]. These RH functional units are relatively easy to use in software, and communication tends to be "directly" handled through the processor's register file. Because, many of these devices do not allow the RH to directly access memory the processor is needed to manage the RH's input and output data [86, 130, 10, 133]. This limits the RH's I/O bandwidth, and complicates sharing the RH across multiple processor cores.

Other reconfigurable computing systems incorporate the RH as a peer of the processor(s). In these systems, the CPU often uses the system bus or a peripheral bus to communicate with the RH [82, 57, 51, 21, 120, 121, 77]. Because the system must write data out on a bus, latency between the processor and RH is much greater than in a reconfigurable functional unit approach, resulting in larger setup overheads. Although these RH coprocessors work well for long-latency coprocessor operations, they cannot efficiently accelerate kernels with short execution times due to this overhead [51]. The work in this thesis focuses on the creation of an RH coprocessor rather than using the RH in a functional unit.

In some RH coprocessor systems, the RH can directly access the processor's physical memory [82, 57, 51, 77]. In other systems, the RH accesses a private physical memory, or must communicate through buffers attached to he RH [21, 120, 121]. In both cases, a software routine produces and copies data into the RH's address space, and then must consume the RH's output data. In some systems, buffers in the RH's address space can be used in a hardware pipeline to allow multiple RH kernels to share data independently of the SW. However, SW routines often provides data to

the first hardware module and receive results from the last module in the pipeline. The overhead associated with SW routines producing and consuming these buffers can both slow down the system and increase its energy consumption.

Synchronizing the host processor(s) with the RH kernel(s) is also a difficult challenge. Although it is relatively simple to stall the host processor when executing an RH kernel, designing a system (and applications) that allow hardware and software to execute concurrently is more difficult [21, 7]. The SCORE system [21] facilitates such synchronization by using built in buffers that automatically stall both hardware kernels and software threads when their input buffer is empty, or their output buffer is full. Other work focuses on a more traditional thread-based synchronization model. The hthreads system [7] adds hardware-level support for using many of the synchronization primitives in the software pthreads library from a hardware thread, allowing hardware threads to communicate with SW threads and/or other hardware threads.

Several systems that share the processor bus also allow the RH to initiate DMA transfers, alleviating some of the burden from the main processor [82, 111]. One major drawback with DMA-like communication paradigms is that they either require OS intervention, or expose the processor's physical memory directly to the RH kernels. Although these approaches can work on specialized systems, general-purpose systems require more flexibility. Prior research has suggested that the OS should facilitate hardware-software communication using dedicated hardware structures [127, 120, 40] or through a form of message passing [93]. Chapter 2.7.1 examines these methods in more detail.

2.6 Dynamic Reconfiguration

One of the biggest advantages RH has over traditional ASICs is that it can be dynamically reconfigured to implement new functionality. This has been used extensively to allow systems to adapt not only to changing standards (implementing RH accelerators for functionality unknown when the chip was originally fabricated), but can also be used to reconfigure the hardware based on runtime behavior of the application(s) currently executing on the system.

Many commercially-available RH devices such as field programmable gate arrays (FPGAs) have been designed primarily as ASIC replacements. These devices are often used for hardware designs that were not economical to implement as custom ASICs, as well as prototyping new hardware designs. Many of these systems are statically configured when powered on, only "updating" the hardware to fix potential bugs or add new functionality. Although these "static" reconfigurable devices have gained significant traction in the market, they do not fully exploit the reconfigurable nature of the device.

Because of this, many FPGAs were difficult to efficiently use in a dynamically reconfigurable system. Many of these devices have little or no support for "partial" reconfiguration. Therefore if the system wants to reconfigure any of the hardware on the device, it must first stop all execution on the RH, and reconfigure the entire device. This method of reconfiguration can greatly slow down the the entire system if it only needs to swap out a single kernel on the device.

Although commercial support for dynamically reconfigurable devices has traditionally been minimal, many different research groups have proposed partially reconfigurable systems [57, 51, 133, 35]. Some of these systems examined

the use of "relocatable" hardware blocks [29, 123, 73]. This allows previously place and routed hardware modules to be configured in different positions on the device. On these systems, only a single configuration of the RH kernel is needed, rather than having separate configurations of the RH kernel for every possible placement of the device. More recently, commercial FPGA vendors started adding support for partial reconfiguration [131, 4], allowing portions of the FPGA to be reconfigured while other portions of the RH operate.

2.7 Operating System Support

2.7.1 OS Support For Communication

Wigley et al. [127] designed an early operating system (OS) for RH processors. This work suggests that the OS should provide an interface for communicating between application components. It also stated that this interface needs to exist in hardware for performance reasons. However, the implementation of such a feature was left for future work. Nollet et al. [93], provided OS support for using message passing between the processor and RH. The hardware determined the exact implementation of the message passing, however they did not allow for the RH to directly communicate with main memory.

Lubbers et al. [83] examined how an RH kernel could directly access portions of the OS' API. Similarly, the HybridOS [73] system provided support for applications to use the OS' APIs. In addition, this work examined the tradeoffs involved in using the OS to access the RH kernels. Similar to the HiPPAI work [111], they found that the overhead of the OS, and the data transfers required when using it significantly slowed down the operation of RH kernels.

some research has also examined how to allow RH coprocessors to access virtual memory. Vuletic et al. extended the Altera Excalibur rSoC [6] platform to support RH virtual memory access [121, 120]. In this work, the RH issued memory requests to the local address space of its parent process. A window management unit (WMU) cached multiple pages of the processor's memory in a memory buffer attached to the RH . If the RH issued a memory request to an address that is not cached in this buffer, the WMU copied the entire page of data from memory into the buffer, facilitating a buffer communication scheme similar to that used in the SCORE system [21].

While using virtual memory windows allow for a more transparent buffer management than that used in the SCORE system [21], they still do not efficiently support random access to main memory due to the large costs associated with copying entire pages of memory. This coarse grain (page-level) sharing of memory between processors and reconfigurable units simplifies the interface design, but results in larger memory latencies for applications that access small amounts of memory from different pages.

Like the work from Vuletic et al. [120, 121], my work allows the RH to directly access virtual memory addresses, but at a cache-line granularity. Other recent research also allows coprocessors cache-line level access to virtual memory. The HiPPAI project [111] added support to the RH coprocessor so that it could translate virtual memory addresses; however, this aspect of my work predated that project [43].

2.7.2 OS Support for Dynamic Reconfiguration

There are many different ways that a system can allocate the RH fabric amongst RH kernels. In Resano et al. [103] applications issued "hints" so that the system could prefetch RH kernels before using them. This method relies on the programmer knowing which kernels will be used next and knowing how much RH is available on the system. Additionally, it does not scale well to systems where multiple processors share an RH fabric. In my work, the OS is responsible for selecting which RH kernels should be configured on the RH.

My work uses the RH scheduler designed in Fu et al. [39, 42, 41, 40]. This scheduler periodically evaluates which RH kernels should be loaded in the following time interval by examining applications recent past behavior [42]. This scheduler assumes that application behavior in the next interval will be similar to the previous one, and therefore attempts to select the kernels that would have provided the greatest benefit during the previous scheduling interval. When intervals are short (on the order of milliseconds), application behavior for the upcoming interval is likely to be similar to its behavior in the previous interval due to the stability of program phases [42].

This scheduler dynamically profiles the execution usage of all kernels, and uses a knapsack-based algorithm to determine which kernels would provide the most value by being configured on the RH (knapsack) [72]. The scheduler uses an exact knapsack solver because the problem size is sufficiently small that it does not create significant overhead. Additionally, due to the constraints of the problem, the exact solution can be solved in pseudo-polynomial time. In this algorithm, the cost of a kernel is equal to the amount of RH that it occupies. Fu et al. [40] examined several functions to calculate the "value" of a kernel. At the time, the best-performing value function measured the application speedup that a kernel would provide, assuming all other kernels executed in software. This scheduler used the number of times software called a kernel, its execution time in software, and its execution time in RH to estimate the application speedup that would be obtained when using a given RH kernel. In Chapter 10, I propose a new RH kernel scheduler optimized for multicore reconfigurable computing platforms.

2.8 Summary

This chapter examined many of the previous reconfigurable computing systems, and the methods that the general-purpose processor(s) use to communicate with the RH and vice versa. The works covered in this section provide the background needed to understand how the design decisions made for the reconfigurable computing systems proposed in this thesis. Many of the chapters in this thesis will perform a more detailed examination of prior work relevant to the experiments examined in the chapter.

Chapter 3

System Model

This chapter describes a new RH coprocessor architecture, and the basics of how it operates. Section 3.1 gives a brief overview of the system. Section 3.2 describes the design of the general-purpose processor(s) used in this system. Section 3.3 describes the organization of the RH coprocessor. Next, this chapter describes the two ways that the CPU(s) and RH coprocessor communicate. Section 3.4 describes how the RH coprocessor accesses the shared memory hierarchy, and Section 3.5 describes how CPU(s) can directly load/store values to the RH coprocessor. Section 3.6 describes the simulation platform used to develop and test the reconfigurable computing system. Finally, Section 3.7 discusses some of the shortcomings of the proposed reconfigurable computing model, and examines future work that could be done to better refine the system.

3.1 System Overview

In this thesis, I propose a reconfigurable computing system that directly couples one or more processor cores with a reconfigurable fabric. Figure 3.1 illustrates how two general-purpose processors can share a RH fabric in my proposed system.

In this model of reconfigurable computing, the RH acts as a coprocessor. However, unlike many traditional coprocessors, the RH interconnects with the processor using a low-latency interconnect, and user-applications can directly communicate with the coprocessor. This new platform shares properties of coprocessors that communicate via the memory bus, as well as those of functional units within a CPU core.

Most communication between the RH and cpus is done using a shared memory hierarchy. This memory is coherent between the CPU(s) and RH, however RH kernels executing on the RH coprocessor can only access data in their virtual memory address space, preventing them from directly accessing physical memory. In addition to this shared virtually-addressable memory, RH kernels have a limited direct communication with the processor cores allowing kernels to query and setup RH kernels (amongst other functions).

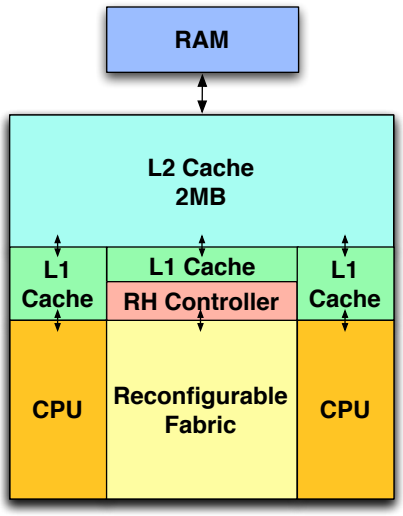


Figure 3.1 High-level model of a chip containing two general-purpose CPU cores, and a shared RH fabric.

3.2 Processor Model

The microprocessor design used in my system are similar to those currently in high end embedded systems such as the ARM Cortex A9, or the Octeon MIPS64 core [9, 22] which can both contain multiple cache-coherent processor cores. Due to the constraints of the simulation platform (see Chapter 3.6), I simulate an UltraSparc processor core instead of an ARM or MIPS architecture more commonly found in embedded systems. However, the simulated processor(s) perform similarly to the referenced high end embedded processors.

Table 3.1 shows the configuration parameters of the simulated UltraSparc processor cores. All of the simulations performed for this thesis use this model, unless otherwise specified. The systems had enough DDR2 RAM to contain the working sets of all the executing applications, avoiding swapping memory out to disk. The L2 cache uses a point to point network to connect it to the CPU cores, as well as the RH controller. This means every core (as well as the RH controller) has a fixed L2 latency, regardless of the number of cores in the system.

3.3 Reconfigurable Coprocessor

The RH coprocessor consists of two primary components: a set of RH tiles that can be configured independently, and an RH controller that allows the RH tiles to communicate with the rest of the system. RH kernels occupy one or more RH tiles. Each RH tiles can be configured independently, however the system may only configure one tile at a time (all other requests are queued). The tiles are interchangeable such that all of the tiles in the system appear the same; therefore a placed and routed circuit can be configured to occupy any set of RH tiles in the system. RH kernels configured on the tiles use the concept of “virtual hardware”, which allows the OS to select whether a kernel

Frequency	900 MHz
L1 D/I-Cache size	32 KB
L1 Associativity	4-way
L1 Latency	2 cycles
Unified L2 cache size	2 MB
L2 Associativity	16-way
L2 Latency	∼16 cycles
Cache line size	64 bytes
Memory Model	450MHz DDR 2
Issue Width	2
Instruction Window Entries	32

Table 3.1 Simulated Processor Configuration.

is executed on the RH coprocessor, or executes in SW [17]. The RH tiles are shared amongst all of the processor's in the system, and the advantages of sharing a RH fabric are fully elaborated in Section 11.

The work in this thesis, does not concern itself with the detailed design of the actual RH fabric, and instead focuses on how multiple processor cores can best use the shared RH fabric. Because of this, the architecture of an RH tile is not limited to just FPGA-like hardware [30, 47], but could alternatively be elements of GPUs [100, 25], custom arrays of processors [19], vector processors, or any other type of "configurable" heterogeneous resources. In this thesis, I modeled the RH coprocessor as an FPGA-like device to obtain performance/area estimates,

I modeled the RH tiles on the fabric found on the Xilinx Virtex-5 FPGA. Each tile contains 256 Virtex-5 slices, one Virtex-5 DSP unit and two Virtex-5 block RAMs. These resources correspond to roughly 1% of the Xilinx Virtex-5 LX155 FPGA [131]. We synthesized, and place and routed each RH kernel on the Virtex-5 device to obtain both the maximum clock frequency that it can run at, as well as the number of tiles that it occupied.

Figure 3.2 shows how the systems may select to configure three RH kernels simultaneously on the RH coprocessor. Although the kernels in the figure are "contiguous" on the RH fabric, this is done for clarity only, and the system imposes no restrictions on the placement of RH kernels on the RH tiles. Each tile has its own independent clock, allowing RH kernels with different clock rates to execute simultaneously. A series of FIFOs that can operate across multiple clock domains [8] handles synchronization between the RH kernels and the rest of the system. Section 3.4 describes the FIFOs in more detail. Although we do not consider all of the potential issues involved with the independent placement of kernels, or the organization of the FIFOs used to communicate between the RH kernels and the rest of the system, Section 3.7 examines these problems in more detail, and proposes potential solutions to the problems.

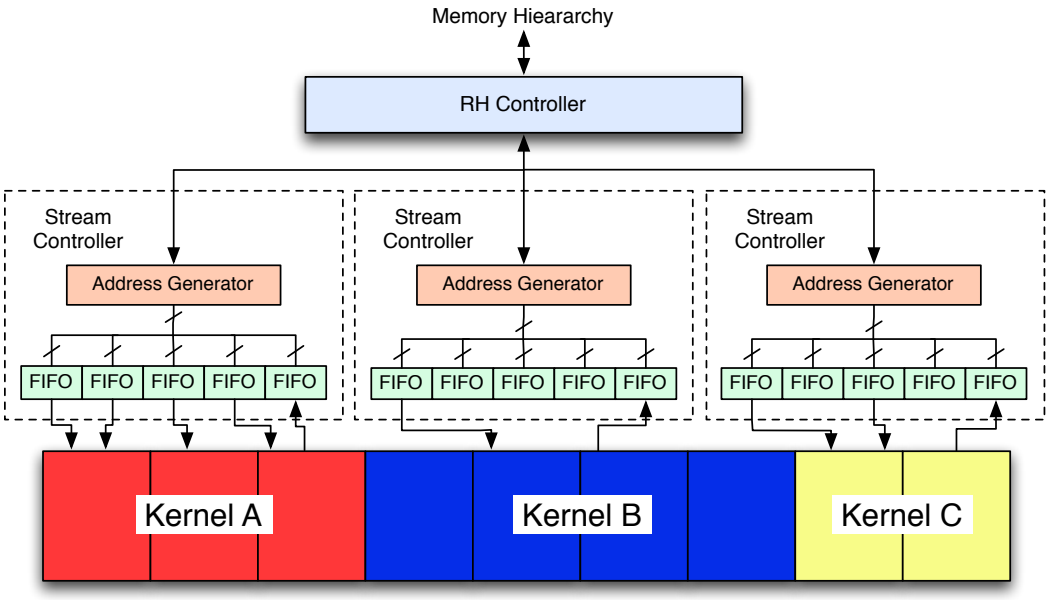


Figure 3.2 Interconnection of the memory hierarchy, stream controllers, and the configured RH kernels on a shared RH fabric.

3.4 Shared Memory Communication

Communication between the RH kernels and the processor core(s) occurs primarily through a cache-coherent shared memory hierarchy using a MOESI protocol. This model ensures that all coherent nodes see the same data in memory [59]. Most contemporary multiprocessor systems communicate using a shared memory. Although alternative communication schemes can get buy using only message passing, on a multicore platform, this limits the flexibility of the system. Cache coherent systems can easily and efficiently implement message passing APIs, but the reverse is not true [58, 70].

In addition to supporting shared memory, all of the memory that the RH kernels can access is virtually addressable. Because of this, programmers doe not have to explicitly transfer data into dedicated buffers, or worry about whether data is contiguous in physical memory. Using virtual memory provides process isolation because RH kernels cannot directly access physical memory, preventing kernels from accessing data owned by another process or the OS. Most modern operating systems for both high-end embedded systems, and desktop computing systems require process isolation. Although other methods exist for providing process isolation, virtual memory is the most common method in modern systems.

An RH controller facilitates all communication between the RH kernels and the CPUs or the memory hierarchy. RH kernels executing on the RH fabric use the RH controller to issue memory requests to the virtually-addressable,

cache-coherent memory subsystem. In most of the tested systems, the RH controller communicates directly with its own private L1 cache, however the RH controller can interface with the memory hierarchy at any point.

Kernels on our system use a "streaming" model of computation, similar to the model proposed in the SCORE system[21]; however the streams read/write to their parent process' virtual address space, and not directly to other kernels or scratchpad memories. RH kernels communicate with the RH controller using a stream controller that generates memory addresses, and buffers data for each "stream" of data that the kernel requires. Figure 3.2 shows how the stream controllers interact with the rest of the system. The stream controllers were designed for reading windows of data, representing one- or two-dimensional arrays of data in memory. The stream controller's address generators are based on the ones in the MoM-2 system [61], where they proved to be useful for many different streaming and media processing applications. The current set of possible address generator parameters has been sufficient for all of the existing benchmarks; if future kernels require more complex addressing patterns, modifications would need to be made to the design of the address generator [61, 52, 81, 53]. Application software is responsible for "programming" the stream controller's address generation units. Chapter 4.2 provides details on how applications both configure, and use these stream controller parameters.

Each tile in the system contains a single stream controller capable of loading/storing data from one of five different input/output streams (five FIFOS). The stream controller's address generation logic is time-multiplexed with each of its five attached FIFOs; only one of the FIFOs can issue a request to the RH controller per (processor) clock cycle. Each request can be up to 32 bytes in length, and does not have to be word-aligned. Each stream can have up to eight entries queued within it (eight-deep FIFO). Not all requests can be serviced immediately. The RH controller can service up to two memory requests per cycle, however each stream controller can only issue a single memory request per cycles. Because the RH kernels execute at a slower clock frequency than the stream controllers; a stream controller can issue memory requests from multiple streams during a single *kernel* clock cycle, limiting the advantage of serving more requests per cycle. Section 3.7.1 discusses some of the potential problems with the stream controller organization, and suggests possible future work to better optimize the hardware.

Figure 3.2 illustrates how the RH kernels communicate with the rest of the system. In this example, there are three kernels configured on the RH fabric, using a total of nine tiles. A system with nine tiles has nine available stream controllers; for simplicity only the three that are needed (one per kernel) are shown. Kernel A has four input streams and one output stream, Kernel B has a single input and single output stream, and Kernel C contains two input streams and a single output stream. Because each stream controller can service up to five streams, only one stream controller is needed per kernel. During each processor clock-cycle, the RH controller can start processing up to two memory requests from the various stream controllers. A round-robin scheduler selects which RH kernels (amongst those that are active) is allowed to issue a request. This method ensures that all kernels have "equal" access to memory. The RH controller can queue up to sixteen outstanding memory requests at a time. When the queue is full, RH kernels cannot issue memory requests until at least one pending request completes.

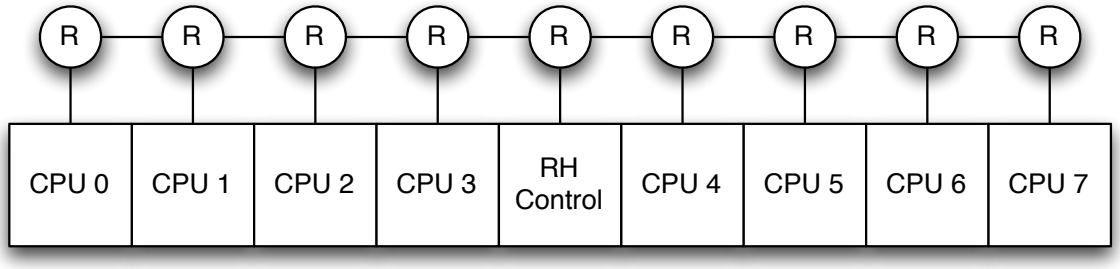


Figure 3.3 Illustration of the ring-based network used for direct communication between the RH controller and the CPUs on an 8 processor system, in this figure, each R corresponds to a router capable of forwarding packets to all of its neighbors

The RH controller also performs additional operations on data before forwarding the request to the cache. First, the controller translates the virtual memory address into a physical memory address. Chapter 4 outlines how these translations are performed. The controller then determines if the request straddles a cache line. If it does, a cache alignment unit splits the request into multiple load or store operations that it will issue to the cache. When a cache request is serviced and returned to the RH controller, the cache alignment unit combines data from multiple loads (if necessary) into a single data unit, aligns the data, and sends it back to the appropriate stream buffer.

3.5 Direct Communication

The previously-described shared memory access mechanism for CPU-RH communication works well for long-latency data transfers initiated by the RH kernels, but cannot efficiently handle all of the communication between the CPU and the RH kernels. Therefore I added support for an additional communication mechanism that uses memory-mapped I/O; this I/O space allows applications to directly issue commands and submit queries to an RH kernel. This form of communications is primarily used to initialize RH kernels, query if a kernel is present on the RH fabric, and to determine if an RH kernel has finished executing.

A dedicated on-chip ring network handles the direct communication between the CPUs and the RH kernels [14]. Figure 3.3 illustrates what this network looks like. Each link on the network takes one cycle to traverse, and each packet spends at least one cycle at each router to forward the packet (the routers are designated by an R). Each link on the network is bidirectional, and only one packet can traverse down each link (in each direction) at a time. To simplify the design of the network, each router had unlimited buffers; however in practice this network is very lightly loaded, and buffering along the links is rarely used. Each node on the network can only send out a single packet each cycle. Likewise, each node can only receive a single packet every cycle. This effectively limits the rate that the RH controller can respond to requests from the processor cores.

All loads and stores issued from a CPU to the RH controller's address space are sent over this network. This model, assumes that stores to the RH controller require only one directional traffic, and loads to the RH controller must traverse the links to the RH controller, process the data for a cycle, and then send the resultant data back across the network.

In addition to allowing the applications to query the status of RH kernels, this network can also be used to read back a single-word value from an RH kernel that has finished its execution. Chapter 6.6 examines the impact that direct access to RH kernels has on the execution time of RH kernels, and describes how this is implemented.

Although using such a network might be overkill for communication on one and two-processor systems (where a point-to-point network would be simpler and provide better performance), on four- and eight-processor systems some form of network would likely be necessary. Additionally, on real systems, it is likely the this network would be combined with the existing cache coherency networks, making better use of limited resources. However the implementation and analysis of such a system is left for future work.

3.6 Simulation Platform

To perform the experiments in this thesis, I developed Alexandrite, a custom full-system simulation platform that models the proposed reconfigurable computing system. Some of the initial infrastructure was provided by Fu et al. [39]. This early platform concentrated on measuring the impact of scheduling RH kernels on a single-core processor. Because of this, the modeling of the RH kernels, and the communication between the RH kernels and the rest of the system was of a secondary importance. I have greatly expanded upon this infrastructure, concentrating on providing a more detailed memory interface that models the interactions that a multi-core system might have with a reconfigurable coprocessor.

I initially developed Alexandrite to model a RH coprocessor on a single-core platform [43, 45], but later expanded upon it to model multicore systems [44, 46]. This simulation platform models the performance of single chip systems containing one or more UltraSparc processors, the reconfigurable hardware, the RH kernel's access to main memory, and the direct communication between the CPU(s) and the RH controller.

Alexandrite uses the Simics [84] platform to provide functional accuracy of the host processor(s). GEMS 2.1 [87] provides Simics with extensions that model the timing of the out-of-order superscalar processor(s) (Opal) as well as the cache-coherent memory hierarchy (Ruby). The simulator used the Alexandrite module in conjunction with GEMS to simulate the behavior of the processor cores, cache-coherent memory hierarch, RH controller, stream controllers, and the configured RH kernels.

Alexandrite models the timing of the RH kernels by using Verilator [110], which translates the Verilog kernels into steppable C++ modules. Verilated modules functionally simulate the kernel faster than would be possible using event-based simulation of Verilog code. The Verilated modules includes timing hooks, allowing the simulator to step the hardware through its operation. Verilated modules also include communication hooks that interface with the modeled

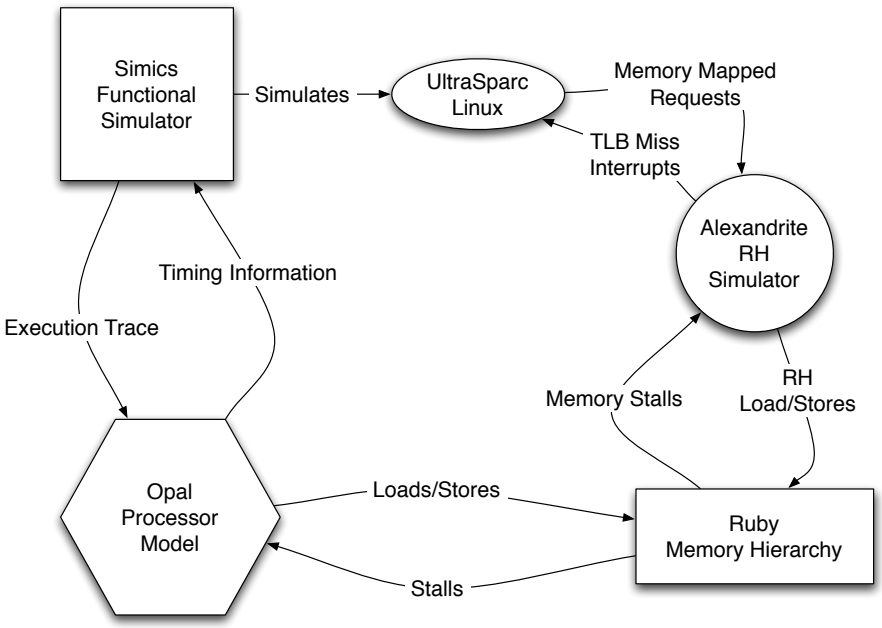


Figure 3.4 Block diagram of the Alexandrite reconfigurable computing simulator.

stream controllers, providing data to the RH kernels. Alexandrite issues these memory requests to Ruby's timing-accurate cache-coherent memory model. An overview of the execution flow of the simulator can be seen in Figure 3.4.

Because clock frequencies of RH kernels are generally slower than the clock frequency of the host processor(s), Alexandrite ticks the RH kernel's clock forward less frequently than the clock of the host processor(s). For example, if the RH kernel executes at 300MHz, and the host processor executes at 900MHz, Alexandrite will tick the RH kernel once every three CPU cycles. This allows Alexandrite to accurately model the relative execution speeds of the RH and software. The exact clock multiplier depends on the achievable clock frequency of the synthesized RH kernel.

3.7 System Limitations

Not all of the problems encountered when creating a platform from scratch were able to be adequately solved in my research. Two unsolved problems remain that need to be addressed for this system to be implementable. The first is the design and placement of the stream controllers that transmit data between the RH kernels and the RH controller, and the second, related problem, involves the placement of RH kernels that span multiple RH tiles. This section examines these problems in depth, proposing multiple potential solutions for each.

3.7.1 Stream Controller Allocation

In the current system, each stream controller contains an address generator, and five FIFOs that that contain eight entries of up to 32-bytes in size. Each RH tile in the system contains a single stream controller, simplifying the interconnection between a tile and the stream controller.

However directly mapping a stream controller to each RH tile results in an inefficient allocation of resources. All of the RH kernels used in this work require a very limited number of input and output streams. Most only required one or two input streams, and a single output stream (only one of the RH kernels required more than two input streams). These requirements are unrelated to the size (in tiles) of the RH kernel in question; a kernel occupying 13 tiles is likely to have the same number of input and output streams as a kernel occupying a single tile. In these situations, many of the stream controller resources go unused, resulting in wasted resources. Additionally, although a handful of RH kernels use streams with 32-byte data (the maximum data size of the FIFOs), most require only a fraction of this, many only accessing eight bytes of data at a time. These factors result in a large portion of the FIFO's SRAM bits going unused. This system would also require more I/O "pins" between an RH tile and its associated stream controller than could realistically be placed on the chip. The current system would require 1280 data "pins" to connect each stream controller with its associated RH tile, despite the fact that none of the examined RH kernels use more than 512 data "pins" (and this is for a kernel that occupies five RH tiles). The most "pins" used by an RH kernel that occupies only a single RH tile is 320, therefore, the vast majority of data "pins" go unused, resulting in a large amount of wasted resources.

In a real system, a more efficient allocation of stream controllers and data wires would be required. The current system assumes that the stream controllers' FIFOs are implemented using "hard" logic (non-reconfigurable), however it might be better to use the block RAMs associated with each RH tile to implement the FIFOs (these block RAMs would require some modification to support asynchronous access required for data to travel between both the RH controller's and the RH tile's clock domains [8]). In such a system, a single "hard" address generator could be associated with each RH tile or each tile could use a "soft" address generator that uses resources on the RH (such as the DSP block). In this scenario, a limited number of "pins" would be available to connect the RH controller to the RH tile. Logic to select which "stream" the data is from could either be provided on separate "pins", or each stream could have its own unique set of data "pins".

Another potential optimization to the stream controllers would be the creation of a "reconfigurable" FIFO. Many kernels do not require a FIFO that operates on 32-byte data entries. Instead, smaller FIFOs could be linked together to create a single larger FIFO. This optimization could help deal with the wasted space in each FIFO, but wouldn't address the problem of unused stream controllers associated with each RH tile.

Although these solutions can make for a more efficient stream controller, they still have their own forms of waste. Another potential solution would be to have a limited pool of stream controllers that can be used at a given time. When an RH kernel is initialized it would have to specify not only how many tiles it requires, but also how many stream controller resources it requires. Later, when the OS determines which RH kernels should be loaded onto the RH fabric, it would have to take into consideration both the number of tiles, and the number of stream controller resources required by an RH kernel. The OS could also be unaware of the stream controller allocation, and instead a static pool of stream controllers could be available for use by all of the RH kernels that are currently configured. The RH controller could dynamically allocate these stream controllers to an RH kernel when it is first requested. If an RH kernel is requested when there aren't enough stream controllers available to support it, the system would either have to delay execution of the RH kernel, or execute a software alternative. Because many RH kernels are only executing a fraction of the time they are configured (See Chapter 5.1), this could result in a better allocation of resources, although it would likely come with its own share of problems.

Much future work is needed to investigate how to allocate the stream controllers to the RH kernels. The work in this thesis, focuses on the viability of the platform as a whole, and so it has ignored some of the implementation details involved in the stream controller's allocation, relying instead on a simplistic model that is able to estimate the performance of the RH kernels, and their associated applications.

3.7.2 RH Kernel Placement

In the current system, RH kernels can occupy any set of tiles on the RH fabric. However, implementing this on a system would require a crossbar to interconnect all of the tiles on the system. Although this is feasible for systems

with a small number of tiles, it would likely be infeasible for larger systems. Better ways to partition RH kernels across multiple tiles and place them on the tiles are therefore needed.

Although much research has targeted the placement, routing, and packing of separate circuits on an FPGA, this is often done as an offline analysis [126]. Real-time reconfiguration of devices is currently in its infancy. Newer FPGAs have tools to allow for the reconfiguration of portions of the chips, but these tools often require separate place and routes for every position on the chip that a circuit might be located [126, 125, 123], although other research has examined the issues involved with moving pre-placed and routed circuits to a different portion of a reconfigurable device [91, 54].

Unlike typical FPGA systems, the RH computing system described in this thesis is partitioned into multiple tiles that can be individually configured. This is similar to prior work that partitioned single circuits across multiple FPGAs [99]. A similar procedure could be done for on chip tiles, however this would complicate the tool flow, as partitioning must be considered. Additionally, it is not currently known how much bandwidth would be required between the RH tiles on the chip, or how best to organize them [122].

One potential solution requires that the tiles are laid out in a one-dimensional array, and each RH kernel occupies a contiguous set of tiles. However, depending on how many tiles are available on the system, and the average number of tiles each RH kernel occupies, this could result in fragmentation of the RH fabric, and require the OS to "move" RH kernels that are currently configured to make room for more RH kernels. Depending on the system design, moving an RH kernel could require a complete reconfiguration of the RH kernel, making the RH kernel temporarily inaccessible. Obtaining the best performance in such a system might require a more advanced OS scheduling algorithm that takes into account the current placement of kernels, and the number of tiles that would be displaced when deciding on a new scheduler allocation.

Another potential solution to this new scheduling problem is to use a thresholding algorithm to prevent the OS from reconfiguring the RH fabric if a new allocation of kernels does not obtain a large enough speedup over the current allocation. The scheduler could then periodically perform a "global" schedule that wouldn't use thresholding to select the best RH kernel selection/placement. Such a thresholding scheme would likely improve performance on the current system, however if the system required RH kernels to be placed in adjacent tiles, the threshold value would need to be larger to avoid thrashing of the RH fabric. Although this would present an imperfect solution, depending on the workload, such a system might perform similar to the current system.

Evaluating the best way to handle the communication between tiles and the placement of RH kernels on the tiles require extensive future work. However, these problems are not insurmountable. Much of the work presented in this thesis was done to first justify the use of a RH computing system, and also show that the memory and system arrangement is feasible for future multicore systems. Without this justification being in place, there is little incentive to examine the best ways to handle inter-tile communication, and the placement of RH kernels on the tiles.

Chapter 4

Application Programming Interface

The use of a reconfigurable hardware (RH) coprocessor requires a rich interface to enable communication between the user applications, the RH kernels, the RH controller, and the operating system (OS). This chapter describes the communication mechanisms used to communicate between the RH and processor(s), discusses the security issues involved in the communication, and discusses the mechanisms used to ensure that contiguous packets are received int he order that they are sent.

Access to physically configured RH kernels is abstracted using the concept of "virtual" RH kernels [32, 40, 39]. Virtual/physical RH kernels are similar to the usage of virtual memory, however, unlike virtual memory, when a requested kernel is not configured on the RH fabric, the software (SW) can instead elect to execute a SW-only version of the RH kernel. This avoids the long latencies associated with reconfiguring the device. When the OS configures a physical RH kernel, it updates the associated virtual kernel so that it maps to the configured physical kernel. Section 4.3 describes how the OS configures a physical RH kernel, and performs the mapping of virtual to physical kernels. In Chapter 9 a single physical kernel can be shared by multiple virtual RH kernels. Therefore the mapping of physical to virtual RH kernels is not always one to one. The RH controller will then map these actions onto the appropriate physical kernel (if applicable).

All communication between a user application and the RH fabric takes place using virtual RH kernels. This limits the application's access to the physical hardware while still allowing user applications to obtain the status of an RH kernel, setup the RH kernel, tell it to execute, and determine if the physical RH kernel is still executing. In contrast, the OS is given full access to all of the virtual and physical RH kernels, allowing trusted access to the physical hardware.

4.1 Memory-Mapped I/O Segments

The CPU(s) use memory-mapped I/O to access the RH controller, as well as the actual RH kernels. Memory-mapped I/O is commonly used to communicate with devices, however this communication is normally done through the OS. In this thesis, I use memory-maps to allow user applications "direct" access to RH kernels that they have requested. This is similar to the work done in the SHRIMP project [15, 34]. Multiple memory segments are defined in this work to allow for process separation. Separating the memory into segments allows the OS to perform actions on

RH kernels that cannot be performed by individual applications. This means user application can only access a subset of operations that the RH controller can handle. Additionally, these memory segments mean that an application can only access virtual RH kernels assigned to it.

At least two different memory segments are needed to support both physical and virtual RH kernels. However, in this work, it was found that using four separate memory segments allows for a much cleaner application programming interface (API). These memory segments are the global segment, physical segment, virtual segment, and the virtual OS (VOS) segment.

Each of these segments are of different sizes, depending on how many entries are required. However, because the page size on the UltraSparc platform is 8KB, the memory segments are divided on 8KB boundaries. Although this might seem larger than necessary (when taking into consideration the number of required entries), these memory segments only correspond to a portion of the processor's address space, and does not have to correspond with physical memory.

4.1.1 Global Memory Segment

The first memory segment is the global segment that is used to set and retrieve global attributes of the RH controller. This segment corresponds to a single page of "memory" and is primarily used when the OS first initializes the RH controller. For instance, the OS uses this segment to query how many tiles are on the system (allowing a single device driver to support multiple devices, each containing a varying number of RH tiles). Interrupt routines associated with the RH controller also use this segment to determine the cause of an interrupt.

4.1.2 Physical Memory Segment

The physical memory segment must be able to access each physical kernel that could exist on the RH fabric. Each page in this segment represents a single physical RH kernel. Because a configured kernel must occupy at least one tile on the RH, the maximum number of "entries" in the physical memory segment is the number of tiles on the system (with each entry occupying a single page of the physical RH kernel's memory segment).

The physical memory segment is used by the OS to perform operations on a physical RH kernel. For instance, in a real system, this memory segment would be used to specify the address, and length of the configuration bitstream of an RH kernel. It would also be used to specify which tiles on the system the configuration should occupy. Any operation directly associated with a physical RH kernel is mapped into this memory segment.

4.1.3 Virtual Memory Segment

A portion of the the virtual memory segment is mapped to an application when the application first initializes an RH kernel (see Section 4.2). Each page of the segment corresponds to a single virtual RH kernel. This segment contains user-accessible parameters that an RH kernel needs to operate, allowing an application to read back if a

kernel is available, set stream controller parameters, start an RH kernel, and query if a kernel has finished operating. For details on how these steps are done, see Section 4.2 .

Each kernel in the system is assigned a single 8KB page of the virtual memory segment. Because hardware is needed to support each virtual kernel on the system, a hard limit is placed on the number of virtual RH kernels that a system can contain. Therefore this memory segment contains a single page for each virtual kernel that can exist in the system.

4.1.4 Virtual OS Memory Segment

Although a memory segment for each virtual RH kernel already exists, this is not sufficient to handle all of the operations that can be performed on an RH kernel. This is because portions of the virtual memory segment are mapped into user applications' address spaces. Therefore user applications can access all parameters associated with a mapped virtual RH kernel. However, the OS must be able to perform additional operations on virtual RH kernels that the user application should not be able to perform.

We therefore created a separate VOS memory segment that allows the OS to access portions of a virtual RH kernel that user applications cannot access. Pages in this memory segment are only mapped into the OS' address space, and are primarily used to map virtual RH kernels to physical RH kernels, to initialize the virtual kernel's memory management unit (MMU), and to gather statistics about a virtual RH kernel that can be used to schedule which RH kernels should be configured on the device.

4.2 Initializing and Using RH Kernels

An RH kernel must be initialized by the OS before it can be used by an application. To do this, the application first opens up a virtual kernel device that is created in the `/dev/` directory when the OS boots up. Once the "file" is opened, the application writes a structure containing the RH kernel's information to the file. The OS uses this information to setup the virtual kernel on the RH controller, initialize the virtual kernel's MMU data, specify the type of kernel that is to be used, etc. After the OS sets up the virtual kernel, execution is transferred back to the requesting application.

Next, the application must request a "memory-map" of the file that it opened. The OS will then map the 8KB page virtual kernel memory-map to the requesting application's address space that the application can use to access the virtual RH kernel. Access to this returned virtual kernel memory-map will be referred to as loading and storing data to/from a virtual RH kernel.

Once the application has obtained the virtual kernel, it is ready to be used. Because the OS is responsible for deciding which RH kernels are loaded on the RH fabric at a given time, the application must check to see if the RH kernel is available on the RH before using it. The application does this by reading back a value from the "Kernel Status" offset of the virtual RH kernel.

```
1  do
2      Status = QUERYRH(VirtualKernel)
3  while (KERN_BUSY(Status))
4  if (NOT_CONFIGURED(Status))
5
6      Software Routine. . .
7  else
8      INITKERNPARAMS(...)
9      LAUNCHKERN(VirtualKernelID)
10     WAITKERNDONE(VirtualKernelID)
```

Figure 4.1 Pseudocode illustrating how an application accesses a virtual RH kernel, and launches it if it is available.

The kernel can currently be in one of three states: *Available*, *Not Configured*, or *Busy*. The *Busy* state indicates that the RH kernel is currently configured on the fabric, however another application is executing on the physical RH kernel (for details on how multiple applications can share an RH Kernel see Chapter 9). If the kernel is in the *Available* state, the RH controller will "reserve" the RH kernel for the requesting application. When an RH kernel is reserved, another application cannot access it (it will be in the *Busy* state), and it cannot be deconfigured until the RH kernel has finished executing (however the OS can tell the RH kernel to automatically deconfigure itself when it is done executing).

Figure 4.1 shows pseudocode for the typical usage of an RH kernel. In this example, if the RH kernel is busy, the application will wait until the hardware is available. If the kernel is not present on the RH, a SW version of the kernel is executed instead, and if the RH kernel is available, the application initializes and executes the RH kernel.

Before executing a kernel, many different parameters must be initialized. In particular, each of the stream controllers associated with the RH kernel must be initialized. Each stream controller currently requires the programmer to set 13 different variables to be useable. However, many of these variables are either fixed for the kernel (never change), or are the same during a single instance of a program's execution. For example, many of the video processing kernels always look at 8x8 blocks of data, and always execute for the same number of iterations. This part of the kernel never changes. However, these kernels also need to know how "wide" the frame of data they are looking at is so that the stream controller can calculate the correct stride between consecutive rows in the frame. This portion of the kernel is dependent upon the video being encoded, however, once this value is set, it is unlikely to change throughout the kernel's execution. The remaining parameters (such as the start memory address of the data a kernel is consuming/producing) are likely to change every time the kernel is called.

Because many of the parameters do not change often, multiple optimizations have been made to the stream controllers that allows them to be used without resetting all of the parameters contained in them. This saves multiple cycles each iteration, and in the case of some of the short-running Xvid kernels, performance on the RH would be

much worse if all of these parameters had to be set for each stream controller (in many cases, the overhead of setting all these parameters is even greater than just the kernel's SW execution time).

