
# Reconfigurable Architectures for Chip Multiprocessors

by Matthew Ace Watkins

---

# RECONFIGURABLE ARCHITECTURES FOR CHIP MULTIPROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Matthew Ace Watkins

August 2010

RECONFIGURABLE ARCHITECTURES FOR CHIP MULTIPROCESSORS

Matthew Ace Watkins, Ph.D.

Cornell University 2010

Prior research in chip-level reconfigurable computing has involved augmenting a single processor core with reconfigurable logic. Despite significant performance gains for some applications, the area and power costs can easily outweigh the benefits, especially when considering the breadth of applications run on a general purpose processor and the benefit they receive from reconfigurable logic, from orders of magnitude benefit to no benefit at all. Moreover, this prior work focused almost exclusively on uniprocessor systems and did not address the unique requirements of parallel applications.

This dissertation proposes novel reconfigurable architectures for chip multiprocessors (CMPs). In our approach, the reconfigurable fabric is shared among multiple threads from both sequential and parallel applications to amortize the area and power costs and increase fabric utilization. To further reduce the overhead, we propose a heterogeneous CMP where different regions are optimized for different tasks, including regions with shared reconfigurable fabrics, and other regions with only conventional cores. Within a reconfigurable region, the architecture dynamically manages the use of the shared fabric and includes mechanisms that accelerate parallel applications and enable parallelization of otherwise sequential applications.

We first identify a number of features from previous proposals that enable efficient sharing of reconfigurable logic. With these features in mind we design Specialized Programmable Logic (SPL), a reconfigurable fabric specially tailored

for sharing among multiple cores, and evaluate and optimize the SPL under a range of both single- and multi-threaded applications.

As with other shared structures, shared SPL must be intelligently controlled in order to achieve optimal performance. We propose a number of sharing schemes and find that, with proper management, shared SPL achieves performance similar to providing each core with its own large, private fabric, while substantially reducing area and peak power costs.

When multiple single- and multi-threaded applications are running on multiple SPL clusters, the assignment of threads to clusters and the dynamic partitioning of the fabric significantly impact performance. To address these issues, we propose a number of management algorithms that control both thread scheduling and SPL sharing.

Finally, the shared nature of the SPL makes it well suited for communicating among the attached cores. We propose modifications to the baseline SPL design that allow it to provide a means of fine-grained interthread and barrier communication among cores sharing the fabric. Performing communication through the SPL provides the additional benefit of allowing computation to be performed on the data while it is in-flight to the recipient. When incorporated into a heterogeneous CMP, the combined computation and communication abilities of the SPL provide significant benefits over a CMP with only traditional cores.

## BIOGRAPHICAL SKETCH

Matthew Watkins was born in Youngstown, Ohio and spent most of his childhood growing up in Saratoga Springs, New York. He graduated as Valedictorian from Saratoga Springs High School in 2001. He attended the University at Buffalo as a Distinguished Honors Scholar and graduated in 2005 with dual degrees in Computer Engineering and Electrical Engineering. He was active in the university's jazz ensemble and served as president of the Tau Beta Pi and Golden Key chapters on campus.

Matt has been interested in computers from an early age. As a child he enjoyed playing on his family's Commodore and early IBM computers. This interest in computers, coupled with his strength in math and science, led him to pursue a degree in computer engineering. Of all of the courses he took as an undergraduate, his favorites were digital logic, computer organization, and computer architecture. Although Buffalo had no active research in computer architecture, Matt decided to pursue this area for his graduate studies.

Out of a number of prestigious universities that accepted him, Matt chose to attend Cornell for his graduate work due to the professors' sincere interest in working with him and their active efforts to recruit him. During his first year and a half he worked on leveraging on-chip optical interconnects to enhance power and performance of large scale CMPs. For his thesis he concentrated on reconfigurable computing. In particular, focusing on how the move to chip multiprocessors allows more efficient integration of reconfigurable logic into a general purpose processor.

In his limited free time Matt enjoys playing the trumpet and piano and swing and ballroom dancing. Matt has been involved with music for most of his life, but prior to arriving at Cornell, had almost no dance experience. Despite this,

when the dance bug bit him, it bit him hard. He joined the ballroom team at the beginning of his second semester, unsure if he wanted the time commitment it involved. By the end of the semester, the nearly daily ballroom practices were the highlight of his day (and still are to this day). He found he had an affinity for the standard/smooth style of dances and by the time of his graduation he was dancing those styles at one of the highest levels in collegiate ballroom dancing.

After graduating, Matt hopes to find a position in academia. He is especially interested in the teaching side of university life where he looks forward to passing on his knowledge and interest in computers to future students. He also looks forward to mentoring students both in their research and as they decide on their future career path.

*To my parents,*

*for their never-ending love and support.*

## ACKNOWLEDGEMENTS

First and foremost, my thanks to my advisor, Prof. David Albonesi, for his guidance and direction throughout the last five years. Working for a Ph.D. is a challenging experience under any circumstances. Your advisor, however, can make the difference between a difficult, but rewarding experience, and some of the most unpleasant years of your life. I could not have asked for a better advisor than Prof. Albonesi. In addition to caring about my professional success, he was also concerned with my life outside of work, understanding, and even encouraging, my pursuit of activities outside of office life.

I would also like to thank the other members of my committee, Rajit Manohar and José Martínez, for their input on my thesis and their help in general over the years. Thanks also to the other faculty of Cornell CSL, past and present, especially Sally McKee for helping me turn a class project into a real paper, getting that paper published, and supporting me in my first (and third) overseas conference.

To the students of CSL, thank you for your advice and feedback and for keeping CSL a nice place to work. Special thanks to Mark, Paula, Basit, and Jonathan, my office mates throughout my tenure at Cornell, for keeping the office fun but productive.

Special thanks is also due to the people I interacted with outside of the office. These people where key to keeping me sane for the last five years and making sure I did something other than work all of the time. Of particular note are all of my friends from ballroom, especially my partners Julie, Ingrid, Margaret, Xiumin, and Jenn. It's amazing how far we progressed in the last four and half years.

To my family, near and far, thank you for your support over the years and for your interest in my work and my progress (even if you didn't understand half of what I was doing). It was always nice to be able to talk to someone who could actually talk back (sadly, computers haven't mastered that ability yet). Special thanks to my grandparents, Greem and Pop-Pop, for their constant interest in what I was doing and how life was progressing, for celebrating my successes, supporting me through the difficult times, and in general for being my number one fans throughout my education.

Lastly, my unending thanks to my parents, Michael and Sarah, for providing constant support and encouragement throughout my years of schooling, for creating an environment at home that encouraged learning and achievement (and sometimes overachievment), for listening and encouraging when times where tough, for keeping me connected to the real world and making sure I stayed informed of family happening, and for just being the best parents anyone could ever hope for.

## TABLE OF CONTENTS

## LIST OF TABLES

## LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

The microprocessor industry is rapidly transitioning to incorporating multiple processing cores on a single time. While the move away from single complex cores is indisputable, the most profitable way to organize these chip multiprocessors (CMPs) is a topic of ongoing debate. Given the variety of single- and multi-threaded applications that will run on a particular design, it will be extremely difficult to determine the optimal hardware/software system *a priori* at design time.

One attractive approach is to intermix conventional processing cores with reconfigurable logic that can be programmed to assume multiple specialized functions. This approach provides a form of *reconfigurable heterogeneity* that can be matched to changing workload characteristics at runtime. While the physical design may be homogeneous, with identical cores and attached programmable logic fabrics, each fabric can be configured at runtime to perform a different function according to the tasks assigned to the cores, or in the case of homogeneous threads from a parallel application, they may be configured identically.

Numerous proposals [6, 20, 22, 34, 39, 64, 65, 68, 91, 92] exist for integrating reconfigurable logic with general purpose cores. All of these proposals, however, were developed during the single core era and so the fabric was only attached to a single processor. As such, these designs typically suffered from a large area overhead and limited application coverage. The fabric might only speed up 50% of applications, yet consumed 50-75% of the chip area [34, 41, 68, 90, 91].

The recent move to chip multiprocessors, along with continued technology scaling, motivates us to reconsider the integration of reconfigurable logic on chip. Increased transistor density allows the integration of multiple cores and reconfigurable logic on the same die and provides the opportunity to share a single fabric among multiple cores. This amortizes the area overhead, increases the likelihood of use by at least some of the cores, and opens up optimization opportunities that are not possible with private fabrics. With this in mind, we propose RACM[1], a Reconfigurable Architecture for Chip Multiprocessors designed for the multicore era. RACM tightly integrates reconfigurable logic with traditional general purpose cores. Unlike previous reconfigurable proposals, which were designed for single core processors, RACM, and its associated *Specialized Programmable Logic (SPL)*, is specifically designed with today's chip multiprocessors in mind.

Figure 1.1 provides an overview of RACM, including (a) the integration of the SPL in a heterogeneous multicore system, (b) a high level view of the SPL architecture, (c) the design of the SPL computation cells, and (d) the interaction between the SPL and the RACM cluster manager. As a whole, RACM provides:

- A tightly integrated, row-based reconfigurable fabric to accelerate computation (Chapter 3);
- Mechanisms to temporally and spatially share the fabric among multiple cores (Chapter 3);
- Dynamic cluster management policies to manage the assignment of threads to cores and the partitioning of the fabric (Chapter 4);
- Fine-grained intercore data communication (Chapter 5);

[1]Pronounced "rack 'em."