The first optimization allows an RH kernel developer to set "default" values for each of the stream controllers parameters. This way the application does not have to initialize static stream controller parameters. Secondly, the RH kernel's stream controllers cache all of their parameters between calls to them. This cache is only cleared (to default values) when an RH kernel is reconfigured on the RH fabric, or if kernel sharing is enabled, and the physical RH kernel was last used by a different application as described in Chapter 9.

To accommodate this second optimization, a kernel can be in a fourth status when requested by an application: Available but not initialized. When an RH kernel is in this state, the kernel is still reserved, however the application should ensure that all of the stream controllers are reinitialized. With these optimizations in place, the application only sets the parameters that change between calls to the kernel. Many kernels only need to update the memory address that each of the kernel's stream controllers access data from.

After all of the stream controllers, and the RH kernel itself has been initialized, the application can start executing the RH kernel. This is done by writing to the StartKernel offset in the virtual RH kernel. Once the kernel starts executing, applications can poll the status of the virtual RH kernel to see if the RH kernel has finished executing. Although applications could continue executing useful code while also executing on the RH coprocessor, the applications examined in this work do not do this because it would require significantly altering the application source code. Such an expensive redesign is unlikely to be done for many of the short-running RH kernels used in this work.

RH kernel continue executing until all of their output stream controllers have finished processing their data, or the RH kernel sends the RH controller a done signal. However, the RH controller does not mark an RH kernel as done, and "unreserve" it until all of the kernel's outstanding memory requests have been committed to the cache hierarchy. This acts as an implicit memory barrier, ensuring that memory is updated when a SW application reads back that the kernel is done.

4.3 OS Control of the RH Kernels

The OS is responsible for the control of the physical RH kernels. The OS must be able to initialize RH kernels, configure the RH kernels, and later respond to interrupts generated by the RH controller on behalf of RH kernels. Most of the OS communication with the RH controller takes place using the base, physical, and virtual OS memory maps. Whenever the OS accesses a virtual or physical RH kernel, the OS first obtains a lock on the device to ensure atomicity of the OS' actions.

The OS is first invoked when an application opens up a virtual RH kernel from the `/dev/` directory. During this routine, the OS checks to see if the virtual kernel has been opened by another application. If it hasn't been, the OS sets up the device, and modifies the appropriate OS structures so future requests will be answered properly. The OS also initializes the virtual RH kernel with the RH controller. In doing this, it sets the PID of the application, as well as

the location of the application's translation storage buffer (TSB) and page table address of the application. These are stored using the virtual OS address space associated with the opened virtual kernel. In Section 4.5, the these structures are used to help translate virtual memory addresses issued from RH kernels.

The OS also implements a "write" routine so that the application can write information about the requested RH kernel to the OS. This structure contains run times of the RH kernels (both in software as well as hardware), the size of the RH kernel (in tiles), the type of RH kernel (in a real system this would correspond to a bitstream file), as well as the number of equivalent instructions it takes for the SW to run the RH kernel (for detailed information on the equivalent instructions metric see Chapter 5.3). These values are saved in the OS and written out to the kernel's virtual OS address space.

The next routine the OS implements is the memory map routine. In this routine, the OS determines the physical memory address of the virtual RH kernel, and maps it to a single page of virtual memory in the requesting application's address space; allowing the user application "direct" access to their virtual RH kernels.

The last step when initializing a virtual RH kernel is to setup an address translation thread. The behavior of this thread is described in Section 4.5.

The OS is also responsible for deciding which RH kernels should be loaded on the RH at a given time, and subsequently configuring the selected virtual RH kernels onto the physical RH. Configuring a selected RH kernel on the RH is a fairly straightforward process that can be done dynamically at runtime (as described in Chapter 2.7.2), or statically when an application initializes the kernel. To start the configuration process, the OS needs to know the virtual RH kernel to configure, the bitstream configuration of the physical RH kernel, as well as the physical kernel position that the RH kernel will be configured on. The OS then tells the RH controller to configure the physical RH kernel at the appropriate location on the RH. This causes the RH controller to queue up the kernel in its configuration slot, and will start the configuration of the RH kernel as soon as possible. The OS than writes to the VOS address space of the virtual RH kernel to update which physical RH kernel it corresponds to. Once the RH controller has finished configuring the RH kernel, an application that requests the kernel will be informed that the RH kernel is configured, and ready to use.

The OS also has the capability of querying RH kernels to read back statistics about the execution of the RH kernels. This is done to provide the OS with a more accurate picture of the RH kernel's execution time when compared to the time stated when the kernel is initialized. This prevents an application from saying that an RH kernel is much more valuable than it really is so that it will be more likely to be configured, resulting in better allocation of the application's kernels (but a worse overall allocation of kernels) [41]. To support this, the OS reads back the average time the kernel takes when executing in both hardware and software, and the number of times the RH kernel was called both in hardware and software. The OS is also able to reset these values. These values are all located in the VOS address space.

4.4 Hardware Support

Unlike most hardware, the RH controller was designed to interact directly with "non-privileged" applications, and must take extra precautions to protect system memory. For the system to operate correctly, the RH controller must be able to answer all requests to memory segments corresponding to it, forwarding them to the appropriate portion of the RH controller when appropriate.

Therefore, the RH controller must be able to keep track of all of the physical RH kernels, as well as all of the virtual RH kernels in the system. Because of limited device constraints, this puts a hard limit on the total number of virtual kernels in the system. However future research could examine methods that would allow the RH controller to support more virtual RH kernels by dumping the contents of known RH kernels to known main memory locations (provided by the OS). For now, the number of total virtual and physical kernels that can exist in the system is set to be far greater than the number of RH kernels used in any of the workloads.

4.4.1 Support for the RH Controller's Memory Segments

The hardware to support the RH controller's memory segments is separated into three categories: hardware to handle virtual RH kernels, hardware for the physical RH kernels, and hardware to respond to requests to the RH controller itself.

Requests to the base memory segment are often fairly simplistic requests. The most common is used to find out which virtual RH kernel caused a TLB miss. This can be handled by a single register that is updated upon a TLB miss. The details of the TLB miss handler are described in Section 4.5. Other requests to the global address region are either to hard coded values (such as the number of tiles contained in the system), or are used for debugging of the simulation platform.

Memory requests corresponding to virtual (both virtual and VOS memory segments) RH kernels are more difficult to handle, as each virtual RH kernel can be in many different states, and many requests addressed to a virtual RH kernel must be forwarded to the physical RH kernel that is associated with the kernel. Each virtual RH kernel is either available on the RH, or not. Therefore, they must keep track of which physical RH kernel they are associated with. Although a virtual kernel can be associated with multiple physical RH kernels, in this work, the first of these physical kernels was designated to be the "master" physical kernel. Other copies of the kernel are linked from physical copies of the kernel (see Section 9).

4.4.2 Querying RH Kernels

When a kernel is first queried it is put into a "reserved" state. To do this, the RH controller must keep track of which physical kernel the virtual kernel is currently executing on (this way it does not have to check all potential physical kernels that it could correspond to). If this value is not set, then the kernel is not currently reserved. When a

kernel is reserved, many requests will be forwarded to the physical kernel which will handle them appropriately (such as storing the values to the stream controllers). Most requests to a "unreserved" kernel are undefined, and will be sent a NULL response.

RH kernels execute on the physical RH, but applications query the status of virtual RH kernels to determine if the physical RH kernel is still executing. Therefore, when a physical kernel has completed its execution, it informs its owning virtual RH kernel that it is done. When doing this it will pass any "buffered" value that an application might request after the kernel has completed to the virtual RH kernel. The virtual RH kernel then marks the RH kernel as unreserved, and can later respond to requests for the buffered value.

Different strategies have been examined to determine how an application should check when a kernel has finished its execution. In a naive implementation, the owning thread could continuously query the RH controller to determine if the kernel has finished executing. Although this method works well, and can return results relatively fast, it requires extensive communication on the interconnection network between the RH controller and the processors (this interconnection network is described in detail in Chapter 3.5). These messages require extra power, and can, in larger systems executing many hybrid RH/SW applications, saturate the network, causing network performance to plummet [14].

To optimize these queries, the RH controller queues requests that are querying when the RH kernel has finished its execution. In this scenario, the RH controller will buffer a query for up to 200 cycles to wait and see if the RH kernel finishes its execution. If the RH kernel is not done executing after 200 cycles, the RH controller will return that the RH kernel is not done executing. The RH controller does not have infinite buffering capability, and can only buffer a single request per virtual RH kernel. Therefore, a modification must also be made to the CPU to prevent it from issuing multiple requests querying the RH kernel at the same time. This logic stalls requests issued to the RH controller if there is already an outstanding request to the same address. The logic is similar to that required to combined multiple loads into a single, wider L1 cache request [129], however, instead of forwarding the data to the different loads, a subsequent load is stalled until the first load is finished. Upon finishing the first load, the second can than be issued. In multithreaded applications, only one thread should issue queries to see if an RH kernel is done executing.

By performing these optimizations to the query logic, in the common case, RH kernels can immediately return when they have finished executing, requiring only the return-trip latency of the direct-communication latency, rather than the full request latency. Additionally, far fewer requests need to be sent over the network to inform an RH kernel has finished its execution. In the case of short-running RH kernels (those that usually execute in under 200 cycles), branch prediction can be improved, and the processor can resume its execution faster once the RH kernel is done executing. Using this technique for stalling RH kernel queries improves the performance of the simulated RH kernels, however, this thesis does not present these results.

4.4.3 Configuring RH Kernels

Hardware support is also needed for the RH controller to change which physical RH kernels are loaded on the RH fabric. The OS informs the RH controller to change a physical kernel configuration by setting a new address/length pair to the kernel configuration bitstream offset. After doing this, the hardware is responsible for loading the new configuration bit stream.

To do this, the RH controller must first have a mechanism to queue up multiple physical kernel configuration requests. The RH controller then checks the first entry in the queue to make sure the physical kernel is done executing (the OS must ensure that no virtual kernels point to a physical kernel before it initiates a deconfiguration request). Once the old kernels are finished deconfiguring, the RH controller can start loading the new physical kernel's bitstream from main memory (the OS is responsible for ensuring that the bitstream is located in contiguous physical memory pages).

Hardware support is also needed in the virtual RH kernels to check if the physical RH kernel that they are associated with has actually been configured. This can be done by having the virtual RH kernel poll the physical RH kernel when it is queried. An additional optimization could be made where the virtual RH kernel does not perform this check after acknowledging that the RH kernel is loaded as long as the physical kernel value on the virtual kernel is not changed by the OS.

4.5 RH Kernel Virtual Address Translation

When RH kernels access memory, the RH controller is responsible for translating the issued virtual memory addresses. This is done using a TLB that is shared by all of the physically configured RH kernels. If a valid translation cannot be found in the RH controller's TLB, the RH controller interrupts the CPU to obtain a translation. The OS then executes a custom interrupt handler, described in pseudo-code in Figure 4.2. On multi-core systems, the RH controller interrupts the host CPU that is running the application that requested the RH kernel to reduce the impact of page translation interrupts on the performance of other processes executing on the system.

The interrupt routine was designed to mimic the trap routine that the UltraSparc processor enters upon a software TLB miss, however the interrupt routine must first read back which virtual RH kernel caused the miss from the RH controller's base memory segment. The routine then reads back the virtual memory address that caused the miss. Like normal TLB misses during software operation, the routine first checks the UltraSparc's TSB for the faulting address. If the address is not mapped in the TSB, the interrupt handler walks the page table to find the translation. For some of the studies performed for this thesis, the RH controller directly accessed the RH kernel's TSB, for details on how this is done, and its impact on performance, see Chapter 7.3

Interrupt routines in Linux have limited capabilities: they cannot block (to the OS) waiting for input like a normal process can. Therefore, on a page table miss, the interrupt handler is unable to retrieve the faulting page from disk (or wherever it may be) [104]. Because of this, the RH kernels rely on the user application to handle page faults. To

```
INTHANDLER(MissAddr, IsRead)
1  hash = TSBHASH(MissAddr)
2  if (TSB_hash.Addr = MissAddr)
3       if (VALID(TSB_hash.phys)
4       return TSB_hash.phys
5  phys = PAGETABLEWALK(MissAddr)
6  if (VALID(phys))
7       return phys
8  WAKEUP(FaultHandlerThread, MissAddr, IsRead)

FAULTHANDLERTHREAD(MissAddr, IsRead)
1  while (1)
2
3       if (IsRead)
4            tmp = *MissAddr
5       else *MissAddr = *MissAddr
6       INTHANDLER(MissAddr, IsRead)
7       SLEEP()
```

Figure 4.2 Interrupt handler routine that is called upon a TLB-miss in the RH controller.

handle page faults, user applications spawn a fault-handler thread when initializing the RH kernel (applications only need to spawn a single fault handler thread regardless of how many RH kernels the application has). This thread waits in a loop that blocks to the OS until the interrupt routine experiences a page table miss. On a page table miss, the interrupt handler awakens this fault-handler thread. After waking up, the fault-handler thread queries the OS for the logical address of the faulting load or store, and attempts to read or write from it (as appropriate). Upon accessing this address, the thread will produce a page fault that invokes the OS' page fault handler. After the thread has finished reading/writing from the faulting address the fault-handler thread blocks to the OS once again. The OS will then provide the page translation to the RH controller so that the address can be added to the shared TLB.

Unfortunately, this method is slower than if the interrupt routine fully handled the page fault. However, in a well-tuned application, page faults should be a rare occurrence. Additionally, page faults are already extremely expensive to handle (on the order of multiple milliseconds) [109], so the extra time required to swap in the fault-handler thread should not result in significant additional overhead.

After translating an address (whether within the interrupt routine or the fault-handler), the OS sends the translated address to the RH controller. The RH controller stores this address inside its TLB, and reissues the virtual address translation.

4.6 Memory Ordering on the Reconfigurable Hardware

On this system, the RH acts very much like a peer of the processor cores from the viewpoint of main-memory. Both processor cores and the RH kernels share virtually addressable memory, and both can use the processor's cache subsystem to store data that is frequently reused.

However, in the current system the RH kernels cannot operate on their own, and require support from an associated SW thread. Application software is expected to launch RH kernels, determine when the kernels are done executing, and ensure data is ready for the RH kernels to process before starting the RH kernel. Because of this, we optimized our system for performance at the expense of ensuring loads and stores issued by the RH kernels are ordered properly. Once requests are issued to the cache subsystem, they maintain total store ordering, however this is not the case within the RH controller. This is problematic because the RH kernels should calculate the same values that would have been calculated by software. If the memory accesses in the system are not consistent, loads and stores can be reordered with respect to one another, and a load may read back "stale" data from memory. This can result in race conditions where the CPU and RH kernel contain data that is logically inconsistent [59].

Although the processors, and caches support total store ordering, the RH kernels support a much weaker release consistency model where access to memory is acquired when the CPU launches the RH kernel, and is released when an RH kernel has finished executing. To ensure that hybrid RH/SW applications will execute properly on this platform data accesses made within the RH system or between the CPU and RH must be repeatable if the same test was run again.

Because our RH system ensures no ordering of data loads and stores made by RH kernels, we have placed restrictions upon how RH kernels can access main-memory to ensure that our hybrid RH/SW applications are data race free. To provide for this, a RH kernel can never write data that it will later read, as this could result in a read after write hazard [59], where the data read back does not match the data the kernel previously wrote out. RH kernels can easily work around this problem by internally caching any data they will need later before committing it back to memory (data can be read from memory multiple times, as long as it is not read again after the kernel writes to it), however this was not necessary for the RH kernels used in this thesis. Additionally, RH kernels are not allowed to directly communicate with concurrently executing RH kernels. If they did, a race condition could occur where kernel A writes data first to memory location 1, and then to memory location 2. A second kernel B might read back these values, first reading the new value from location 1, and then reading the old value from location 2. Therefore software programmers must ensure that RH kernels do not read data that is being produced by a currently executing RH kernel. However, this is only necessary on multithreaded applications, because, as I will explain below, the SW in our system spin-waits while an RH kernel is executing as a method of providing the release consistency model used in CPU RH communication. In a multithreaded system, threads generally cannot write to data memory unless they have an exclusive lock on the data [109], so by extension the RH kernels called by each thread would not be reading data that another thread has an exclusive lock on.

In addition to ensuring our RH kernels were data race free, we also had to ensure that communication between the SW and RH kernels was data race free. Because our RH kernels followed a release consistency model, the RH kernels must have complete control of the memory they accesses during their execution, and the SW application must have complete control over that memory when it executes. To prevent SW from reading back stale data from finished RH kernels, we placed an implicit memory barrier at the end of each RH kernel (when releasing the RH kernels access to memory). This prevents SW from reading back that an RH kernel is done until all of the kernel's memory accesses have been serviced by the cache subsystem; ensuring that the RH kernel properly releases its access to memory. Therefore, as long as the application SW does not read data that is currently being processed by an RH kernel, or output by an RH kernel, the hybrid RH/SW application is data race free, and will execute correctly. This is implicitly the case in our system, as the SW applications spin-wait while executing RH kernels, which prevents the SW from accessing any of the data the RH kernels are processing. In our multithreaded application we locked the data being written by individual threads, or theRH kernels launched by the thread. This ensures that other SW threads cannot access the data that another RH kernel writes (and the RH kernel likewise cannot read data being written by other SW threads).

In future systems, it is possible that limited forms of store ordering could be utilized within the RH controller, however the optimizations made in the stream controllers make complete ordering of stores extremely difficult. Future work could also examine the use of special atomic primitives and barriers within the RH kernels that would allow multiple RH kernels to safely communicate.

4.7 Library and OS Support

To support this reconfigurable computing system, I have developed a Linux kernel module, as well as a C library (LibRH) that implements all of the communication between the user applications, the OS, and the RH controller. Using these modules allows application developers to create new RH kernels without having to concern themselves with the direct RH/SW communication necessary to use the RH kernels.

The Linux device driver handles the initialization and configuration of RH kernels. It also performs all of the actions discussed in Section 4.3, and services interrupts from the RH controller. The module was developed for the 2.6.17 version of the Linux kernel.

The library SW API consists of a set of functions and macros that user applications can use to abstract away the details of the communication with the RH kernels. This library is responsible for initializing RH kernels, automatically spawning a single page table miss handler thread (see Section 4.5) for the application, and allows for the initialization and calling of RH kernels.

The C library uses a LibRH structure to represent an RH kernel. This structure keeps track of the file descriptor for the virtual RH kernel that was opened, and contains a pointer to the virtual RH kernel memory map that the library uses. Additionally, it encapsulates access to the stream controllers in a C-style, presenting the application developer with an array of structures representing the stream controller. Using this allows applications to set fields in the stream controller in the same style that they would change values in any structure (for example, the user could change the starting address of stream 0 of a kernel by the following line of code: `KERNEL->Streams[0]->StartAddr=Value`).

Chapter 5

Benchmarks, Workloads, and Testing Metrics

I tested a range of applications on my system to model "real-world" workloads. To do this, I developed multiple hybrid RH/SW benchmarks, and combined them with software-only benchmarks to evaluate the performance of the reconfigurable computing platform.

This chapter first describes the benchmarks used in this work (Section 5.1). Then, Section 5.2 discusses how I combined these benchmarks to create "realistic" system workloads. Finally, Section 5.3 describes some of the metrics used to evaluate the research performed in this thesis.

5.1 Benchmarks

I evaluate the proposed reconfigurable computing system using a variety of benchmark applications. Because the execution model of the platform differs from that of traditional software-only applications, I heavily modified many of the benchmarks so that they executed efficiently on the platform. This section first describes how I developed these benchmarks (Section 5.1.1). Then it examines the hybrid benchmarks that have actually been implemented. Finally it describes the software-only benchmarks used in some of the experiments performed in this thesis.

5.1.1 Benchmark Development

Developing benchmarks to execute on the proposed reconfigurable computing platform required a significant amount of work beyond designing the interface and API described in Chapter 4. Most of the benchmarks started from open-source software-only applications, which simplified the development process.

I only ported applications likely to be executed on a high-end embedded reconfigurable platform to my new platform. To create these benchmarks, I first decided which portions of the application should be accelerated by the RH fabric. To do this, I executed the software versions of the benchmarks using the GNU profiler (gprof). This allowed me to determine which functions in the application represent the majority of the application's execution. I used a workstation computer to perform the Initial profiling of the applications. For these measurements, custom SIMD instructions were disabled, because we envision the RH coprocessor being used in place of a SIMD unit. Although this machine

was much faster than the target embedded platform, the approximate distribution of runtimes of each function in the application tends to be similar across platforms, providing a rough estimate of the application's runtime distribution.

After profiling the application, I evaluated candidate portions of the application to determine the suitability of creating RH kernels for them. This process involved carefully examining the functions that represented the largest percentage of the application's execution time (based on gprof's measurements). This often involved examining not only the function shown in gprof, but also the functions that called the candidate function. This is because C functions don't always translate directly into RH kernel implementations. Additionally, in some instances it was easier to create a Verilog implementation of only a portion of a C function, allowing similar application coverage with simpler RH. If the candidate kernel was deemed suitable for a RH implementation, we created a Verilog implementation of the kernel.

To create a Verilog implementation of the kernels, I modified the software to output test input vectors (and the corresponding outputs) that could be input into the verilog test bench. Once we created fully working and tested Verilog implementations of an application's kernels, I modified the software's source code to call the newly created RH kernels.

For some kernels, this process was straightforward, as the Verilog kernel was as a drop-in replacement for all, or most of a standalone C function. In these situations, all that I only needed to add code that used the LibRH API (Chapter 4.7) to call the RH kernel where appropriate. In other cases, this proved to be a more difficult challenge. For instance, the hardware implementation of the FDCT kernel in Xvid scales the output in a very different manner than the software version. Therefore, I had to modify a later portion of the application to detect if the SW or the RH generated the inputs, applying different scaling appropriately. In other RH kernels, I created new buffers to facilitate the transfer data to/from the RH kernel.

The benchmarks developed for this work, including their RH kernels are all part of the ERC Bench (embedded and reconfigurable computing benchmarks) project [24]. Some of these benchmarks were developed specifically for the work done in this thesis, while others have existed in varying states of readiness, and were modified to run on the Alexandrite simulation platform. In the future, it is likely that RH can be developed using "high-level" hardware description languages that can create hardware implementations from a high level description of an algorithm, or software code [37, 112]. Additionally, a hardware software codesign environment could allow developers to directly include hardware and software modules in the same project, simplifying the design and test of hybrid RH/SW applications.

5.1.2 Hybrid RH/SW Benchmarks

This thesis analyzes the performance of four hybrid RH/SW benchmarks: Xvid encoding, Tremor decoding, AES encryption, and a WiMaX back-end receiver. Table 5.1 describes the RH kernels used in these applications, including their size (in tiles), clock frequency, percent of their host applications execution time (when running in SW), percent

Application & Kernels		Area (tiles)	Frequency (MHz)	Percent of SW-only exec	Percent of Hybrid execution	SW Runtime (cycles)	Average RH Speedup
Xvid	SAD8	1	500	31.4%	8.6%	652	8.5x
	SAD16	1	475	10.0%	1.8%	2,274	15.7x
	interpolate 8x8 6-tap	4	225	7.6%	3.1%	8,950	6.9x
	FDCT	9	225	6.4%	.89%	19,063	20.4x
	interpolate8x8 avg-4	1	450	6.2%	1.1%	2,061	12.5x
	interpolate8x8 avg-2	1	500	4.3%	1.1%	1,444	8.7x
	transfer 8 to 16 sub	1	500	2%	.75%	990	7.7x
	total	18	NA	68.6%	24%	NA	NA
AES		5	225	99.9%	99.9%	247,000	17.2x
Tremor	mDCT	5	150	32.5%	2.4%	127,622	21.2x
WiMaX	DFT	12	300	7.1%	.53%	56,086	41.1x
	Viterbi	12	67	62.0%	1.9%	491,342	100.5x
	total	24	NA	69.1%	2.4%	NA	NA

Table 5.1 Breakdown of the kernels used in the hybrid RH/SW benchmarks, when executing on the RH coprocessor platform (data is not given for SW runtime and average RH speedup for full applications, as these metrics are used to describe individual kernels).

of host application's time spent executing the kernel during hybrid RH/SW execution, the SW runtime, and the overall application speedup on a single processor baseline system that contained 2MB of L2 cache, and sufficient RH tiles to fit all of the benchmark's RH kernels. I calculated these values after enabling all of the optimizations described in this thesis, so the results in Table 5.1 represent the best runtimes these kernels had on a single-core version of the platform.

The Xvid benchmark uses the Xvid 1.0.3 encoder to encode videos using the MPEG-4 advanced simple profile, and contains seven different RH kernels that cover $\sim$69% of the application's software runtime. All of the simulations that run the Xvid benchmark encode the same 720x400 input video, however each copy of the benchmark might be encoding a different portion of the video frame. The xvid encoder encoded multiple frames of the input video before I created checkpoints of workloads to ensure that I only measured Xvid's steady-state behavior.

The Xvid benchmark is an example of a hybrid RH/SW application that exhibits different phases of execution, where each phase of execution calls a different subset of the application's RH kernels. In the first phase of execution, the encoder calls the interpolate 8x8 6-tap kernel to create interpolated images of the video frame. The second phase repeatedly calls the SAD16 routine across the image to obtain an estimate of the motion vectors used in the next phase. The third phase performs the motion estimation. This phase regularly calls the SAD8 kernel, and also calls both the interpolate 8x8 average-2 and interpolate 8x8 average-4 kernels to estimate the value of subpixels. In the last phase of execution, the video is encoding into a bitstream. This phase calls 0the FDCT kernel to transform the video, and the transfer 8 to 16 sub function to transfer data between macro blocks in the video streams..

The Tremor benchmark decodes Ogg/Vorbis music files for playback. This benchmark contains a single RH kernel that performs the modified discrete cosine transform.

The AES benchmark is an example of a high-bandwidth streaming application that encrypts a single large file (much larger than the cache on the system). Applications that handle secure information commonly use the AES encryption algorithm. The hybrid application contains a single kernel that represents the vast majority of execution. Each call to the AES kernel encrypts 8KB of data (either in RH or in software). When simulating AES, the plaintext input data, as well as the buffer containing the output data is pre-loaded in main memory (but not the cache) to ensure that the data is found in the process page table. This enables the measurement of AES's steady-state behavior, and not artifacts of the beginning of its execution.

Unlike the other hybrid benchmarks examined in this thesis, I developed the WiMax back end receiver benchmark specifically for heterogeneous multicore systems. The operations performed in the WiMaX receiver are common to many different software defined radio protocols. I used a reference receiver WiMaX receiver developed in Matlab [74] to model the WiMaX back end receiver. Because I did develop a front end receiver, the control signals necessary to change the data rate were not implemented. In the current implementation of the WiMaX benchmark, a single packet of data is continuously processed in the same mode of operation. This benchmark is multithreaded, spawning threads for the DFT, denormalizing, demodulating, deinterleaving, Viterbi decoding, Reed-Solomon decoding, and derandomizing stages of the WiMax back end receiver.

When possible, existing library code was used to perform the operations contained in the pipeline stages of the WiMaX backend. The DFT uses the KISS FFT library, which implements a fixed-point version of the DFT algorithm. Viterbi decoding and Reed-Solomon decoding uses Phil Karn's Forward Error Correction library. The other modules use custom C routines to implement their functionality. The hardware implementation of the DFT kernel uses a modified version of one generated using the Spiral project [94].

5.1.3 Software-only Benchmarks

In addition to the four hybrid RH/SW benchmarks, some of the workloads contained SW-only benchmarks. I used a subset of the SPEC 2006 integer benchmarks [60] for the SW-only benchmarks, including mcf, gobmk, and xalancbmk. mcf is a benchmark that models the scheduling and routing of a fleet of vehicles using a network simplex algorithm. gobmk is a solver for the game of Go, and is representative of AI algorithms that a game might contain. xalancbmk is an xslt processor that transforms xml documents into html. I chose these benchmarks because they have different memory access patterns and they are similar to software applications that might execute on a high-end embedded platform.

5.2 Workloads

Many of the evaluated systems contained multiple processor cores, so I created workloads that run multiple benchmarks simultaneously. I created workloads containing a combination of the benchmarks for One-, two-, four-, and eight-processor systems to ensure every CPU on the system was busy..

In an attempt to make the workloads more "realistic", only certain benchmarks were executed simultaneously. In particular, all of the workloads contain at most one copy of the Tremor benchmark, and at most one copy of the WiMaX benchmark. This is because real systems are unlikely to play back multiple audio files simultaneously, or process multiple WiMaX streams simultaneously.

The workloads used in this thesis, will be referred to by the number of copies of each benchmark followed by the first letter of the benchmark, with a hyphen to separate the benchmarks. Therefore 2X- 4A-1T-1W refers to a workload containing two copies of Xvid, four copies of AES, one copy of Tremor, and one copy of WiMaX. For any workloads where SW-only is included, it indicates that three separate workloads were executed, each containing one of the SW-only benchmarks. All of the workloads were simulated for 2-billion cycles (just over two seconds of wall clock time) to get a measure of the workloads steady-state behavior. Unless otherwise specified, the same checkpoint was used for each workload, regardless of how the system executing it was setup. Therefore (at least initially) the systems will execute the exact same code, at the same point in time, allowing direct comparisons between the different system models.

5.3 Testing Metrics

When executing these workloads, a handful of metrics were used to compare the relative performance of different systems. These metrics are based off those proposed in Rupnow et. al [106].

I measured the performance of single benchmarks within a workload using the equivalent instructions per cycle (EIPC) metric. This metric is similar to the IPC metric used in traditional processors. The equivalent instruction count (EIC) of the benchmark is needed to calculate the benchmarks normalized EIPC (NEIPC). The EIC measures how many instructions the application would have executed for if the application executed everything in SW. This EIC is then divided by the number of cycles that the benchmark executed for to obtain the EIPC. The EIPC metric is often used in this thesis for comparing performance of a single application within a workload.

However, on many workloads, this metric is not sufficient for comparing performance, and instead we use the NEIPC metric. Because workloads can contain multiple threads executing across multiple processors, it is important to normalize each of the benchmarks to "spread-out" the OS' overhead amongst all of the benchmarks (and not just the benchmark the OS interrupts the most). This is done by summing up the number of cycles each of the benchmarks in the workload executed for, and dividing it by the total number of cycles the simulation ran for times the number of processors in the simulation. This gives us the "efficiency" of the workload, representing the percent of time an average

thread in the workload was performing actual work, and not in the OS. To calculate the NEIPC of each benchmark, the efficiency of the workload was multiplied with the workload's EIPC. Equation 5.1 illustrates how I performed this calculation.

$$NEIPC_i = \frac{EIC_i}{Cycles_i} \times \frac{\sum_i Cycles_i}{NumProcessors \times TotalCycles} \tag{5.1}$$

The NEIPC of each application in a workload is useful, but it can be cumbersome to compare each benchmark individually for each of the many tests that were executed. To do this, I calculated the geometric mean of the workload's benchmark's NEIPC values. This allows the direct comparison of values across different system models.

Chapter 6

Cache Organizations for Reconfigurable Computing Systems

Cache-coherent reconfigurable coprocessor's memory systems can be designed in many different ways. However, in all of these designs, an ordering point is necessary so that the system can ensure that the memory the RH coprocessor accesses contains the same data that other CPUs would see. This chapter examines multiple different cache organizations, and analyzes how the cache organizations impact the performance of both the RH coprocessor, and the general-purpose processor.

I first examine the difference between RH kernel memory accesses and CPU memory accesses. Then, I propose various different cache topologies and compare the performance of systems using these different cache topologies and examine the impact of the cache size on performance. Then I examine the dynamic energy consumption of the memory hierarchy for each of the topologies. Finally, I propose changes to the access model that allow the CPU to directly access the results of RH kernels; allowing some of the RH kernel's memory requests to bypass the cache hierarchy altogether.

6.1 How Do RH Kernel Memory Accesses Differ From a CPUs?

RH kernels in this system access memory in a very different manner than traditional CPUs. For instance, RH kernel load/store operations tend to operate on larger data sizes than a general-purpose processor. In this system, an RH kernel can access up to 32 Bytes on a single load/store operation. In contrast, standard microprocessor load/store operations operate on word-sized data; allowing them to access one, two, four, or eight bytes of data in a single cycle. Because of this, RH kernels often require fewer memory accesses to perform the same operation.

In addition, RH kernels can store more of their data internally. When a general purpose processor executes an algorithm, many of the memory accesses it makes are to state variables contained on the application's stack. These accesses tend to hit in the processor's L1 cache due to both spatial and temporal locality. In an RH kernel, many of these memory accesses are unnecessary because the RH kernel implicitly keeps track of these variables. In addition, some algorithms will repeatedly access the same data memory values when performing computations. RH kernels are much more likely to cache this operational data internally, requiring fewer memory accesses.

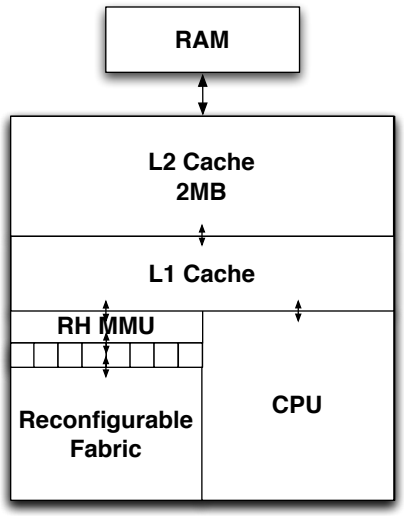


Figure 6.1 CPU/RH topology where the RH and the CPU share a single L1 cache.

Because of this, it is anticipated that an RH kernel will require fewer cache accesses than its software counterpart. It is also expected that RH kernels will have a higher L1 miss rate; although it is still likely that an RH kernel will have fewer L1 cache misses than the kernel's SW implementation. To examine this, I modeled a single CPU system where both the RH and CPU share a single L1 cache. This is similar to the Molen processor proposed in [79]. Figure 6.1 illustrates how this topology might look on a single processor system. Although this model is overly optimistic with respect to the cache's latency (and has further issues when scaling to multicore processors), it allows us to directly compare the L1 cache hit rate of kernels when executed in the RH, and on general purpose processors.

I first use this shared-L1 topology, to compare the L1 cache's hit-rate between the RH and SW implementations of the kernels. Figure 6.2 shows the L1 data cache's hit rate for all of the RH kernels used in the tested applications. As expected, the SW versions of these kernels had substantially better hit rates than the RH versions. Figure 6.3 shows the number of L1 cache misses per 1,000 equivalent instructions both when the kernels executed in SW, as well as on the RH. The RH implementations tend to have similar hit rates for short-running kernels, and for the long-running kernels (mDCT, FFT, Viterbi, FDCT) I observed significantly lower hit rates due to the ability of the RH to locally cache data. These results show that the RH kernel's lower L1 hit rate is due to both the reduced number of accesses made by RH kernels, and the greatly reduced execution time of the RH kernels.

Although the RH kernels are capable of generating many more requests per cycle than the SW, this rarely happens in the kernels that I tested. Figure 6.4 shows how many L1 data requests each kernel generated per cycle for both the RH and SW implementations. This figure shows that most of the RH kernels generate far fewer requests per cycle than SW versions of the same kernel. In particular, the highly data-parallel algorithms accessed the cache $\sim$1/5th as often as the software would. The only exceptions to this general rule of thumb occurred in some of the Xvid kernels. Many

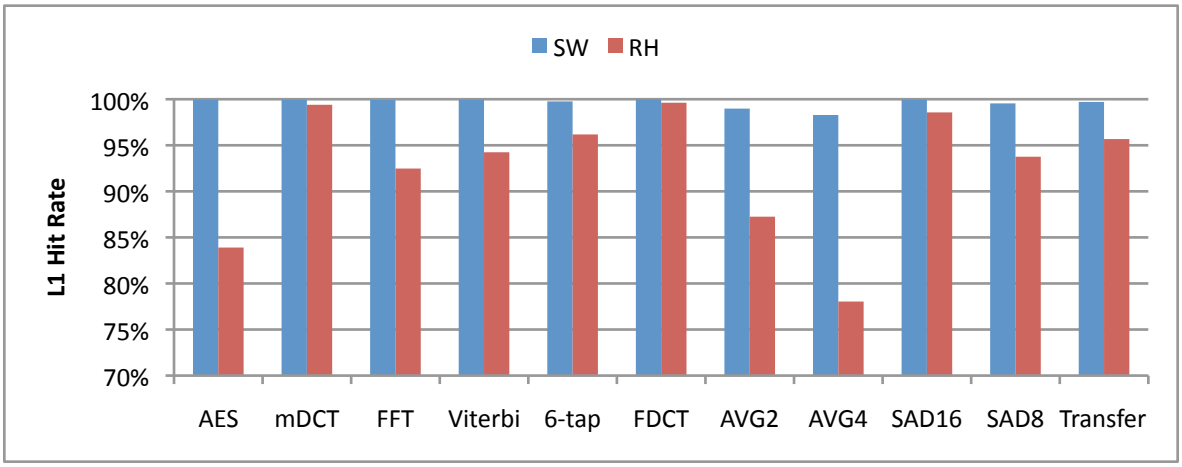


Figure 6.2 L1 data cache hit rate for the kernels when executed by RH and by SW

of these kernels read in the same data as the SW versions of the kernels, and only use the data in a single operation. Therefore both the RH and the SW only have to read this data in a single time. Because of this, many of the requests rates are similar to those of the SW, and in the case of AVG2, the RH kernel requests slightly more data per cycle than the SW.

This data suggests that the kernels would not "overload" a standard cache hierarchy, and even suggests that a single cache has enough bandwidth to handle multiple RH kernels that are running simultaneously. Chapter 7 will examine the scalability of the cache hierarchy, however this chapter focuses on how the cache hierarchy should be designed.

6.2 Examined Cache Topologies

Although previous work showed that sharing one or more cache levels between cores on a processor [9, 63] allows for the effective sharing of data on a multicore processor, it is unclear if this will hold true in RH computing systems. In this section I propose multiple different ways that a RH coprocessor can use shared-memory to communicate with a general-purpose processor.

I examined a variety of topologies to determine whether particular cache levels are advantageous for RH performance. For all tested topologies, the CPU always has a private L1 cache and either a shared or private L2 cache. The RH's cache hierarchy varies more significantly than the CPU's cache hierarchy, both in terms of number of cache levels and sizes. Table 6.1 lists the nine cache topologies examined in this work. All designs have a 2-cycle L1 access time and require 13 additional cycles for L2 accesses. In shared L2 hierarchies, if a request from the RH misses in the L2 cache, but is located in the CPU's L1 cache, the access will require an additional 8-15 cycles (varying due to queuing delays). In split L2 hierarchies, if the data is not found in the RH's L2 cache, but is in the CPU's L1 or L2 cache, it requires an additional 40-50 cycles to check/update the cache directory, submit the request to the CPU's L2 cache, and

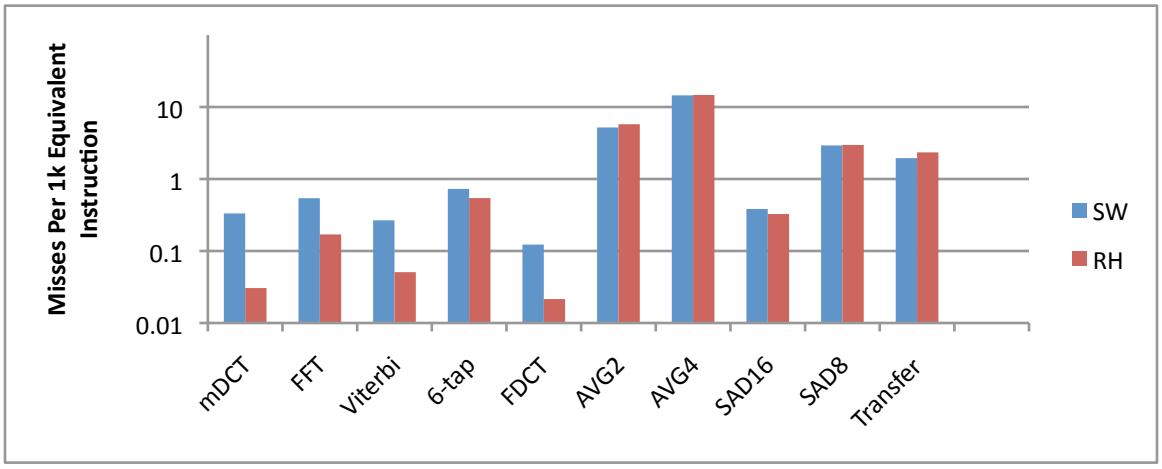


Figure 6.3 Number of L1 data cache misses per 1,000 equivalent instructions

transfer the data between the caches. These latencies are calculated by GEMS (discussed in Section 3.6) and include the timing of the coherence messages needed to perform these operations. I modified the GEMS simulator to provide timing support for the structurally asymmetric topologies that were tested. For example, requests in topologies lacking an RH L1 cache that hit in the L2 incur a latency of only 13 cycles, the number of cycles it normally takes to transfer data between the L1 and L2 caches.

In some of these topologies, the RH uses a small spatial locality buffer instead of an L1 cache. This buffer is similar to an L1 cache, however it is much smaller (2KB vs 32KB), more associative (16-way vs 4-way), and has a single-cycle latency. The goal of this buffer is to exploit the spatial locality found in many RH kernel memory requests without the overhead of a full cache. In general, the I did not modify the CPU's cache hierarchy apart from exploring both private and shared (with the RH) L2 caches. All other variations are to the RH's cache hierarchy, and include eliminating either or both the L1 or L2 cache levels, and comparing shared to private L2 caching. The total L2 cache size in the topologies is always 2MB; if it is split, the CPU and RH each have a 1MB private L2 cache. When the RH does not have an L2 cache, the CPU contains a 2MB private L2 cache.