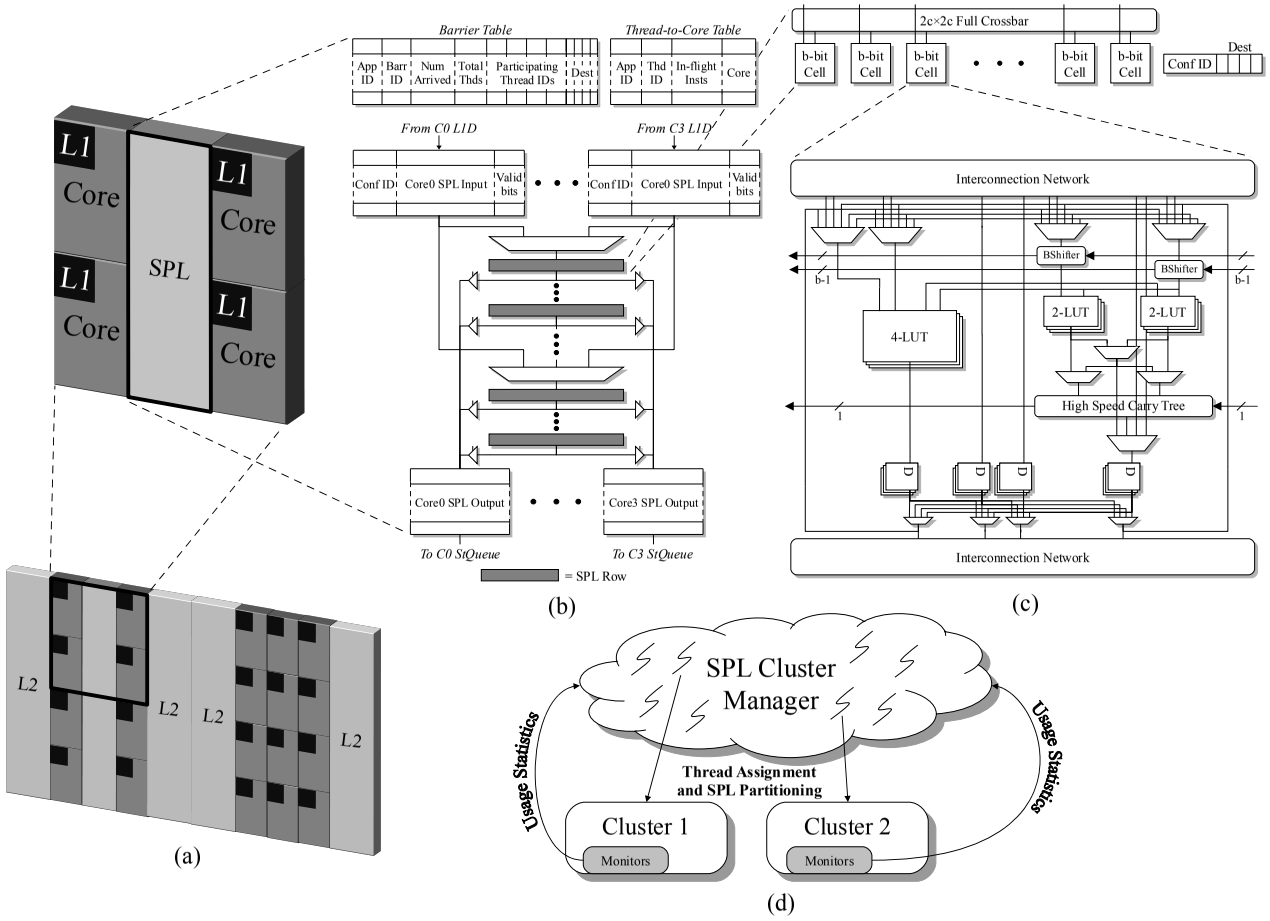



Figure 1.1: Overview of a RACM CMP. (a) Depiction of overall chip, with two SPL clusters and one conventional cluster, and blow-up of one SPL cluster, (b) four-way shared SPL including tables required for communication, (c) design of SPL cell (unless otherwise noted all data paths in the SPL are $b$ bits wide, and (d) SPL Cluster Manager.

- Fine-grained barrier synchronization (Chapter 5);
- The ability to perform computation on data that is being communicated (Chapter 5).

Each of these features will be discussed in the coming chapters.

The next chapter reviews literature related to all of these areas. We first discuss the numerous proposals for reconfigurable fabrics and architectures, focusing particularly on those that deal with the high level integration of a reconfigurable fabric with general purpose processors. Next, we examine prior work that has investigated thread scheduling and dynamic spatial partitioning as it relates to other shared structures, especially caches. Finally, we discuss proposals for fine-grained interthread communication and synchronization.

Chapter 3 details the design of the SPL and those features that we find to be particularly amenable to fabric sharing. Multiple dynamic sharing schemes are proposed and their performance is compared to the performance of providing each core with its own large, private fabric. We find that, with intelligent control policies, shared SPL is able to obtain the same performance as large private fabrics while providing a 4X reduction in fabric area and peak power.

Due to contention and communication issues, the SPL should only be shared among a limited number of cores. As such, there will likely be multiple SPL clusters in future many-core CMPs. The mapping of threads to clusters and the dynamic partitioning of each cluster can play a significant role in the performance obtained by each application. A number of dynamic management schemes are developed in Chapter 4 which control the assignment of threads to clusters and the dynamic partitioning of the SPL in order to maximize per-

formance. The best scheme, Hybrid Heuristic-Hill Climbing (H3C), combines elements of phase analysis, stability detection, and machine learning. Dynamic management with H3C outperforms both the per-workload oracle best static schedule and a system where the SPL is ideally replaced by additional cores.

In addition to reducing area and power overhead, sharing the SPL among multiple cores allows for optimizations not possible with private fabrics. In particular, the shared SPL provides a good platform for performing fine-grained communication between cores. Chapter 5 describes how RACM uses the fabric to allow fine-grained interthread communication and fine-grained barrier synchronization. This fine-grained communication allows parallelization of algorithms that might not otherwise realize a benefit from the additional cores provided in today's CMPs. Using the SPL for communication has the added benefit that computation can often be performed on the data while it is in flight to the consumer, providing performance enhancements not possible with dedicated hardware communication schemes. We show that, in the context of a heterogeneous CMP with both traditional and reconfigurable clusters, RACM can provide significant performance and energy benefits compared to what can be achieved with an area equivalent set of more powerful cores and idealized communication hardware.

CHAPTER 2

# RELATED WORK

## 2.1 Reconfigurable Computing

Several survey papers (e.g., [15, 37, 85]) provide an overview of the contributions of prior reconfigurable computing projects. We focus on those proposals that address the design of reconfigurable fabrics tailored to inclusion with general purpose cores, the characteristics of prior approaches that we find particularly amenable to shared fabrics, and the integration of the fabric within a CMP.

### 2.1.1 Fabric Design and Core Integration

Numerous proposals exist for integrating reconfigurable logic with a general purpose processor. Designs such as PRISC [63, 64], OneChip [92], Proteus [18, 19, 20], CoMPARE [68], Stretch [79], Chimaera [39, 93], and DPGA [21, 22] tightly integrate the fabric with the processor as a specializable execution unit. In DISC [90, 91] and NAPA [65] the fabric predominates with the processor serving largely to feed the reconfigurable hardware. Garp [6, 7, 41], PipeRench [34, 35, 69], and Tartan [56] fall in between.

Garcia and Compton [31] investigate a reconfigurable system in which the processor and reconfigurable fabric communicate via virtual memory. Carrillo and Chow [8] and Dales [18, 19, 20] evaluate the high level integration of a reconfigurable fabric with a single core processor and the overall system performance it provides. All of these, however, only investigate the integration with a

single core, although Garcia and Compton [31] state that their technique could be extended to a multicore system.

In [32], configuration data for a reconfigurable coprocessor is shared among multiple cores with the goal of increasing fabric utilization by allowing a larger number of configurations to concurrently exist in the fabric. Chen et al. [11] investigate the benefits of including reconfigurable ISA support in a multicore processor and find that combining program parallelization with custom ISA support provides larger speedups than the product of the two techniques applied in isolation.

The aforementioned designs have largely focused on accelerating certain application classes that see significant benefits from reconfigurable hardware. In addition to application acceleration, integrated reconfigurable hardware has been shown to be useful in other areas, such as aiding fault tolerance [74] and creating instruction path coprocessors to perform trace construction and optimization [14].

Reconfigurable computing has recently been gaining increasing attention from industry. Both Intel and AMD allow tighter integration of FGPAs with general purpose processors through HyperTransport, QuickPath, and licensing of front side bus technology [26, 27]. Convey Computer's HC-1 pairs an Intel processor with a reconfigurable coprocessor and allows different instruction sets to be loaded into the coprocessor [17].

### 2.1.2 Fabric Characteristics for Sharing

Several efforts have developed reconfigurable fabrics whose characteristics – in particular *higher computation granularity*, *row based design*, and *virtualization* – we find to be highly amenable to efficient SPL sharing.

Multiple works utilize coarser computation granularities than the bit-level configurability supported by FPGAs. Garp [41] and PipeRench [34], for example, use two and eight bits, respectively, as the smallest computation granularity. Using larger granularities reduces power, area, and configuration information at the cost of flexibility and density in design mapping. Most general purpose applications, however, do not require bit-level manipulation and so the savings tend to outweigh the costs. While this does not directly ease sharing *per se*, it does significantly reduce area and power.

Row based reconfiguration is employed by a number of designs [34, 90, 93] for several reasons. The cycle time of the fabric is set by the row delay and thus remains constant for all configurations. Using row based fabrics makes hardware design and application mapping easier and significantly reduces the routing complexity over traditional FPGA architectures. It also makes partial reconfiguration of the fabric easier as designs occupy a certain number of rows; so long as that number of rows is available in the fabric, it can be reconfigured by reprogramming only those rows.

Virtualization, such as that employed by PipeRench [34, 35], allows the fabric to handle configurations that require more rows than are physically available in the fabric. The costs of virtualization are degraded throughput, since the design

can no longer be fully pipelined, and higher power. Despite these drawbacks, virtualization is a key component of efficient shared fabrics.

## 2.2 Thread Scheduling

The benefits of dynamic thread scheduling in small scale CMP/SMT systems has previously been explored for a number of purposes, including cache-aware scheduling [10, 25, 47, 82] and thermal and power management [16]. Most of these efforts deal with temporally scheduling threads between time slices where the number of threads is greater than the number of processor contexts.

Cache-aware scheduling aims to minimize contention or maximize sharing between threads scheduled in the same interval. Tam et al. [82] create sharing vectors based on performance monitoring unit (PMU) data to cluster threads that share data in order to minimize long latency cross-core or cross-chip cache accesses. Fedorova et al. [25] use balance-set scheduling to create L2 conscious schedules. Chen et al. [10] investigate a parallel depth first task scheduling algorithm to increase data locality for certain parallel algorithms.

Constantinou et al. [16] investigate the performance impact of saving different portions of the processor state, including cache and predictor state, when migrating threads at various scheduling granularities. Such migrations might be used to minimize thermal variations or, in a heterogeneous CMP, to improve power efficiency by placing a thread on the core best suited to its current needs.

Thread assignment for a shared reconfigurable fabric shares some similarity with SMT scheduling. Snavely and Tullsen [76] use a trial based technique of

sample and "symbios" phases in which they sample the performance of a small number of possible schedules for a short period and choose the best performing of these schedules to run for the longer "symbios" phase. El-Moursy et al. [24] investigate continuous, on-the-fly thread scheduling based on a number of different possible metrics.

In the realm of reconfigurable computing, previous work has investigated the issue of configuration scheduling [2, 29]. Configuration scheduling tries to determine which implementation (either software or one of possibly many hardware implementations) should be employed for a particular function given current performance, area, and power restrictions. Our work is orthogonal to configuration scheduling and the two could be combined to further reduce conflicts for the SPL and improve performance.

## 2.3 Dynamic Resource Sharing

Previous research has proposed sharing other architectural components among multiple cores. L2 caches are likely the most prominent shared component with numerous works addressing how to best allocate cache space among multiple threads [12, 46, 80]. Sun's UltraSPARC T1 [81] shares a single floating point unit among its eight SMT cores. Kumar et al. [48] investigate sharing floating-point units, crossbar ports, and L1 instruction and data caches between two cores. Their work, however, focuses largely on temporal sharing, and does not consider dynamic spatial techniques such as splitting a cache in half if inter-thread conflicts are too high.

Prior work that optimized resource allocation during different program phases [23, 73] focused solely on single, sequential applications. In our multithreaded environment, each thread has its own current phase and we must deal with optimizing thread assignment and resource allocation as phases change across multiple applications.

Our situation is more difficult than any of this previous scheduling and shared resource work as we must concurrently manage both the assignment of threads to clusters and the dynamic partitioning of the fabric.

## 2.4 Fine-Grained Interthread Communication

StreamIt [36, 84] is a programming language and compiler infrastructure aimed at easing the use of pipeline parallelization. Decoupled Software Pipelining (DSWP) addresses hardware options for implementing fine-grain communication [61, 62], automatic extraction of streaming threads [57], data parallelization of pipeline stages [60], and speculative DSWP [87]. The Synchronized Pipelined Parallelism Model [88] parallelizes programs into a series of producing and consuming threads to keep data on-chip for as long as possible to minimize off-chip accesses.

SCORE [9] employs a stream computing model and targets reconfigurable systems. The design incorporates a single CPU and multiple reconfigurable blocks and streaming occurs between reconfigurable blocks over a dedicated interconnect. In our work, communication occurs between CPUs and the shared reconfigurable fabric is used to perform the communication.

None of this prior work evaluates the energy efficiency implications of streaming. Energy usage is a non-trivial concern given the fact that streaming tends to provide less than ideal speedups.

## 2.5 Fine-Grained Synchronization

Beckmann and Polychronopoulos [3] and Shang and Hwang [72] both propose hardware mechanisms for performing barriers using dedicated interconnect and hardware tables. IBM's Cyclops architecture [5] provides dedicated hardware support for barriers through a special purpose register and wired-OR. Sampson et al. [67] propose barrier filters to eliminate the dedicated interconnect required in most barrier synchronization proposals. The Multi-ALU Processor [45] provides an explicit barrier instruction in the ISA and supports register to register communication between clusters.

CHAPTER 3

# SHARED RECONFIGURABLE ARCHITECTURES FOR CMPS

Despite its potential, reconfigurable logic as an attached customizable unit has not yet been widely embraced by mainstream microprocessor manufacturers. One major impediment is the large power and area costs of FPGA technology relative to fixed functionality hardware [50]. While researchers have made significant progress in devising specialized fabrics to bridge this gap, many microprocessor architects still view the costs as too high to justify their mainstream adoption.

RACM addresses this gap by sharing a *Specialized Programmable Logic (SPL)* fabric among multiple cores. In this chapter, we describe the microarchitecture and low-level hardware control for shared SPL. In a multicore system where each core is coupled with its own fabric there are inevitably periods where one fabric is highly utilized while another lies largely or even completely idle. Thus, by *sharing* SPL fabric resources among multiple processor cores, programmable logic can be much more efficiently integrated – at far less power and area cost – into future multicore microprocessors.