6.3 Performance

This study examines the performance of the four hybrid RH/SW benchmarks described in Chapter 5.1. I measured the performance on all of the topologies listed in Table 6.1, and compare them with the performance on the "ideal" architecture (SL1). Figure 6.6 shows the ratio of each cache topology's performance to that of SL1. I chose this comparison to normalize the results for all of the applications. In all situations, the SL1 topology performed as good or better than the other topologies. This figure shows that most of the applications were not overly-sensitive to

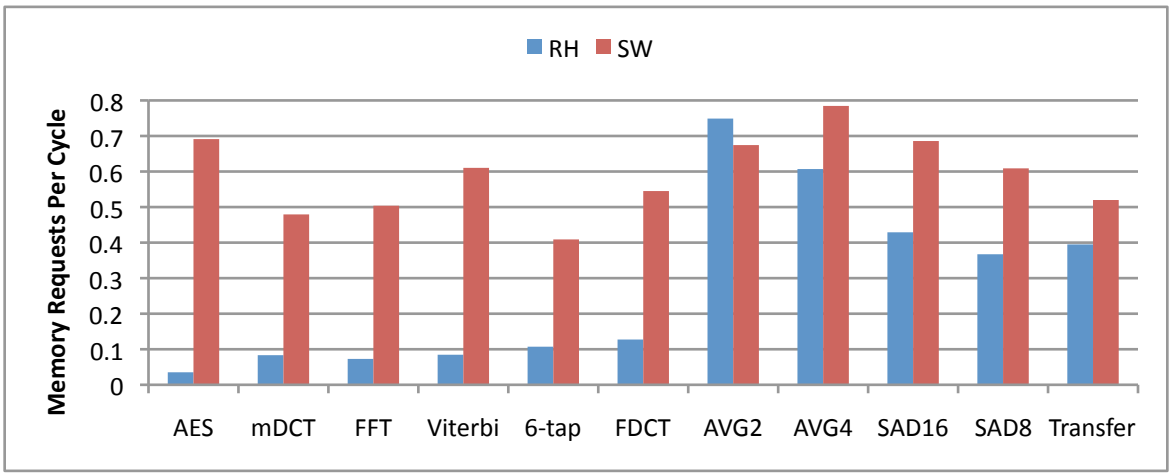
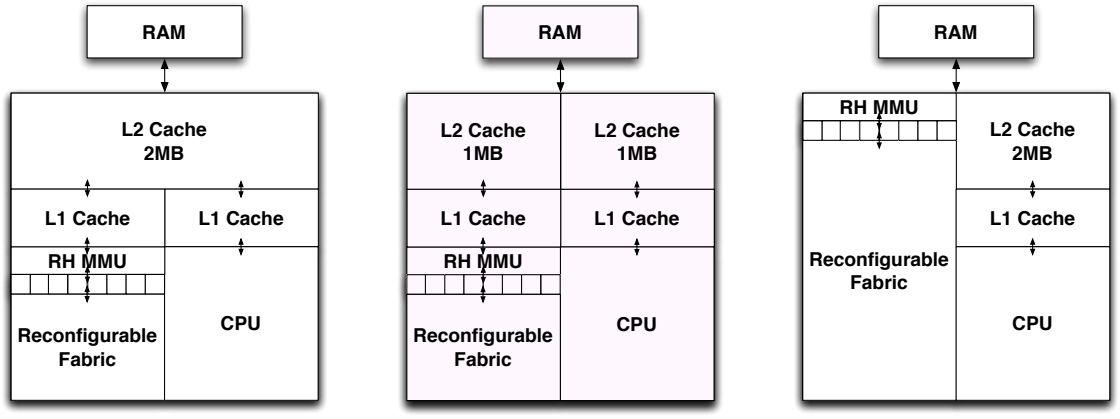


Figure 6.4 The average number of memory requests per cycle each kernel makes when implemented in SW and in RH



(a) RH and CPU have private L1s, share a 2MB L2

(b) RH and CPU contain private L1 and L2 caches

(c) RH contains no cache, CPU has private L1 and L2

Figure 6.5 Some of the cache hierarchies examined in this thesis

Name	Description
SL1	"Ideal" model where the RH coprocessor and CPU share a single L1 cache, and the rest of the memory hierarchy.
SL2-PL1	Shared 2MB L2 cache, the CPU and RH each have a private L1 cache.
SL2-NoL1	Shared 2MB L2 cache, the CPU has a private L1 cache, the RH does not have an L1 cache.
SL2-NoL1-B	Shared L2 cache, the CPU has a private L1 cache, the RH has a spatial locality buffer instead of an L1 cache.
PL2-PL1	The CPU and RH each have their own private L1 and 1MB L2 caches.
PL2-NoL1	The CPU and RH each have private 1MB L2, CPU has private L1, RH has no L1.
PL2-NoL1-B	The CPU has a private L1 and L2 cache; the RH has a small spatial locality buffer plus a private L2 cache.
NoL2-PL1	The CPU has a private L1 and 2MB L2 cache; the RH has a single cache level the size of the CPU's L1 cache.
NoL2-NoL1	CPU has private L1 and 2MB L2 cache; the RH has a spatial locality buffer instead of a cache.

Table 6.1 The nine shared-memory cache topologies that were tested. L2 cache sizes are given, the RH's L1 cache size is 32KB unless otherwise stated.

the cache topology selected, however the Xvid benchmark showed a wide variance of performance on the different topologies.

The AES benchmark was essentially unaffected by the choice of topology due to its highly streaming nature and because it did not reuse data brought into the caches. Furthermore, because each AES memory access is 32 bytes, and a cache line is 64 bytes, improved data locality has limited benefit.

Likewise, the Tremor benchmark was relatively unaffected by the cache topology used because the mDCT kernel uses static memory buffers for its data. Although the cost of moving this data across caches is non-zero, the mDCT kernel executes for a relatively long time, allowing the stream controllers to hide much of this latency. From the software's perspective, a small penalty is associated with accessing data written into this buffer, but because the kernel does not represent the majority of the application's execution time, and Tremor must perform a significant amount of computation on the data, the cost of moving it makes up a very small percentage of Tremor's overall execution time. Figure 6.6 shows that split topologies perform slightly worse than shared topologies on the Tremor benchmark. This is primarily due to the extra time required to copy data buffers between the RH and the CPU's caches.

The WiMaX application also saw very little change in performance due to the cache topology used. This is due to multiple factors. First the RH kernels execute for a relatively long time, so the stream controllers can hide the majority of the kernel's memory latency. Additionally, the RH kernels (in particular the Viterbi kernel) performs a significant amount of computation on each byte brought into it. Because WiMaX is multithreaded, when executing on a single-processor system, the data produced by any given stage in the overall computation is unlikely to be in the L1 cache

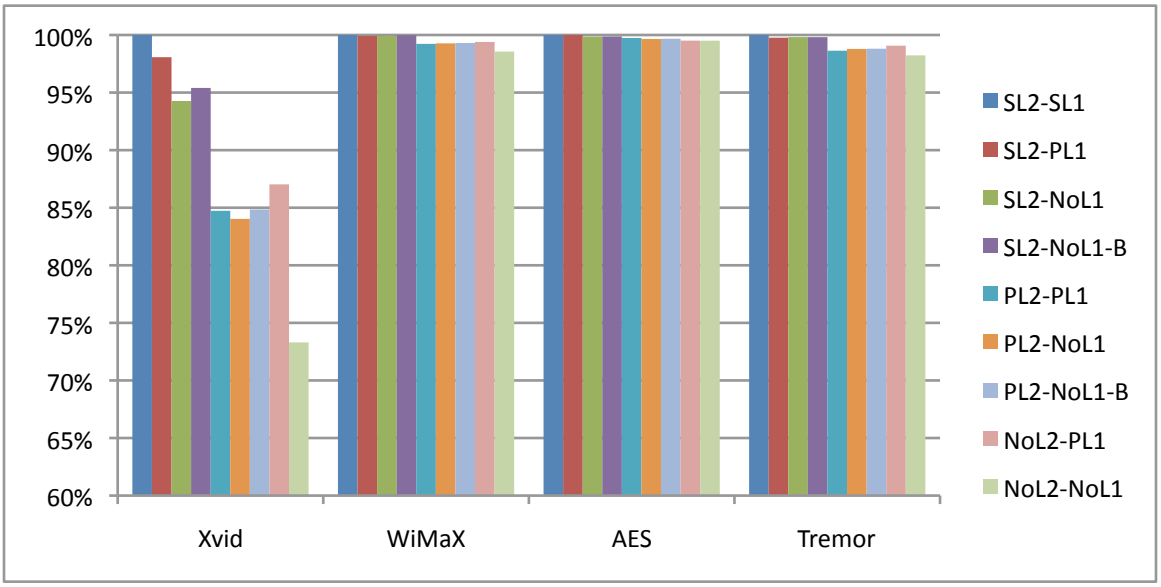


Figure 6.6 Performance of each cache topology normalized to the performance of SL1 for each of the hybrid RH/SW benchmarks

during the next stage of computation. This means that the data produced by a kernel is unlikely to be in the CPU's L1 cache regardless of whether the kernel ran in SW or on the RH. Like in Tremor, the split topologies performed a little worse than shared topologies due to the extra latency required to transfer the data between the caches. In these two benchmarks it was also observed that NoL2PL1 performed slightly better than PL2PL1. This is because the NoL2PL1 topology allowed the software access to a larger L2 data cache, increasing its performance. However, as these software applications are not overly sensitive to the L2 cache size (in the 1MB-2MB range), this only provided a small performance boost over the topologies with a private L2 cache.

Unlike AES, Tremor, and WiMaX, Xvid contains multiple RH kernels, many of which execute for a short period of time. When these kernels execute, their stream controllers are unable to prefetch the data ahead of time. Because of this, the performance of the Xvid kernels varies greatly depending upon the cache's performance. This shows that for the Xvid benchmark, it is important to consider the cache organization of the system.

Figure 6.7 shows the percentage of Xvid memory requests (CPU and RH) that missed in all cache levels and had to retrieve data from main memory. The NoL2-NoL1 topology had a significantly higher miss rate because the RH has only a very small (2K) buffer. Having just a 32KB L1 cache (and no L2) significantly improved the RH cache miss rate. Topologies with a shared L2 cache had the fewest main memory requests. In the private L2 topologies, data may be replicated between the two private caches; the shared L2 topologies have the same total L2 size, but with no replication a larger number of unique addresses fit in the L2, allowing Xvid to have a lower cache miss rate in shared topologies

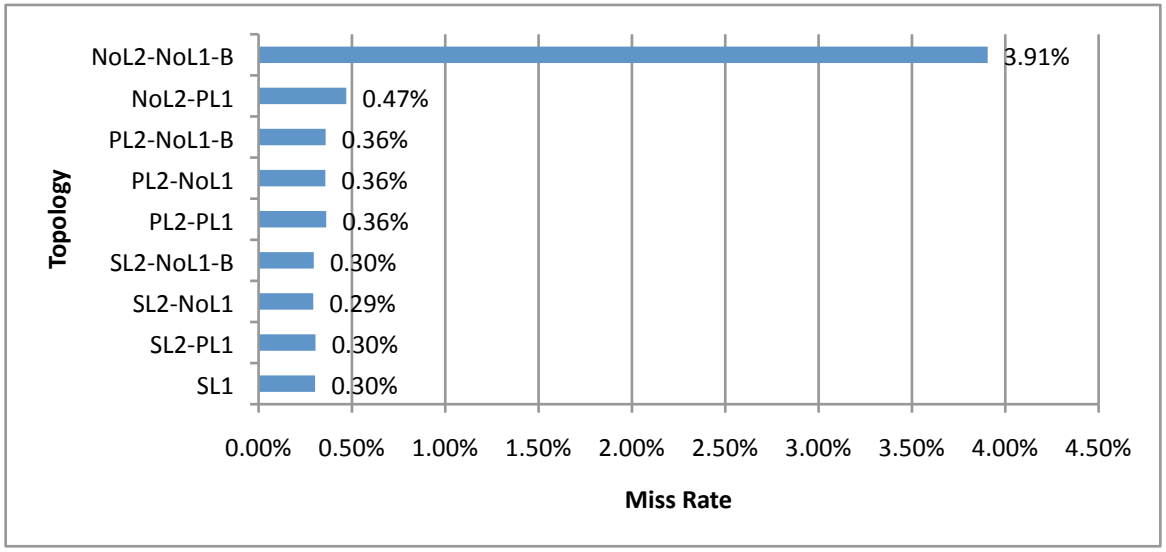


Figure 6.7 Overall cache miss rate for the Xvid application (combined RH/CPU accesses)

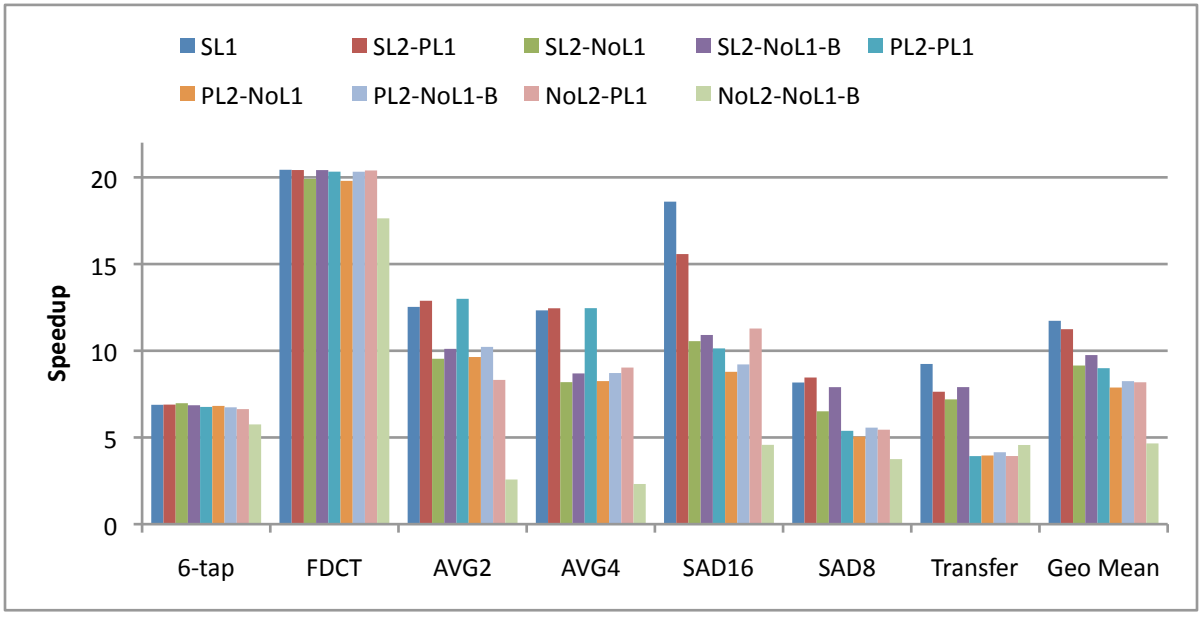


Figure 6.8 Speedup of Xvid kernels on the different topologies

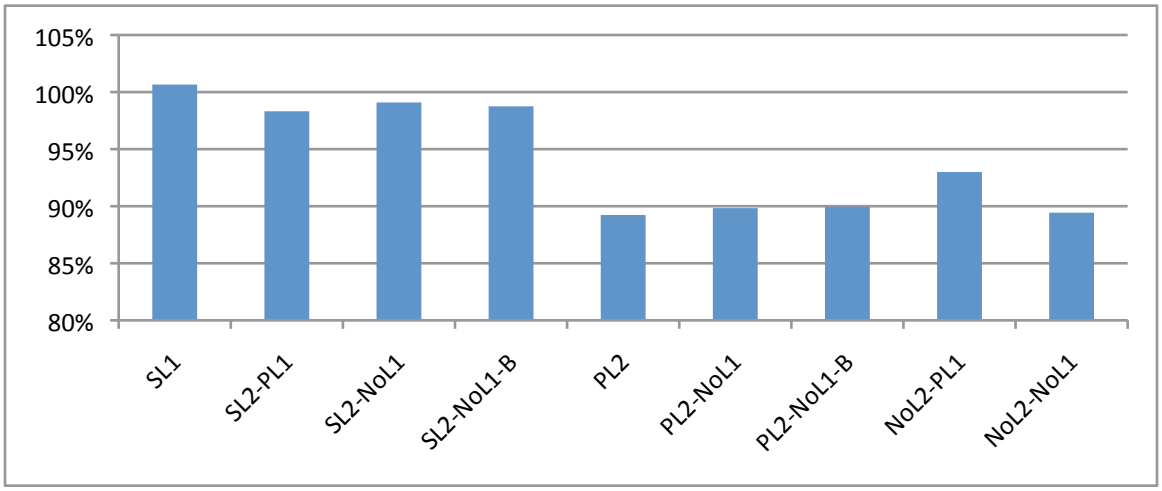


Figure 6.9 Cache topology's effect on non-kernel performance within Xvid.

Xvid performance varies noticeably across the different topologies. Although at a macro level, Xvid is a streaming application that processes a single frame of data at a time, there is still a great deal of data reuse when processing a given frame, and much of the processing is interleaved between the software and the RH. The high degree of CPU-RH communication results in higher performance for shared L2 topologies compared to those with private L2s or no RH L2. One interesting result, seen in Figure 6.6, is that the NoL2PL1 topology outperformed the PL2PL1 topology. To examine the cause of this unexpected outcome, the speedup of each individual kernel in the hybrid applications is first examined.

Figure 6.8 shows the speedups of each of Xvid's kernels on each of the examined cache topologies. This figure shows that the performance of most of the kernels was the same on many of the different topologies, although, a consistent drop in performance is seen for split topologies. Additionally, topologies where the RH has no cache performed significantly worse on most of the kernels. However nothing in the speedups of these kernels would suggest why the NoL2PL1 topology would outperform the PL2PL1 topology.

If the RH kernel's are not performing significantly better, than the difference in performance must be due to the non-kernel portions of Xvid's execution. Figure 6.9 shows the relative performance of the non-kernel portions of Xvid when compared to the same non-kernel portions of a SW-only execution of Xvid. In most of the topologies, the non-kernel portions of Xvid took longer to execute than they did in the SW-only execution of Xvid. The only exception to this is in the SL1 hierarchy. This is because the RH kernels in SL1 are loading the same working-set data into the CPU's (and RH's) L1 cache as the SW-only execution would; however the RH kernels do not have to load some of the unnecessary control data into the cache. The difference in performance between the non-kernel execution of Xvid in SW and SL1 is only .66%. On other topologies performance in the non-kernel parts of the benchmark was worse than the SW-only tests. This is because data written by the RH kernels that is requested in the non-kernel sections of

the application will not be found in the processor's L1 cache. Topologies with a shared L2 cache performed relatively similar to each other, although topologies without a full L1 cache outperformed those with one in the non-kernel sections. This is because the data written by the RH kernels is more likely to be found in the L2 cache when the CPU requests it. This saves time when transferring the data to the CPU, and results in the .78% performance improvement SL2NoL1 has over SL2PL1 in the non-kernel portions.

In the non-kernel sections of Xvid, topologies where the RH had a private L2 cache performed worse than those containing a single shared L2 cache. This is due to two factors, the size of the CPU's L2 cache, as well as the extra time required to transfer data between the RH's caches and the CPU's caches. The performance of the non-kernel portions of NoL2-PL1 explain why the overall performance of NoL2-PL1 outperformed PL2. In the accelerated version of Xvid, the non-kernel portions of the application dominate the application's runtime, and therefore a 4.22% performance improvement in the non-kernel section explains why NoL2PL1 outperforms PL2.

These results reinforce the notion that a shared L2 cache is important for application performance; the large amount of RH-CPU communication benefits from reduced communication latency and increased bandwidth. Private L2 caches make it much more difficult to retrieve data that is located in the other L2 cache, or its associated L1 caches. These results also show that using a shared L1 cache between the CPU and RH will result in the best overall performance. However, while this approach can be done in single-CPU scenarios, the scaling of such a topology to multiple processors is difficult. It would also require more dramatic changes to the CPU's architecture (as the CPU tends to load word-sized data from the cache while the RH kernels can request much larger sized data), and could potentially introduce additional latency to the L1 cache. Because the performance of the SL1 topology is not significantly better than SL1-PL1, these improvements are reduced further), for the work done in the remainder of this thesis, SL2-PL1 architectures will be used.

6.4 Cache Sizing

When considering using a shared cache hierarchy for transferring data between RH kernels and the CPU, it is important to examine how the new hybrid applications react to the sizing of the system's caches. This section examines the cache sizing's impact on both the RH kernel's execution, as well as on the SW applications. This work suggests that L1 cache size makes almost no difference on overall performance, and even for Xvid, SL2-NoL1-B only performed $\sim$3% worse than SL2-PL1. Therefore, I did not expect that changing the RH's private L1 cache size would greatly change overall performance. Figure 6.10 shows Xvid's EIPC as I varied the size of RH's L1 cache between 2KB and 128KB (in these experiments the CPU's L1 cache size was fixed at 32KB). This shows that Xvid's performance rapidly levels off when the RH's L1 cache is larger than 8KB in size. Increasing the cache's size to 128KB did not noticeably speedup Xvid, suggesting that the RH's L1 cache size may not need to be as large as the CPU's to obtain similar performance. However, it is unknown how the RH's L1 cache should be sized when it is shared amongst additional

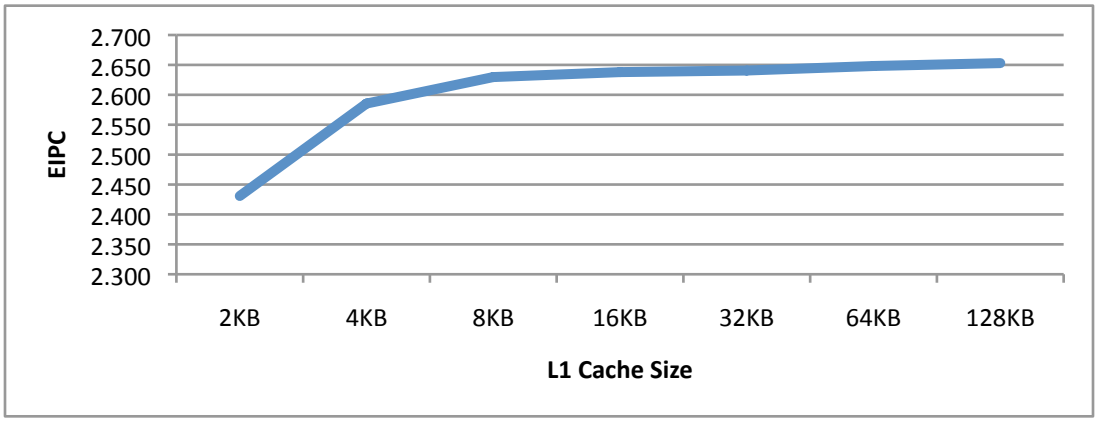


Figure 6.10 SL2-PL1 Xvid speedup as the RH's L1 cache size is varied

CPU cores that could execute multiple RH kernels simultaneously. The impact of the RH's cache size on the other workloads was also tested; however, little change in performance was seen as the RH's L1 cache size varied.

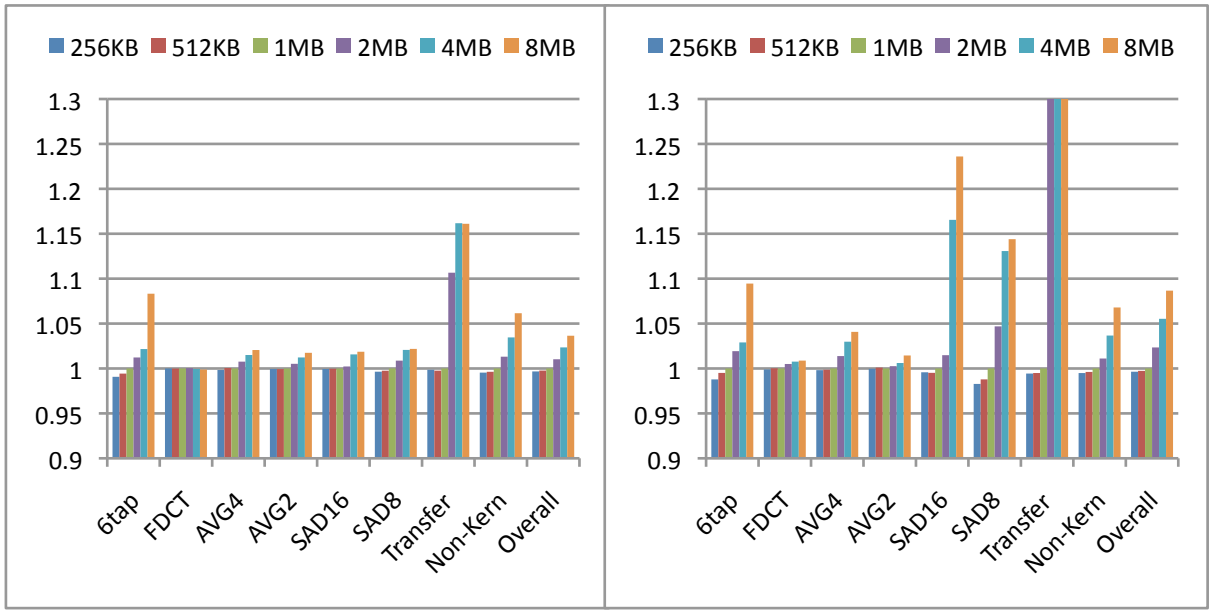
My experiments also suggest that the performance of the RH kernels and the SW-only portions of Xvid improve dramatically as the L2 cache is varied. In Split L2 topologies, performance of the non-kernel sections reduced significantly due to the "halving" of the CPU's L2 cache. To test the L2 cache's impact on performance, I varied the L2 cache size between 256KB and 8MB. On the AES, WiMaX, and Tremor workloads, I saw no noticeable change in the performance based on the L2 cache size. Therefore, this section will focus on Xvid, which makes extensive use of the processor's L2 cache.

Figure 6.11 shows the relative performance of each of Xvid's kernels, the non kernel portions, as well as the overall performance of the application as the size of the on-chip L2 cache was varied. The performance in these figures is normalized to that of a system containing a 1MB L2 cache. One surprising result of this study is that the hybrid RH/SW system was *more* sensitive to the L2 cache's size than SW-only execution. Increasing the L2 cache size from 1MB to 8MB increased Xvid's performance by 3.6% on SW-only systems, and by 8.6% on hybrid systems.

The reasons for this are two fold. Figure 6.11(a), shows that, with a few exceptions, the non-kernel portions of Xvid's execution are more sensitive to the cache size than most of the kernels. The only exceptions are the 6-tap kernel, and the transfer kernel. However, as shown in Chapter 5.1, the 6-tap kernel makes up $\sim$7.3% of overall execution time, and the transfer kernel only occupies 2.3% of execution time. Because the transfer kernel is the only kernel that performs significantly better than the non kernel portions when the cache size was increased, it is clear that the overall benefit obtained by increasing the L2 cache size is due to the non-kernel execution. However, on a SW-only version of Xvid (with a 2MB L2 cache), this accounts for only 31.4% of total execution, so the speedup of the full application's execution is limited.

Two factors greatly impacted the hybrid system's execution time: first the kernels were sped up by a greater degree as the L2 cache size was increased, and the non-kernel portions of the application made up a larger percent of overall

execution time. In particular, SAD16, SAD8, and the transfer kernel were greatly sped up by increasing the cache size. Two factors account for these kernel's performance increasing so significantly. First, these kernels execute for a very short time, so the RH's stream controllers do not have adequate time to prefetch data for them. Secondly, these kernels access a lot of data that is likely to be found in main memory. Both the hybrid and the SW-only execution of the kernels are likely to generate the same number of cache misses, however because the RH kernels base execution time is so much shorter, a fixed latency penalty is much larger (in terms of percentage of execution) for the RH kernels than for the SW kernels. The transfer 8 to 16 kernel also experiences a significant number of cache misses, both in the SW-only and the hybrid execution. Increasing the L2 cache size greatly increased the transfer kernel's performance, yielding a speedup of 1.48x with a 2MB L2 cache, and of just over 2x for systems with both 4MB and 8MB L2 caches.



(a) SW-only

(b) Hybrid RH/SW

Figure 6.11 Relative performance of SW-only and hybrid execution as the L2 cache size (with a baseline of a 2MB L2 cache) is varied

These results show that although the performance of RH kernels and applications are not heavily dependent on the size of the RH's L1 cache, the L2 cache's size can influence their execution to a greater degree. They also show that the L2 cache sizing influenced hybrid Xvid's performance more than it did the SW-only execution. This suggests that it is still important for hybrid RH/SW systems to maintain a large L2 cache, both for the purpose of legacy SW-only applications, and also to further accelerate hybrid RH/SW applications. However, due to the large die size of L2 caches, it might make sense for future reconfigurable computing systems to trade L2 cache size for increased reconfigurable computing resources. Although doing this would reduce the performance of hybrid applications, the

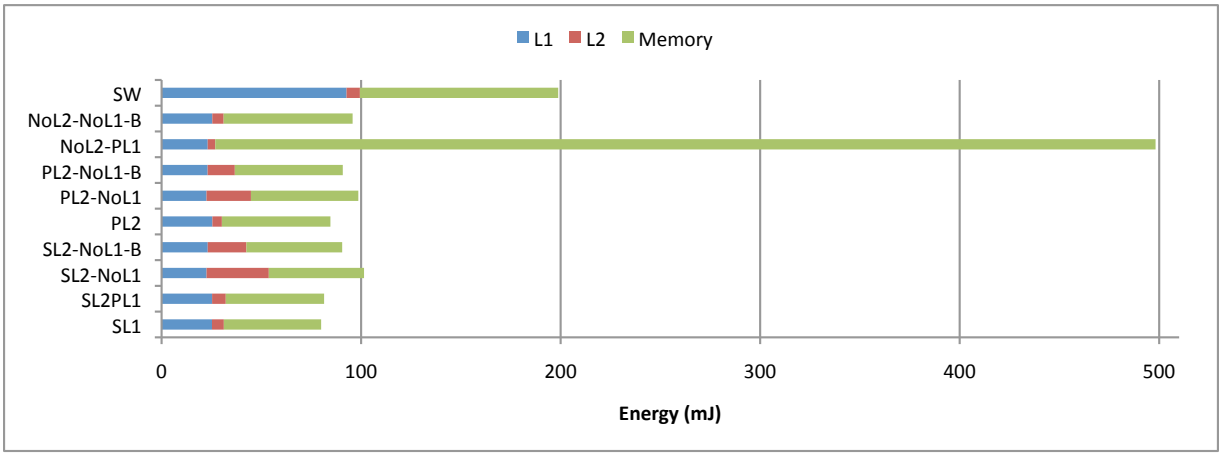


Figure 6.12 Energy consumption of the cache hierarchy for each of the tested topologies.

benefits of RH acceleration for hybrid applications far outweighs the performance losses that they would experience due to a smaller L2 cache size.

6.5 Power

Although system performance is a key metric when choosing a cache topology, energy consumption is also a consideration, particularly in embedded devices. Therefore, I modified the simulator to keep track of the cache hierarchy's energy consumption.

Cacti 5.3 [114] was used on a 65nm technology node to model the relative memory access energy for all of the caches used in the topologies as well as the spatial locality buffers. The L1 cache was configured with two read/write ports and the L2 had a single read/write port. Based on these parameters, a 2MB 16-way associative L2 cache access consumes almost 10 times the energy of a 32KB 4-way associative L1 access (.077nJ versus .769nJ). Accessing data from the DDR2 main-memory is even more expensive, consuming almost 50 times more energy than accessing the 2MB L2 cache(36nJ)[90]. These numbers suggest that, for power efficiency, it is important to obtain as much data as possible from small caches, and to minimize accesses to main memory.

I modified the simulator to keep track of memory accesses in an attempt to estimate the energy consumption of the memory hierarchy. In this model, every data memory access requires an L1 cache access except for those from RH cache topologies that lack an L1 data cache (SL2NoL1 for instance). Loads that missed in the L1 cache count as a read from the L1 plus a read from the L2, and a write to the L1 cache. However, due to limitations of the simulation infrastructure, it could not be determined whether a load that missed in the L1 cache was found in the L2 cache, or had to be read back from another CPU's L1 cache instead. However, errors accounting from this should be minimal, as the energy requirements for an additional L1 cache access are much smaller than those required for the L2 cache access.

Stores made into the cache hierarchy were a bit harder to measure. Upon a store L1 miss (that hit in the L2 cache), the energy of an L1 write was added, as well as an L2 read (as the data must be read back from the L2 cache). The simulator also counted the number of L1 writebacks, and each of these counted those as L2 writes.

The simulator counted load and store main memory requests to the DDR2 memory directly from the memory controller. Using these values, I calculated the total energy consumption of the memory hierarchy for each cache topology. These energy consumption values do not take into account the energy required to transmit data across the chip, and instead only focus on the access energy. Although this study does not measure all of the energy consumption in the cache topologies, they are more than adequate for showing the general trend in energy consumption for each of the topologies. Because each cache topology executed a different number of equivalent instructions, the energy consumption of each hierarchy was normalized relative to that required to execution two billion cycles of the SW-only version of Xvid.

Figure 6.12 shows the total dynamic energy consumption of the memory hierarchy for the full hybrid Xvid benchmark (CPU and RH) for each cache topology. Examining this figure shows that most of the topologies (with the exception of NoL2PL1 which required a tremendous number of memory requests) used less dynamic memory energy than the software when performing the same work. This is due in a large part to the greatly reduced number of requests to the L1 cache. It also shows that the RH should access an L1 cache, or even a spatial locality buffer if only to reduce the number of L2 cache accesses required. In this case, the SL2NoL1 topology used 20% more memory hierarchy energy to perform the same work as the SL2PL1 topology. Although some of this is due to the better performance of SL2PL1, the majority of this increase is from the increased number of L2 cache accesses.

These results suggest that a RH cache topology should contain an L1 cache to limit the dynamic energy used in the memory hierarchy. Having a 32KB L1 cache attached to the RH reduced the dynamic energy consumption of the memory hierarchy by 20%. Although this is nowhere near the amount that would be saved by a SW-only application (not having an L1 cache would *greatly* increase the energy consumption of such an application's execution), it is still a significant figure. Because an L1 cache is fairly small in comparison to an L2 cache (and because the RH does not need as large of an L1 cache), the RH kernels examined in this thesis should not directly access an L2 cache. Even when the kernel's L1 cache hit rates are low, as is the case for many of the RH kernels, an L1 cache reduces overall dynamic power of the system.

6.6 Reading Data Directly From The RH Kernels

Although the shared memory access for CPU-RH communication works well for long-latency data transfers initiated by the RH kernels, this is not always the best way to communicate data between the CPUs and RH kernels. As described in Chapter 3.5, the CPU can issue commands and requests to the RH kernels using an on chip data network. This allows for relatively low-latency communication between the CPU and RH when transferring small amounts of data between them. In examining the RH kernels, I realized that this network would also be ideal for

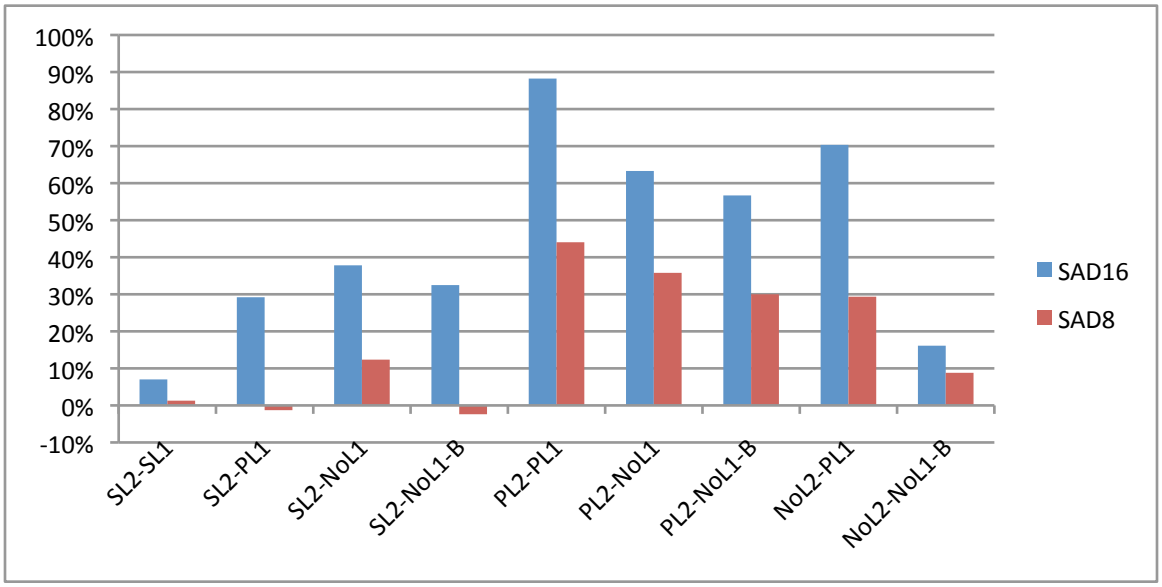


Figure 6.13 Performance improvements of SAD kernels when the CPU directly reads back values from them.

reading back single data values from an RH kernel. For instance, the SAD8 and SAD16 kernels used in the Xvid application (described in Chapter 5.1) read in multiple words of data (suitable for submitting memory requests directly to main-memory) but produce only a single word of data as a result. Writing a single word to a cache-coherent memory-hierarchy is relatively expensive (in terms of latency); especially if the data is being written to a cache line that is also present in another cache (this is likely because this data is used by both the RH kernel and the application software). Additionally, reading back this data from the software requires additional cache coherency transitions, as the RH's cache will contain an exclusive copy of the data. These latencies can greatly reduce the absolute performance of the SAD kernels.

Therefore I extended the pure shared-memory communication model to allow RH kernels to use an alternate form of communication with their host CPUs. I implemented this by attaching a single word of memory to each virtual RH kernel, and associating this buffer with the virtual kernel's memory segment. When an application reads back that the RH kernel is done executing, the application issues a memory request to read back this word of data, bypassing the cache coherency transitions required had the data been written into the cache hierarchy.

This mechanism is currently only used in the SAD8 and SAD16 kernels. All of the other kernels generate a larger amount of data, and therefore write it straight to the cache hierarchy. To test the impact of bypassing the shared memory subsystem in these two kernels, the Xvid benchmark were ran again on all of the tested topologies when the SAD kernels could read directly read back the resultant data. These simulations used the exact same checkpoints as the original runs, so the data can be directly compared to the tests where the CPUs had to read back the result values from the memory subsystem.

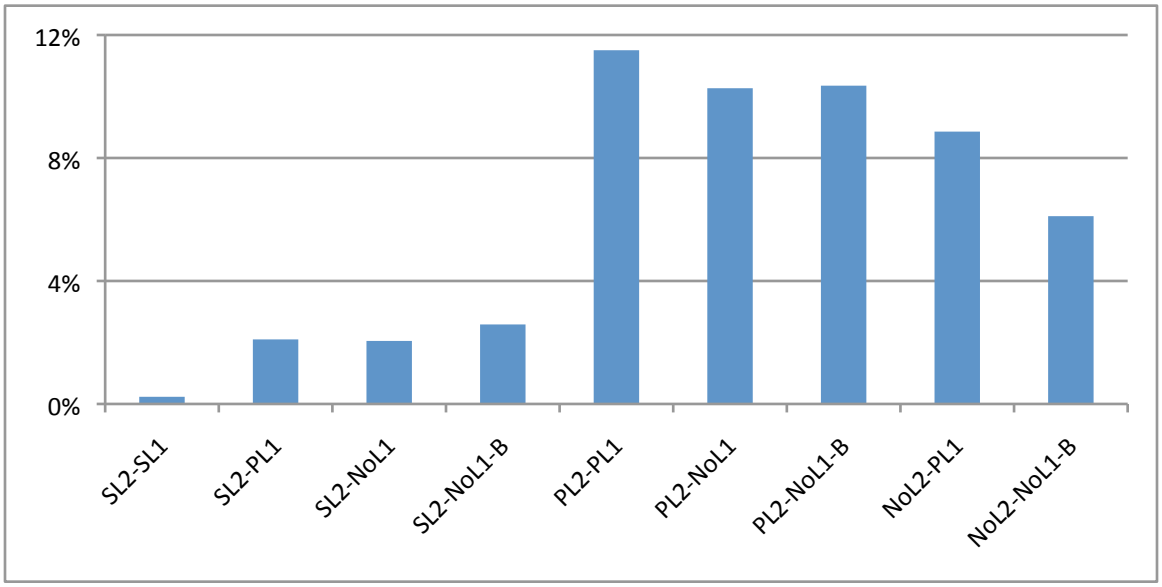


Figure 6.14 Overall Xvid performance improvements when the CPU directly reads back values from them.

Figure 6.13 shows the performance improvement seen in the SAD kernels when data was directly transferred. The direct performance improvements in these RH kernels is only part of the overall improvement in system performance. The non-kernel portions of the application are also able to execute faster, further speeding up the application. Examining these numbers shows a relatively moderate speedups on shared cache topologies, and much larger improvements on split topologies. This is because the cache transitions required to transfer data are much more expensive on split cache topologies. Reducing this latency will greatly reduce the kernel's runtimes.

Figure 6.14 shows the overall application performance improvement when the SAD kernels could directly transfer data to the CPUs. As expected, SL1 has almost identical performance in both cases (the cost of having to access the shared data from the CPU's own L1 cache is negligible). For most of the SL2 topologies, performance gains are $\sim$2%. On split topologies this increases dramatically, though still not enough to offset the drop in performance seen when using a split cache topology.

Because of this, it is recommend that kernel designers examine their kernels, and when appropriate, allow the applications to read singleton values directly from the RH kernels, bypassing the shared cache. This is particularly important when using "short" kernels that only execute for 100s of cycles. In the case of SAD8 and SAD16, the cache coherency transitions can, in certain instances dominate the runtime of the kernels. Avoiding this overhead is highly advisable, and this method allows the software to directly access the RH kernels with a minimum change to the programming model, and no changes to the data path that already exists between the RH kernels and the CPU(s).

6.7 Conclusion

This chapter examined the difference in memory accesses between RH kernels and general purpose processors. Using this knowledge, I examined the performance impact of the cache topology on hybrid RH/SW applications. Although the performance of some of the applications was relatively unaffected by the cache topology chosen, the performance of Xvid changed greatly based on the cache topology used.

In the examination of the cache topologies, I showed that using a topology containing a shared L2 cache was key for obtaining the best performance. This is due to a combination of the fact that the SW-only aspects of the application could make better usage of the L2 cache than the RH kernels, and because of the much smaller latencies incurred when accessing data in a shared L2 cache. I also showed that the performance of a hybrid RH/SW application did not decrease significantly as the size of the RH's L1 cache was reduced. Topologies where the RH connected to a 2KB spatial locality buffer performed similarly to those with a full 32KB L1 cache.

However, despite the fact that the RH kernels did not need an L1 cache to obtain similar performance to topologies containing one, the L1 cache was important, as it reduced the overall energy consumption of the memory hierarchy. It therefore seems to be a fair tradeoff to connect a reasonably sized L1 cache to the RH controller.

This chapter demonstrated that some of the applications can benefit from being able to read data directly back from the RH kernel. Bypassing the cache coherent memory hierarchy altogether for some memory accesses provides significant performance improvements to some of the RH kernels. Although these benefits are greatest on cache topologies containing private L2 caches (or where the RH does not connect to an L2 cache), a 2% application performance improvement was still obtained on a cache topology with a shared L2 cache, and private (32KB) L1 caches.

Because of these observations, I use the SL2-PL1 cache topology in the rest of the work performed in this thesis. If an application is likely to perform its best on this topology on single processor systems, it is reasonable to assume that on a multiprocessor system, this topology would also be the best. Additionally, throughout the rest of this thesis, the Xvid benchmark will always use the on-chip CPU-RH network to read back data from the SAD8 and SAD16 kernels, bypassing the cache hierarchy altogether.

Chapter 7

Scaling to Multicore Systems

This chapter builds upon the work in Chapter 6, extending the model of reconfigurable computing to devices that contain multicore processors. This chapter first examines how the memory subsystem's performance scales as more processors share the RH fabric. It then examines tradeoffs in the memory sub-system design, including the ability of the RH controller's TLB to translate memory addresses on multicore platforms. The chapter then examines how the performance of the RH coprocessor's memory interface scales when an increasing number of CPU cores shares the RH coprocessor.

Once the performance of hybrid RH/SW applications on this multi-core system is establish, this chapter examines the performance of software-only applications that execute alongside multiple hybrid RH/SW applications. Hybrid systems will likely need to execute legacy software-only applications. Furthermore, even though the runtime of an application may be dominated by one or more kernels in the original software version, when accelerated by RH, these kernels take less time to execute, and thus represent a smaller percentage of the accelerated application execution time. Thus it is important to know to what degree the RH kernels' memory requests hamper the performance of software code. In future systems, an RH-aware OS scheduler could even use this information when deciding both what applications to run simultaneously, as well as how much CPU time should be allocated to each application [42].

It is anticipated that software code will perform worse when executed alongside hybrid RH/SW applications than when executed only with other software code. Accelerated kernels require accelerated access to memory, potentially evicting other application's data from shared caches. I examine the impact of hybrid RH/SW applications on software-only applications in Chapter 7.3.2 and propose a mechanism to limit the impact that high-bandwidth streaming RH kernels have on system performance are then proposed and evaluated.

7.1 Prior Work

The best way to couple multiple processor cores with an on-chip reconfigurable hardware (RH) coprocessor has not been examined extensively. Watkins et. al [124] proposed a reconfigurable multicore architecture that shared a specialized programmable logic (SPL) fabric between multiple cores. However, in this work, cores communicated with the SPLs using dedicated FIFOs, and did not directly share memory. Although this work discussed multiple ways

Benchmark	two-cores												four-cores							eight-cores						
Xvid	2	0	1	1	1	0	0	0	0	1	0	0	4	0	2	1	3	0	1	8	0	2	4	7	3	0
AES	0	2	0	1	0	1	1	0	1	0	0	0	0	4	2	1	0	3	1	0	8	4	4	0	2	7
Tremor	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0	0	1	0
WiMaX	0	0	1	0	0	1	0	1	0	0	0	1	0	0	0	1	0	0	1	0	0	1	0	0	1	0
SW-only	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	0	0	0	0	1	1	1

Table 7.1 A listing of the benchmarks executed in each of the executed workloads for two-, four-, and eight-core systems.

to use their communication primitives, it did not discuss the speed of their interface, or how the interface scaled with the number of processor cores.

Chen et. al [26] proposes a multicore reconfigurable ISA, where a separate RH unit is directly attached to each processor core. The dual-core system thus acts like two separate reconfigurable processors, so the communication between a processor and RH is no different than on a single- core system. Similarly, Williams et. al [128] and Syed et al [113] created a multicore reconfigurable computing system the used the message passing interface (MPI) to facilitate communication between processor cores. This work implemented an MPI system on a reconfigurable fabric, however this work did not examine the use of MPI from an RH accelerator (although, an RH kernel could technically use the interface). Additionally, the processors did not share the RH fabric, and the system allocated the RH fabric statically at design time.

In Yan et. al [132], a reconfigurable multicore processor is also examined. In this work, a multicore reconfigurable computing system is designed. This system contains multiple reconfigurable processor units are combined with multiple processor cores. However, communication is handled using local buffers, and no data is given on the efficiency of their communication infrastructure.

7.2 Application Workloads

This chapter uses a combination of the hybrid RH/SW and the SW-only benchmarks described in Chapter 5.1 to create workloads that will run on two-, four-, and eight-core systems. Each workload contains as many benchmarks as there are processor cores to create realistic workloads for high-performance embedded systems. I simulated these workloads to determine how the system's performance scales with an increasing number of processors.

Table 7.1 shows the number of copies of each hybrid benchmark that were in each of the workloads that I tested. Cells with a zero in them imply that no copy of that benchmark was executed. Columns that include the SW-only benchmarks indicate that I created three workloads; each containing one of the SW- only benchmarks, and all of the workloads included the other hybrid RH/SW benchmarks. I refer to workloads to by the number of copies of each

Description	Area mm^2	Read Energy (nJ)
32-entry fully-associative TLB	.031	.015
64-entry fully-associative TLB	.053	.025
128-entry fully-associative TLB	.101	.050
256-entry fully-associative TLB	.186	.090
32KB 4-way associative L1 cache	.550	.057

Table 7.2 Area and read energy for different TLB and cache configurations

benchmark followed by the first letter of the benchmark, with a hyphen to separate the benchmarks. Therefore 2X-4A-1T-1W refers to a workload containing two copies of Xvid, four copies of AES, one copy of Tremor, and one copy of WiMaX.

7.3 Results

Because this work's goal is to determine how the performance of our reconfigurable computing platform scales with an increasing number of RH kernels, I attempted to maximize the impact of the RH kernels on the memory subsystem. To achieve this, my system has sufficient RH tiles available to fully accelerate every RH kernel requested by the workloads. In later chapters of this thesis, I will examine how the OS can make better usage of limited RH resources on multicore systems. The workloads examined in this chapter are all executed for two-billion processor cycles on all of the different system setups

7.3.1 Hardware TLB Miss Handler

In [43] I proposed a method for handling TLB misses in the RH controller. This method involved interrupting the processor to obtain virtual memory addresses, and is described in Chapter 4.5. In that work, whenever a memory request's virtual address could not be found in the RH kernel's TLB, the RH controller would interrupt the CPU to obtain a translation. However, this work was originally performed on a single processor platform, and thus it was unclear how this method would scale to multiple processor systems.

When I extended my platform to use with multicore processors, some workloads had extremely poor performance due to TLB thrashing. In examining the bevavior, I saw that interrupting a CPU to provide an address translation is a relatively expensive operation, taking about 400 cycles on most workloads (in workloads with many TLB misses, this number is reduced due to caching effects). This is an order of magnitude longer than the time it takes to handle software TLB misses on the platform (approximately 20-40 cycles). Because of the problems associated with TLB thrashing, I found that the sizing, and policy for filling the RH controller's TLB can be very important to the performance of hybrid RH/SW applications.

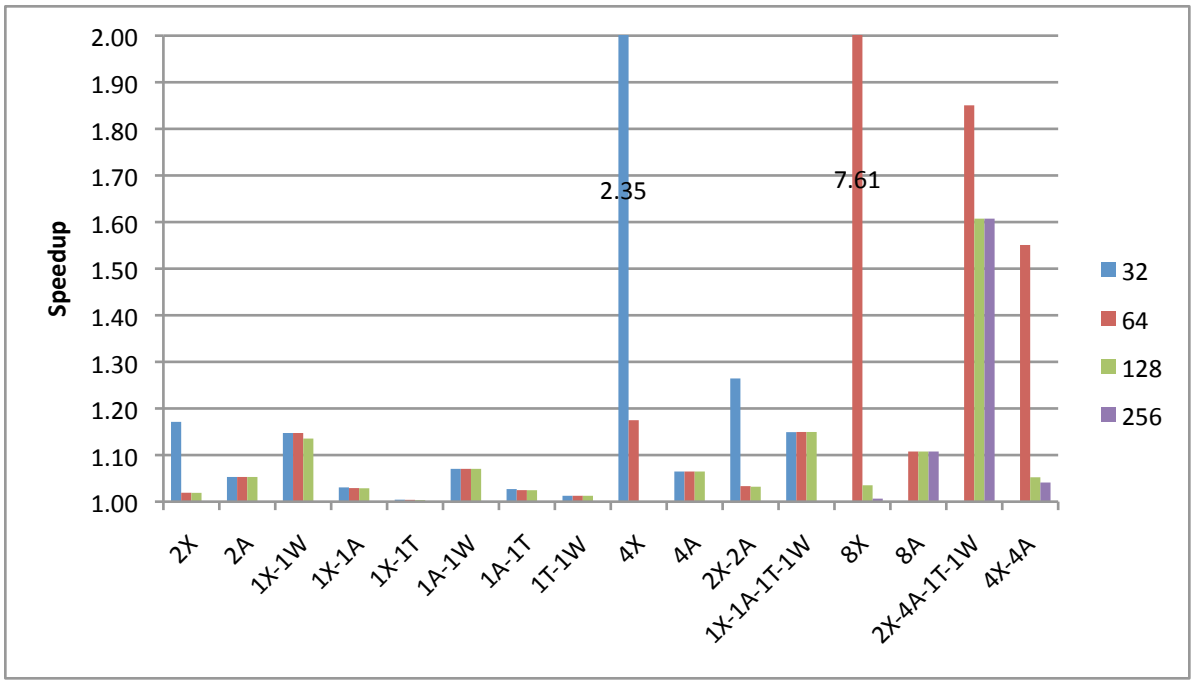


Figure 7.1 Workload speedup using proposed HW-based TLB miss handler instead of interrupting the CPU

Although larger TLBs may avoid thrashing on these specific workloads, larger TLBs are more expensive, and future application mixes could still cause TLB thrashing. To demonstrate the tradeoffs involved in selecting different sized TLBs, I used Cacti 5.3 [114] to model the caches and TLBs (at 65nm) used in the system. Table 7.2 shows both the read energy and area of various sized TLBs as well as the 32KB L1 cache for comparison. This table shows that a 256-entry fully associative TLB is 6 times bigger, and consumes 6 times more energy per access than a 32-entry fully-associative TLB. Furthermore, a lookup in a 256-entry TLB requires 1.58x the power of an L1 cache (32KB, 4-way associative) access, and occupies 34% of the area of the 32KB, 4-way associative L1 data caches. These results indicate that increasing the TLB size is not a good solution to prevent TLB thrashing.

Instead of just increasing the size of the RH controller's TLB, I added a hardware TLB miss handler to the RH controller to improve the performance of the RH controller's memory subsystem. On the simulated UltraSparc platform, I modeled a hardware structure that could access the (already existing) translation storage buffer (TSB) lookup from memory. The TSB is a software construct resembling a very-large direct-mapped TLB. The simplistic structure of the TSB makes the hardware to calculate the new address very simple. The TSB can be up to 1MB in size, allowing it to hold 64K entries, enough to cover 512MB of an application's virtual address space. In the (relatively rare) event that the address cannot be translated in the TSB, the RH controller interrupted the CPU to perform the address translation. Although TSBs are not present on many other computer ISAs, hardware pagetable walks can be implemented on most of these platforms [68, 64]. By implementing the hardware TSB lookup, I reduced the average latency of a TLB miss

to between 10 and 40 cycles, depending on the workload (workloads with higher TLB hit rates tend to take longer to perform a TSB translation and vice-versa).

All of the results reported (including those already reported in Chapter 6) use the hardware TSB miss handler. Because the hardware TSB lookups have a much greater impact on the performance of multicore systems, I do not examine the performance advantage of using the hardware TLB midd handler on single-processor systems.

Figure 7.1 shows the performance advantage of using the HW TLB miss handler for systems containing TLBs of various sizes across all of the hybrid RH/SW workloads. Workloads that used the new HW TLB miss handler always outperformed those executing on a system that used the software-based interrupt routine. Workloads containing WiMaX in particular performed better with a HW TLB miss handler. This is because WiMaX is a multithreaded application, and the OS must regularly swap out WiMaX threads. When this happens, the OS flushes the RH controllers TLB, which in turn causes a sequence of compulsory TLB misses. Many workloads that contained the WiMaX benchmark saw no benefit at all from increasing the size of the TLB because the OS would flush the TLB so frequently that it could never be filled.

A couple of the workloads presented had extremely poor performance when not using the hardware TLB miss handler. In particular, the 4X workload performed poorly on a system with a 32-entry TLB, and the 8X workload performed very poorly when executing on a system with a 64-entry workload. In the later case, the performance was so bad that the Xvid applications actually performed worse than the SW-only baseline. In these two cases significant TLB thrashing occurred. By enabling the hardware TSB lookup mechanism, the applications ran much closer to their theoretical maximum performance as will be shown in the next section.

For most workloads, the speedup of using the hardware TSB lookup mechanism ranged between 1% and 15%. Because the hardware TSB lookup is a relatively minor addition to the reconfigurable computing platform proposed in this work, it is highly recommended that future reconfigurable computing systems where RH kernels directly access virtual memory have a fast method to translate memory addresses; only relying on the OS to perform the translation in rare circumstances.

7.3.2 Multiprocessor Performance

This section determines how efficient the memory subsystem in the reconfigurable computing platform outlined in Chapter 3 is. In particular it compares the platform against one containing an ideal RH memory subsystem. In the platform containing a ideal RH memory subsystem all RH kernel memory requests (including those generated by the HW TLB miss handler) were serviced in a single processor clock cycle. The ideal platform also contained a very large TLB, and used the hardware TLB miss handler described in Chapter 7.3.1.

On the ideal platform, each time a RH kernel issues a memory request, a prefetch for the data was issued to the memory subsystem. Although the data is immediately written to the RH kernel, these prefetch requests operate like normal cache requests. Therefore, these prefetch requests acted to preload data that might by used by the CPU at a

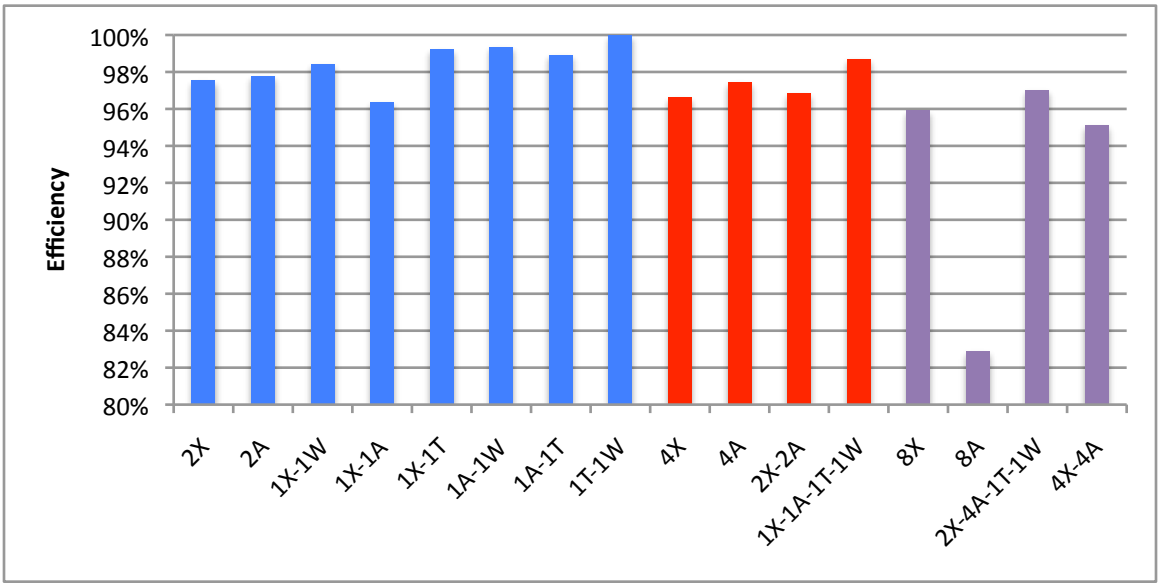


Figure 7.2 The "efficiency" of the hybrid RH/SW workloads on the proposed reconfigurable computing platform

later time. This is useful because most of the hybrid RH/SW benchmarks communicate with their host processor using shared memory, and data written by an RH kernel is likely to be read by the processor in the near future. Issuing these prefetches helps to ensure that the SW portions of hybrid applications have good performance. In this ideal platform's baseline, the performance of the CPUs memory hierarchy is not ideal, and CPU-initiated memory requests (including those needed to access the RH controller) have their usual latency. It is important to note that this "ideal" system still observes the same latency when initiating requests with the RH kernels. All communication over the network described in Chapter 3.5 occurs at the same speed for both the normal, and ideal systems.

I used the performance of this "ideal" workload to calculate the "efficiency" of the workload, which is the ratio of the actual performance of the workload to that of the workload containing an "ideal" memory system. Equation 7.1 shows how I computed this value.

$$\textbf{Efficiency} = \frac{NEIPC_{workload}}{NEIPC_{ideal}} \tag{7.1}$$

Figure 4 shows the efficiency of the hybrid workloads running on the RH computing platform. Performance of the RH system described in Chapter 3 was very good; approaching that of a system where the RH has immediate access to the memory hierarchy. Most of the workloads obtained between 95% and 99.9% of the ideal limit performance for this shared memory organization, with the only exception being the 8A workload.

When examining the 8A workload, it becomes apparent that the bottleneck was not the RH controller's memory subsystem, but rather the inability of the chip's memory subsystem to supply the RH kernels with data. This issue would occur on *a*ny RH computing platform that using the same main-memory structure. To overcome this slowdown

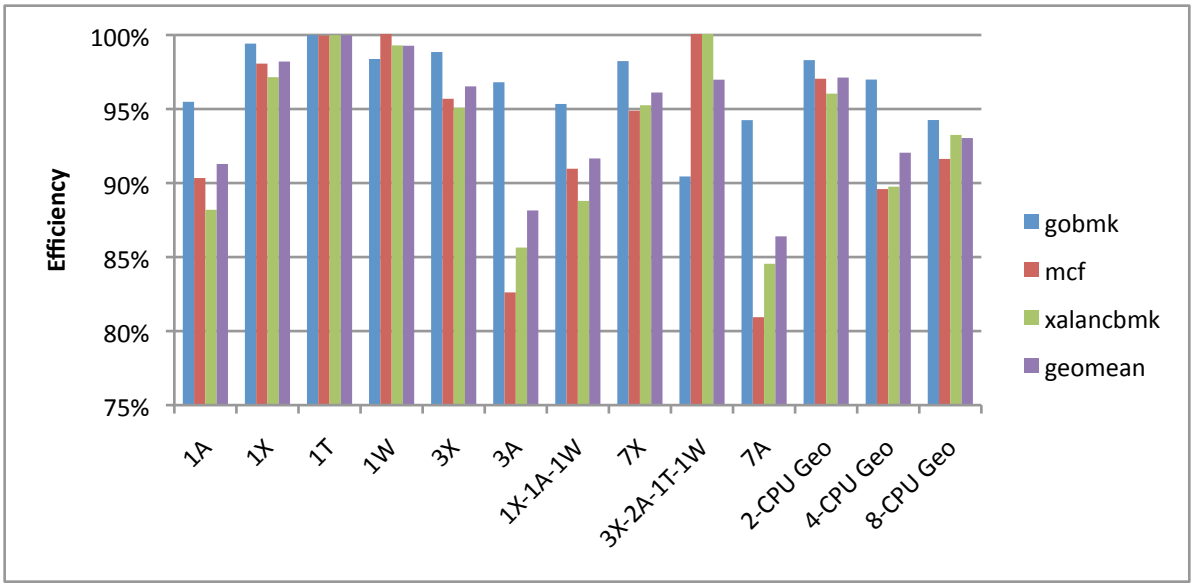


Figure 7.3 SW-only benchmark "efficiency" when run alongside hybrid RH/SW workloads

in performance, the system would need a higher performance memory subsystem with potentially lower main-memory latency, greater bandwidth, and/or a more-optimized memory controller. However, in this study, the 8A workload was intended to represent peak streaming capability, and was not necessarily representative of the data patterns observed in real workloads.

With the exception of workloads containing only AES benchmarks, no further degradation of performance was seen as the number of processors was increased from two to eight. This suggests that the RH memory interface can scale to at least eight CPU cores without a problem, illustrating that the same RH controller used on a single processor system can also be used on multicore systems.

7.3.3 Impact on Software-Only Applications

The final multicore study of this chapter examines the impact that hybrid applications have when run alongside traditional SW-only applications. The performance of these SW-only applications is important not only on legacy systems that are executing hybrid applications alongside SW-only applications, but also on systems that exclusively run hybrid applications. This is because non-kernel code often dominates the runtime of accelerated hybrid applications.

To determine this impact, I calculated the EIPC of each SW-only benchmark both when executing alongside fully accelerated hybrid RH/SW applications, as well as when the hybrid applications executed entirely in software. The ratio of these two EIPC values represents the efficiency of the software-only benchmarks. Unlike previous experiments, these tests are *not* using the normalized EIPC values when comparing the speed of the SW-only application.

The primary reason for using the EIPC for measuring performance is because of the behavior of the WiMaX hybrid RH/SW benchmark. In calculating the NEIPC value, the overhead of the OS is distributed equally amongst all applications. This works well when applications are given different time slices by the OS, and when the vast majority of the time spent in the OS is "overhead" that should be charged to the system as a whole. However, in the case of the WiMaX benchmark, this assumption no longer holds true. Due to the multithreaded nature of this benchmark, much time is spent swapping threads and handling some of the synchronization between the threads. This overhead can be directly attributed to the WiMaX application's execution, and therefore should not be evenly distributed amongst all of the applications.

If I distributed this overhead amongst all of the applications (as the NEIPC metric does), then, if using the NEIPC metric, it would appear that *every* application on the system performed worse because the WiMaX application spent more time in the OS. All of the SW-only benchmarks executing alongside WiMaX executed for approximately the same number of cycles regardless of whether WiMaX was accelerated. However, the calculated NEIPC value when executing a RH-accelerated version of WiMaX was much lower due to WiMaX spending more time in the OS. To avoid this problem, I use the EIPC metric to measure the performance of the SW-only application in these workloads. This provides a more accurate evaluation of the SW-only applications' performance.

Figure 7.3 shows the efficiency of the SW-only workloads run alongside hybrid RH/SW applications. The SW-only applications perform worse when run alongside hybrid RH/SW applications. Unsurprisingly, these SW-only workloads performed even worse as the number of the hybrid RH/SW applications executing on the system increased. In particular, SW-only applications perform worse when run alongside workloads containing the AES benchmark. The AES benchmark continuously reads and writes large blocks of memory, overwriting potentially useful data located in the L2 cache. When accelerated, the AES application reads and writes significantly more data in a given span of time, exacerbating L2 cache evictions.

Because the AES kernel has little temporal cache locality, it is not useful for the processor to cache the data that the AES kernel accesses in the L2 cache. Therefore an extension to the reconfigurable computing system was made that allowed input or output streams from an RH kernel to declare themselves to be "streaming" with no temporal locality. The system marks all cache lines accessed from theses streams with a flag that tells the cache controller to evict those lines first if an eviction is necessary. This modification is relatively simple, only requiring one extra bit to be sent to the cache. When this bit is set, the cache does not update the least recently used field of the cache, causing the cache to think that the line is still the oldest line in its associativity set, and thus the one that should be evicted if needed. Applications can inform the RH controller that an input or output stream is "non-cacheable" by marking a field in the virtual kernel's memory segment. Only the AES kernel used the non-cacheable flag is only used because all of the other kernels have some form of temporal locality, and thus, their performance would be degraded by using this flag.

General-purpose processors often contain special prefetch instructions that inform the processor that the data has little temporal locality, and can be flushed. Unlike prefetch instructions in traditional computer systems, the streaming

flag in the RH coprocessor is associated with all loads from a given stream and do not require additional instructions that could potentially slow down execution. Additionally, because RH coprocessors have accelerated memory accesses, the usage of a streaming flag is even more important than on general-purpose processors.

Figure 7.4 shows the performance improvement of the SW-only thread when the AES kernel's memory accesses are non-cacheable. Using this flag greatly creased the performance of SW-only applications; for example, in the case of SW benchmarks running alongside seven copies of RH-accelerated AES, performance improved by almost 30%. In some cases this was actually better than the performance of the SW-only benchmark when the AES applications executed only in software, where it would still cause unnecessary evictions, but at a slower rate than the accelerated version.

In this graph, the 3X-2A-1T-1W workload seems strange when mcf is running. At first glance, it appears that enabling the non-cacheable flag actually decreased the workloads performance, as mcf performed worse than when this flag wasn't enabled. However, the reason for this can be explained by examining the performance of the hybrid RH/SW applications in the workload. This performance can be seen in Figure 7.5. In the case of the 3X-2A-1T-1W workload running with mcf, the performance of the hybrid applications increased by over 2% due to the superior performance of the hybrid Xvid applications. Because the Xvid applications could perform better, more pressure was put on main memory, increasing latency. Additionally, the Xvid applications used up more of the processor's shared L2 cache.

Another advantage of using the non-cacheable flag is that it did not adversely impact the performance of the AES benchmarks. In all of the runs, AES had nearly identical performance regardless of whether it's data was cached. I also observed that using the streaming flag also accelerated, to a lesser degree, other hybrid RH/SW applications executing on the system. These performance numbers show that application developers should consider the impact of their RH kernels on other applications, and if possible, annotate RH kernel data streams as streaming or streaming, as appropriate. Doing this can, in the extreme case, improve the performance of SW-only applications by as much as 30%.

7.4 Scalability Conclusions

This chapter examined how the performance of my proposed reconfigurable computing system scales with the number of processor cores using it. It also examined some of the issues encountered when scaling to eight processor cores.

The results presented in this chapter have shown that although using the OS to obtain TLB translations was sufficient on a single processor system, hardware translation techniques are necessary on multicore systems to obtain high performance. When using hardware to perform the TLB translations, the workloads executing on the system performed quite well. Even on eight-cpu systems, most of our workloads performed within 95% of their ideal performance. This shows that the shared memory model used in our RH coprocessor scales well to eight cores, and that

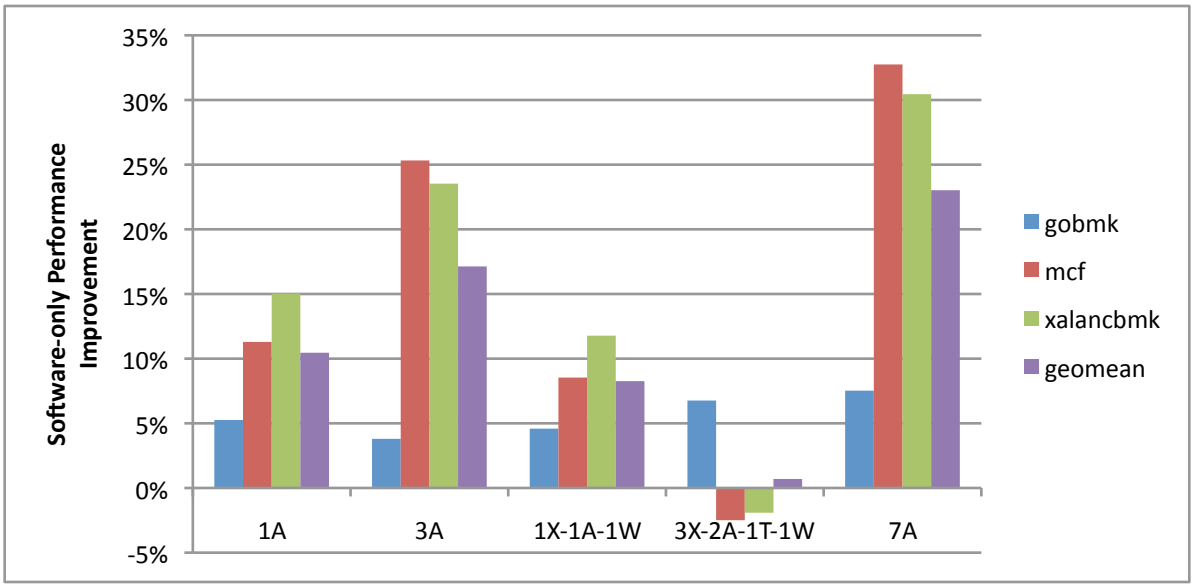


Figure 7.4 Performance improvement of SW-only application in workloads workloads where AES is set to be non-cacheable

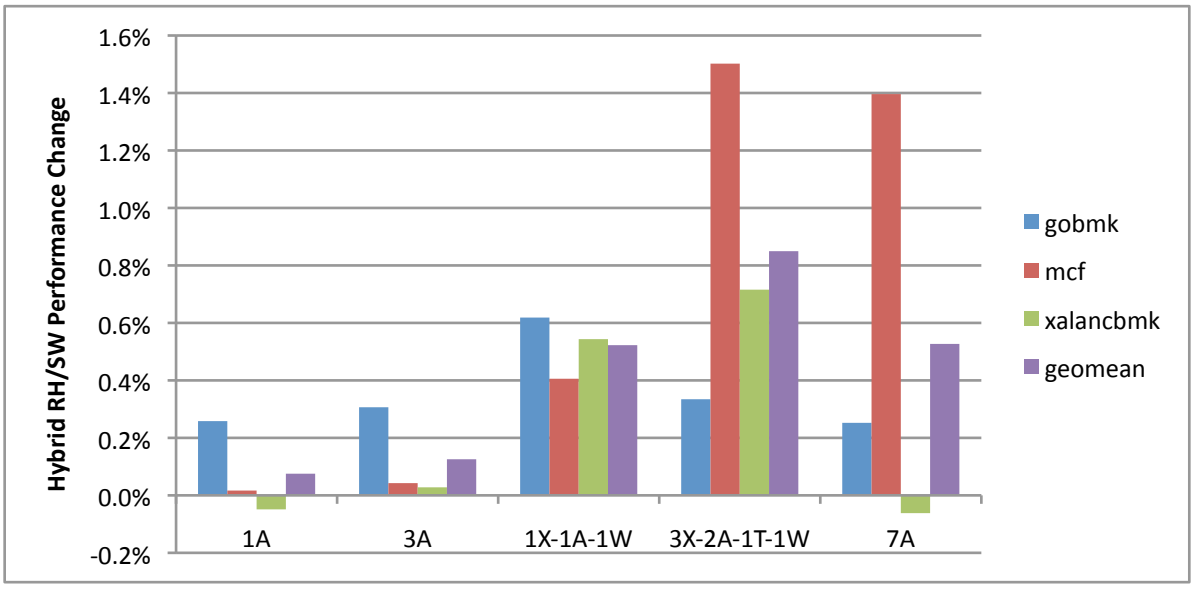


Figure 7.5 Performance change of the hybrid RH/SW applications when AES is set to be non-cacheable

the ability of RH kernels to access the memory subsystem is not a limiting factor. This is important not only for our system, but also future accelerator platforms that might use a shared memory model.

Additionally, we have shown that streaming RH kernels can greatly impact the performance of other applications running on other processors in the system. However, if the loads and stores issued from these RH kernels are marked as non-cacheable, the impact of these kernels on SW-only systems can be greatly reduced. Future RH computing systems should therefore implement a policy to prevent kernels with little cache locality from overwriting useful data in the processor's cache.

Overall, this chapter demonstrated the usefulness of the proposed multicore reconfigurable computing system. Our proposed coprocessor can accelerate multiple applications almost as efficiently as it can a single one. Additionally, due to the reduced rate of memory accesses from RH kernels, the memory subsystem can easily handle the added load of additional processor cores. This helps to prove the viability of our coprocessor, illustrating that a shared memory coprocessor can have high performance, even when multiple processors are all executing short-running kernels simultaneously.

Chapter 8

Reconfigurable Computing on Simultaneous Multithreaded Processors

Previous chapters of this thesis have dealt with the design of an RH coprocessor, and examined its performance on multicore systems. Although multicore systems have become common, it is also important to examine how this shared reconfigurable coprocessor performs on other multithreaded platforms. This chapter examines the performance of SW-only applications coscheduled alongside hybrid RH/SW applications on simultaneous multithreaded (SMT) processors. Additionally, it proposes extensions to the processor to more efficiently share the processor core's resources, helping to balance the performance of hybrid threads with SW-only threads.

8.1 Prior Work

SMT processors differ from more commonly found CMP processors because an SMT system allows multiple threads of execution to share a single processor core's resources [116, 118, 36]. By sharing many of the resources contained within an out-of-order superscalar processor amongst multiple threads, greater throughput can be obtained without a major increase in the size of the processor cores [76, 116, 118, 36]. Additionally, SMT processors obtain high performance of single-threaded applications (when running in isolation), as well as good performance of multithreaded applications.

In a typical superscalar processor, many of the processor core's functional units are idle at any given time. This occurs both because an application's instruction mix rarely utilizes all of the processor's functional units, and because long-latency events, such as cache misses, delay the processor from executing subsequent instructions that are dependent on the long-latency instruction. SMT processors allow additional threads to use functional units (as well as other processor resources) that are not currently being used by other instructions. Although each thread executing on an SMT processor cannot operate as fast as it could if executing alone on the processor core, the aggregate performance amongst all threads is higher.

Although prior work extensively examined SMT platforms, little work has been done to combine these platforms with reconfigurable computing. Uhrig et. al [119] integrated a MOLEN-based [79] reconfigurable coprocessor on a real-time SMT system, focusing on allowing the prioritization of memory requests from the RH coprocessor. However this work did not analyze the performance impact of sharing the processor resources between SW-only applications

and hybrid RH/SW applications. Mamidi et. al [85] augmented an interleaved multithreaded processor with a reconfigurable functional unit. This work focused on software-defined radio applications, and the use of the RH fabric to accelerate tasks within them. Although it provided mechanisms for different threads to access the RH, results were only presented for single applications executing at a time.

A survey described the existing models for multithreaded reconfigurable computing exist, OS support for the architectures, and methods for scheduling tasks to the RH [134]. Although this survey provides a broad overview, it did not examine the performance of both hybrid applications or SW-only applications coscheduled alongside hybrid applications.

8.2 Execution of Hybrid RH/SW workloads on SMT Processors

Due to the ability of SMT processors to share processor resources, they seem like an ideal match for the reconfigurable computing applications examined in this thesis. In these applications, a thread executing an RH kernel waits in a spin loop until the kernel has finished executing. On processor cores that can only execute a single-thread at a time, this results in wasted processor time and under-utilized resources. Although the currently active thread could be swapped out during an RH kernel's execution, this would not be feasible during the execution of most our RH kernels because the kernels' RH runtimes are often less than the time required for the OS to perform a full context switch [109]. Because a hybrid thread requires few of the processor's functional units during its execution, I anticipate that coscheduled threads should perform better than if the SW-only thread was scheduled alongside another SW-only threads.

I configured the SMT processor used in this study exactly like the single-threaded processor core described in Chapter 3, but with two copies of the thread state (register file, etc). In this SMT processor, both threads share the 32-entry instruction window, and are each guaranteed at least one entry in the window. The processor's fetch unit fetches new instruction from from the thread containing the fewest number of entries in the shared instruction window. However, if only one of the threads can fetch instructions in a given cycle, that thread will be granted access to the core's fetch unit, regardless of how many entires in the instruction window it is using.

This study executes each of the hybrid benchmarks alongside each of the SW-only benchmarks, examining a total of twelve different workloads. I executed each workload (both in SW-only mode as well as hybrid RH/SW mode) on both a two-threaded SMT processor, and a dual-core CMP system, and use multiple different comparison metrics to relate the performance of both the coscheduled SW-only thread, as well as that of the entire workload.

The first measurement used will be referred to as the efficiency of the coscheduled SW-only thread, which is the ratio of the EIPC of the coscheduled SW-only thread on the SMT processor to its performance on the CMP processor[1]. Figure 8.1 shows the cosecheduled threads' efficiency for each of the workloads, both when everything

[1]The EIPC is used when comparing single thread performance in a multithreaded workload for the same reasons given in Chapter 7.3.3.

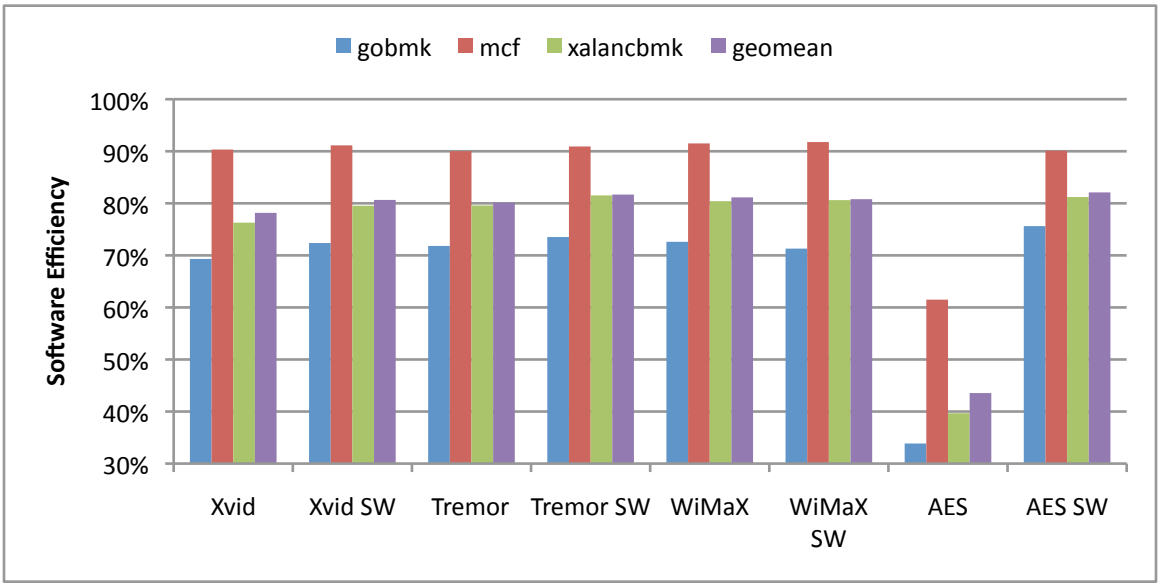


Figure 8.1 Efficiency of the coscheduled SW-only thread's performance

runs in software, and when accelerating all of the hybrid application's RH kernels. The second measurement computes the overall workload efficiency by calculating the ratio of the geometric means of the NEIPC of the workloads on the SMT processor to that on the CMP processor. This measurement is shown in Figure 8.2.

These graphs show an unexpected trend; despite the fact that AES spends 99.9% of its execution time in an RH kernel, SW-only threads coscheduled alongside AES perform far worse than those coscheduled alongside other applications. I expected that SW-only threads running alongside a thread in an RH kernel would have performance similar to that when the thread runs on its own private processor core, because the hybrid thread will not make use of the core's functional units. Therefore, I decided to further analyze of the workloads containing the AES benchmark.

Although the AES thread does not use the processor's shared execution units often, it could still occupy a large portion of the processor's shared instruction window. Figure 8.3 shows the percent of the 32-entry instruction window that, on average, the AES thread occupied on the SMT processor. Despite the fact that the AES thread only occupied about 40% of the core's instruction window entries when everything executed in software, during hybrid execution, the AES thread used nearly 90% of the core's instruction window entries.

During the AES kernel's execution, many of the AES benchmark's instructions in the instruction window were loads to query whether the RH kernel had finished its execution. Each of these instructions has a large latency, waiting up to 200 cycles for the RH kernel to finish. Additionally, because the branch performance of AES was quite good the processor rarely had to flush its instruction window. Although the fetch unit would not fetch the AES thread's instructions when the coscheduled thread could also fetch instructions, AES could quickly fill up most of the window when its coscheduled SW-only thread could not fetch instructions. Once an AES instruction was placed in the window,

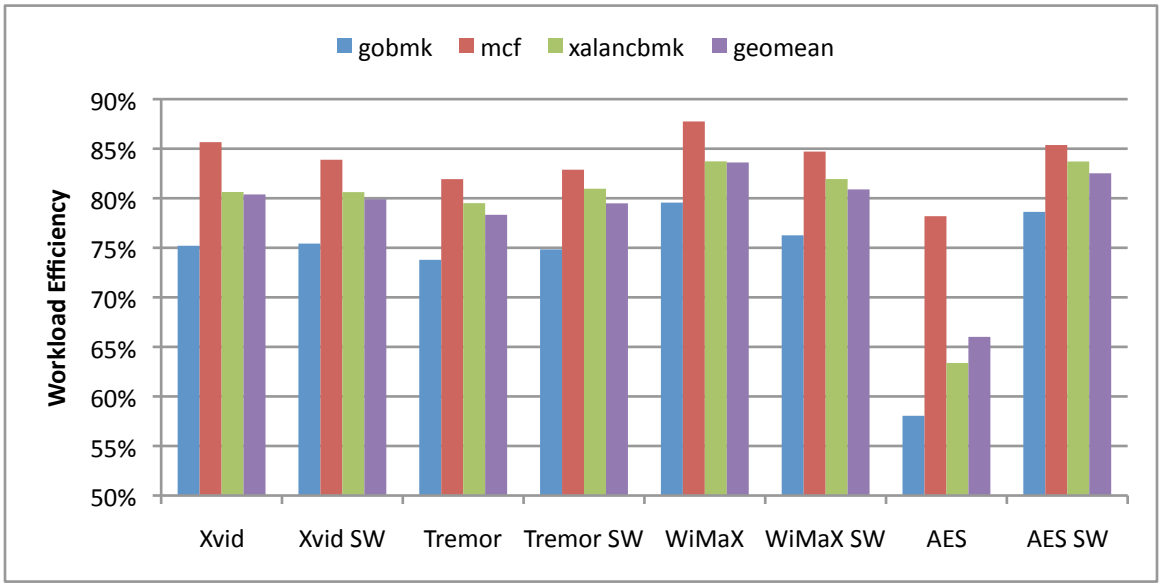


Figure 8.2 Efficiency of the overall workload's performance

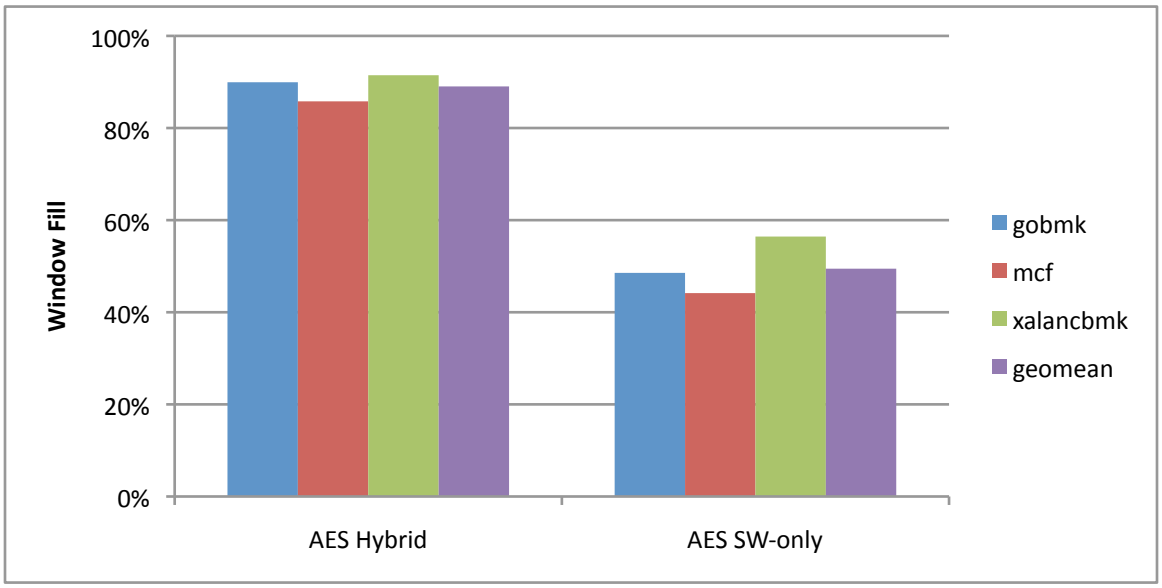


Figure 8.3 Average percent of the processor's shared instruction window occupied by the AES thread

it was likely to sit idle in the queue for a long time. Although hybrid RH/SW applications were not using the core's functional units very often, they managed to fill up much of the processor's shared instruction window.

8.3 Limiting RH thread's Resource Usage

Although sharing an instruction window between applications makes sense for many workloads, it can result in a poor allocation of resources when applications experience a long stall event. Tullsen et. al [117] experienced a similar problem when ordinary software threads had long-latency events such as a cache miss. This work examined mechanisms that flushed an application's instruction window upon suspected cache misses. In two-processor workloads they found that using this mechanism could greatly improve performance when one of the coscheduled threads experienced many cache misses.