In RACM, a number of standard cores share a common pool of SPL. Depending on the particular needs at any given point in time, each pool is dynamically partitioned among the cores, either *spatially*, where the shared fabric is physically partitioned among multiple cores, *temporally*, where the fabric is shared in a time multiplexed manner, or a combination of both. We show in Section 3.3 that pooled SPL configurations guided by effective control policies have little impact on performance for both parallel and coarse-grain multithreaded work-

loads – compared to private SPL attached to each core – while significantly reducing SPL area and energy costs.

## 3.1 SPL Architecture

In this section, we first motivate the need for research in more efficient SPL integration in CMPs, and quantitatively demonstrate the overall benefits of our approach. We then describe the RACM architecture, including high level system integration, the SPL microarchitecture, and our dynamic sharing control policies.

### 3.1.1 Motivating Example

We motivate our work by demonstrating the drawbacks of a straightforward application of prior SPL approaches to a CMP and the substantial benefits of intelligent SPL sharing. Figure 3.1(a) depicts a floorplan – with the L2 omitted but remaining areas drawn to scale – for a portion of a large-scale multicore chip[1] with eight single-issue out-of-order cores, each of which is coupled with an SPL fabric. Each SPL contains 26 rows of programmable logic, just enough to avoid virtualization for all eight applications. (Application statistics and modeling methodology are described in Section 3.2.)

The SPL area is roughly twice that of each core. Granted, a more complex core would consume more area, and we explore such options later in this chapter. Still, from an area and utilization perspective, there is clearly room for improvement. Figure 3.2 shows the utilization (percentage of the total number

[1]Other regions of the die may contain only cores and no SPL.

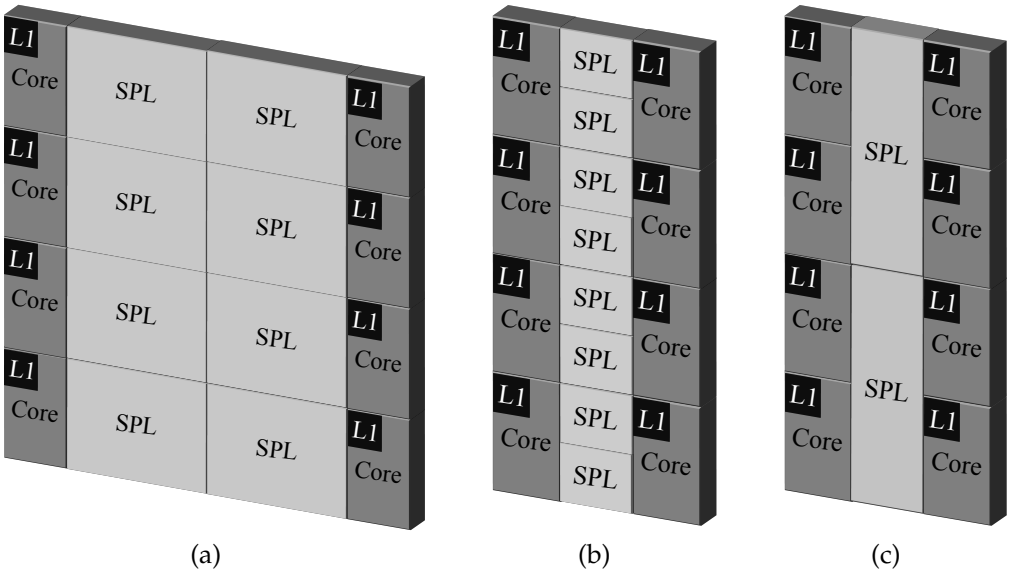



Figure 3.1: Sample floor plans for a portion of a multicore chip with (a) 26-row private, (b) 6-row private, and (c) 4-way shared SPL.

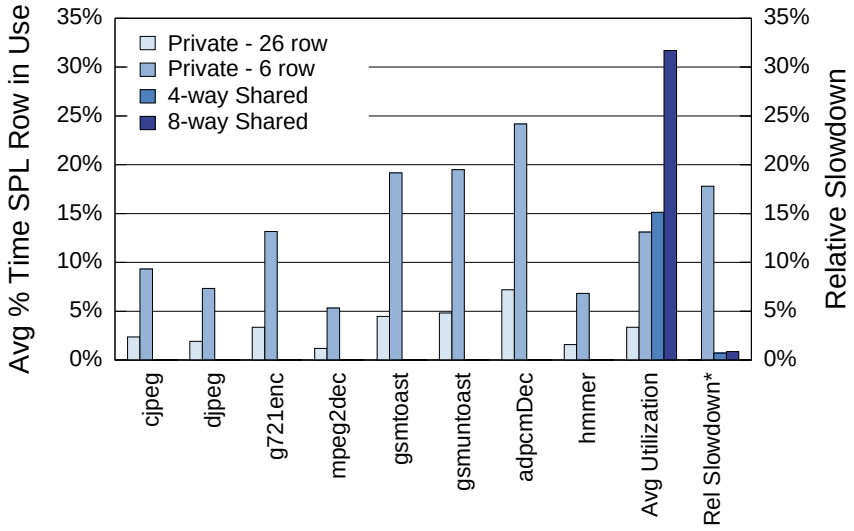



Figure 3.2: SPL utilization for private and shared SPL organizations. *The last set of bars shows percentage slowdown relative to 26-row private SPL.

of rows that are in use on average) and performance of several SPL configurations for a coarse-grain multithreaded workload of eight single-threaded applications, each of which runs on one of the eight cores. The leftmost bars for the individual benchmarks show the utilization of the 26-row configuration of Figure 3.1(a). The utilization of all eight of the SPL fabrics is less than 10% , and

the average SPL utilization is only 3.4%. Reducing each SPL to 6 rows (next set of bars) markedly increases SPL utilization for some benchmarks; moreover, the area is greatly reduced as shown in Figure 3.1(b). However, this comes at a high cost: an 18% overall performance loss, since all benchmarks use more than 6 rows.