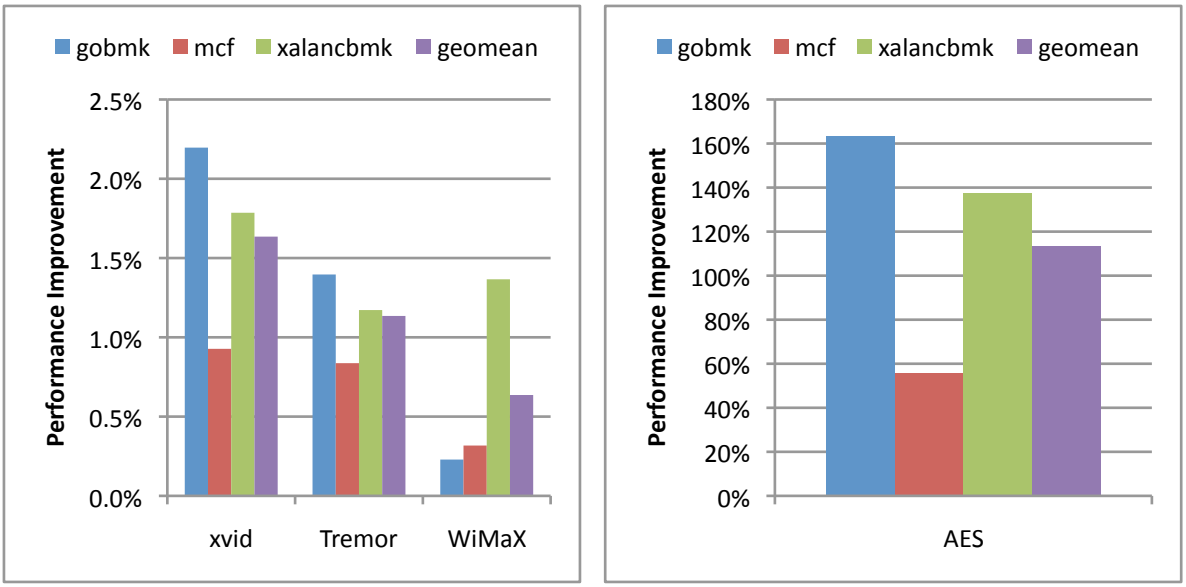
I therefore implemented a similar scheme to prevent threads executing on the RH from monopolizing shared processor resources. However, unlike detecting cache misses as examined in [117], is is trivial for the processor to detect RH kernel execution. As described in Chapter 4.4.2, applications in the system spin-wait while RH kernels execute. During this spin loop, the SW issues requests to the RH controller to determine if the RH kernel is done. The RH controller buffers these requests for up to 200 cycles, and only one request can be outstanding at a time. Therefore, these instructions are likely to be stalled in the pipeline for a long time.

In this section, I examine multiple methods that help prevent threads executing in an RH kernel from monopolizing the processor core's shared instruction window. Because RH kernel execution can be detected earlier than cache misses, our stalling mechanisms elected to not fully flush the thread's pipeline as was done in Tullsen's prior work [117]. In Section 8.3.2 I will show that slowing down a thread's execution too soon can reduce the performance of RH kernels, so I elected not to fully flush stalled thread's pipelines.

8.3.1 Dynamic Fetch Stalling While Executing RH Kernels

The first method employed to improve performance of coscheduled applications on an SMT processor is dynamic fetch stalling. This policy prevented thread from fetching instructions when they had an active kernel query request. When the outstanding kernel query returned, the hybrid thread could resume fetching instructions. Additionally, fetch stalling was disabled upon any exception (interrupt, TLB miss, etc) to prevent starvation.

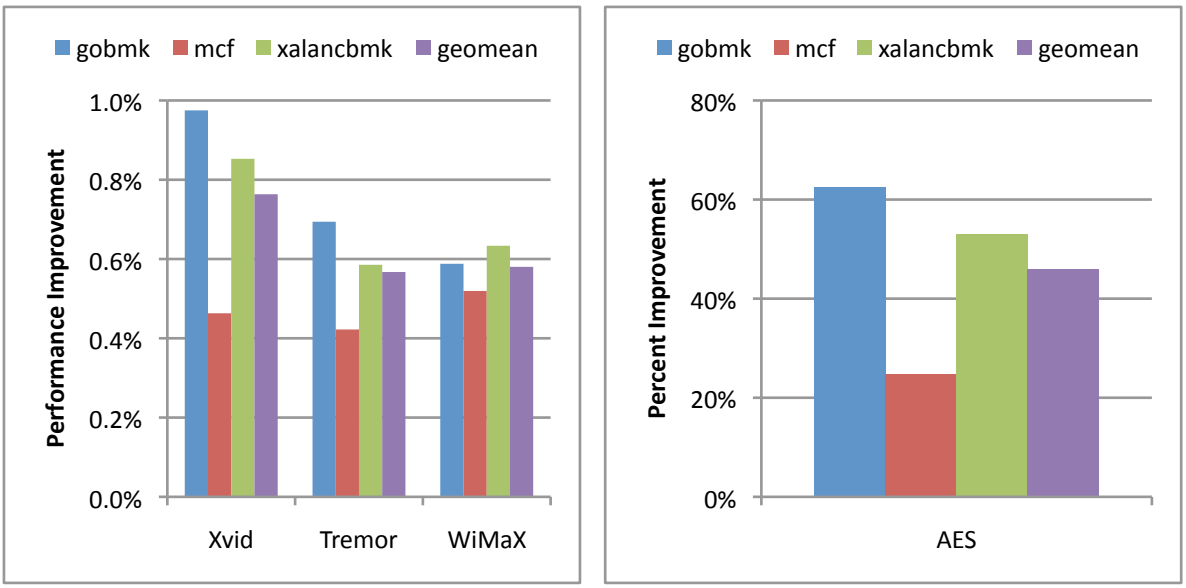
Figure 8.4 shows the performance improvement (using the EIPC metric) that dynamic fetch stalling has on SW-only threads coscheduled alongside hybrid RH/SW applications. Performance of the SW-only threads were always improved by the usage of this mechanism. More important than just the performance of SW-only threads however, is overall system performance. Figure 8.5 shows the percent change in the workload's NEIPC performance when using the dynamic fetch stall mechanism. This shows that not only was the SW-only thread accelerated, but performance of the entire workload was better when using dynamic fetch stalling.



(a) Performance improvement

(b) Performance improvement of AES

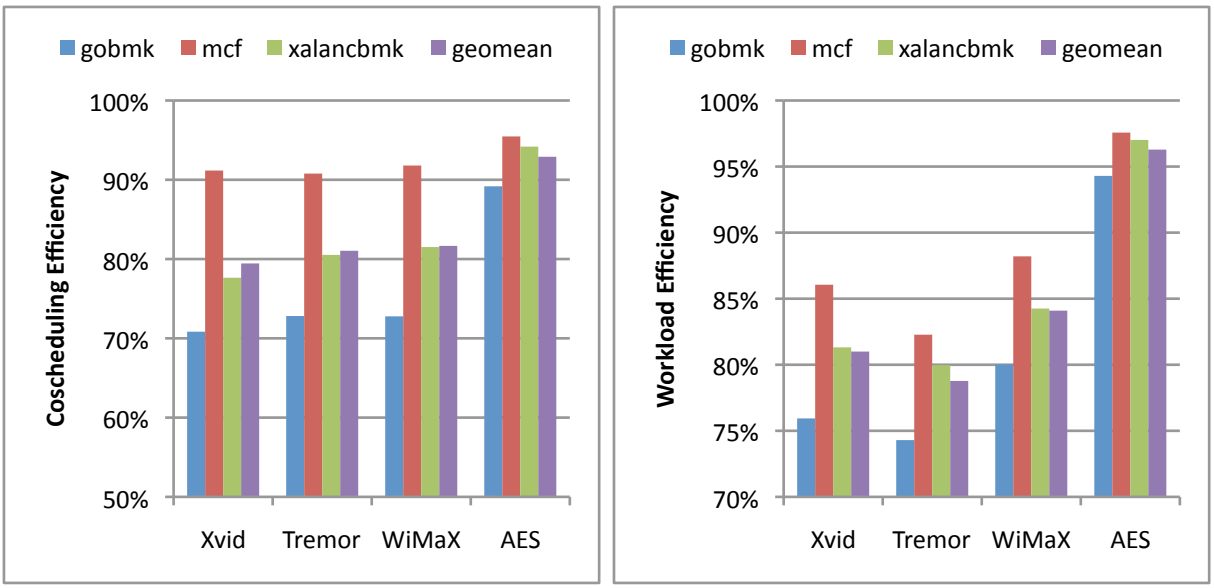
Figure 8.4 Percent improvement of coscheduled benchmarks' EIPC when using dynamic fetch stalling



(a) Performance improvement

(b) Performance improvement of AES

Figure 8.5 Percent improvement of workloads' NEIPC when using dynamic fetch stalling



(a) Coscheduled thread's efficiency

(b) Workloads' efficiency

Figure 8.6 Efficiency of coscheduled threads and workloads when using dynamic fetch stalling

Using the dynamic fetch stalling mechanism improved performance of threads coscheduled alongside AES much more than that of threads coscheduled alongside the other hybrid RH/SW benchmarks. This is due to the fact that the AES benchmark consists of a single RH kernel that is called repeatedly. Preventing this thread from occupying the core's instruction window, when the AES kernel is executing, can greatly improve the performance of coscheduled threads. Even for the other hybrid workloads, the performance of both the coscheduled SW-only benchmark, as well as the workloads as a whole improved when enabling dynamic fetch stalling, however the performance improvement was nowhere near as dramatic due to the short execution time of many of the other kernels, and the (relatively) small percentage of overall execution time spent in the RH kernels.

Figure 8.6 shows the efficiency of the coscheduled thread when using dynamic fetch stalling. I calculated the efficiency measurements the same as I did in Section 8.2. When enabling dynamic fetch stalling, threads coscheduled alongside AES outperformed the same threads when AES ran purely in software. However, with the other hybrid RH/SW benchmarks, coscheduled threads sometimes performed slightly worse than when everything ran in software. This can be explained by the behavior of the RH kernels. Many of the kernels (particularly in Xvid) performed poorly when executed on an out-of-order superscalar processor. Because of this, an SMT processor could better take advantage of coscheduling another thread alongside it during those times, improving overall performance. When hybrid execution is enabled, the coscheduled thread is now more likely to be executing alongside code that performs better on out-of-order processors. This results in slightly worse performance for the coscheduled thread. Although the coscheduled thread also executes alongside the hybrid benchmark during the execution of RH kernels, this time is

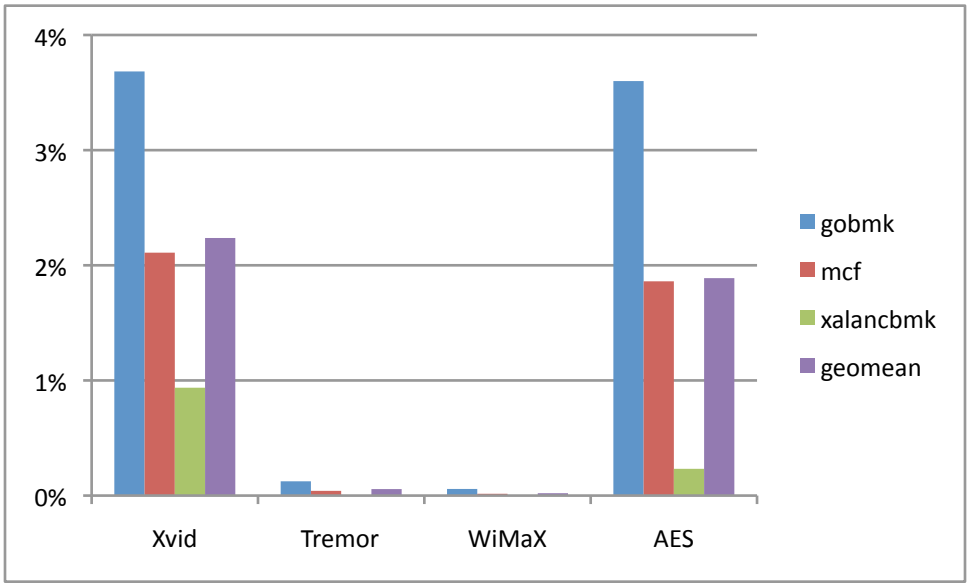


Figure 8.7 Performance improvement of SW-only coscheduled thread when using both dynamic and static fetch stalling instead of just dynamic fetch stalling

a relatively small portion of overall execution, and thus cannot improve performance enough to offset the difference. Additionally, many of the RH kernels in the Xvid workload execute for a very short time, so they do not execute long enough for the dynamic fetch stall mechanism to significantly impact the hybrid thread's instruction window usage.

8.3.2 Improved Fetch Stalling

Although disabling the ability of a thread to fetch new instructions while waiting for an RH kernel to execute helps the performance of coscheduled applications, it does not do enough to minimize the stalled thread's instruction window usage. In particular, threads coscheduled alongside AES only achieved $\sim 93\%$ of their CMP performance. I therefore evaluated further methods to stop a stalled thread from using the instruction window. In this section, I introduce a second mechanism to help improve the performance of coscheduled threads.

I refer to this new mechanism as static fetch stalling, and changes the method the processor used to select which thread to fetch from. When using the modified fetch mechanism, the processor reduces a hybrid thread's fetch priority as soon as an RH kernel is reserved, causing SW-only threads to always have priority. This mechanism comes into effect as soon as the hybrid thread reserves an RH kernel, and does not wait until the application has started executing on the RH fabric to take effect.

On its own,static fetch stalling is not particularly useful because it still allows hybrid threads to fill up the processor core's shared instruction window during RH kernel execution. However, when used in conjunction with the stalling technique described previously, it can help further improve the performance of coscheduled threads. Figure 8.7 shows

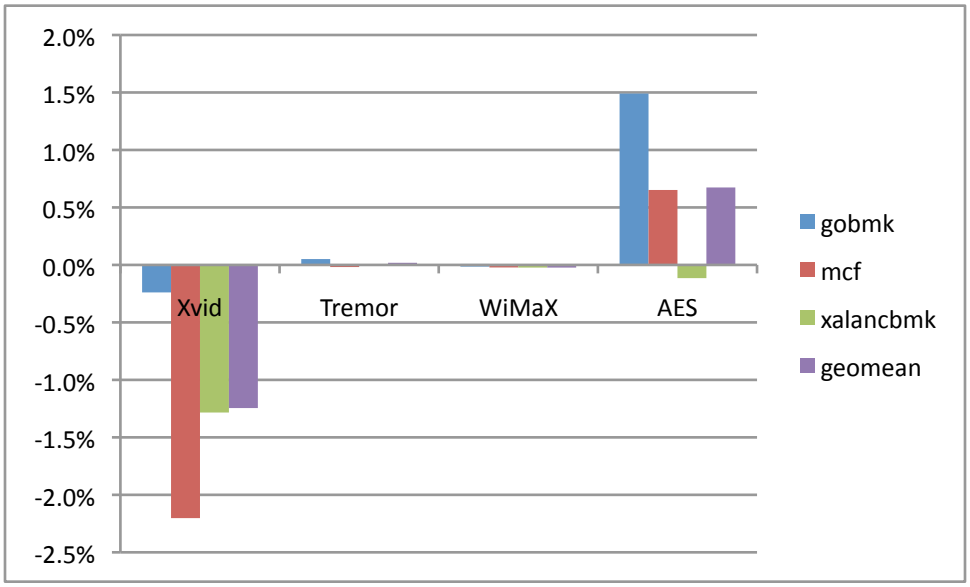


Figure 8.8 Performance improvement of workloads when using both static and dynamic fetch stalling instead of just using dynamic fetch stalling

the performance improvement of the coscheduled threads' EIPC when using both static and dynamic fetch stalling over just using dynamic fetch stalling.

Performance of threads coscheduled alongside AES and Xvid improved when the system used both static and dynamic fetch stalling. Performance gains for each software-only benchmark was around 2%. For both the WiMax and Tremor benchmarks, performance of coscheduled threads does not change significantly when using static fetch stalling, because the accelerated versions of both Tremor and WiMax spend little of their execution time in RH kernels.

Although static fetch stalling always improved performance of the coscheduled SW-only thread, it does not always improve the performance of the workload as a whole. Figure 8.8 shows the performance "improvements" of the workloads when using both static and dynamic fetch stalling compared to just using dynamic fetch stalling. Although AES workloads' performance improved all around, Xvid's performance actually decreases significantly. Many of the RH kernels in Xvid execute for a very short time period; slowing their fetch means that it takes longer to start some of the RH kernels, and thus it will take longer to complete the kernel's execution. Figure 8.9 shows the efficiency of the AES workloads when using all of the different fetch mechanisms.

Using static fetch stalling improved the performance of coscheduled SW-only, however it also increased the run-time of the RH kernels. This increase in RH kernels' runtimes is relatively constant, causing it to impact short-running kernels much greater than long-running kernels. With this in mind, I created a new fetch mechanism that dynamically decided whether to apply static fetch stalling

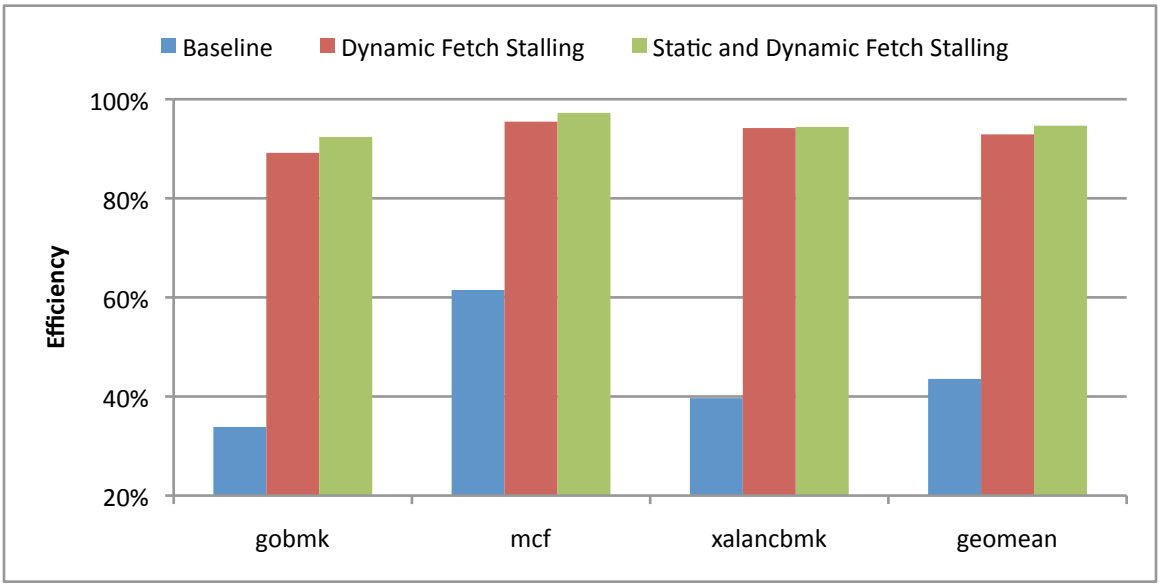


Figure 8.9 Efficiency of thread coscheduled alongside hybrid AES for the three proposed systems

This policy used the weighted average of an RH kernels' execution time to determine how long each call to an RH kernel executes for, and only enabled static fetch stalling when the kernel's runtime was expected to be over 1,000 cycles. Otherwise the CPU only used dynamic fetch stalling. This new fetch stalling mechanism allowed applications coscheduled alongside applications that executed long-running RH kernels to perform at nearly the same speed as they do on a CMP processor, while hybrid applications with shorter-running RH kernels were not penalized through the use of static fetch stalling

The results from these runs are not shown. The performance of workloads containing the AES benchmark was the same as when using both dynamic and static fetch stalling. The other workloads performed the same as they did in Section 8.3.1. Although I chose the 1,000 cycle number arbitrarily for this policy, it managed to accelerate workloads containing long-running RH kernels, while not degrading the performance of workloads containing short-running RH kernels. A more thorough study across a wider range of workloads is needed to know exactly when the fetch priority should be reduced, however this has been left for future work.

8.4 Conclusions

This chapter showed that SMT platforms can very efficiently execute hybrid RH/SW threads. However, modifications to the processor are needed to prevent hybrid thread's from filling up the shared instruction window. Unlike prior work work that examined flushing the pipeline upon a cache miss [117] , we chose to instead modify the fetch policies to prevent threads from monopolizing the shared instruction window. This was possible because we did not need to

wait to predict if an instruction was a long-latency cache misses, but rather knew if long-latency RH kernels were executing. Through the usage of a new fetch policy, the performance of SW-only threads improved while not degrading hybrid RH/SW threads performance. Future RH systems that execute long-running RH kernels should consider using SMT processors to better utilize processor resources. Using such systems can results in performance similar to CMP systems, but using a fraction of the area, and with lower power consumption.

Although this chapter showed the results of coscheduling two threads together on a single core system, it did not attempt to examine the complexities of a multicore SMT system. On such a system, the OS should be aware of which threads are likely to execute RH kernels, how long their RH kernels execute for, and what percent of hybrid execution is spent in the RH kernels. This way, the OS could better select which hybrid applications should execute together. Not using this information could result in a poor balancing of workloads. For instance, on a dual core two-way SMT system (two cores, each executing two threads) running two copies of a SW-only application, and two copies of AES, scheduling the two AES threads on the same core would likely result in worse overall performance than if each core executed an AES thread, and a SW-only thread. Much future work is still needed to determine which applications should be coscheduled together, building upon the work done in [105].

Chapter 9

RH Kernel Sharing on Multicore Systems

In this thesis I first examined how a RH kernel should communicate with the general-purpose processors in a single-processor system (Chapter 6). It then extended this model to multiprocessor systems, examining workload behavior as I increased the number of applications and processors in the system (Chapter 7). I designed these studies to examine the RH's memory subsystem. Therefore, every RH kernel was physically configured on the RH fabric to maximize the strain placed on the memory subsystem.

However, in a real reconfigurable computing system, the RH will likely be a constrained resource, and it is unlikely that all of the RH kernels belonging to all of the applications can fit on the RH at one time. These constrained systems require policies to allocate the RH to the various RH kernels. One way to allocate resources is to have the OS dynamically allocate the RH resources as discussed in Chapter 2 (and reexamined in Chapter 10). Although this method can be quite effective at selecting which RH kernels are configured, if multiple applications are executing the same RH kernels, it may not be the best method for allocating the limited RH resources. This chapter shows that sharing configured RH kernels amongst the applications results in better performance than our dynamic kernel scheduling algorithm.

9.1 Motivation and Prior Work

With multicore processors becoming prevalent in both embedded and desktop computing, application developers must make their applications multithreaded if they wish to maximize performance. Many of these multithreaded applications will exploit a type of parallelism called single program multiple data (SPMD), where each thread of execution runs the same set of operations on different data. In this scenario, rather than requiring each thread to have its own unique copy of physical RH kernels, threads could theoretically share a single set of configured RH kernels, stalling execution of a thread if it attempts to use an RH kernel that is currently in use by another thread. Doing this can increase the utilization of the RH kernels, and provide a more efficient allocation of the limited RH resources

In addition to applications that exploit SPMD parallelism, RH kernel sharing can be useful in systems that concurrently execute multiple copies of the same application. This could happen in a network appliance encrypting/decrypting multiple streams of data, or a set-top box processing multiple video streams. In these instances, multiple copies of the same or similar processes need the same RH kernel designs. By sharing these configured resources, it is likely that similar performance could be obtained (compared to having sufficient hardware for every RH kernel) using a fraction of the RH resources.

Sharing of RH kernels would also be useful in a software-defined radio (SDR) platform [101]. SDR platforms use programmable processors (often combined with accelerators) to implement wireless algorithms. The WiMaX backend receiver described in Chapter 5.1.2 is an example of an SDR algorithm. In these systems, the processors implement a variety of different radio protocols. In some instances, WiFi tethering of a cell phone for instance, it is necessary to concurrently process multiple wireless protocols simultaneously. Because many wireless protocols use the same basic algorithmic components (FFT, Viterbi or Turbo encoders, reed-solomon encoding/decoding etc) accelerators configured for one SDR protocol might also be able to be used by other wireless protocols.

For most of these applications, it is unlikely that any single RH kernel makes up the majority of the application's accelerated execution time. Therefore, at any given point in time, most RH kernels are idle. For example, in a case where a kernel makes up 95% of the current phase of an application's execution when unaccelerated, the acceleration provided by an RH implementation of the kernel means the kernel will be idle much of the time. If the RH kernel was 20x faster than the software version of the kernel, the RH kernel would be idle over 50% of the time. In this scenario, two threads could still make use of the RH kernel (assuming the overhead of switching the RH kernel between applications is minimal), waiting for access when the other application is using the kernel. However, even in a worst case scenario where an application *always* requests the RH kernel just after another application has requested it, the RH kernel will still provide a 10x speedup, and the application will have an overall speedup of 6.9x instead of 10.3x. However, in scenarios where the RH kernel is not in greater demand than it is available, a thread is equally likely to request the in-use RH kernel at the start of its execution as it is at the end of the kernel's execution. Therefore the average wait time is likely to be half the RH kernel's runtime, as I will verify in Chapter 9.3. Using this new wait time, the kernel speedup would be closer to 13.3x, and the application speedup would be 8.2x. For kernels that account for a smaller percentage of the application's execution time, the "penalty" for sharing an RH kernel will be even less, as the kernel is unlikely to be requested while another application is using it..

Although prior work examined the sharing of hardware resources, these works do not fully consider the impact of sharing *configured* RH kernels amongst applications. Chan et al. [23] describe a mechanism for sharing cryptographic coprocessors between multiple applications. User processes request access to an agent that virtualizes requests to a fixed set of cryptographic coprocessors. However, this system is specialized for encryption, and did not implement process isolation for the coprocessors' memory accesses.

In the time since I first published a method of sharing RH kernels [44], Intel has released their Quick Assist [66] hardware abstraction layer. Quick Assist is designed to handle many of the challenges of coupling a high-performance computing system with FPGA coprocessors. The Quick Assist API provides mechanisms for sharing configured RH kernels between applications. However, they do not describe how they isolate applications sharing a configured RH kernel or examine the impact of "over-subscribed" kernels and how to deal with them.

More recently, Sun Microsystems T1 and T2 processors share cryptographic coprocessors amongst multiple thread contexts on a processor core [65]. Modified cryptographic libraries provide hooks to use cryptographic accelerators on systems that support them, otherwise a software-only version of the library executes. Each processor core has its own cryptographic coprocessor, however, these cores are multithreaded, so the hardware must support the sharing of these accelerators. Process isolation between threads is preserved by flushing any data stored in the cryptographic core whenever it has finished executing.

9.2 Sharing RH Kernels

Sharing configured RH kernels allows multiple copies of the same or similar applications to share previously configured RH resources [44]. In this chapter, I examine the impact of sharing RH kernels between applications using the Xvid benchmark. In the Xvid benchmark, no single RH kernel represents the majority of application runtime. The most dominant kernel in unaccelerated Xvid is SAD8, which accounts for $\sim$31% of Xvid's runtime. If all kernels are hardware-accelerated, the percent of execution changes—the faster-running hardware version of SAD8 accounts for $\sim$8.6% of the accelerated Xvid's runtime. The SAD8 accelerator is therefore idle for $\sim$91.4% of the time, provided it remains configured in RH.

Although the OS could elect to reconfigure the hardware for another purpose during this time, this is actually not effective in all cases, as the time required to reconfigure the RH kernels is often larger than the time between kernel calls. The execution of Xvid helps to illustrate this point. The largest phase of Xvid's execution is the motion estimation routine. This routine accounts for nearly half of the accelerated Xvid's runtime and regularly calls the SAD8, Avg2, and Avg4 kernels. The time between kernel calls is short (orders of magnitude shorter than the time required to reconfigure the hardware), and the time between calls is data-dependent, making it difficult to predict which RH kernels should be scheduled onto the hardware. Therefore, to best accelerate this phase of execution, all of these RH kernels need to be configured in hardware at the same time. In other words, despite the fact that none of these kernels makes up a significant portion of Xvid's accelerated runtime, they all must be loaded $\sim$50% of the time to accelerate their respective computations. Because of this, on multicore machines, an RH kernel scheduler may be forced to not load some of the RH kernels for their entire phase of execution to leave room on the RH fabric for other kernels that the RH kernel scheduler finds more useful (even though they may only be in use a small fraction of the time).

Sharing the RH kernels amongst the different threads or applications that need their functionality can dramatically improve the utilization of these kernels. For instance, if two instances of Xvid are executing, a single physical copy of each RH kernel could be shared so that both instances of Xvid can benefit from them.

9.2.1 RH Kernel Sharing Implementation

RH kernel sharing can be added to the RH computing system describe in Chapter 3 by making a few simple modifications to the RH controller (I described the RH controller in Chapter 3.3). First, I modified the information that the RH controller stores about each physical RH kernel so that it keeps track of the most recent process to use it, and whether or not the physical RH kernel is currently executing. This allows the RH Controller to delay access to a kernel if it is in use by another thread.

RH kernels are accessed using the mechanism described in Chapter 4.2, and applications "spin" wait when a shared RH kernel is in use by another application. I selected this mechanism because even when eight processors are in the same phase of execution, it is relatively unlikely that the spin wait time plus the execution time of the accelerated kernel would be greater than the software runtime (as shown in Chapter 9.3, the "spin" time for the most used RH kernels is rarely much larger than the RH kernels' execution time). In the event that an application queries an RH kernel that is busy, the RH controller queued the request internally for up to 200 cycles to see if the physical RH kernel becomes available. This optimization is similar to the optimization mentioned in Chapter 4.4.2 that delays packets requesting if an RH kernel is done. Queuing RH availability request packets also reduces the network traffic between the processor cores and the RH controller, and also reduces the latency seen when the RH controller notifies an application that the RH kernel is available.

When the RH Controller grants a thread access to a kernel, it sets a special reserve bit associated with the physical RH kernel (physical and virtual RH kernels are described in detail in Chapter 4). The RH kernel stays reserved until it has completed execution, at which point it signals the RH Controller to clear the reserve bit.

If the requested virtual RH kernel differs from the virtual RH kernel that previously reserved the physical RH kernel, the RH Controller clears all cached stream controller parameter. This prevents threads from accessing parameters set by other processes, helping to preserve process isolation. If the RH controller flushes the stream controller's parameters, the executing application must reinitialize the stream parameters. This functionality is provided by modifying the routine that queries if the RH kernel is available. Instead of just indicating if the kernel is ready, it also indicates if the stream controller parameters have been flushed. If so, the software must reinitialize the stream controller parameters.

9.2.2 RH Kernel Sharing Test Cases

I examined six different test cases were used to determine the effect of RH kernel sharing. At the two extremes are the **no RH** and sufficient RH (**sufficient**) cases that will be used as baselines in this chapter. In no RH, multiple

Allocation	# RH Tiles	Description
No RH	0	All the copies of Xvid execute in software
Sufficient	18/36/72	All copies of Xvid have a unique copy of all the RH kernels (2p/4p/8p)
Sharing	18	All copies of Xvid share a single copy of every RH kernel, waiting for access
Static	18	A static allocation of the RH kernels is given to all of the copies of Xvid
Dynamic	18	Every 2mS the OS dynamically decides which RH kernels should be loaded
One	18	One of the Xvid encoders has access to the RH, others do not

Table 9.1 Methods used to allocate the RH, the number of tiles needed for the allocation method, and a brief description of how the RH was allocated.

software-only Xvid processes are executed, and the sufficient RH case models a system where there are enough RH resources to allow every executing Xvid process to have its own unique set of physical RH kernels. In other words, this case fully accelerates all RH kernels, and configured RH resources are not shared. This case is therefore an upper bound on the performance benefit that is achievable with kernel sharing.

The remaining tests in this chapter limit the RH resources to some degree. For example, in the "one accelerated" (**one**) simulation, the OS configures a single copy of the Xvid kernels on the RH, and these kernels are only accessible to a single instance of Xvid (other instances of Xvid execute entirely in software). The static schedule (**static**) case models a situation where kernels are not shared, and uses a static analysis scheduling algorithm to decide which physical RH kernels should be configured. For this case, the system contained the same number of tiles (18) required by the" one accelerated" case. However, rather than implementing one of each kernel type, the kernel scheduler described in Chapter 2.7.2 chooses which kernels will provide the best overall benefit. In addition to the static-scheduling runs, I compared the performance of the sharing system against a dynamically-generated scheduler (**dynamic**) that periodically runs the dynamic kernel scheduling algorithm (every 2ms) [40][1]. Finally, in shared kernels (**shared**), the OS configured a single physical copy of each RH kernel, however all of the Xvid processes share all of these configured RH kernels. This test case used the sharing mechanisms described earlier to share the physical RH kernels. Table 9.1 provides an overview of how each experiment allocated the RH.

We ran each test case on three different test platforms, one where two processors execute two instances of Xvid, one with four processors executing four instances of Xvid, and one with eight processors running eight instances of Xvid. This allows us to examine the impact of sharing RH kernels as the number of processes competing for access to the RH kernels is increased.

[1] Although all tests containing statically configured RH kernels use the same checkpoints, simulations using dynamically scheduled kernels had to use a different checkpoint. This is because dynamically configured kernels have a very different OS' state than kernels that are statically allocated, resulting in different execution paths. This makes it physically impossible to reuse the checkpoints without causing the OS and/or the RH controller to be in an inconsistent state. However, because Xvid's runtime is relatively consistent over a long enough time span (and the checkpoints were from similar points in the applications' execution), the differences in the two checkpoints should be minimal.

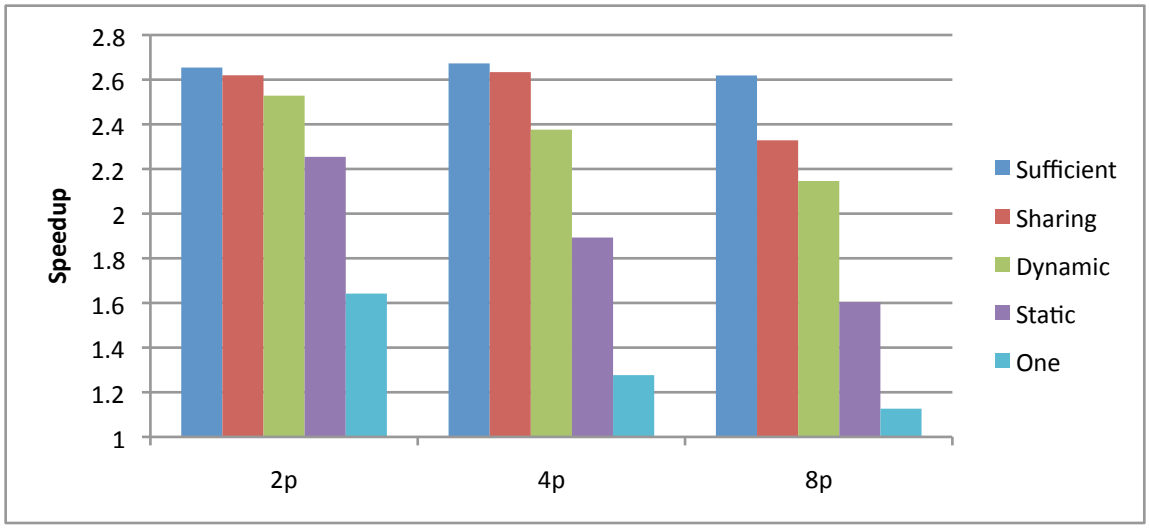
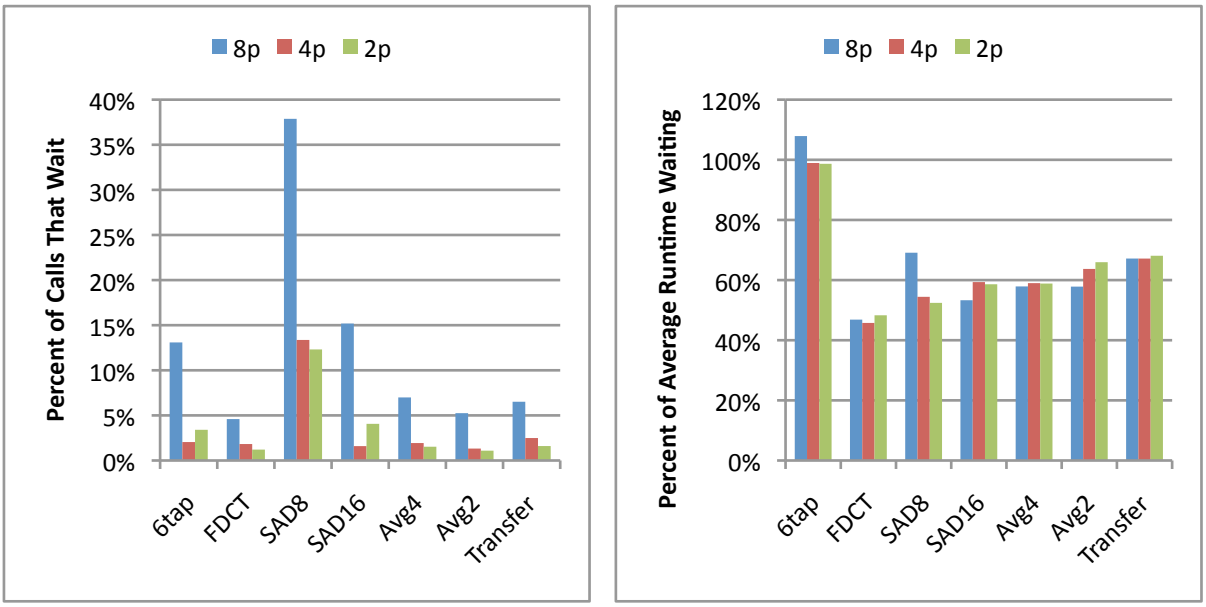


Figure 9.1 Xvid's speedup over no RH for each of the test cases.

9.3 Sharing Performance

Figure 9.1 shows the performance of the test cases on both two- and four-processor systems. These results show that sharing a single copy of the configured Xvid RH kernels results in system performance that is close to that of a system where each application has its own unique copy of the RH kernels, achieving $\sim 98.5\%$ of sufficient RH's performance for both two- and four-processor systems. Only on eight-processor systems did the performance degrade noticeably, obtaining only 89% of the performance of sufficient RH. In addition, all of the sharing cases outperformed the statically and dynamically allocated hardware. In particular, as the number of processor cores increased, the performance advantage over the scheduled RH cases increased. This is because none of the RH kernels are in extremely high demand, so sharing the RH kernels can provide high performance.

This figure also shows that although dynamic scheduling is quite effective when executing on a two-core system, as the system is scaled up to four and eight cores, performance suffers. I expected this, as the demand for the RH has increased, while available resources have stayed constant. Although the same thing happens when sharing RH kernels (the demand for each RH kernel increases as the number of applications accessing them increases), as Figure 9.1 shows, this dropoff occurs at a later point than for dynamic scheduling. Although dynamically scheduling the RH fabric provides coarse grain sharing of limited resources, sharing configured RH kernels exploits a more fine-grained sharing of the resources. Unlike the dynamic system, the shared system does not have to periodically perform a relatively expensive scheduling operation or suffer from the *very* large cost associated with reconfiguring kernels located on the RH. This allows the sharing system model to outperform dynamic allocation of the RH resources. Additionally, as will be shown in Chapter 9.4, by using a few additional resources, performance of shared RH kernels can be improved significantly.



(a) Percent of calls to the a given kernel where it must wait for that kernel to complete execution for another thread

(b) When waiting, the percent of average kernel runtime that the application waits

Figure 9.2 How often there was contention for shared Xvid kernels, and how long the waiting application had to wait to obtain access.

There are many important things to take into consideration when examining the impact of RH kernel sharing. It is useful to know how often an application had to "wait" to access an RH kernel because the associated physical RH kernel was in use. Figure 9.2(a) shows the percent of calls to each RH kernel where the application had to wait at least one cycle to obtain access to the RH kernel. This graph shows that there was rarely much contention when accessing most of the RH kernels. For two- and four-processor simulations, applications had to wait $\sim 13\%$ of the time they called the SAD8 kernel. All other RH kernels were immediately available over 97% of the time. In eight-processor simulations, waiting was more common due to the increased contention for all of the RH kernels. In particular, applications that wished to use the SAD8 kernel had to wait almost 40% of the time. Also, applications that wished to use the interpolate 8x8 6-tap and the SAD16 kernel had to wait about 15% of the time.

However, just knowing how often an application waits to access an RH kernel does not tell the entire story; it is also important to know *h*ow long the application waited to access the RH kernel. Figure 9.2(b) shows the percent of each RH kernels' average runtime an application spent waiting when a kernel was not immediately available. This figure shows that the wait time for most RH kernels was approximately half of the RH kernels runtime. This makes sense, because most of the time, there will be exactly two threads[2] competing for access to the resource. Assuming applications call the RH "randomly", they are equally likely to overlap during all periods of execution, resulting in a wait time of $\sim 50\%$ of the kernel's hardware execution time.

The only exception to this is the interpolate 8x8 6-tap kernel. When calls to this kernel had to wait to access the RH, they tended to wait for the entire execution of the RH kernel. This is because when Xvid is in the interpolation phase of its execution, it calls this kernel hundreds of times, performing almost no calculations between subsequent calls. Therefore, if two copies of Xvid are in this phase of execution at the same time, they will continuously compete for access to the kernel. Because only one copy of Xvid can gain access to the kernel at a time, the other copy of Xvid will have to wait for the first application to finish executing the RH kernel before it can start its own execution. This situation will continue to repeat until one of the copies of Xvid finishes the interpolation phase of its execution.

The time spent spin-waiting for an RH kernel to become available is only part of the overhead associated with busy RH kernels. When RH kernels are relatively uncontested, the applications will predict that the RH kernel will get executed. However, if there is a lot of variability in whether the RH kernel will be available, the processor's branch predictors are more likely to mispredict that an RH kernel is not available when it really is (if the branch predictor predicts that the kernel is available when it is not, the penalty is not very great, as the pipeline flush will occur while the processor is stalled waiting for the RH kernel). Although the impact of this is minimized by the fact that the RH controller will not immediately return that an RH kernel is busy, instead stalling a request for up to 200 cycles to see if the physical kernel becomes available. If the RH kernel is available within this time period, the RH controller returns that the RH kernel is available even though it was not available when the processor initially requested the RH kernel.

[2] Although it is is possible for more than one thread to be waiting for the RH kernel at the same time, in situations where the RH kernel is rarely busy, it is highly unlikely that another thread will already be waiting for the RH kernel.

Additionally, the application must "reset" any cleared stream controller parameters (if not using the default values). Combined, these stalls mean that a busy RH kernel can greatly increase the average runtime of the shared RH kernel.

When examining this data, it is clear that Xvid's RH kernels can be shared rather effectively by two and four applications executing on multicore systems without a significant drop in system performance. However, as the number of applications (and cores) sharing the RH kernels increases, some of the RH kernels can become bottlenecks. In particular, the SAD8 kernel suffered from poor performance when more applications shared the RH kernels. When up to eight applications could be requesting the SAD8 kernel at a time, the average execution time of this RH kernel increased. Because the RH kernel took longer to execute, it now represented a larger percentage of the application's execution time. This further increased the chance that yet another copy of Xvid would request the SAD8 kernel at the same time; further reducing performance, and resulting in a SAD8 execution time that was twice that of sufficient RH's SAD8 execution time.

Counting the time spent waiting for the RH kernel to become available, the eight copies of Xvid combined spent over three billion cycles in the SAD8 kernel. This was longer than the total time that the simulations executed for (two billion cycles). SAD8's reduction in performance (due to time spent waiting), combined with its large percentage of overall execution time make it responsible for the majority of the slowdown experienced when sharing RH kernels. I refer to shared kernels whose performance suffers greatly (such as SAD8's) as "over-subscribed" kernels.

9.4 Kernel Pooling

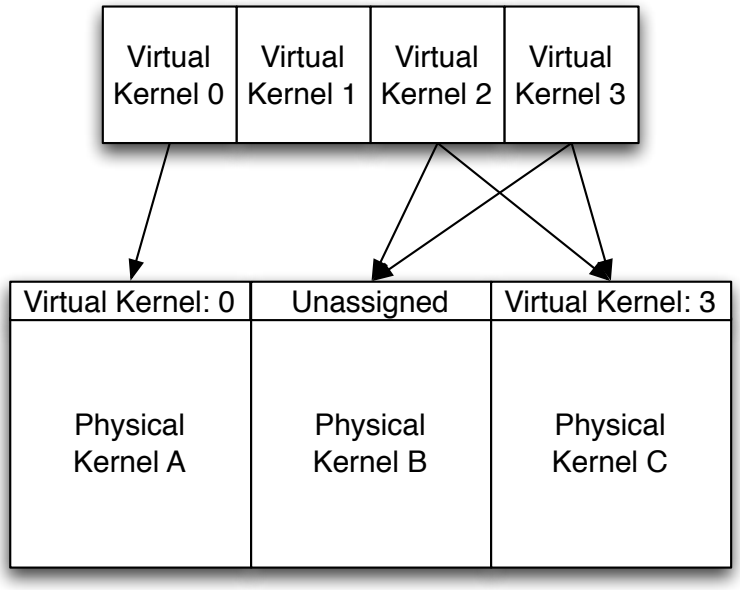


Figure 9.3 Mapping virtual kernels to physical ones using kernel pools

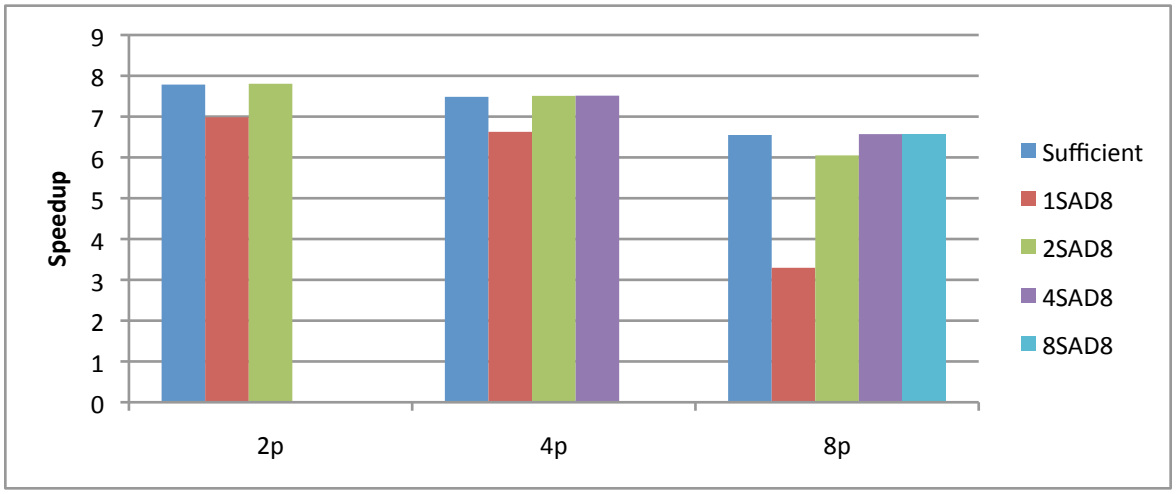


Figure 9.4 SAD8 kernel's speedup when varying the number of SAD8 kernels on the system, and configuring a shared single copy of all other kernels

To counter the problem of "over-subscribed" kernels,I augmented the system with kernel "pools". Kernels pools are groups of multiple copies of physical RH kernels shared by all of the processes that have requested that type of RH kernel. I implemented these kernel pools by modifying the method that the RH Controller allocates RH kernels. When an application attempts to access a virtual RH kernel, the RH controller first checks if any corresponding physical kernels exist on the RH and if any of them are free. If multiple physical RH kernels are available, the RH Controller attempts to reserve the physical RH kernel that the application last accessed.

Figure 9.3 depicts a set of virtual kernels and their mapping to physical kernels in the kernel pool. Virtual kernels mapped to the same physical kernel cannot use the same physical kernel simultaneously. In this figure there are four virtual RH kernels, and three physical RH kernels. Virtual kernel 0 is assigned to physical kernel A, and virtual kernel 1 is currently not associated with any physical kernels. Virtual kernels 2 and 3 over time have shared physical kernels B and C. Currently, however, virtual kernel 3 is assigned to physical kernel C, and virtual kernel 2 is unassigned.

I examined the benefit of kernel pools by executing the same workloads as before, but with a varying number of physical SAD8 kernels configured on the RH. Figure 9.4 shows the speedup of the SAD8 kernel (compared to the execution time of No RH) as we configured more physical copies of the SAD8 kernel. This graph shows that adding a second copy of the SAD8 kernel improves the performance of SAD8 on two- and four-processor machines (on par with the performance of sufficient RH), and greatly improves performance on 8-processor machines. On eight-processor systems, adding a second physical copy of the SAD kernel increases performance from 50.3% of its performance with sufficient RH to 92.3% of that performance. Adding more copies of the SAD8 kernel on eight-processor systems brought the kernel's performance to be on par with that of the sufficient RH system, showing that pooling RH kernels can increase the performance of over-subscribed kernels.

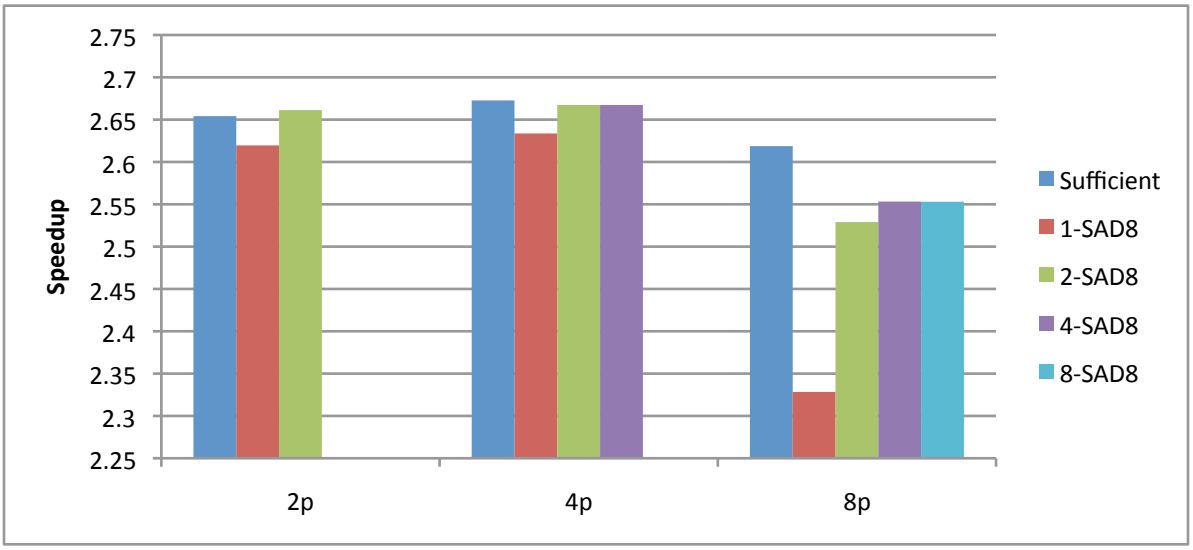


Figure 9.5 Xvid performance when all of the RH kernels are shared, but multiple copies of the SAD8 kernel were pooled together to decrease the demand on the RH kernel.

Figure 9.5 shows that overall Xvid performance improves when pooling multiple copies of the SAD8 kernel . On two- and four-processor systems, adding a second copy of the SAD8 kernel brought performance up to the same level as sufficient RH. On the eight-processor system, adding a second copy of SAD8 increased performance significantly, but even when eight copies of SAD8 were configured on the RH, overall performance was still only 97.5% of the sufficient RH case. Because the SAD8 kernel is only a single tile in size, the cost of configuring an extra SAD8 kernel is minimal compared to the cost of adding all of the RH kernels, making it worthwhile to configure at least two copies of this RH kernels on eight-processor systems. To further increase the performance of the eight-processor workload, the system would need to enable RH kernel pools of the interpolate 8x8 6-tap kernel and possibly the SAD16 kernel in addition to the pools of SAD8 kernels. Doing this would likely bring performance to the same level as sufficient RH, but would use significantly more resources than just sharing an additional one or two copies of the SAD8 kernel. However, this would still obtain significantly better performance than dynamically scheduling the RH fabric with the same number of tiles.

9.5 Kernel Sharing Conclusions

This chapter examined the allocation of limited RH fabric resources on multicore reconfigurable computing systems. In this work, I examined both existing ways that the OS can allocate the RH resources (static scheduling, dynamics scheduling, and one RH), and compared their performance to a new method I developed to share configured RH resources. In this work, I show that multiple copies of the same application can be best served by configuring a single copy of every RH kernel, and sharing them amongst all the applications.

I also showed that many RH kernels can be effectively shared by multiple applications without reducing the performance of the kernels. this is because it is unlikely that another application will be using the configured RH resource when it is requested. However, in other instances, I saw that the performance of RH kernels that are in high demand suffered greatly when shared by all of the applications executing on the system. To help prevent these RH kernels from degrading system performance, I pooled multiple copies of a configured RH kernel together and shared them amongst all applications executing on the system. Through the use of kernel pooling, application performance could be greatly improved using minimal extra resources.

This work suggests that future systems that are likely to execute multiple copies of the same or similar applications should implement the mechanisms proposed in this chapter to safely and securely share configured RH kernels. Sharing these RH kernels can result in performance similar to that of having sufficient RH to fully accelerate every application, while using only a fraction of the resources.

This chapter first showed how multiple applications executing on a reconfigurable computing system can safely share multiple copies of a single physical RH kernel. The additional hardware required to support this is minimal in systems like ours that distinguish the virtual RH kernels that an application is allowed to directly access from the physical RH kernels that are implemented on the reconfigurable fabric. Because of this, only a few additional bits of storage had to be added to each virtual and physical RH kernel to support the sharing and pooling of physical RH kernels.

This chapter also showed that sharing RH kernels can allow a very efficient usage of limited RH resources, even performing better than a dynamic RH kernel scheduler to the tested situations. It then examined situations where a shared RH kernel can become “oversubscribed”, limiting the performance of the application. However, this limitation on performance can easily be overcome by using pools of RH kernels that are high demand. Using pools of shared RH kernels can greatly improve the performance of oversubscribed kernels, bring performance close to that of a system that had sufficient RH such that all of the requested RH kernels could have their own unique copy of the RH kernel configured on the RH fabric. This work showed that for a hybrid RH/SW application containing multiple RH kernels, multiple physically configured RH kernels can be shared amongst multiple copies of the application without significantly impacting performance.

Chapter 10

Scheduling RH Kernels on Multicore Systems

Previous work examined methods to dynamically schedule RH kernels (see Chapter 2.7.2). In this prior work, the OS periodically evaluated which RH kernels should be configured on the RH fabric in order to maximize performance [40, 41, 107]. To perform this operation, the OS must read back how many times each kernel was called (whether the kernel was configured or not) and then use that information, in combination with the kernels software runtime, and RH speedup to determine which RH kernels should be loaded at a given time. Much of this previous work focused on selecting RH kernels for single processor systems. However, when multicore reconfigurable computing systems used this scheduler, the RH kernel scheduler did not always select the RH kernels that maximized system performance [48]. The interdependence of RH kernels caused performance to decline.

This chapter reexamines the RH kernel scheduler proposed in [40], and observes when it selects the "wrong" RH kernels. After examining the cause of the original schedulers shortcomings, this chapter proposed a new hierarchical RH kernel scheduler that properly handles the interdependence of RH kernels. In this chapter, I examine the performance of the new hierarchical RH kernel scheduler under a variety of situations, and compare it to that of the original RH kernel scheduler.

10.1 Shortcomings of Existing RH Scheduling Methods

The knapsack solver in the earlier kernel scheduling work assumes that kernel values are both additive and independent. However, that work did not fully examine the implications of running multiple applications that each contain multiple RH kernels, on a multiprocessor system. In these situations, the real application speedup contributed by each kernel depends on which other kernels from the same applications are also accelerated by the RH. The fact that kernel values are actually interdependent breaks the knapsack solver, causing it to sometimes select one kernel when another would provide more total acceleration to the system. The speedup of individual RH kernels could also be interdependent due to increased data locality, but this type of interdependence is an issue for future work. This chapter only examines the effects of kernel interdependence; therefore the runtimes of individual kernel calls in this work are unaffected by the RH allocation; only the rate at which the applications' call kernels changes.

Application	Kernel	Num Calls	T_{SW}	T_{RH}	Size (tiles)	Independent Application Speedup
1	A	90	10,000	1,000	1	5.26x
	B	20	5,000	1,000	1	1.09x
2	C	30	10,000	2,000	1	1.32x

Table 10.1 Example of two applications executing on a dual- processor system. All kernel execution times are listed as a count of CPU cycles.

Table 10.1 illustrates an example of a situation where the interdependence of RH kernels causes the original knapsack solver to select a set of RH kernels that does not maximize overall system performance. In this example, two applications execute on a system containing two RH tiles. If a scheduler used the independent application speedup value function it would select kernel A from application 1 and kernel C from application 2, resulting in a speedup of 5.26x for application 1, and 1.32x for application 2. However, had the scheduler instead chosen both of application 1's kernels, application 1's speedup would be 9.09x, and application 2s would be 1x (executing entirely in software). Accelerating kernel A causes application 1 to execute kernel B more frequently, and vice-versa. The total system speedup (calculated using the geometric mean) is higher for the second option, indicating a greater benefit to overall system performance. Although the knapsack solver exactly solved the problem it was given, the interdependent kernel values meant that the value of each RH kernel did not reflect its actual value, causing the knapsack solver to select RH kernels that do not maximize performance.

Another issue with the prior scheduling approach relates to the fact that the knapsack scheduler optimizes for the sum (arithmetic mean) of speedups the RH provides to the applications. However, a system-level scheduler may instead want to optimize for the geometric mean of application speedups, which has been suggested as a better method for aggregating overall system performance [88, 27]. The geometric mean strikes a balance between the arithmetic mean, which generally targets maximum single-application performance, and the harmonic mean, which targets minimum variance of application performance.

10.2 Example Problematic Scheduling Cases

In this section I compare the scheduling decisions and resulting performance of both the original scheduler as well as an optimal scheduler, which, for the purposes of this chapter, is defined as a scheduler that, based on the dynamically-profiled system behavior for the previous scheduling interval, selects the RH kernels that would maximize system performance for an identical interval (same applications in the same program phases). This allocation may not be optimal if the behavior in the next interval differs. However, the goal is to evaluate whether or not the the scheduler selects the best solution for the given data, and not to evaluate whether past behavior accurately models future behavior. Thus, for the remainder of this chapter, when the word optimal is used, it refers to a scheduler that

Variable	Description
P_k	Percent of application time (for the current phase of execution) that kernel k executes for in software.
S_k	Speedup that the RH implementation of kernel k has over its software implementation.
P_{ALL}	Percent of application's software execution time (for the current phase) covered by RH kernels
T_{SW_k}	Execution time of kernel k in software
T_{RH_k}	Execution time of kernel k on the RH
N_k	Number of times kernel k was called in the previous interval

Table 10.2 Some of the variables used in this chapter.

behaves as described above. For this study, the examined problems each represent a single scheduling interval, and the system can only re-allocate RH at the start of the interval.

In this chapter, I use a simplified model of hybrid RH/SW applications to focus on the issues involved when scheduling for multicore systems. Because of this, kernels have a constant runtime both when executing in software, and when executing on the RH. Additionally, the remaining SW-only portions of an application have a constant execution time regardless of what other applications are executing, or what RH kernels might be configured on the hardware. I calculated the speedups assuming that the next interval of execution has the same ratio of calls to the RH kernels, and that all of the selected RH kernels are available for the entire interval. These simplifications isolate the impact of the scheduling algorithm on performance, allowing the direct comparison between the two RH kernel schedulers. It is left to future work to examine the impact of these other factors on the scheduler's performance. I use multiple variables in this chapter to describe the behavior of applications, and their kernels. Table 10.2 lists many of the variables that will be used in this chapter.

To demonstrate situations where the original scheduler chooses a suboptimal allocation, I examine a scenario where two identical applications execute concurrently. In this scenario, each application contains two RH kernels (A and B). The RH implementation of each kernel occupies a single tile on the RH fabric, and the system contains exactly two tiles. Thus, for the given interval, the scheduler can select two of the four kernels for acceleration. For the first several examples I only evaluate situations where, in isolation, kernel A provides more application speedup than kernel B. In these examples, P_A and P_B represent the percent of execution time that the application spent in those kernels when executing entirely in software. Combined, they represent the percent of the application that can potentially be accelerated. S_A and S_B refer to the speedup of the RH implementation of these kernels over their software implementations. Note that the requirement that kernel A provides more application speedup than kernel B does not dictate that S_A is greater than S_B, but rather that the combination of S_A and P_A provide more speedup than

the combination of S_B and P_B. Equation 10.1 shows the condition under which kernel A provides a better speedup than kernel B.

$$1 - P_A + \frac{P_A}{S_A} \leq 1 - P_B + \frac{P_B}{S_B} \tag{10.1}$$

For the tests performed in this section, I set the value of an RH kernel to it's independent application speedup. The equation for this speedup is based off Amdahl's law, and is given in Equation 10.2. Although there are many different ways to calculate the value of an RH kernel , these other value functions also suffer from interdependencies amongst RH kernels. Chapter 10.3 examines an alternative value function, and explains why it also fails due to interdependent kernels.

$$IndependentApplicationSpeedup = \frac{1}{1 - P_k + \frac{P_k}{S_k}} \tag{10.2}$$

In the examples discussed in this section, the scheduler must choose between configuring kernel A for both applications, or configuring both kernel A and kernel B for one application and neither for the other application. The scheduler will never select kernel B for both applications because it provides less speedup to the application than kernel A. In this example, the original RH scheduler will always select the first option, configuring one copy of kernel A for each application, because the calculated value of kernel A is greater than that of kernel B. Equation 10.3 expresses the condition under which it is better to only accelerate the one application; it compares the geometric means of overall application performance for each option described above. In this equation, the left side represents the geomean of system performance when accelerating a single application, and the right side represents the geomean of system performance when accelerating a single kernel from each application.

$$\frac{1}{\sqrt{1 - P_A - P_B + \frac{P_A}{S_A} + \frac{P_B}{S_B}}} < \frac{1}{1 - P_A + \frac{P_A}{S_A}} \tag{10.3}$$

To demonstrate when this condition can occur, P_A was set to P_B. In this case, the minimum speedup kernel B must have (S_B) for the inequality of Equation 10.3 to hold is found. Figure 10.1 shows the value of S_B as I varied both P_A and P_B (which are equal in value). The required S_B value does not grow linearly with an increase in S_A for any of the tested coverage values. As total coverage grows, the required S_B also decreases for a fixed value of S_A.

Figure 10.2 takes a different approach, and fixes P_B at 30% while varying both P_A and S_A. The smallest S_B needed for the original scheduler to select a non-optimal set of RH kernels is found for each P_A and S_A combination.This graph shows that if P_B is constant, S_B increases as P_A increases, but also that as $P_A + P_B \rightarrow 100\%$, the minimum problematic S_B actually decreases because overall possible application speedup increases. For instance, when P_A equals 40% and S_A equals 10x, S_B only needs to be 4.3x to cause the original kernel scheduler to produce a suboptimal schedule. When P_A is very small, the wrong kernels are selected even for small values of S_A. However, when P_A is much smaller than P_B, it becomes increasingly unlikely that kernel A will be more useful than kernel B.

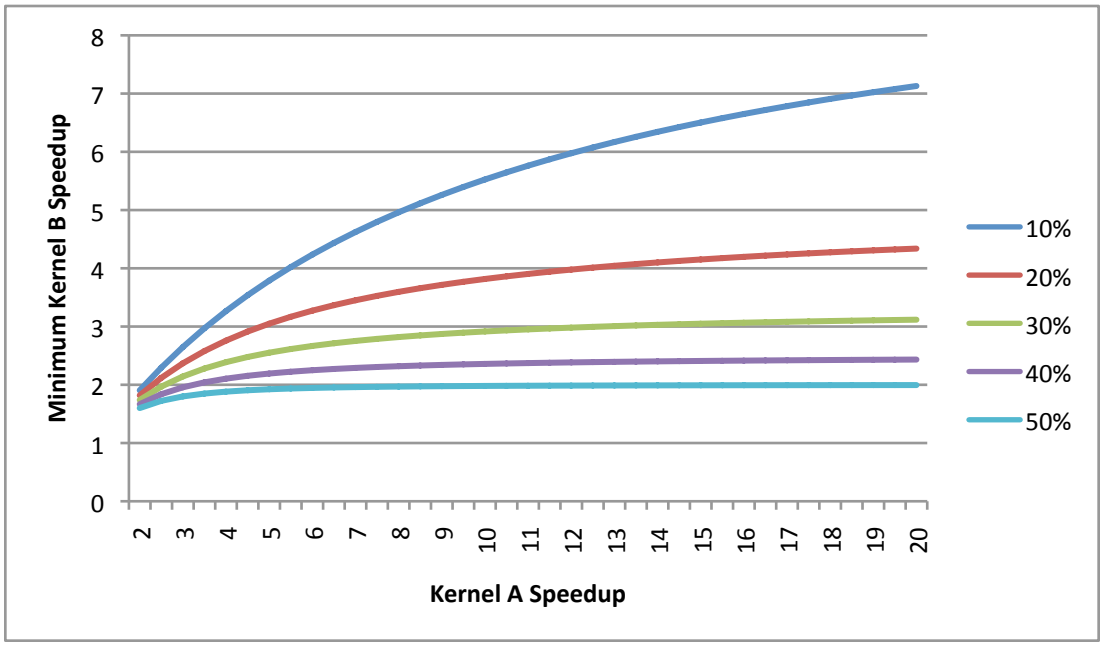


Figure 10.1 Conditions under which the old scheduler selects RH kernels that do not maximize system performance, where the percentage represents $P_A = P_B$

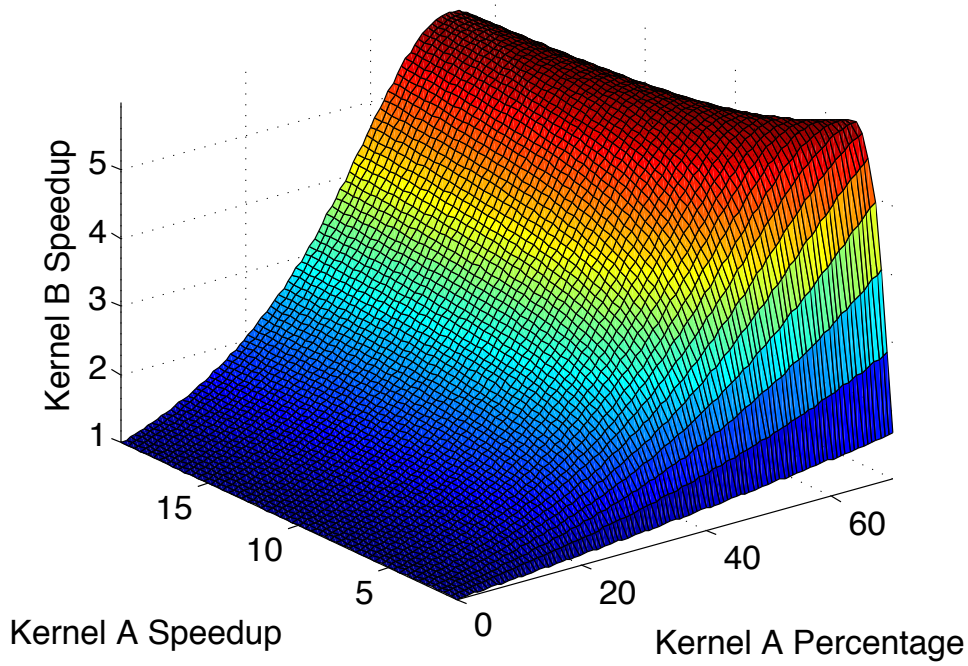


Figure 10.2 Kernel B speedup needed for old scheduler to perform sub-optimally

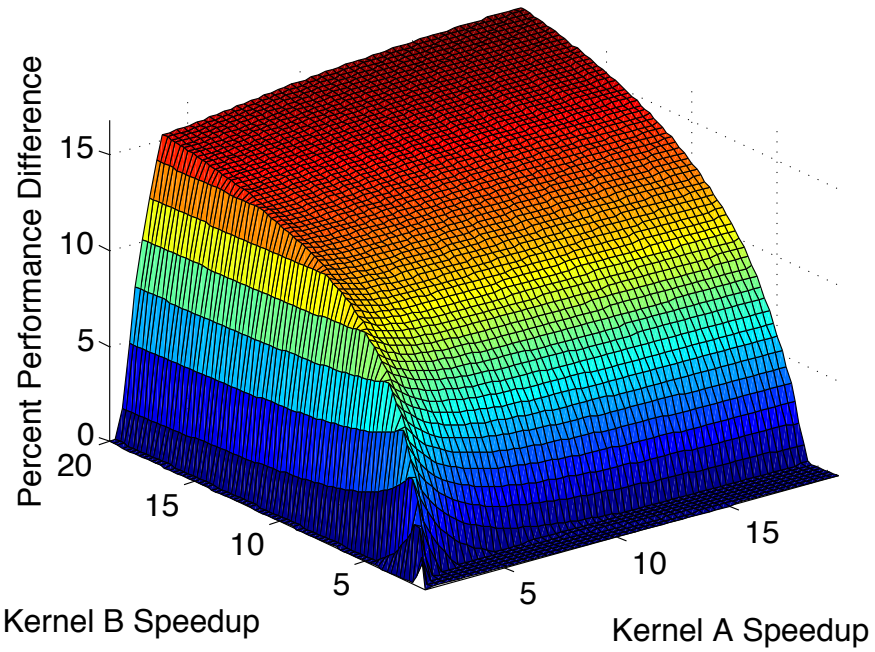


Figure 10.3 Performance difference between original and optimal scheduler when P_A = 45% and P_B = 35%

Although these graphs show that the original scheduler computes suboptimal allocations in some situations, they do not indicate the effect that these suboptimal schedules have on overall system performance. Figure 10.3 illustrates the performance degradation caused by using the original RH scheduler. In this figure, P_A is fixed at 45% and P_B is 35%, for a combined application coverage of 80%. For this experiment, kernel A is no longer constrained such that it provides a greater application speedup than kernel B. This graph shows that the old scheduler performs worse relative to the optimal scheduler as the kernel speedups increase. The performance degradation is over 12% when S_A and S_B are both 10x, and is almost 17% when S_A and S_B are both 20x. In other experiments, I varied both P_A and P_B and saw that the performance difference between the original schedulers decision and the right decision increased as the total application coverage increased ($P_A + P_B \rightarrow 100\%$).

10.3 Alternative Value Functions

In the previous Section, the original knapsack scheduler used the independent application speedup value function given in Equation 10.2. However, this is not the only method that a knapsack scheduler could use to calculate an RH kernel's value. Multiple different value function were evaluated in Fu et. al [40], with the best performing being the MCKP_TP value function. This value function is equivalent to the independent application speedup algorithm, which, as I showed in the previous section,did not create optimal selections.

The value function for an RH kernel scheduler should directly reflect the benefit that an RH kernel has on the system. In this work the RH kernel scheduler's goal is to maximize the performance of the system as a whole. Therefore the scheduler does not consider fairness, power, or any other number of factors that could be of benefit to the system. Any value function that the RH kernel scheduler uses should therefore directly reflect the performance benefit to the individual applications. One alternative to the independent application speedup value function is the cycles saved value function shown in Equation 10.4. This function is relatively straightforward, and shows the value of using an RH kernel as the number of cycles that would have been "saved" if the kernel executed on the RH instead of in SW.

$$CyclesSaved_k = N_k \times (T_{SW_k} - T_{RH_k}) \tag{10.4}$$

On first glance, this value function might appear to work in all cases, however it suffers from the same problem that the independent application speedup value function suffers from. This function calculates the number of cycles saved for a single kernel assuming that the application performs a given amount of work. When an RH kernel accelerates the application, the application performs more work during the same amount of time. This means that when the RH accelerates even a single kernel within an application, the number of cycles saved calculated using Equation 10.4 is not correct, because the the RH kernel would be called more often (increasing N_k), causing the application to perform more work. Within a single application, this is not a big deal, because the number of times the application calls a kernel is scaled by a constant factor (the overall application speedup) for all of the kernels in the application. This means that this value function still selects the optimal set of RH kernels when the system is executing a single thread.

When the system is executing multiple threads, the cycles saved value function may not obtain the correct solution because the "scaling factor" for the number of times an RH kernel has been called would be different amongst the various threads. This could result in a kernel with what appears to be a small number of cycles saved improving overall system performance more than one with a larger number of cycles saved value. This is similar to the example in Table 10.1 where a kernel with a relatively small independent application speedup was shown to be more useful than a kernel with a larger independent application speedup.

When reevaluating the value function, I made another observation: if all applications executing on the system contain only a single RH kernel, than the independent application speedup is the actual speedup of the application. This is because the actual application speedup cannot change because the scheduler cannot select another RH kernel belonging to the application. Because of this, the independent application metric can be used to exactly solve the scheduling problem when each application on the system contains only a single RH kernel. This fact will be useful when coming up with the hierarchical RH kernel scheduler in the next section of this chapter.

Although an RH kernel scheduler could be developed that dynamically calculates the value of the set of RH kernels selected each time it compared one selection to another, this would result in a problem that would not fit into the knapsack framework. Although the knapsack problem is in general an NP complete problem [72], when the

```
1  for each Application A
2      for each RH kernel k in A
3          Value_k = TimesCalled_k × (T_SW_k − T_HW_k)
4      Implementations_A = KNAPSACK_APP(A)
5      for each implementation I in Implementations_A
6          TotalValue_A[Tiles_I] = Cycles_A / (Cycles_A − CyclesSaved_I)
7  Sched = MODIFIED_MCKP(TotalValue, A)
```

Figure 10.4 Pseudocode describing the hierarchical RH kernel scheduler. KNAPSACK_APP computes the knapsack solution for an individual application; MODIFIED_MCKP combines the knapsack solutions into a single schedule.

weight of each object is a constant non-negative integer, a dynamic programming solution exists that can be solved in pseudo-polynomial time. Because of this, the existing knapsack solver's runtime is proportional to the number of kernels in the system times the maximum weight that can fit in the knapsack. Using a generalized knapsack solver to get around the problems associated with kernel interdependence would result in a solver that could be *m*uch slower than the current implementation. Solving such an NP complete problem would be infeasible for systems containing many kernels (8-CPU systems can easily have 50 or more kernels).

10.4 New Hierarchical RH Kernel Scheduler

Due to kernel value interdependence, the original knapsack RH scheduler does not produce optimal (as defined in Chapter 10.2) RH allocations when multiple applications, each containing multiple kernels, execute simultaneously. However, using the properties from the different value functions in Chapter 10.3, I decomposed the scheduler into two successive knapsack problems that combined, produce an optimal allocation of RH kernels. Figure 10.4 illustrates how I broken the problem down into multiple steps

First, I applied the knapsack solver to each application individually. Next, the OS calculated the value for each kernel in the application using the cycles saved function given in Equation 10.4. The knapsack solver then exactly determines which RH kernels should be loaded for each possible tile allocation for the application. One side-effect of the dynamic programming implementation of the knapsack solver is that, when calculating the solution for a given tile count it determines the knapsack solution at each smaller tile count. This means that the scheduler retains intermediate solutions that can be used in the next step to quickly determine which RH kernels each application should implement when they are allocated a set number of tiles.

The second level of the algorithm, uses a modified multi-choice knapsack problem (MCKP) solver [72] to find the overall best set of kernels for the total available RH tiles. The MCKP chooses not only which items to include in the knapsack, but also which version of those items to include (at most one version of a given item can be chosen). The second stage combines the possible application versions (the solutions calculated in the first stage) to find the

best global RH allocation solution. The "weight" of each possible application version is equal to the number of tiles it occupies, and the value of each version is the total application speedup provided by the combination of kernels it represents. This application speedup can easily be determined and is static at each potential tile size.

However, these changes alone do not allow the scheduler to exactly solve for the best mix of RH kernels. The knapsack solver maximizes the sum of values given it. Although this would be adequate if the goal was to maximize the average application performance, it does not optimize for the geometric mean of application performance. There are multiple ways to modify the algorithm to maximize the geometric mean of application speedups. One approach is to modify the values of each "implementation" of the application. Instead of setting these values to the speedup of the application implementation, they could be set to the log of the application's speedup. Doing this would cause the MCKP solver to actually maximize the products of application speedup because the addition of logs is equivalent to multiplication of the base numbers.

Using the log method is not ideal for a number of reasons. Calculating the log of a number can be relatively expensive, and when using fixed point calculations, as required in Linux kernel code, precision will be reduced, resulting in kernels that don't have the same speedup appearing equal after calculating their log. To get around these issues, we used a modified MCKP solver that instead optimized for the maximum product of values, rather than sum of values. This modification to the dynamic programming solution was relatively simple, and results in as good of speedups as using the log method. Making this change to the second-level MCKP solver enabled the new hierarchical RH kernel scheduler to select the RH kernels that maximize the geometric mean of each application's performance.

10.5 Setup

Prior chapters in this thesis used the Alexandrite full system simulation platform to analyze the performance of new features. Similarly, I also implemented the hierarchical RH kernel scheduler in the Linux kernel on the simulated system. I did this to verify that the scheduler worked on a real system, and to to ensure that the overhead of the new hierarchical RH kernel scheduler was similar to that of the original RH kernel scheduler. Although this full system simulation platform is extremely useful for some tests, it is also extremely slow, and provides far more detail than needed to analyze the performance of the schedulers.

In this chapter, I evaluate millions of different application scenarios to ensure that the hierarchical scheduler produces the correct results not only on the workloads already developed, but also on future workloads that may be encountered. This expansive of an evaluation is not feasible on the Alexandrite simulation platform. Furthermore, my goal was to test whether or not the scheduler makes the correct decisions based on the input data given it, rather than specifically motivating reconfigurable computing as a platform for acceleration. Therefore I used a simplified testbed to evaluate the hierarchical RH kernel scheduler.

10.5.1 Simulation Infrastructure

This work relies on a new simulation testbed designed to both generate synthetic workloads and analyze the resultant performance of the workloads when the RH kernel scheduler selects which RH kernels are loadedr. Millions of synthetic workloads can be generated on this simplified simulation testbed, a feat which would not be feasible on the full- system cycle-accurate Alexandrite simulation platform. Testing on so many workloads ensures that the scheduler functions correctly not only for the actual workloads that have been developed, but also for other workloads that may be encountered in the future. Using these synthetic workloads allows the modeling of a very large number of different simulations where kernel value interdependency could have an effect.

The simulation testbed used to verify the hierarchical RH kernel scheduler focuses only on the scheduling problem itself; modeling the RH kernel allocations produced by the periodic RH kernel schedulers across a large range of tile sizes. This testbed generated synthetic workloads containing multiple applications that are representative of real hybrid RH/SW applications. This testbed also performs the necessary scheduling operations for each workload to determine which RH kernels each scheduler would select at each tile size. The testbed randomly generates a set of kernels for each application in the workload. Each kernel has associated with it, the number of times it was called, its runtime in software, the runtime of the kernel when executed on the RH fabric (normalized to a count of CPU cycles), and the number of tiles that the kernel occupies.

The data generated about each application (and kernel) is the same data that a scheduler executing on a real system would use. Using this data, the testbed calculates the kernel allocation for the generated profile data, using both the original and hierarchical kernel scheduler. The testbed then estimates the workload performance for each selected kernel allocation.

10.5.2 Workload Generation

I generated fifteen million workloads to model the execution of two-, four-, and eight-processor systems (five million each). Each workload includes as many hybrid RH/SW applications as there are processors in the system. Like the hybrid RH/SW benchmarks described in Chapter 5.1, the main control flow of these applications is in software, with compute intensive kernels being able to execute on the RH fabric if selected. Each application spends the majority of its unaccelerated execution time in its kernels, and each kernel in the application has a meaningful impact on application performance when implemented in RH. This approach models the fact that designers are only likely to implement kernels that represent a significant portion of runtime and achieve a significant speedup when executed on the RH fabric.

For each application, I first genarated the P_{ALL} value (the percent of that applications software execution that could be accelerated in RH). Because the P_{ALL} value is very influential on the overall performance of the scheduler, I created multiple sets of tests with different distributions of P_{ALL}. Regardless of the range of P_{ALL}, I used a

Parameter	Distribution	Mean	Range
P_{ALL}	Normal	varies	varies
Number of kernels in application	Normal	varies	[1,varies]
S_k	Log-normal	12x	[2x,53x]
Kernel Size	Log-normal	4	[1,13]

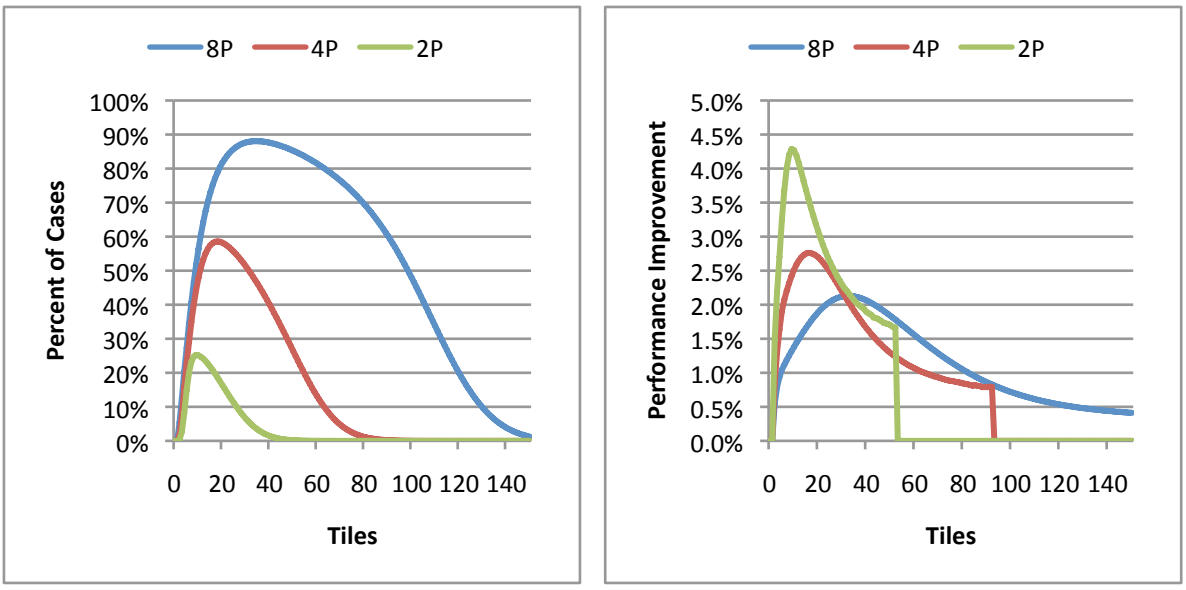
Table 10.3 Parameters used to generate the synthetic applications

normal distribution to select the P_{ALL} values. I set the mean and standard deviation of this range such that 99.7% of the application's P_{ALL} values fit within the range used (values generated outside that range were clipped). For the baseline test I set P_{ALL} to be between 50% and 100%. This range gives a mean of 75%, and is similar to many of the hybrid RH/SW workloads examined in Chapter 5.1. I also simulated workloads with a P_{ALL} range between 90% and 100% and an average P_{ALL} of 95%. I did this to see how the hierarchical RH kernel scheduler performs on applications that are very amenable to RH acceleration.

I selected the number of kernels in each application using a normal distribution between one and a factor multiplied by the application's P_{ALL} value. The test generated generated the number of kernels (NumKerns) such that applications that have a large coverage are more likely to have more RH kernels. Although this is not the case for the AES benchmarks examined in Chapter 5.1, AES is a special case because the entire application executes inside a single kernel. In real systems, a kernel like AES is more likely to be used in conjunction with other code that produces data that needs to be encrypted. Xvid works as a much better example of a hybrid RH/SW application. The RH coverage of Xvid is dependent on how many compute-intensive functions in the application have RH implementations. In most of the tests, I used a multiplication factor of 10, so if an application had $P_{ALL} = 100\%$, it would contain between one and ten RH kernels with a mean of five. However, in some of the workloads, I modified this factor to examine how the number of RH kernels in an application impacts the performance of the RH kernel schedulers.

Next the testbed determined the coverage, size, and speedup of the individual kernels in each application. The sum of the coverages of the kernels in an application must sum to the P_{ALL} chosen for that application. To calculate kernel coverages for an application, P_{REMAIN} (the coverage remaining to be included in one or more kernels) is initially set to P_{ALL}. The testbed then iterates over NumKernels, and for each kernel K generates P_{KERNEL} (the coverage of the individual kernel) using a normal distribution in the range [2%, $P_{REMAIN} - (2\% \times (NumKernels - K))$]. The testbed assigns the last kernel in the application a coverage of P_{REMAIN}. This method produces applications that tend to contain one or two kernels that correspond to the majority of the time spent in kernels, along with multiple kernels that account for a smaller portion of the application's execution.