By intelligently sharing the SPL among multiple cores, the average number of rows for each core *can* be reduced to six with little performance loss relative to the private 26-row configuration. Figure 3.1(c) shows a floorplan with two pools of SPL, each of which contains 24 rows and is shared – using control policies that we describe in Section 3.1.7 – among four cores. This configuration reduces the SPL area and peak power costs by over 4X. Furthermore, as shown by the third AvgUtilization bar[2] in Figure 3.2, the average utilization of the fabrics increases as well. These benefits come with virtually the same performance as the 26-row private configuration. The rightmost bars show that a single 48-row SPL shared among all eight cores further improves utilization, and also suffers negligible performance loss.

The contrast in performance between the private six-row and four-way shared SPL configurations – which have the same total number of SPL rows – motivates the need for good sharing policies. The six-row private configuration can be viewed as a spatially shared SPL organization with a naïve control policy that equally divides the SPL among all cores at all times. The shared configurations use a more intelligent policy that eliminates the 18% performance loss of the naïve approach.

[2]We do not show utilization for individual benchmarks since the fabrics are shared among multiple benchmarks.

### 3.1.2 System Overview

Figure 1.1(a) shows an overall depiction of a 20 core CMP with three *clusters*[3], with the external interface not shown for simplicity. Each of the two clusters on the left hand side consists of four single issue out-of-order processor cores[4] sharing a common pool of SPL. The SPL is a highly pipelined, row-based reconfigurable fabric and is described in more detail in Section 3.1.4. In the "conventional" cluster on the right hand side of Figure 1.1(a), each fabric has been replaced by two additional cores, giving 12 cores in total. Applications that are not compiled to use the SPL run on this conventional cluster, while those that exploit SPL run on one of the two left clusters.

### 3.1.3 Core/Fabric Integration

As with a number of previous designs [20, 64, 79, 93], the SPL fabric is tightly integrated with the processor core as a reconfigurable functional unit. However, rather than consume additional register file ports, we use a queue-based decoupled architecture to interface the SPL to the memory system. The SPL input queue matches the row input width and special SPL load instructions place values into the queue at a particular data alignment. Likewise, instead of writing to the register file, the SPL writes to an output queue that is then written out to the Store Queue using special SPL store instructions. Since the normal LSQ/cache datapath is used for data transfer, no additional steps are needed to handle memory dependences with the processor core.

[3] Although relative sizes of the cores and SPL are accurate, this is not intended to represent an actual floorplan.

[4] An analysis of different complexity cores and the reasons for selection of single issue out-of-order cores for the final design will be discussed in the following sections.

As discussed in Section 2.1, row-based designs employing virtualization are highly amenable to sharing. Figure 1.1(b) shows how our row-based SPL is modified to enable both spatial and temporal sharing between two cores. For spatial sharing, additional muxes select input bits at the entry point of the SPL and at each point where the SPL pool might be partitioned. Furthermore, an additional set of tristate drivers tap off of each row output to drive the sharer's output queue. Finally, there is additional wire overhead to get data to and from multiple cores. These wires can be pipelined if necessary to match the SPL clock frequency at the cost of additional pipeline initiation time. However, with deeply pipelined row-based fabrics, the cost is small and is outweighed by the efficiency gains.

For temporal sharing, all rows of the fabric are available to all cores in a time multiplexed fashion. Therefore, only the muxes at the head of the fabric are required.

### 3.1.4 Fabric Microarchitecture

An SPL fabric consists of a collection of rows. Each row contains $c$ cells, and each cell computes $b$ bits of data. Figure 1.1(c) shows the cell and row design. The major cell components are a main 4-input look-up table (4-LUT), a group of 2-LUTs (equivalent to one 3-LUT) plus a fast carry chain to compute carry bits or other logic functions if carry calculation is not needed, barrel shifters to properly align data as necessary, flip-flops to store results of computations, and an interconnection network between each row. Within a cell, the same operation is performed on all $b$ bits. These $b$-bit cells are arranged in a row to form a

$c \times b$-bit row. Each cell can perform a different operation on its inputs and a number of rows are grouped together to execute an application function. The clock frequency is set such that the longest possible computation within a row completes in a single SPL clock cycle.

As shown previously [34], several trade offs dictate the optimal choice of cell width, row width, and number of rows. Increasing the cell bit width decreases area and power at the cost of less configuration flexibility. Increasing the row width allows more computation in a single cycle but also increases the likelihood of less than 100% resource utilization if not all of the cells in a row can be put to use. Furthermore, the fabric width should match the ability of the memory system to supply data at a fast enough rate. Finally, reducing the number of rows has linear area and power benefits but increases the number of functions that must be virtualized.

To quantitatively evaluate these trade offs, we create analytical area, latency, and power models for SPL in 65nm technology. The model combines estimates for the different components of the fabric. We use Cacti 4.2 [83] to model LUTs, Orion [89] for between row interconnect modeling, the models of [42] for local wiring and between cell wires (such as for the barrel shifter and carry logic not included in the Orion model), and the work of [40] for the carry chain logic implementation. Finally, various bit level components, such as transistor delay, area, and power, used to compute estimates for muxes and small SRAM cells, are taken from the ITRS [70].

To validate our model, we compare scaled values of area, latency, and power available from previous reconfigurable fabric designs [41, 69, 39] with predictions by our model for these fabric architectures. The results are given in Ta-

Table 3.1: Comparison of data from actual reconfigurable fabrics (scaled to 65nm) and the analytical model.

| | Scaled Actual | Model | % Diff |
|---|---|---|---|
| PipeRench - 8-bit, 16-cell, 16-row - 180 nm | | | |
| Area ($mm^2$) | 1.5 | 1.59 | 6.0% |
| Frequency (MHz) | 350 | 460 | 31.4% |
| Power (W) | 0.929 | 0.832 | -10.5% |
| Chimaera - 1-bit, 32-cell, 32-row - 0.6 $\mu$m | | | |
| Area ($mm^2$) | 0.805 | 0.751 | -6.7% |
| Frequency (GHz) | 1.27 | 1.06 | -16.5% |
| Garp - 2-bit, 24-cell, 32-row - 0.5 $\mu$m | | | |
| Area ($mm^2$) | 1.32 | 1.06 | -19.7% |

ble 3.1. The Scaled Actual Area values in this table are derived by scaling the fabric area of each design by the square of the ratio of the technology factors. For frequency, the reported values are linearly scaled by the technology ratio. The Scaled Actual Power value for PipeRench is derived by scaling the reported power value by the square of the ratio of the PipeRench voltage to the SPL voltage, by the ratio of the PipeRench frequency to the SPL frequency, and by the ratio of the technology factors (to account for capacitance scaling). The model achieves good correlation except for the frequency in PipeRench and Chimaera and the area of Garp. For PipeRench, the PipeRench paper notes that the circuit design was not highly optimized [69], and therefore we expect that a frequency closer to our higher value could be achieved in an industrial PipeRench design. Given the seven technology generations between our design and Chimaera, a 16% error is not unexpected. For the area disparity with Garp, the information available for Garp is limited, making good correlation with the Garp design difficult. Specifically, only the area of the entire Garp chip is provided and we determine the area consumed by the fabric by determining the ratio of fabric area to total chip area from the provided floorplan.