The testbed generated the speedup of each RH kernel using a log-normal distribution with an expected value of 12 and a standard deviation of 7.1. Thus most kernels will have close to, but slightly below a 12x speedup, and a few



(a) Percent of time the hierarchical scheduler outperformed baseline (b) Performance improvement when hierarchical scheduler was better

Figure 10.5 Advantage of the hierarchical RH kernel scheduler over the original RH kernel scheduler when P_{ALL} is in the range of [50%,100%], and the kernel multiplier factor is 10

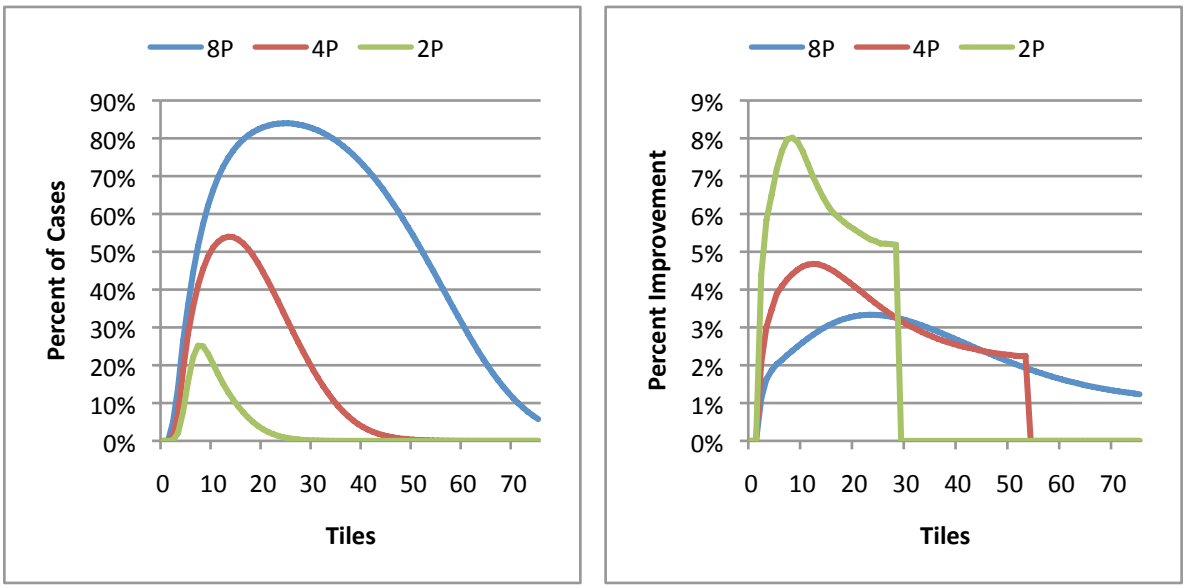
will have much higher speedups. The testbed clipped the speedup values such that they are between 2x and 53x. This models actual kernel speedups seen in the hybrid RH/SW applications in Chapter 5.1 where many RH kernels have speedups of less than 10x and a few have much larger speedups.

Finally, the testbed generates each RH kernel's size using a log-normal distribution with an expected value of four tiles and a standard deviation such that 99.7% of values are between one tile (the minimum allowed), and thirteen tiles. The testbed rounded fractional tile count down to the next-lowest integer. This creates a distribution with many small RH kernels and a few very large ones, similar to the distributions seen in the benchmarks presented in Chapter 5.1. The testbed does not correlate the size of the RH kernel with the speedup of the kernel, because this has not been observed in the hybrid RH/SW benchmarks examined in Chapter 5.1[1]

10.6 Results

In the baseline case, workloads have a P_{ALL} in the range of [50%, 100%], and a kernel multiplier factor of 10 (when $P_{ALL} = 100\%$, there will be between one and ten RH kernels in each application, with an average of 5). Figure 10.5(a) shows the percent of cases where the new scheduler chose an allocation that resulted in a better system performance than the old scheduler. For the remaining cases, both schedulers generated the same schedules. When

[1] Although, alternate implementations of the same RH kernel are likely to be faster when they occupy more area, large and small kernels tend to have a similar distribution of speedups.



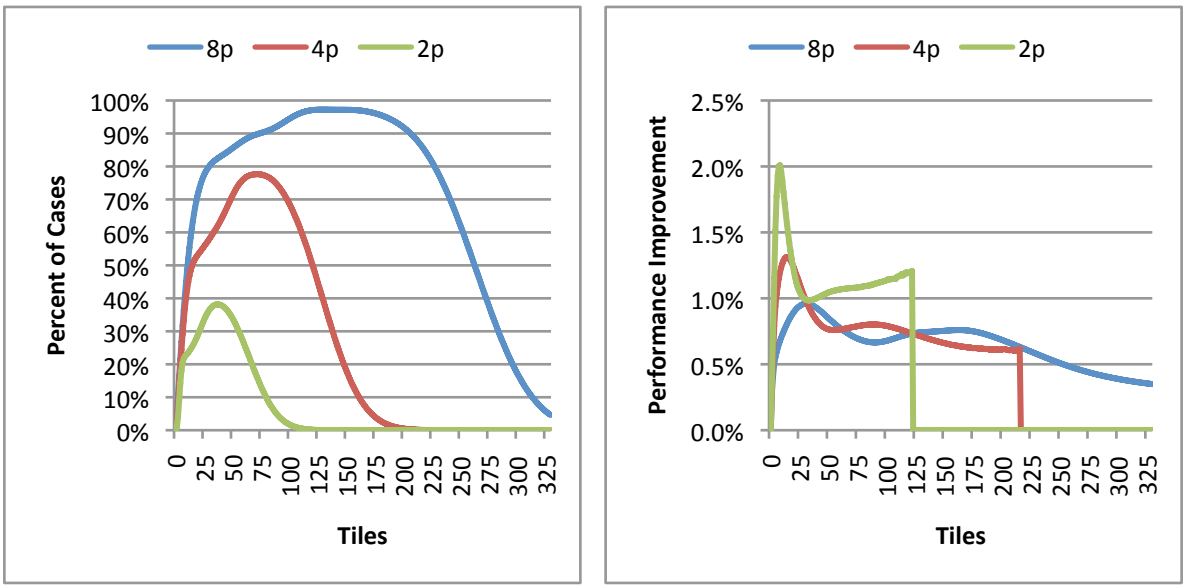
(a) Percent of time the hierarchical scheduler outperformed baseline (b) Performance improvement when hierarchical scheduler was better

Figure 10.6 Advantage of the hierarchical RH kernel scheduler over the original RH kernel scheduler when P_{ALL} is in the range of [50%,100%], and the kernel multiplier factor is 5

few RH tiles were available, many kernels would not fit on the RH fabric, constraining the scheduling decision. Thus, the two schedulers are more likely to choose the same allocation than they would if more tiles were available. As more RH tiles are added to the system, the new scheduler has more room with which to leverage the performance benefits of interdependent kernels. The old scheduler does not perceive these interactions, and thus cannot make use of them. As the number of RH tiles further increases, more of the kernels simultaneously fit in RH (in many cases, all of the kernels can fit on the RH), reducing the number of different possible allocations and increasing the likelihood that the old scheduler will make the right decision. This figure also shows that systems containing more processors are more likely to produce schedules that differ between the original and hierarchical RH kernel schedulers. This is because systems containing more processors have far more possible schedules to choose from, increasing the likelihood that processors will have interdependent kernels that do not get properly selected when using the old scheduler.

Figure 10.5(b) shows the performance improvement of the hierarchical scheduler over the original scheduler in the cases where the two schedulers produced different schedules[2]. Performance followed a similar trend for two-, four-, and eight-processor workloads. These plots also followed the same trend observed in Figure 10.5(a), with the biggest difference in performance occurring when the workloads were most likely to have different schedules. Unlike Figure 10.5(a) however, workloads containing more processors performed better than those containing few processors. This

[2]For all of the performance results in this chapter, only data for situations where more than .1% of cases had differing schedules are shown. For situations where less than .1% of the tested cases showed a performance difference, the performance improvement was set to 0%.



(a) Percent of time the hierarchical scheduler outperformed baseline (b) Performance improvement when hierarchical scheduler was better

Figure 10.7 Advantage of the hierarchical RH kernel scheduler over the original RH kernel scheduler when P_{ALL} is in the range of [50%,100%], and the kernel multiplier factor is 25

is because in workloads containing many processors, kernel interdependence is likely to only impact the scheduling of a couple of the applications. Therefore the other applications will likely be relatively unaffected. Due to the fact that I measured performance using the geometric mean, the performance difference on most of these eight-processor workloads is likely to be less than they would on a two-processor workload. In general, the more processors the system contains, the more important it is to use a hierarchical RH kernel scheduler.

In the next tests I constructed the workloads differently to examine the impact that having more or less RH kernels in each application has on overall system performance. In Figure 10.6 the kernel multiplier factor was decreased to five. Therefore, applications in these workloads have approximately half as many kernesl in them as the workloads examined in Figure 10.5. Similarly in Figure 10.7, I increased the kernel multiplier factor to 25, increasing the number of RH kernels in each application by approximately 2.5x. Comparing these graphs, a trend emerges showing that the more RH kernels an application has, the more likely the schedules are to be different. This is because applications with more kernels are more likely to have interdependent kernels. When the kernel multiplier is low, applications are more likely to have only a single kernel. In these cases it is impossible for kernels to be interdependent.

More importantly, these graphs show that the difference in performance between the hierarchical RH kernel scheduler and the original RH kernel scheduler is likely to be much greater when applications contain few kernels. When applications contain few kernels, adding a single kernel to the application (when one already exists), is likely to have

a bigger impact on performance than if the application contained many RH kernels. Applications with many RH kernels are likely to contain many kernels that cover only a small portion of the application's execution time. If these RH kernels are interdependent, the hierarchical RH kernel scheduler will perform better than the original RH kernel scheduler, however the difference in performance will be less because the added RH kernel(s) will not impact application performance as drastically.

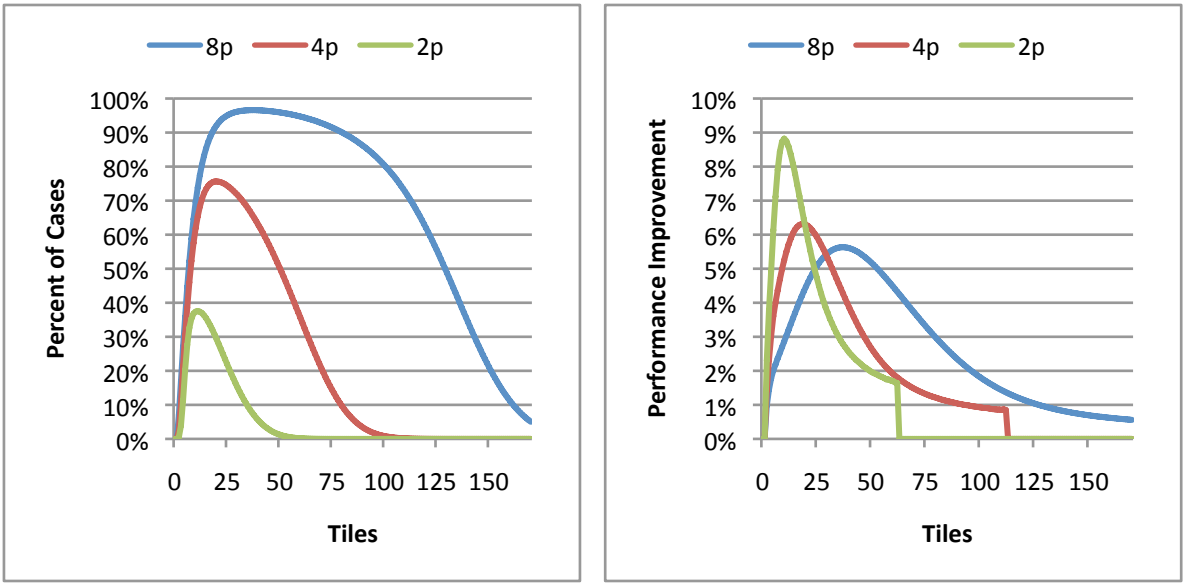
In addition to adjusting the number of RH kernels that are in each application, I also adjusted the range of P_{ALL}. Figure 10.8 P_{ALL} shows the results of the system when using a P_{ALL} in the range [90%,100%]. This is representative of applications that have a much larger portion of their software execution time covered by RH kernels, and are likely to have larger application speedups than applications with low P_{ALL} values. Increasing this value both increased the probability that the hierarchical RH kernel scheduler would outperform the original scheduler, and increased the performance benefit that the hierarchical RH kernel scheduler could achieve. The is due to having slightly more kernels available (increasing the chance that the hierarchical scheduler outperforms the old one), and the increase in application coverage. Increasing the coverage of the application means each RH kernel covers more of the application, and applications are likely to have very large speedups. Adding a kernel that covers 5% of the software's execution time to an application that already has 90% of its execution covered by configured RH kernels is likely to have a much bigger impact on the application's performance than adding the same kernel to an application that only has 75% of its software-only execution covered by configured RH kernels.

Figure 10.9 shows the impact of increasing the application's P_{ALL} value, while simultaneously decreasing the kernel multiplier factor. Doing this increases both the probability that the schedulers will produce different schedules, and increases the performance difference between them. The effect of both increasing P_{ALL} and decreasing the total number of RH kernels appear to be additive in nature. This figure shows a relatively extreme example of difference in performance between the two RH kernel schedulers.

10.7 Hierarchical Scheduler Conclusions

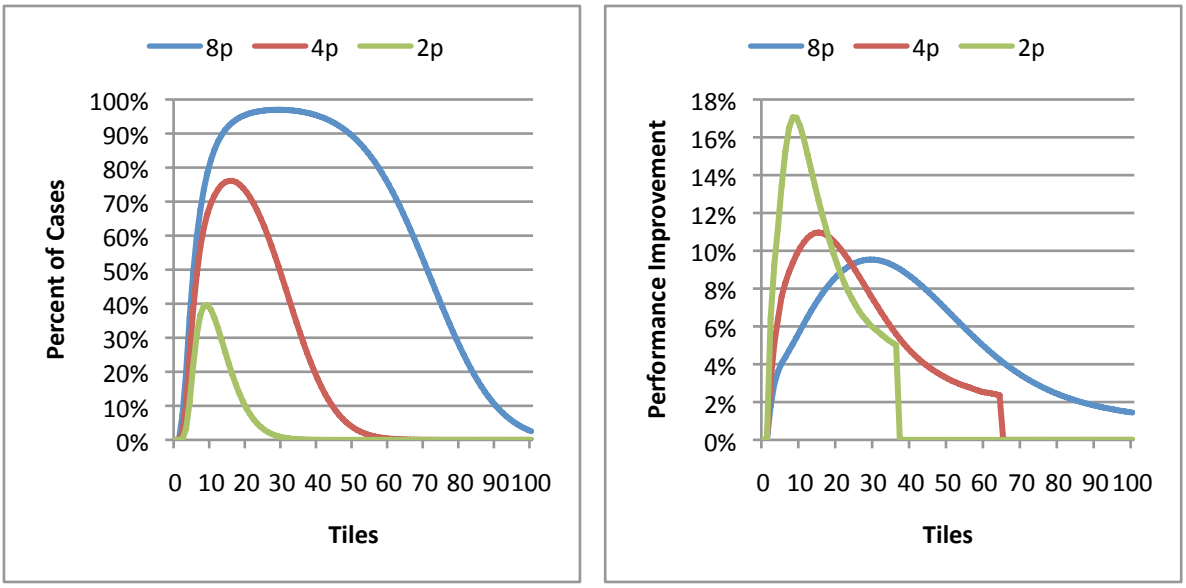
In this chapter, I first examined the original RH kernel schedulers, and noted that it could generate non-ideal solutions to the scheduling problem when executing workloads where multiple applications containing multiple RH kernels. This was because the performance benefit of an RH kernel depends on what other RH kernels the application is using. I then examined why this happens to help come up with a solution to the problem.

I found that properties of the RH kernel scheduler allowed it to always generate the correct schedulers under certain controlled conditions. Because of this, I could reformulate the scheduling problem into two separate knapsack problems. This new hierarchical RH kernel scheduler always outperformed the original RH kernel scheduler. In examining the results, I saw that the performance advantage of using the new RH kernel scheduler was great on systems containing more cores because of the increased likelihood that applications on these systems would contain dependent RH kernels that altered the scheduling decision of the original RH kernel scheduler. Therefore future



(a) Percent of time the hierarchical scheduler outperformed baseline (b) Performance improvement when hierarchical scheduler was better

Figure 10.8 Advantage of the hierarchical RH kernel scheduler over the original RH kernel scheduler when P_{ALL} is in the range of [90%,100%], and the kernel multiplier factor is 10



(a) Percent of time the hierarchical scheduler outperformed baseline (b) Performance improvement when hierarchical scheduler was better

Figure 10.9 Advantage of the hierarchical RH kernel scheduler over the original RH kernel scheduler when P_{ALL} is in the range of [90%,100%], and the kernel multiplier factor is 5

multicore systems should use the new hierarchical RH kernel scheduler to obtain the best performance from the RH coprocessor.

Although this work illustrated the usefulness of the proposed hierarchical RH kernel scheduler, if multiple copies of the same application were executing, or multiple applications contained the same kernel(s), this scheduler might not best utilize the RH fabric. This is because sharing these RH kernels, as was done in Chapter 9 , could result in better system performance. Future work is necessary to evaluate schedulers that recognize the presence of shared RH kernels, and estimates system performance when sharing these kernels. However, the added complexity of this problem means that is unlikely to fit nicely within the knapsack framework. It is likely that a heuristic-based scheduler would be required to perform the scheduling operations necessary within a reasonable amount of time.

Chapter 11

Sharing an RH Fabric

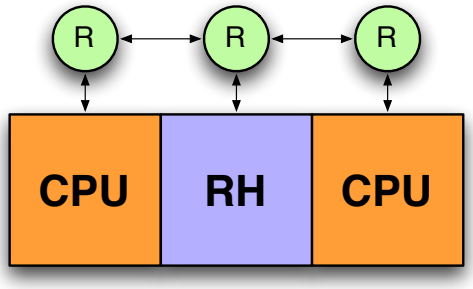
Although this thesis has, up till now, focused on multicore platforms that contain a single shared RH fabric, I have not yet presented data to justify why the RH fabric should be shared by all of the CPU cores. Intuitively, sharing a RH fabric seems logical, particularly on systems that execute a mix of hybrid RH/SW applications alongside traditional software-only workloads. By sharing the RH on these systems, RH space is not "wasted" when one of the cores executes software-only code. For example if a system has two cores, and is only executing one hybrid RH/SW application, the single hybrid application can make use of the entire RH fabric. This would not be possible if each core had its own private RH fabric. In this situation, it is clear that having a shared RH fabric can result in better performance, however it is unclear how advantageous this strategy is on systems executing only hybrid RH/SW applications.

This chapter addresses these oversights, and compares systems that share a RH fabric with those that don't. In particular, it examines the performance improvements of using the dynamic RH kernel scheduler on a shared fabric has over a partitioned RH fabric. By performing this analysis, I better justify the design decision of using a single shared RH fabric in the proposed multicore RH computing system.

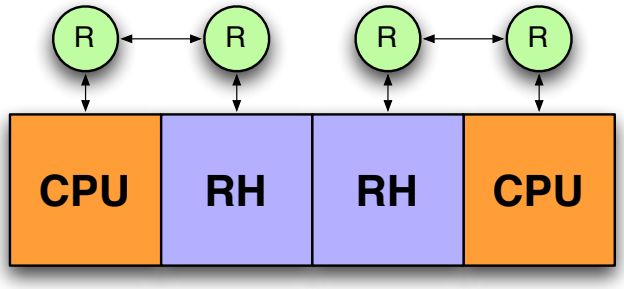
11.1 System Properties

Chapter 2 examined many different ways that CPU could communicate with the RH fabric on a single-chip shared-memory system. For the purpose of this study, I only consider systems similar to the one proposed in Chapter 3. In this system, direct communication between the CPU and the RH fabric uses the direct-communication network to query RH kernels and determine if they have finished executing, and the shared cache hierarchy is used for all other data transfers.

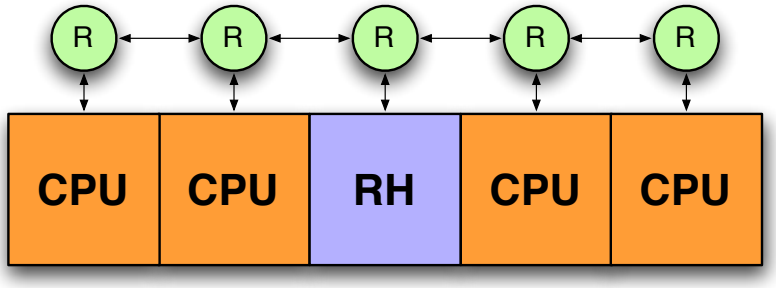
Figure 11.1 shows some of the ways that I could partition the RH across multiple processor cores. This chapter examines not only the simple dual core cases where the cores share a single RH fabric (Figure 11.1(a)) or where each core has its own private RH fabric (Figure 11.1(b)), but also situations where multiple cores share one or two RH fabrics. For example, Figures 11.1(c) and 11.1(d) show two different ways four CPUs share an RH fabric. Note, this chapter only examines situations where N cores share an RH fabric, or $\frac{N}{2}$ cores share a partitioned RH fabric.



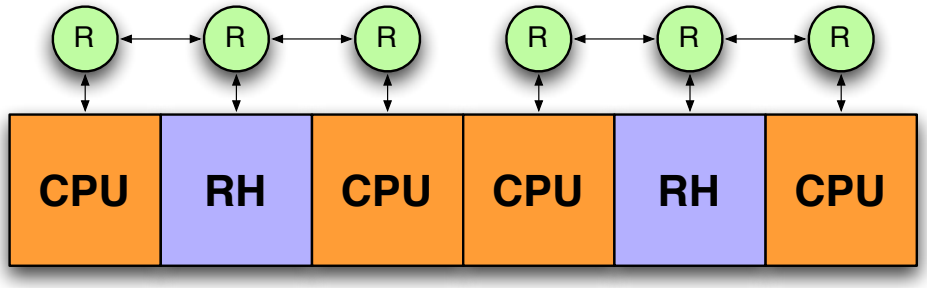
(a) Example network topology when two processor cores share a single RH fabric



(b) Example network topologies when two processor cores each have a single RH fabric



(c) Example network topologies when four processor cores share a single RH fabric



(d) Example network topologies when four processor cores share two RH fabrics

Figure 11.1 Examples of different ways the processor cores can access the RH on both two- and four-processor systems

Distance	Additional Latency (processor cycles)
2	3
3	6
4	10
5	13
6	15
7	18
8	20

Table 11.1 Additional latency experienced by RH kernels that are more than one hop away from the RH fabric

This chapter compares systems that contain two-, four-, eight-, and sixteen-processor cores. In the first set of experiments, the runtime of RH kernels is constant regardless of how many CPU cores share the RH fabric. Under this baseline, systems with an RH fabric shared by all of the processor cores always performs as good as, or better than systems with a partitioned RH fabric . The remainder of the systems consider the impact of the additional direct-communication latency needed on systems using a shared RH fabric. Because RH kernels only use the direct communication mechanism at the beginning and end of an RH kernel's execution, I have set the latency to access this buffer to a constant value, regardless of an RH kernel's execution time. Therefore this latency will be far more pronounced on RH kernels with short execution times.

I used the "distance" between the RH controller and the CPU in question to calculate the direct communication latency. Although different RH kernels could experience different penalties for accessing this network (for instance, it is slightly greater in the SAD8 kernel than many others), I used a constant value in this study for all RH kernels of the same distance. I estimated these constants by evaluating the impact of the distance between the RH controller and CPU cores on the runtime of RH kernels on the Alexandrite platform. Table 11.1 shows the additional number of processor cycles added to each configured RH kernel's execution time based on the RH's fabric's distance from the processor core that initiated the RH kernel.

The experiments in this chapter do not model the impact that the number of processor cores (and subsequently RH kernels) sharing an RH fabric have on the execution time of the actual RH kernels. I chose to do this because the multicore experiments performed in Chapter 7 do not show a significant slowdown in most of the workload's "efficiency" beyond that which exists on single and dual-processor systems.

11.2 Setup

A modified version of the simulation infrastructure developed in Chapter 10 measured the performance of the different system setups examined in this chapter. This allowed for the examination of a wide variety of scenarios at

Parameter	Distribution	Mean	Range
P_{ALL}	Normal	.75	[.5,1]
Number of kernels in application	Normal	varies	[1,varies with a maximum of 10]
S_k	Log-normal	12x	[2x,53x]
Kernel Size	Log-normal	4	[1,13]
RH Kernel Runtime	Normal	varies	varies

Table 11.2 Parameters used to generate the synthetic applications

a level of detail that focuses on the ability of the RH kernel scheduler to select appropriate RH kernels when using a shared RH fabric as well a partitioned one.

I setup the simulations for these experiments similar to those in Chapter 10. Table 11.2 lists the parameters I used to generate the synthetic workloads. These workloads added a new parameter to the setup: RH kernel runtime. This is the runtime of the kernel when executed on the RH fabric. This value was needed because the static communication latency overhead impacts RH kernels with short execution times the most. The testbed generated the RH kernel's runtime using a normal distribution in one of four ranges: [50, 2000], [75, 1000], [75, 500], and [75, 200]. I selected to use the distribution of 50 to 2,000 cycles because those are similar to the distribution of RH kernel runtimes seen in the RH kernels described in 5.1, but I also examine lower values to show the effect on systems where the setup and finish time of the RH kernels can dominate their execution.

The testbed generated workloads for two-, four-, eight-, and sixteen- core RH computing systems. Each workload contained as many hybrid RH/SW applications as there were cores on the system. Once the testbed generated each workload, it measured the performance on both systems with a shared and partitioned RH fabric. In cases with a partioned RH fabric, each partition contained half of the RH tiles that the single shared RH fabric had. Therefore both systems contained an equal number of tiles of the RH fabric. The testbed generated 250,000 workloads for each system, allowing the examination of a wide range of synthetic benchmarks.

11.3 Results

Figure 11.2 shows the speedup when using a single shared RH fabric over a partitioned one for two-, four-, eight-, and sixteen-processor workloads. For this test, I set the runtimes of each RH kernel to a constant value, regardless of how far the RH was from the CPU core.This graph shows that partitioning a system with few processor cores has the greatest impact on performance. As more processor cores share a single RH fabric, the penalty for splitting the RH fabric into partitions is greatly reduced. This is because the performance characteristics of a multiple applications are likely to be similar to other collections of applications, even if individual applications performance characteristics can vary significantly.

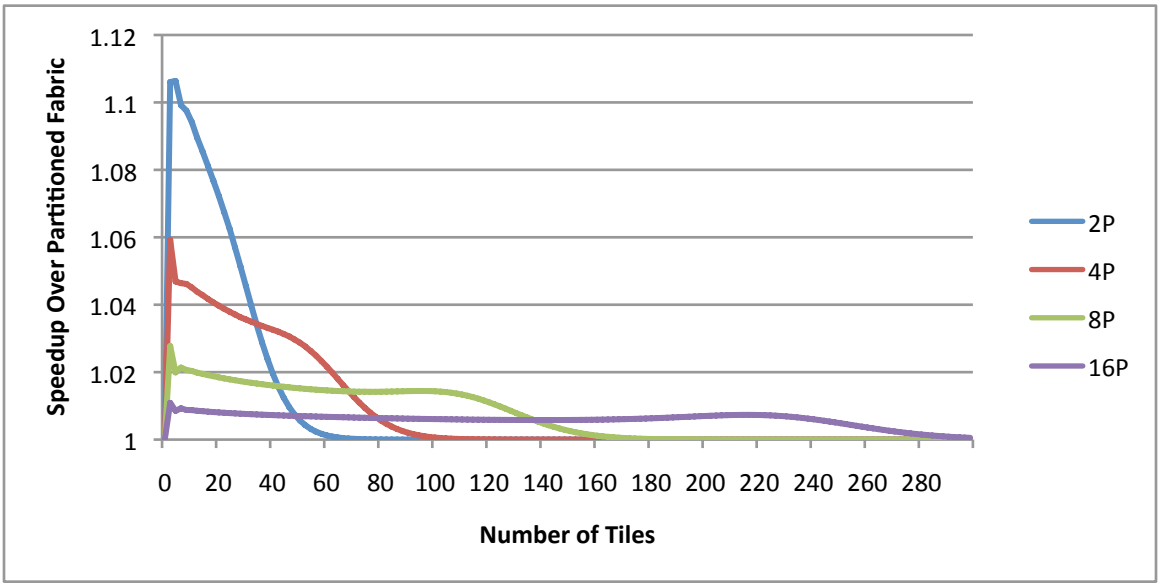
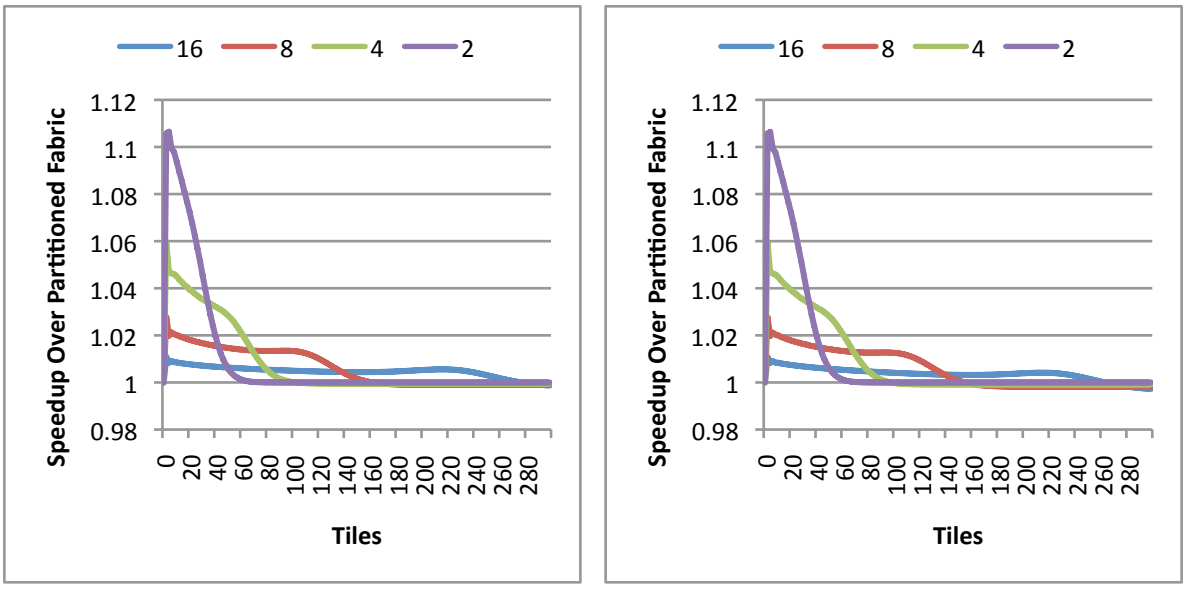


Figure 11.2 Performance advantage of using a shared RH fabric over a partitioned RH fabric when discounting communication overhead

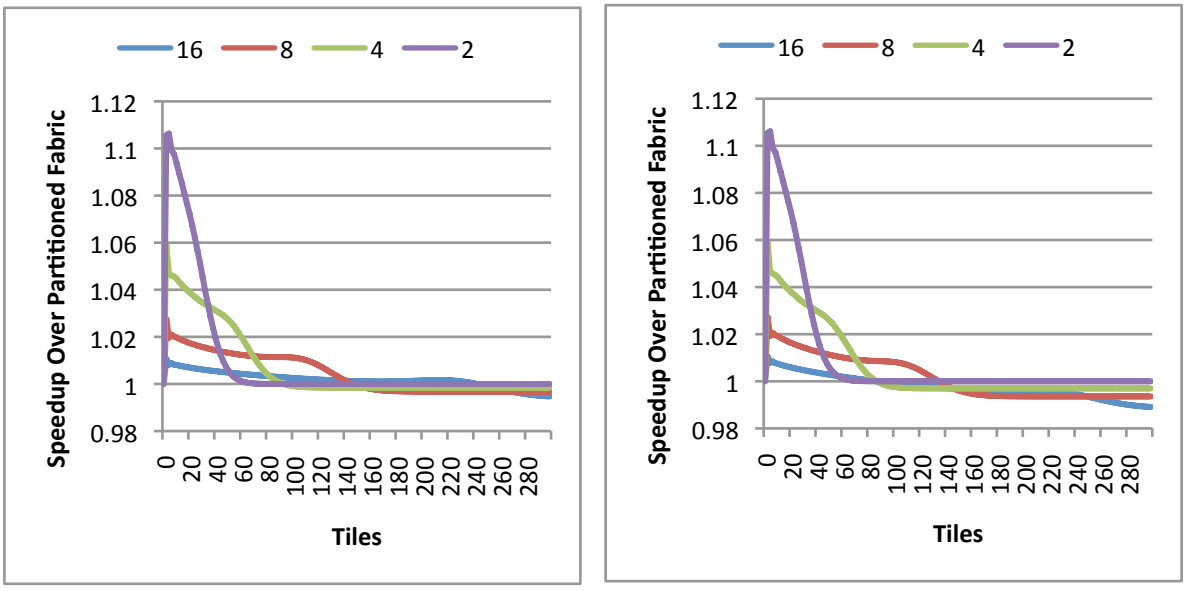
Experiments with each of the other processor core counts followed similar trends. The performance advantage of sharing the RH fabric is greatest when few RH fabric tiles are available in the system, peaking when only two tiles are available. This peak is caused because many RH kernels cannot fit on fabrics containing few RH tiles, whereas on a shared fabric a single RH kernel could be configured. The remainder of the performance difference is due to having a larger selection of RH kernels to choose from when selecting RH kernels. As more RH tiles are available, the performance difference drops. With few processor cores, this performance drop-off is rapid, however in systems containing many cpu cores, the drop-off is delayed significantly, with relatively even performance across a wide range of tile sizes (only dropping significantly when sufficient RH resources are available such that all of the RH kernels can fit simultaneously).

Figure 11.3 shows the impact of varying the average RH kernel runtime on the speedup of using a shared RH fabric over a partitioned fabric. These graphs are very similar to that of Figure 11.2, however, the performance of the partitioned applications is slightly higher than the shared fabric when sufficient RH resources are available. This is due to the increased latency larger systems experience when accessing the RH kernels. Figure 11.3(d), presents an extreme case, where the average runtime of the RH kernels is greatly reduced, to the point where the RH kernel runtimes are similar to the SAD kernels in the Xvid benchmark. This is an important data point, as it shows that even for workloads containing only short-running RH kernels, the shared RH fabric outperforms the partitioned fabric at most tile sizes that are appropriate for the number of processors sharing the RH fabric. In this extreme case, the sixteen-processor core workloads performed $\sim 1\%$ worse when using a shared fabric with sufficient RH resources. However, when operating with smaller tile counts (which are more likely to be available on systems), the shared fabric outperformed



(a) Speedup when RH kernels runtimes are in the range of 50–2,000 CPU cylces

(b) Speedup when RH kernels runtimes are in the range of 75–1,000 CPU cycles



(c) Speedup when RH kernels runtimes are in the range of 75–500 CPU cycles

(d) Speedup when RH kernels runtimes are in the range of 75–200 CPU cycles

Figure 11.3 Speedup of a fully shared RH fabric versus one where the RH fabric is split into two partitions when varying the runtimes of the RH kernels.

the partitioned fabric, albeit by a much lesser degree when compared to systems that executed RH kernel's with shorter average runtimes.

11.4 Shared RH Fabric Conclusions

This chapter analyzed the impact of sharing a RH fabric amongst all processor cores in the system versus partitioning the fabric so that only half of the processor cores can access it at a time. This data shows that an optimized dynamic RH kernel scheduler allows processors that share an RH fabric to obtain higher performance than processors that use a partitioned fabric. These results have also shown that, even when RH kernels have very short execution times (only a couple hundred of cycles), the penalty for using an RH coprocessor located further away is more than offset by the improved performance of a dynamic RH kernel scheduler. Using a private RH fabric for each core is not a good idea in systems where RH resources are constrained, because a dynamic RH kernel scheduler can obtain much better performance on shared fabrics. However, as the number of processors on the system increases, the performance advantage of using a single shared fabric is minimized (compared with partitioning the fabric in two). Because of this, systems with limited RH resources should share their RH resources amongst as many processor cores as feasible. If they must partition the RH fabric, they should attempt to still share each RH coprocessor amongst at least four processor cores.

Chapter 12

Comparison with Vector Processors

Single instruction multiple data (SIMD) or vector instruction extensions are commonly used to accelerate general-purpose processors. The SIMD instructions work by running multiple copies of an instruction in parallel. For instance, a 64-bit SIMD instruction can execute eight 8-bit additions at the same time. The most well known of these instruction extensions are the MMX and SSE instruction set extensions for the x86 architecture. However, many common general-purpose processors support SIMD extensions, including the ARM Neon, Power PC Altivec, and UltraSparc VIS extensions. SIMD instruction. These SIMD extensions sets have proven to be very useful in many application domains, particularly in multimedia processing [102, 80].

12.1 SIMD Performance

In this chapter, I compare the performance of RH kernel accelerators with SIMD accelerators on the x86 platform. Table 12.1 lists the platforms that I examined.

12.1.1 Xvid Performance

The first tests I performed were from the Xvid 1.0.3 video encoder. I profiled all of the Xvid kernels listed in Chapter 5.1, including: SAD8, SAD16, FDCT, Transfer 8 to 16, Average-2, Average-4 and interpolate 8x8 6-tap kernels. Each of these kernels had two different implementations, the "standard" one written in C, and an assembly-optimized version that took advantage of the x86 processor's SIMD extensions.

Vendor	Model	Speed
Intel	Core i7	2.8GHz
Intel	Pentium 4	3.66GHz
Intel	Atom N280	1.66GHz
AMD	Opteron	1.6GHz

Table 12.1 Processors used for the SIMD comparison

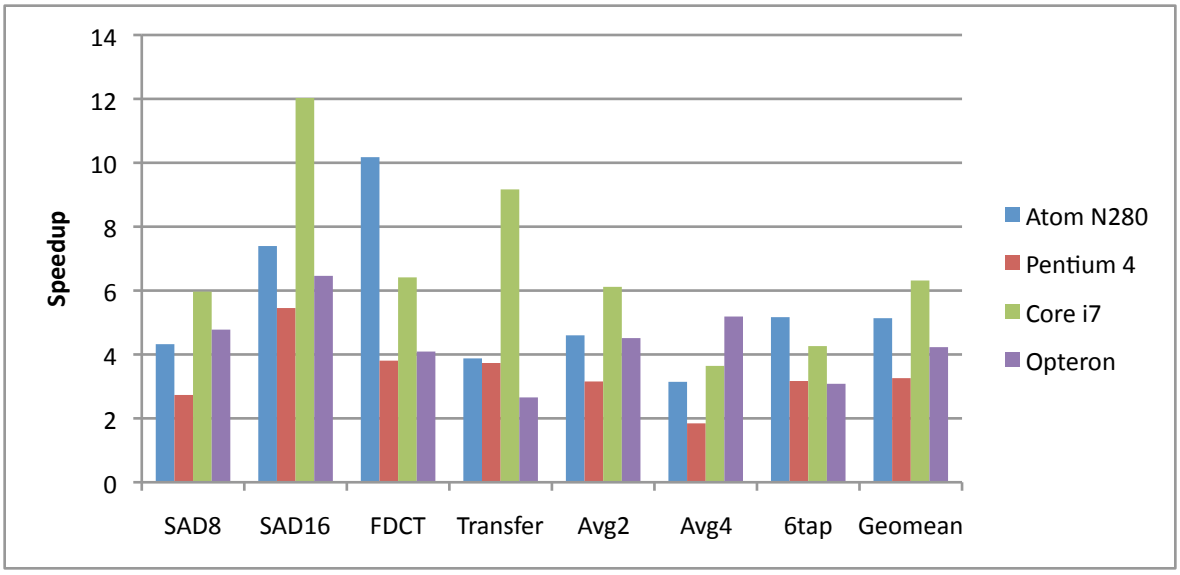


Figure 12.1 Vector coprocessor speedups for various Xvid kernels when executing on PC hardware.

Figure 12.1 shows the performance of each RH kernel on each of the tested SIMD platforms. The SIMD speedup of each kernel varied across platforms due to the varying ability of each processor's core to exploit instruction-level parallelism in the unoptimized C-code. This section focuses on the Atom processor's performance because it is designed for the high-end embedded market. Therefore the processor's performance is likely to closely resemble the CPU cores modeled in this work. Because of this, its SIMD unit should more closely resembles one that would compete with the proposed RH coprocessor.

I used the speedup of the SIMD kernels on the Atom processor to estimate the "SIMD-accelerated" execution time of each kernel in Xvid. To do this, I set each kernel's SIMD runtime to be its unaccelerated runtime on the simulated platform divided by the speedup of the kernel using SIMD acceleration on the Atom platform. Using these new runtimes, I calculated the SIMD-accelerated runtime that the Xvid application would be likely to have on the processor cores described in this thesis. In this SIMD model, the non-kernel portions of Xvid executed at the same speed as they originally did. This is in contrast to the non-kernel portions of hybrid RH/SW applications, which, as Chapter 6.3 showed, ran slightly slower. Although the Xvid encoder used in this study contained SIMD-accelerated versions of kernels besides the seven examined in this thesis, these additional kernels did not have RH kernel counterparts (although such kernels could be implemented), and are therefore not included. This allows for a fair comparison between the existing RH kernels and the SIMD versions of the same hardware kernels.

Figure 12.2 shows the breakdown of Xvid's execution on both the modeled SIMD system and a reconfigurable computing system with sufficient RH tiles to hold all the RH kernels simultaneously. Using the RH coprocessor reduced Xvid's execution more than the modeled SIMD extensions did, resulting in a reduced execution time. In

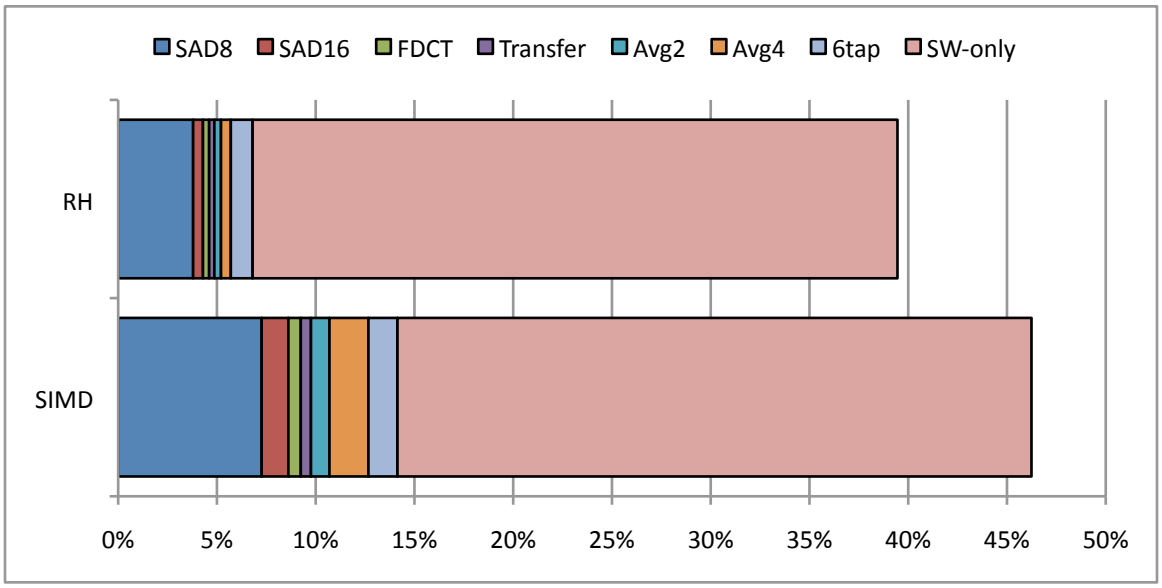


Figure 12.2 Percent of the original SW-only execution time of Xvid (and its kernels) when accelerated using RH (top) and SIMD instructions (bottom).

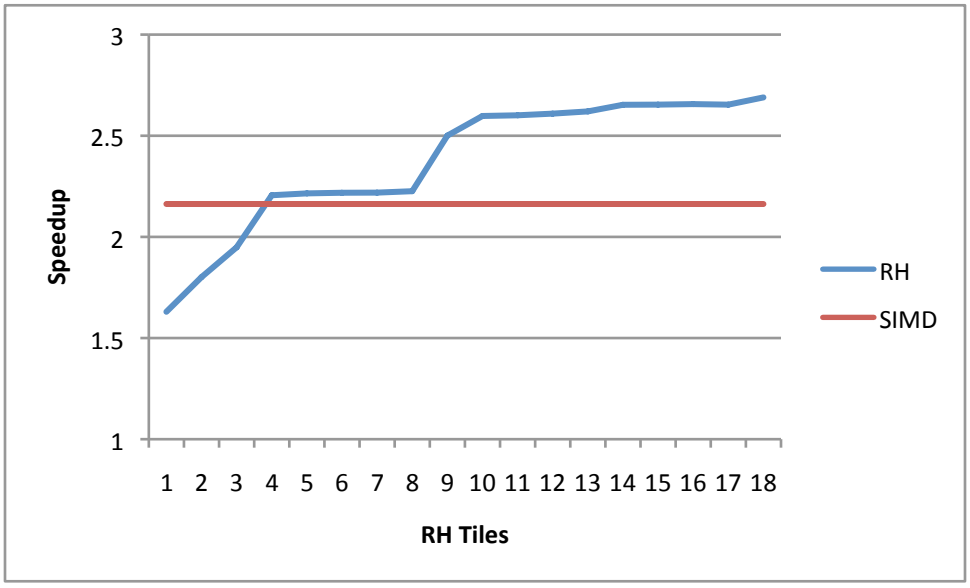


Figure 12.3 Speedup of reconfigurable computing system as I varied the number of RH tiles available

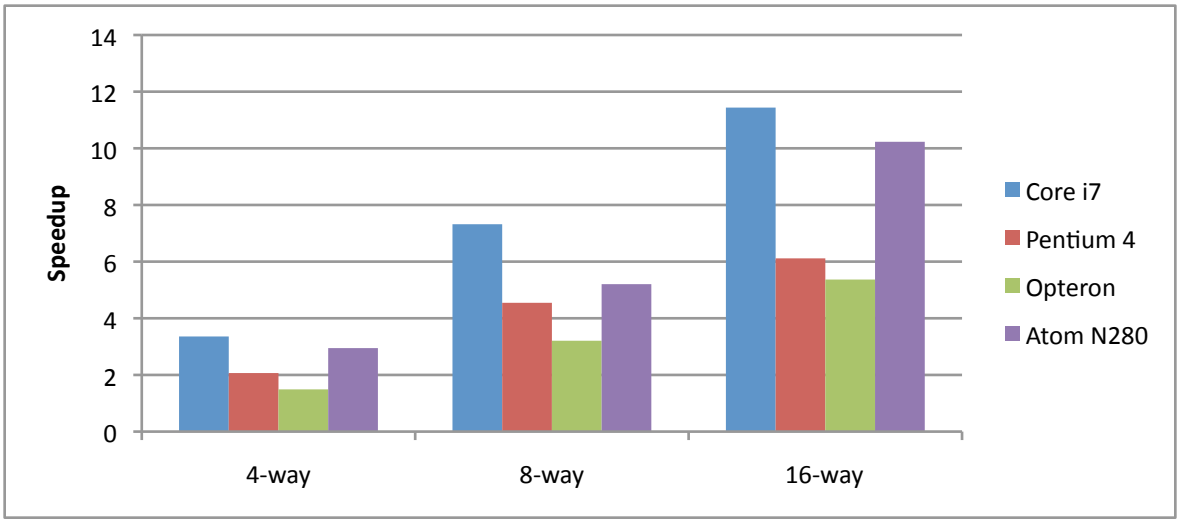


Figure 12.4 Vector speedups when running the Viterbi encoder on PC hardware (RH speedup is 100x)

particular, the RH accelerated the SAD8, SAD16, and the average-4 kernels to a larger degree. This shows that, despite the fact that the x86 SIMD units have instructions designed specifically for multimedia processing, RH coprocessors can still outperform them.

Although this comparison shows the RH's performance advantage when all of the RH kernels can be loaded simultaneously, this scenario is not always realistic. In many reconfigurable computing systems, the RH is a constrained resource, and not every RH kernel can fit simultaneously. Figure 12.3 shows Xvid's performance as the number of RH tiles on the system increases when using the RH kernel scheduler described in Chapter 2.7.2. This graph shows that the RH coprocessor system outperforms the SIMD system whenever there are at least 4 RH tiles available. However, as we convert more kernels to both SIMD and RH implementations, this tradeoff point is likely to change.

12.1.2 Viterbi Performance

I also examined the performance of the Viterbi algorithm on both the modeled SIMD and RH systems. Figure 12.4 shows Viterbi's speedups when using SIMD-optimized algorithms generated by the Spiral project [33]. These results shows that Viterbi's performance scales as I increased the width of the SIMD unitis increased, achieving a speedup of over 10x when using 16-way SIMD instructions. Although these results seem quite impressive, the RH implementation of Viterbi achieved a speedup of over 100x on the proposed reconfigurable computing platform, despite running at a clock frequency of only 67MHz when executing on the RH fabric. In this comparison, it is clear that the Viterbi algorithm can be accelerated much more effectively using a RH coprocessor instead of a SIMD coprocessor. In addition, due to the reduced clock frequency, and greatly reduced execution time, the Viterbi algorithm is likely to consume much less energy on the RH.

12.2 Limitations of SIMD Processors

Although prior work has proven that SIMD ISA extensions usefulness for many classes of applications, they are not useful for all applications. SIMD processors are designed to accelerate coarse-grained parallelism when applications execute the same instruction sequences on many data inputs. They do not tend to perform well on other applications that exploit more fine-grained parallelism, or that require extensive data-dependent loads and/or stores. In contrast, RH has been able to show impressive speedups across a wide range of applications [47].

For instance, many cryptographic algorithms fail to take advantage of the highly parallel nature of SIMD coprocessors. The advanced encryption standard (AES) algorithm, for example, relies on many bit-level manipulations that are not amenable to vector acceleration [69]. Although impressive speedups have been achieved on AES by tweaking the algorithm to the architecture, this has not relied heavily on SIMD instructions, as the AES algorithm requires an extensive number of data-dependent load instructions to read data in from table lookups [12, 50]. Other research has looked into using bitsliced implementations of AES [89]. These have managed to have impressive gains, and appear to scale with SIMD width; but they have some significant drawbacks. In current implementations using 128-bit SIMD registers, performance is $\sim 40\%$ faster than a traditional software algorithm, and appears to scale with bit-width, although, they only compared 64-bit and 128-bit bit-sliced implementations in this study. However, using a bitsliced implementation of AES requires extensive reformatting of data to interleave it for bit-sliced execution, and requires fixed-size data (2048 byte blocks) to achieve this speedup. Because of the inability of standard SIMD instructions to greatly accelerate AES, system designers have instead added custom instructions to accelerate the algorithm [92].

12.3 Conclusions

In this section I examined the performance of various algorithms on both SIMD and RH coprocessors. I demonstrated that, even when SIMD units execute multimedia algorithms, RH coprocessors can obtain better performance. In other algorithms, such as Viterbi, RH's performance was much higher than that of the SIMD unit. I examined other algorithms that were poor candidates for SIMD implementations. Because of this, RH coprocessors are an attractive option for accelerating applications.

Although the development time for SIMD kernels tends to be lower than that of RH kernels, SIMD kernels often require hand-optimized assembly that is also difficult to develop and validate. The development time of current RH kernels tends to be limited by the tool chain available. This tool chain has been primarily designed for developing ASIC-replacements using FPGA technology. Because of this, design-time is not as important as minimizing the size and power consumption of the circuits, and maximizing the circuit's performance. However, through the use of better tools, and higher-level hardware description languages the development-time of RH kernels can be reduced [13, 7]. Recent hardware description languages have been designed to convert small sections of code into RH implementations.

Even though these kernels may not run for as long as routines on traditional off-chip coprocessors, I have demonstrated that my reconfigurable coprocessor works remarkably well with such kernels.

Additionally, using SIMD units and a RH coprocessor are not mutually exclusive ideas. In a processor containing both SIMD units and a RH coprocessor, the RH coprocessor could be primarily used for algorithms that perform poorly on SIMD units (for example, AES), or where the SIMD units cannot accelerate the algorithm sufficiently for the application (Viterbi).

Chapter 13

Broader Impact

Although this thesis has specifically targeted RH coprocessors, many of the techniques developed in this research can be applied to other accelerator technologies. Moving forward, it is likely that no single accelerator technology will be ideal for all application and usage domains. Prior work examined using custom ASIC coprocessors, SIMD vector processors, GPUs, massively parallel processor arrays, and other techniques to accelerate computation. In this chapter I will briefly examine some of these technologies and describes how the research performed for this thesis is applicable to other systems.

13.1 Use with ASIC Coprocessors

Much of the research presented in this thesis can directly be applied to on-chip custom "ASIC" coprocessors by replacing RH kernels with hard kernels implemented on the chip. In particular, the cache and memory organization used in my thesis could directly be applied to ASIC coprocessors, and many of the ideas proposed in my work [43, 45] have since been examined for the use in generic coprocessor architectures [111].

Using a shared cache architecture similar to the one I proposed in Chapter 6 allows for a fast communication path between the processor(s) and any type of coprocessor. Additionally, using the cache-coherent communication model allows the coprocessor to "naturally" access the data it needs, without having to rely on specialized buffers or memory fences when communicating with their host process. This also simplifies the allocation of the hardware, because the OS does not have to specially allocate coprocessor memory into user space, or be used to transfer data into a dedicated buffer associated with the coprocessor The idea of using cache-coherent coprocessors has even been gaining momentum in industry, with ARM's Cortex A9 processor [9] allowing up to four processors to share a coherent memory with a custom hardware accelerator through the use of their "accelerator coherence port".

My work also allows coprocessors to access to the virtual memory subsystem, providing methods to translate virtual to physical addresses within the RH controller (Chapter 3). In most modern operating systems, user applications cannot directly access physical memory, limiting what can be done on a copocessor without OS intervention. Although short-executing accelerators could be implemented that uses only registers for communication, such an accelerator would be core-specific, or require separate register sets for each processor in the system. An alternative model could

use the OS as an intermediary to access the custom hardware accelerator, however such a technique can be significantly slower, and limits the performance of the accelerator [111]. The techniques outlined in my thesis can be applied to on-chip hardware accelerators, allowing applications to directly access a coprocessor without violating process isolation.

Custom ASICs can also directly make use of my work on sharing configured RH kernels [44]. New high-performance embedded devices are likely to contain multiple general-purpose CPU cores, and they will need a way to share the coprocessor resources. These coprocessor resources would likely be shared by every application in the system. Therefore these systems could take advantage of mechanisms, developed for this thesis, to safely share coprocessor resources, arbirtrate access to the coprocessors, and methods to query the coprocessors status.

13.2 Use with Vector Processing Units

Although much of this thesis directly applies to ASIC coprocessors, much of it can also be applied to other types of coprocessor architectures. Vector processors are one potential use for this work. In recent systems, vector, or SIMD units tend to be used as extensions to the processor's ISA. Because of this, they are are not shared by multiple processors and so they do not need special provisions to access memory. Although SIMD coprocessors can be shared in SMT systems, this sharing is no different than sharing any other function unit within a processor [36, 118].

However, there are plenty of reasons to want to share SIMD units amongst multiple processors. AMD's Bulldozer architecture will do this, however sharing is limited to two processors due to latency constraints. A more relevant example of sharing SIMD units can be seen in IBM's Cell architecture [71]. In the cell microarchitecture, multiple "synergistic processing elements" (SPEs) are shared amongst a small number of CPUs. Each SPE contains 128-bit SIMD unit, and has access to its own private scratchpad memory, and a DMA engine. This system shares multiple SPEs amongst general-purpose processor cores; however, the usage of DMA engines and scratchpad memories complicate process isolation, forcing application designers to directly manage the communication between the processing elements.

It is conceivable that a future SIMD architecture could build upon the work of this thesis, coupling one or more vector processor units with their own private L1 cache. This would allow coprocessor codes to execute in the same virtual address space as the original code, without relying on the OS to transfer control to the coprocessor. Applications executing across the host processor, and the SIMD units would not have to explicitly transfer data between the units. In systems containing a large on-chip L2 or L3 cache, indirectly accessing the memory hierarchy through an L1 cache can allow the vector units to access data without having to access (high latency/energy) main memory.

Future vector coprocessors might also involve design tradeoffs making them less functional compared to those on current systems. In a future system, a small instruction scratchpad memory could be attached to a vector unit; providing an instruction stream that allows the coprocessor to operate on data contained in a scratchpad memory or large register file. An attached stream controller, similar to the one used in this work could load data to/from a small scratchpad memory or register file attached to the coprocessor. Extensions to the coprocessor would be necessary,

as it is likely that the processor would want to have more direct control over data access patterns. However, using a stream controller like the one proposed is advantageous to DMA engines as they provide a more fine-grained ability to access memory, reducing the need for the CPU to populate buffers or reprogram a DMA engine to obtain new data. Additionally, the data alignment unit could improve performance to unaligned regions of memory.

With on-chip shared SIMD units, there is no reason that each kernel would have to execute for tens of thousands of cycles at a time to be useful. Using a model similar to the one in this thesis, a SIMD unit could be programmed to accelerate very short kernels in an application's execution, possibly taking the place of only hundreds of processor instructions. This is similar to the number of "equivalent instructions" executed in many of the Xvid benchmark's kernels.

It might also be useful to have multiple of these SIMD units shared amongst all of the processor in the system. This way multiple CPUs can use the SIMD units simultaneously. In this model it might prove useful to combine multiple units with an L1 cache, and a SIMD controller that handles the memory requests from the various stream controllers. Using an attached L1 cache (rather than direct access to the rest of the memory hierarchy) is likely to be beneficial to the energy consumption of these systems, reducing the number of memory accesses to more power-hungry levels of the cache hierarchy. Mechanisms similar to those proposed in Chapter 4 could be used to arbitrate access to the SIMD coprocessors, and these units could be shared using the techniques examined in Chapter 9.

13.3 Use with GPUs

Most GPUs are structured similarly to very wide vector processors. Currently, they tend to be integrated on separate chips due to area and power concerns; however on mobile devices, systems-on-chips readily integrate GPUs with general-purpose processors (amongst other devices) on a single die [97]. Additionally, in the "desktop" computing market GPUs are now commonly integrated on the same die as the CPUs[18, 67].

Although GPUs have mostly been used to accelerate graphics computations, in recent years there has been increased interest in using GPUs to accelerate general-purpose tasks. Many of the techniques developed in this thesis can also be used to better interface CPUs with on-chip general-purpose GPUs (GPGPUs).

GPUs currently use a separate address space from applications, and often use a separate physical memory to hold their own data. Therefore memory transfers between the GPU and the CPU require OS intervention. However, unlike the RH coprocessors examined in this work, it is unlikely that high-performance GPUs will want to exclusively use the CPU's memory hierarchy because graphics operations have been fine tuned to make extensive use of a streaming computation model optimized around fast dedicated memories connected to the GPU. It is, however, possible that future GPUs could directly access both their own local memories and the CPU's memory. Under such a model, the GPU could access the CPU's memory space to directly obtain data for graphics processing, and could use this same memory for accessing data needed to accelerate GPGPU applications.

Current GPGPU systems do not have a strong need for virtually-addressable memory, because only a single application can execute at a time. Because of this, the OS uses a form of cooperative multitasking that allows multiple threads to share the device. However, current work has demonstrated that GPGPU resources can be better allocated when multiple applications can execute on the GPU at the same time [1, 2]. If multiple tasks are allowed to execute on the GPU at the same time, memory protection would be a major concern.

Current GPGPU systems are based on an off-chip coprocessor model [20], and the programming model assumes that this is the case. Because of this, current GPGPU kernels must execute for a relatively long time to amortize the setup costs. These costs typically involve allocating memory on the GPU, and using the OS to transfer data to the GPU's memory both before and after the kernel has finished its execution. In systems where the GPU is integrated on the same die as the CPU, alternative approaches to communicating between the CPU and GPU might be beneficial [111]. Such a system could take advantage of a cache coherent virtually addressable memory for the same reason a SIMD coprocessor might. This would simplify data transfer between applications and GPGPU accelerators, and even allow shorter-running kernels to use the GPU, greatly reducing the overheads associated with OS initiated GPGPU accelerators. Additionally, techniques developed in this thesis could allow for applications to access GPGPU kernels without relying on system calls to the OS.

13.4 Use with Massively Parallel Processor Arrays

The coprocessor architecture proposed in this thesis could also be used with arrays of simple processors. Such devices have been implemented in real products such as the Ambric processor that contains an array of 336 32-bit RISC processors [19]. These devices are currently used as an alternative to FPGAs. In the Ambric chip, a configurable interconnect allows cores to communicate with each other. This interconnect can be reconfigured, allowing the implementation of different kernels on the chip.

Although these chips have primarily been used as off chip accelerators, such massively parallel processor arrays could easily be used within the system model presented in this thesis. Instead of having each tile consist of an FPGA-like fabric, each tile could be an array of simplistic processors. A high level interconnect could then allow processors in different "tiles" to communicate with each other. The OS would then allocate resources to the accelerators based on their requirements. In this model, few direct changes would be needed to my architecture, and most of the work presented in this thesis would directly apply to such a system.

13.5 Usefulness of scheduling

Although much of my work can be directly applied to other types of accelerator architectures, my scheduling work cannot be so easily applied. One of the primary motivators behind scheduling what is configured on an RH fabric is that the configuring the RH fabric is an expensive operation – much too expensive to continuously change what is

configured. Because of this, prior work has examined what applications should have access to the RH fabric at a given time. In these architectures, RH kernels often need to be configured for relatively long (order of ms) time periods to amortize the reconfiguration cost

In other accelerator architectures, the coprocessor may not require reconfiguration at all (e.g., if using ASIC co-processors), or the reconfiguration latency may be quite small (e.g., a context switch to load new instructions in a software-based accelerator. In these situations, it might make sense for applications to use a more fine-grained sharing of the resources, possibly allocating them on demand.

However for many workloads, scheduling of such resources can still result in superior performance. For instance, if one application could obtain much higher performance by using a SIMD accelerator, it might be better to exclusively reserve the accelerator for this applications, preventing other applications from using the resource, even if it is currently idle.

To really gauge the usefulness of such scheduling, it is likely that much future work needs to be done. It is my belief that a scheduler for coprocessor devices needs to be more dynamic than those developed for RH systems due to the relatively long reconfiguration latency. However the scheduler should still consider the impact of sharing single accelerators, or possibly pools of accelerators between multiple applications. In this scenario, the problem could even start to resemble the scheduling decisions that might be necessary for scheduling a RH fabric when multiple copies of the same, or similar applications are executing that could use the same coprocessor resources.

Although the exact scheduling methodologies used in my research might not apply to all of the other coprocessor technologies, I believe they will help provide a framework from which newer, more advanced schedulers could improve upon when scheduling coprocessors that can rapidly be “reconfigured”.

Chapter 14

Conclusion

Consumers are demanding faster, smaller, more power-efficient embedded devices that are increasingly multi-purpose, with functionality that is not always known at design time. To help meet these needs, computer architects have been looking toward heterogeneous systems that integrate more functionality on a single chip. In this thesis work, a new reconfigurable computing architecture has been developed that provides programmers with the ability to dynamically accelerate applications through the use of the on-chip reconfigurable fabric.

Although many different reconfigurable computing architectures have been researched in the past, most of these systems had focused on single-core systems, many of which included the RH as an off-chip coprocessor. This thesis builds upon these previous systems to create a novel multicore reconfigurable computer architecture. This new system has been designed from the ground up to take advantage of the low latency and high bandwidth available on a single chip. To support the low-latency communication available on a single-chip platform, a new programming model was introduced that allow for applications to safely and efficiently access the coprocessor. In addition, multiple optimizations were made to the programming model to allow for a more efficient usage of the shared reconfigurable hardware fabric.

In doing this research, I have made great strides toward creating a shared reconfigurable coprocessor optimized for multicore systems. The following are the systems implemented to perform this research:

- I developed a coprocessor platform that allowed low-latency (less than 20 cycles at 1GHz) communication between CPU cores and a shared RH fabric.
- I created a programming interface that allows applications to safely access RH kernels assigned to it without invoking the OS.
- I created mechanisms to share configured RH kernels amongst multiple applications.
- I created a new, optimized RH kernel scheduler to better select which RH kernels should be configured on multicore systems.

By creating these new systems, and extensions to RH computing, I showed the following:

- Sharing at least one level of cache between the processor and RH coprocessor reduces communication latency between the two.
- RH coprocessors should have at least a small L1 cache, as it can greatly reduce the dynamic energy consumption of the memory hierarchy.
- A hardware TLB miss handler can greatly increase the performance of multicore RH computing systems.
- When eight cores execute hybrid RH/SW applications, my platform performs $\sim 95\%$ as well as a system where the RH kernels access a zero latency memory.
- Preventing data-intensive, streaming RH kernels from polluting shared caches can improve the performance of coscheduled SW-only applications by up to 32%.
- When SMT processors implement RH-kernel-aware thread selection logic, threads coscheduled alongside an application repeatedly executing long-running RH kernels obtain $\sim 95\%$ of their performance when executing on a dual core system alongside the same RH application.
- By sharing configured RH kernels amongst multiple applications, a constrained RH fabric can be used much more efficiently. On an eight-core system that shared RH kernels, applications performed 97.4% as fast as a system containing almost eight times more RH (where each application had its own copy of each kernel).
- A RH coprocessor using my interface can perform as good, or better than dedicated SIMD units; even when executing multimedia algorithms that SIMD functional units have been optimized for.

Through these contributions, I have shown that combining a reconfigurable coprocessor with multiple processor cores on a shingle chip provides a flexible and powerful processing platform that can help meet the needs of tomorrow's computing tasks.

LIST OF REFERENCES

[1] Jacob Adriaens, Paula Aguilera, Katherine Compton, and Nam Sung Kim. The case for gpgpu spatial multitasking. In *TECHON*, 2011.

[2] Jacob Adriaens, Paula Aguilera, Katherine Compton, and Nam Sung Kim. The case for gpgpu spatial multitasking. In *Personal Correspondence*, 2011.

[3] E. Ahmed and J. Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 12(3):288 –298, march 2004.

[4] Altera. Increasing design functionality with partial and dynamic reconfiguration in 28-nm FPGAs. *http://www.altera.com/literature/wp/wp-01137-stxv-dynamic-partial-reconfig.pdf*, 2010.

[5] Altera. Stratix V FPGAs: Built for bandwidth. 2010.

[6] Altera Corporation. Altera excalibur devices. *http://www.altera.com/products/devices/arm/overview/arm-overview.html*, 2003.

[7] David L. Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. The case for high level programming models for reconfigurable computers. In *ERSA*, pages 21–32, 2006.

[8] R.W. Apperson, Zhiyi Yu, M.J. Meeuwsen, T. Mohsenin, and B.M. Baas. A scalable dual-clock fifo for data transfers between arbitrary and haltable clock domains. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15(10):1125 –1134, Oct 2007.

[9] ARM. The ARM Cortex-A9 processors. *http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf*, 2009.

[10] Jeffrey M. Arnold. S5: The architecture and development flow of a software configurable processor. In *IEEE International Conference on Field Programmable Technology*, pages 121–128, Dec 2005.

[11] Michael J. Beauchamp, Scott Hauck, Keith D. Underwood, and K. Scott Hemmert. Embedded floating-point units in fpgas. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 12–20, New York, NY, USA, 2006. ACM.

[12] Daniel Bernstein and Peter Schwabe. New aes software speed records. In Dipanwita Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer Berlin / Heidelberg, 2008.

[13] Shuvra S. Bhattacharyya, Johan Eker, Carl Von Platen, Marco Mattavelli, Gordon Brebner, Jrn W. Janneck, and Mickal Raulet. Opendf a dataflow toolset for reconfigurable hardware and multicore systems. 2008.

[14] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38, June 2006.

[15] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual memory mapped network interface for the shrimp multicomputer. In *International Symposium on Computer architecture*, pages 473–484, New York, NY, USA, 1998. ACM.

[16] Daniel Pierre Bovet and Marco Cassetti. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[17] G. Brebner. The swappable logic unit: a paradigm for virtual hardware. In *IEEE Symposium on FPGA-Based Custom Computing Machines*, page 77, Washington, DC, USA, 1997. IEEE Computer Society.

[18] Nathan Brookwood. AMD fusion family of apus: Enabling a superior, immersive pc experience. March 2010.

[19] M. Butts. Synchronization through communication in a massively parallel processor array. *IEEE Micro*, 27(5):32–40, Sept.-Oct. 2007.

[20] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM J. Res. Dev.*, 54:473–482, September 2010.

[21] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream computations organized for reconfigurable execution (score). In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 605–614, London, UK, 2000. Springer-Verlag.

[22] Cavium Networks. OCTEON plus CN58XX 4 to 16-core MIPS64-based SoCs. *http://www.caviumnetworks.com/pdfFiles/CN58XX_PB%20Rev%201.5.pdf*, 2008.

[23] Herwin Chan, Patrick Schaumont, and Ingrid Verbauwhede. Process isolation for reconfigurable hardware. In *Engineering of Reconfigurable Systems and Algorithms*, pages 164–170, 2006.

[24] Daniel W. Chang, Christipher D. Jenkins, Philip C. Garcia, Syed Z. Gilani, Paula Aguilera, Aishwarya Nagarajan, Michael J. Anderson, Matthew A. Kenny, Sean M. Bauer, Michael J. Schulte, and Katherine Compton. Ercbench: An open-source benchmark suite for embedded and reconfigurable computing. *International Conference on Field Programmable Logic and Applications*, 0:408–413, 2010.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[26] Zhimin Chen, Neil Pittman, and Alessandro Forin. Combining multicore and reconfigurable instruction set extensions. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '10, pages 33–36, New York, NY, USA, 2010. ACM.

[27] Daniel Citron, Adham Hurani, and Alaa Gnadrey. The harmonic or geometric mean: does it really matter? *SIGARCH Comput. Archit. News*, 34:18–25, September 2006.

[28] Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *International Symposium on Computer Architecture*, pages 272–283, Washington, DC, USA, 2005. IEEE Computer Society.

[29] K. Compton, Zhiyuan Li, J. Cooley, S. Knol, and S. Hauck. Configuration relocation and defragmentation for run-time reconfigurable computing. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 10(3):209 –220, june 2002.

[30] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Survey*, 34(2):171–210, 2002.

[31] NVIDIA Corp. Nvidia CUDA: Compute unified device architecture. *http://developer.download.nvidia.com/compute/cuda/2_0/docs/CudaReferenceManual_2.0.pdf*, 2007.

[32] M. Dales. Managing a reconfigurable processor in a general purpose workstation environment. In *IEEE Conference on Design, Automation and Test in Europe*. IEEE Computer Society Washington, DC, USA, 2003.

[33] Frédéric de Mesmay, Srinivas Chellappa, Franz Franchetti, and Markus Püschel. Computer generation of efficient software Viterbi decoders. In *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, volume 5952 of *Lecture Notes in Computer Science*, pages 353–368. Springer, 2010.

[34] Cezary Dubnicki, Angelos Bilas, and Kai Li. Design and implementation of virtual memory-mapped communication on myrinet. In *IPPS '97: Proceedings of the 11th International Symposium on Parallel Processing*, page 388, Washington, DC, USA, 1997. IEEE Computer Society.

[35] Carl Ebeling, Darren C. Cronquist, and Paul Franklin. Rapid - reconfigurable pipelined datapath. In *Proceedings of the 6th International Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pages 126–135, London, UK, 1996. Springer-Verlag.

[36] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–??, /1997.

[37] E. El-Araby, P. Nosum, and T. El-Ghazawi. Productivity of high-level languages on reconfigurable computers: An hpc perspective. In *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 257 –260, dec. 2007.

[38] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *International Symposium on Computer architecture*, New York, NY, USA, 2011. ACM.

[39] Wenyin Fu and K. Compton. A simulation platform for reconfigurable computing research. In *IEEE conference on Field Programmable Logic and Applications*, pages 1–7, Aug. 2006.

[40] Wenyin Fu and Katherine Compton. An execution environment for reconfigurable computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 149–158, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Wenyin Fu and Katherine Compton. Active kernel monitoring to combat scheduler gaming in reconfigurable computing systems. In *Field Programmable Logic and Applications*, pages 611 –614, sept. 2008.

[42] Wenyin Fu and Katherine Compton. Scheduling intervals for reconfigurable computing. In *Field Custom Computing Machines*, page 87, 2008.

[43] Philip Garcia and Katherine Compton. A reconfigurable hardware interface for a modern computing system. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 73–84, 2007.

[44] Philip Garcia and Katherine Compton. Kernel sharing on reconfigurable multiprocessor systems. In *IEEE Conference on Field Programmable Technology*, pages 225–232, Dec. 2008.

[45] Philip Garcia and Katherine Compton. Shared memory cache organizations for reconfigurable computing systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, volume 0, pages 239–242, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[46] Philip Garcia and Katherine Compton. A scalable memory interface for multicore reconfigurable computing systems. In *to appear in IEEE Conference on Field Programmable Technology*, Dec. 2011.

[47] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, and Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *Eurasip Special Issue on Reconfigurable Embedded Computing*, 2006.

[48] Philip Garcia, Kyle Rupnow, and Katherine Compton. A reconfigurable computing scheduler optimized for multicore systems. *International Conference on Field Programmable Logic and Applications*, 0:107–112, 2010.

[49] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11 – 13, may 2005.

[50] Brian Gladman. Aes and combined encryption/authentication modes. http://fp.gladman.plus.com/aes/index.htm.

[51] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. Piperench: a co/processor for streaming multimedia acceleration. In *International Symposium on Computer architecture*, pages 28–39, Washington, DC, USA, 1999. IEEE Computer Society.

[52] D. Grant, P.B. Denyer, and I. Finlay. Synthesis of address generators. In *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pages 116 –119, nov 1989.

[53] Douglas M. Grant and Peter B. Denyer. Address generation for array access based on modulus m counters. In *Proceedings of the conference on European design automation*, EURO-DAC '91, pages 118–122, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[54] Steve Guccione, Delon Levi, and Prasanna Sundararajan. Jbits: Java based interface for reconfigurable computing. 1999.

[55] Marcelo Gtz, Florian Dittmann, and Tao Xie. Dynamic relocation of hybrid tasks: Strategies and methodologies. *Microprocessors and Microsystems*, 33(1):81 – 90, 2009. Selected Papers from ReCoSoC 2007 (Reconfigurable Communication-centric Systems-on-Chip).

[56] R. Hartenstein. Coarse grain reconfigurable architectures. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 564 –569, 2001.

[57] J. R. Hauser and J. Wawrzynek. Garp: a mips processor with a reconfigurable coprocessor. In *IEEE Symposium on FPGA-Based Custom Computing Machines*, page 12, Washington, DC, USA, 1997. IEEE Computer Society.

[58] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the stanford flash multiprocessor. *SIGPLAN Not.*, 29:38–50, November 1994.

[59] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[60] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34:1–17, September 2006.

[61] M. Herz, R. Hartenstein, M. Miranda, E. Brockmeyer, and F. Catthoor. Memory addressing organization for stream-based reconfigurable computing. *Electronics, Circuits and Systems*, 2:813–817, 2002.

[62] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, july 2008.

[63] Lisa Hsu, Ravi Iyer, Srihari Makineni, Steve Reinhardt, and Donald Newell. Exploring the cache design space for large scale cmps. *SIGARCH Comput. Archit. News*, 33(4):24–33, 2005.

[64] Jerry Huck and Jim Hays. Architectural support for translation table management in large address space machines. *SIGARCH Comput. Archit. News*, 21:39–50, May 1993.

[65] James Hughes, Gary Morton, Jan Pechanec, Christoph Schuba, Lawrence Spracklen, and Bhargava Yenduri. Transparent multi-core cryptographic support on niagara CMT processors. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 81–88, Washington, DC, USA, 2009. IEEE Computer Society.

[66] Intel. Enabling consistent platform-level services for tightly coupled accelerators. *http://download.intel.com/-technology/platforms/quickassist/quickassist_aal_whitepaper.pdf*, 2008.

[67] Intel. Intel microarchitecture codename sandy bridge. 2010.

[68] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, tlb-refill mechanisms, and page table organizations. *SIGPLAN Not.*, 33:295–306, October 1998.

[69] C. Jenkins, S. Mamidi, M. Schulte, and J. Glossner. Instruction set extensions for aes processing on a multithreaded software defined radio platform. In *Signals, Systems and Computers, 2007. ACSSC 2007. Conference Record of the Forty-First Asilomar Conference on*, pages 963 –966, nov. 2007.

[70] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. Crl: high-performance all-software distributed shared memory. *SIGOPS Oper. Syst. Rev.*, 29:213–226, December 1995.

[71] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 –604, july 2005.

[72] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[73] John H. Kelm and Steven S. Lumetta. HybridOS: runtime support for reconfigurable accelerators. In *ACM/SIGDA symposium on Field programmable gate arrays*, pages 212–221, New York, NY, USA, 2008. ACM.

[74] M.N. Khan and S. Ghauri. The wimax 802.16e physical layer model. In *Wireless, Mobile and Multimedia Networks, 2008. IET International Conference on*, pages 117 –120, jan. 2008.

[75] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C.-L. Wang. Heterogeneous computing: challenges and opportunities. *Computer*, 26(6):18 –27, jun 1993.

[76] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, 23(2):56 – 65, march-april 2003.

[77] Richard B. Kujoth, Chi wei Wang, Derek B. Gottlieb, Jeffrey J. Cook, and Nicholas P. Carter. A reconfigurable unit for a clustered programmable-reconfigurable processor. In *in Proceedings of the 2004 ACM SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 200–209, 2004.

[78] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. *SIGARCH Comput. Archit. News*, 32:64–, March 2004.

[79] Georgi Kuzmanov, Georgi Gaydadjiev, and Stamatis Vassiliadis. The molen processor prototype. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 296–299, Washington, DC, USA, 2004. IEEE Computer Society.

[80] V. Lappalainen, T.D. Hamalainen, and P. Liuha. Overview of research efforts on media isa extensions and their usage in video coding. *Circuits and Systems for Video Technology, IEEE Transactions on*, 12(8):660 – 670, aug 2002.

[81] N. Lawal, B. Thornberg, and M. O'Nils. Address generation for fpga rams for efficient implementation of real-time video processing systems. *International Conference on Field Programmable Logic and Applications*, 0:136–141, 2005.

[82] Guangming Lu, Hartej Singh, Ming-Hau Lee, Nader Bagherzadeh, Fadi J. Kurdahi, and Eliseu M. Chaves Filho. The morphosys parallel reconfigurable system. In *International Euro-Par Conference on Parallel Processing*, pages 727–734, London, UK, 1999. Springer-Verlag.

[83] E. Lubbers and M. Platzner. A portable abstraction layer for hardware threads. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 17 –22, sept. 2008.

[84] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[85] S. Mamidi, M.J. Schulte, D. Iancu, and J. Glossner. Architecture support for reconfigurable multithreaded processors in programmable communication systems. In *Application -specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on*, pages 320 –327, july 2007.

[86] Suman Mamidi, Emily R. Blem, Michael J. Schulte, John Glossner, Daniel Iancu, Andrei Iancu, Mayan Moudgill, and Sanjay Jinturkar. Instruction set extensions for software defined radio on a multithreaded processor. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 266–273, New York, NY, USA, 2005. ACM Press.

[87] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[88] John R. Mashey. War of the benchmark means: time for a truce. *SIGARCH Comput. Archit. News*, 32:1–14, September 2004.

[89] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 121–134. Springer Berlin / Heidelberg, 2007.

[90] Micron Technology, Inc. DDR2_power_calc_16.xls. *http://download.micron.com/downloads/misc/-ddr2_power_calc_web.xls*, December 2006.

[91] David P. Montminy, Rusty O. Baldwin, Paul D. Williams, and Barry E. Mullins. Using relocatable bitstreams for fault tolerance. *Adaptive Hardware and Systems, NASA/ESA Conference on*, 0:701–708, 2007.

[92] K. Nadehara, M. Ilekawa, and L. Kuroda. Extended instructions for the aes cryptography and their efficient implementation. In *IEEE workshop on Signal Processing Systems*, 2004.

[93] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Designing an operating system for a heterogeneous reconfigurable SoC. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 174.1, Washington, DC, USA, 2003. IEEE Computer Society.

[94] Grace Nordin, Peter A. Milder, James C. Hoe, and Markus Paschel. Automatic generation of customized discrete fourier transform ips. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 471–474, New York, NY, USA, 2005. ACM Press.

[95] Joshua Noseworthy. Enabling communication between an fpga's embedded processor and its reconfigurable resources. In *M.S. Thesis, Northeastern University*, August 2005.

[96] Joshua Noseworthy and Miriam Leeser. Efficient use of communications between an fpga's embedded processor and its reconfigurable logic. In *International symposium on Field programmable gate arrays*, pages 233–233, New York, NY, USA, 2006. ACM.

[97] Nvidia. The benefits of multiple cpu cores in mobile devices. *http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf*, 2010.

[98] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGPLAN Not.*, 31:2–11, September 1996.

[99] Iyad Ouaiss, Sriram Govindarajan, Vinoo Srinivasan, Meenakshi Kaul, and Ranga Vemuri. An integrated partitioning and synthesis system for dynamically reconfigurable multi-fpga architectures. In Jos Rolim, editor, *Parallel and Distributed Processing*, volume 1388 of *Lecture Notes in Computer Science*, pages 31–36. Springer Berlin / Heidelberg, 1998.

[100] J. D. Owens, D. Luebke., N. Govindaraju., M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell. A survey of general-purpose computation on graphics hardware. In *State of the Art Report, Eurographics*, pages 21–51, 2005.

[101] U. Ramacher. Software-defined radio prospects for multistandard mobile phones. *Computer*, 40(10):62 –69, oct. 2007.

[102] P. Ranganathan, S. Adve, and N.P. Jouppi. Performance of image and video processing with general-purpose processors and media isa extensions. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 124 –135, 1999.

[103] J. Resano, D. Mozos, D. Verkest, and F. Catthoor. A reconfigurable manager for dynamically reconfigurable hardware. *Design Test of Computers, IEEE*, 22(5):452 – 460, sept.-oct. 2005.

[104] Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Rielly, third edition, 2005.

[105] Kyle Rupnow. Resource mangement for reconfigurable computing systems. *PhD Thesis, University of Wisconsin-Madison*, 2010.

[106] Kyle Rupnow, Jacob Adriaens, Wenyin Fu, and Katherine Compton. Accurately evaluating application performance in simulated hybrid multi-tasking systems. In *to appear in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2010.

[107] Kyle Rupnow, Wenyin Fu, and Katherine Compton. Block, drop or roll(back): Alternative preemption methods for rh multi-tasking. *IEEE symposium on Field-Programmable Custom Computing Machines*, 0:63–70, 2009.

[108] David Sheldon, Rakesh Kumar, Roman Lysecky, Frank Vahid, and Dean Tullsen. Application-specific customization of parameterized fpga soft-core processors. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 261–268, New York, NY, USA, 2006. ACM.

[109] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[110] Wilson Snyder, Paul Wasson, and Duane Galbi. Verilator. In `http://www.veripool.com/verilator.html`, 2007.

[111] P.M. Stillwell, V. Chadha, O. Tickoo, S. Zhang, R. Illikkal, R. Iyer, and D. Newell. Hippai: High performance portable accelerator interface for socs. In *High Performance Computing (HiPC), 2009 International Conference on*, pages 109 –118, dec. 2009.

[112] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12:66–73, May 2010.

[113] Irfan Syed, John A. Williams, and Neil W. Bergmann. A hybrid reconfigurable cluster-on-chip architecture with message passing interface for image processing applications. In *FPL*, pages 609–612, 2007.

[114] S. Thoziyoor, N. Muralimanohar, J.H. Ahn, and N.P. Jouppi. CACTI 5.1. *HP Laboratories, Palo Alto, Tech. Rep*, 20, 2008.

[115] T.J. Todman, G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung. Reconfigurable computing: architectures and design methods. *Computers and Digital Techniques, IEE Proceedings* -, 152(2):193 – 207, mar 2005.

[116] Nathan Tuck and Dean M. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[117] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 318–327, Washington, DC, USA, 2001. IEEE Computer Society.

[118] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[119] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 4 pp., april 2006.

[120] Miljan Vuletić, Laura Pozzi, and Paolo Ienne. Programming transparency and portable hardware interfacing: Towards general-purpose reconfigurable computing. In *ASAP '04: Proceedings of the Application-Specific Systems, Architectures and Processors, 15th IEEE International Conference on (ASAP'04)*, pages 339–351, Washington, DC, USA, 2004. IEEE Computer Society.

[121] Miljan Vuletic, Laura Pozzi, and Paolo Ienne. Seamless hardware-software integration in reconfigurable computing systems. *IEEE Design and Test of Computers*, 22(2):102–113, 2005.

[122] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 290 – 295, 2003.

[123] H. Walder, C. Steiger, and M. Platzner. Fast online task placement on fpgas: free space partitioning and 2d-hashing. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., april 2003.

[124] Matthew A. Watkins and David H. Albonesi. Remap: A reconfigurable heterogeneous multicore architecture. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 497–508, Washington, DC, USA, 2010. IEEE Computer Society.

[125] Grant Wigley and David Kearney. The first real operating system for reconfigurable computers. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 130–137, Washington, DC, USA, 2001. IEEE Computer Society.

[126] Grant Wigley and David Kearney. The management of applications for reconfigurable computing using an operating system. In *CRPIT '02: Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, pages 73–81, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[127] Grant B. Wigley and David A Kearney. Research issues in operating systems for reconfigurable computing. In *Engineering of Reconfigurable Systems and Algorithms*, June 2002.

[128] J.A. Williams, I. Syed, J. Wu, and N.W. Bergmann. A reconfigurable cluster-on-chip architecture with mpi communication layer. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:351–352, 2006.

[129] Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. *SIGARCH Comput. Archit. News*, 24:147–157, May 1996.

[130] R.D. Wittig and P. Chow. OneChip: An FPGA processor with reconfigurable logic. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[131] Xilinix Inc. Virtex-5 family overview. *http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf*, 2009.

[132] Like Yan, Binbin Wu, Yuan Wen, Shaobin Zhang, and Tianzhou Chen. A reconfigurable processor architecture combining multi-core and reconfigurable processing unit. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 2897–2902, Washington, DC, USA, 2010. IEEE Computer Society.

[133] Zhi Alex Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *International symposium on Computer architecture*, pages 225–235, 2000.

[134] Pavel G. Zaykov, Georgi K. Kuzmanov, and Georgi N. Gaydadjiev. Reconfigurable multithreading architectures: A survey. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 263–274, Berlin, Heidelberg, 2009. Springer-Verlag.