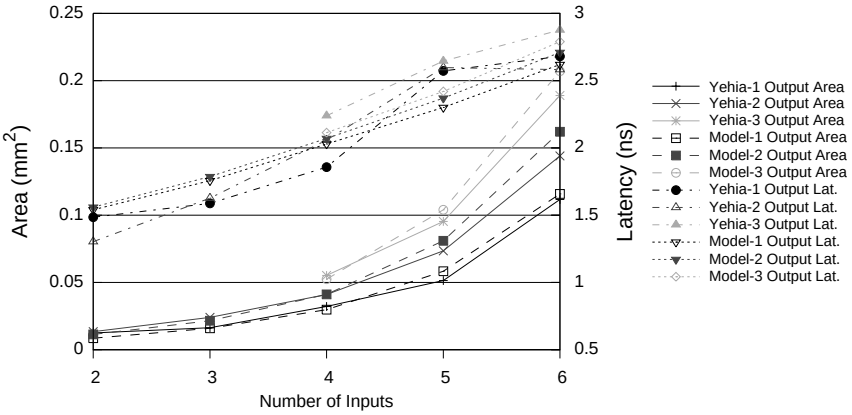



Figure 3.3: Comparison of latency and area predicted by analytical model to results reported in [95].

Yehia et al. [95] perform an area and latency design space exploration for programmable function units with varying number of inputs and outputs (among other parameters). Figure 3.3 shows area and latency comparisons between our model and the results presented by [95] for fabrics with different numbers of row inputs and outputs. The model error is within 15% in all but two cases.

We use our model to estimate the costs of different configurations for the various functions that we map to the SPL (discussed in Section 3.2). An 8-bit wide cell with 128-bit wide rows provides a good compromise between flexibility and area/power cost, and permits significant parallelization. Each SPL function can take in up to 512 bits of input and can produce up to 128 bits of output. This organization achieves a reasonably high frequency (500MHz) relative to the processor core frequency (assumed 2GHz at 65nm, the same as the Pentium Core2 Duo [43] and the AMD X2 Dual-Core [1], both of which are implemented in the same 65 nm technology). At this one-quarter clock speed differential, four quadword load instructions can supply 512 bits to the SPL pipeline every SPL clock period.

Table 3.2: Area and power of different core types and 26-row SPL normalized to IO area and power.

| | Area | Dynamic Power | Leakage Power |
|---|---|---|---|
| IO | 1.00 | 1.00 | 1.00 |
| OOO1 | 1.19 | 1.06 | 1.05 |
| OOO2 | 1.82 | 1.26 | 1.26 |
| OOO4 | 4.87 | 1.66 | 1.63 |
| SPL | 2.49 | 0.66 | 3.02 |

For the baseline 26-row private SPL, we include on-chip storage for eight configurations to allow for fast switching between different configurations. For our workloads, this permits all configurations for any phase to reside on-chip. Thus, reconfiguration latency is not an issue as all configurations are immediately available after the initial configuration overhead is paid.

To gauge the sensitivity of our evaluation to the exact details produced by our model we evaluated an SPL design where the SPL is 50% slower than described above (i.e., one SPL cycle is equivalent to six processor cycles). Despite this 50% increase in latency, our workloads experienced only an average 12-16% lower performance than the 4 cycle baseline depending on the complexity of the attached core. This confirms that our conclusions are not overly reliant on the exact results of our model. Even if the SPL opperates a little slower than predicted, similar performance benefits can still be achieved.

Table 3.2 shows the area and power of a 26-row SPL compared with four conventional core types: a single-issue in-order core (designated as IO), and one-, two-, and four-way issue out-of-order cores (OOO1, OOO2, and OOO4). Results are normalized relative to the IO core. Each core has separate 8 kB L1 instruction and data caches. We adopt the methodology of Kumar et al. [49] to calculate per-core area and power costs. We note that an OOO1 core augmented

Table 3.3: SPL configurations and associated area and power costs.

| | Rows/ SPL | Configs/ Row | Total SPL Area ($mm^2$) | Dynamic Energy/ Row (nJ) | Total SPL Leakage Power (W) |
|---|---|---|---|---|---|
| Eight Private | 26 | 8 | 23.74 | .0600 | 4.59 |
| Four 2-way Shared | 12 | 8 | 5.55 | .0601 | 1.06 |
| Two 4-way Shared | 24 | 10 | 6.03 | .0601 | 1.08 |
| One 8-way Shared | 48 | 12 | 6.62 | .0601 | 1.10 |

with SPL has an area that falls between OOO2 and OOO4, while the area of OOO2 + SPL is slightly less than that of OOO4.

We create shared SPL configurations by pairing OOO1 cores with an SPL that consumes approximately half the area of the sharing cores. OOO1 cores are selected as, of the cores investigated, adding SPL to single-issue out-of-order cores provides the greatest relative benefit (see Section 3.3.1 for further details). A six row SPL consumes slightly more than half the area of an OOO1 core and the combination of the two is smaller than OOO2.

Given these constraints, we arrive at the shared SPL configurations shown in Table 3.3, which also includes the baseline private 26-row SPL organization for comparison purposes. In terms of total area and leakage power, four two-way shared SPLs come at slightly less than a quarter of the cost of the eight private SPLs. The reduced number of rows, however, means that functions will need to be virtualized more often. Indeed, any function requiring all 26 rows of the private SPL will be virtualized in both a two- and four-way shared SPL, even if the other cores are not using the shared SPL at that time. SPL configurations with a higher degree of sharing consume a slightly larger area than the two-way shared configuration but they require less virtualization. When several applications simultaneously reach a point where they do not need the SPL, it can be allocated among the remaining cores so that virtualization can be avoided. The

area increase for higher degrees of sharing comes from the additional datapath hardware and on-chip configurations for higher degrees of sharing, with the latter contributing most of the increase. As with the private fabrics, the shared SPL designs provide enough on-chip configuration storage such that, for our workloads, all configurations for any set of application phases can reside on-chip.

### 3.1.5 Virtualization

Virtualizing reconfigurable hardware was proposed by [6] to allow a fabric to execute a configuration that requires more resources (i.e., rows) than are physically available. Although this ability comes at the cost of reduced throughput when the design must be virtualized, virtualization permits the designer to trade performance for area. As more area becomes available (or for higher-end chips) larger fabrics can be created without requiring any change to the application mappings.

Virtualization is accomplished by using the same physical row to execute multiple virtual rows. Figure 3.4 shows an example in which a function requiring six virtual rows is virtualized over a fabric with only three physical rows. Figure 3.4(a) shows the six virtual rows of the function while Figure 3.4(b) shows the actual physical rows. The number in each cell shows which function row is currently loaded into that row during that cycle. In the first three cycles new data is input each cycle. At the end of the third cycle the first piece of data is ready to move onto the fourth virtual row of the function. Since no more physical rows are available, the first physical row is reconfigured to virtual row four and virtual row one is cached for possible later use. As such, new data cannot

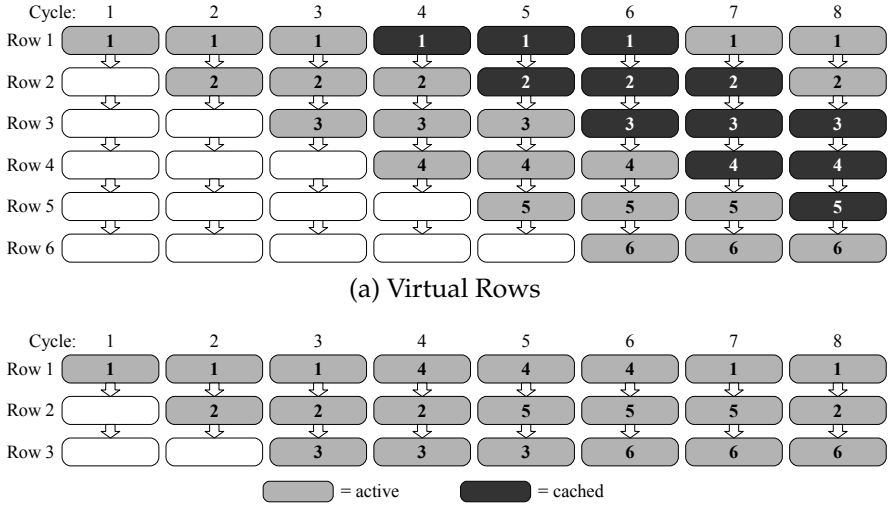



Figure 3.4: Example of a six row configuration being executed on a three row SPL using virtualization [34]. Numbers inside each block indicate the configuration loaded in each row.

be input during this cycle. This reconfiguration process continues until the last row of the function is loaded, at which point the fabric loads the first virtual row of the function again if there is more data waiting to be processed and the sequence is repeated. In this example, the cost of virtualization is a 50% reduction in throughput as results are only output every three out of six cycles.

For shared fabrics the number of rows available to a function is not known at application design time even for a particular fabric implementation as the function may not be allocated the entire SPL. As such, virtualization is especially useful for shared fabrics as it allows all SPL functions to be executed – albeit with possibly different throughput – regardless of the number of rows that are allocated to the function at runtime.

Table 3.4: ISA extensions.

| Instruction | Description |
|---|---|
| `spl_lsize align, offset(reg)` | Load data of size *size* into SPL input queue at alignment *align* |
| `spl_ssize align, offset(reg)` | Store data of size *size* from SPL output queue to memory at alignment *align* |
| `spl_init config` | Invoke SPL using configuration *config* |
| `spl_prefetch config` | Prefetch SPL configuration *config* into configuration memory |

### 3.1.6 Software Interface

We now describe the instruction set extensions required to utilize the SPL. We take the position that sharing, like virtualization, should be transparent to the compiler. Thus, the SPL fabric controller handles sharing conflicts using the policies we describe in the next section.

Table 3.4 shows the instructions used to access the SPL. The SPL load instruction loads an operand of size *size* from the effective address into the SPL input queue at position (alignment) *align*. The instruction executes in the core pipeline but the data is loaded into the SPL input queue rather than the register file. With our SPL microarchitecture, four quadword SPL loads, each offset by a quadword, can fill the full SPL input width in a single SPL cycle (four processor cycles). SPL stores similarly store an operand of size *size* at alignment *align* from the SPL output queue to the memory system (Store Queue). Here also, the effective address is calculated in the core pipeline. The instruction waits for the head SPL output queue Valid bit to be set before performing the transfer.

The `spl_init` instruction invokes the SPL function described by configuration *config*. The corresponding configuration ID is loaded into the input queue along with the data. This ID is used to load the appropriate configuration as the

```
spl_lq 0, 0($6)
spl_lq 16, 16($6)
spl_init dist1
spl_sw 0, 16($sp)
```

Figure 3.5: SPL assembly code for MPGenc `dist1` function. The reference `dist1` is a pointer to the SPL configuration information.

data passes from row to row. Once the `spl_init` command is issued the queue advances such that subsequent loads get placed into the next slot, in preparation for the next `spl_init`.

If the configuration data corresponding to *config* is not present in the on-chip configuration memory, then a delay is incurred while it is loaded from off-chip. Subsequent invocations of this function will pay no penalty if the previously loaded configuration is still present. The `spl_prefetch` instruction can be used to prefetch this information in advance of its first being used [38, 54], analogous to prefetching memory data into cache.

We provide an example to illustrate the use of these instructions. Figure 3.5 shows the SPL assembly code for the *dist1* function from the ALPBench MPGenc benchmark [52]. This function is used for motion estimation in the encoding process and computes the absolute difference between two image blocks. The function operates on eight-bit values. The simplest "if" case requires 16 additions, 16 subtractions, and a variable number of negations (to produce the absolute value). The 16 subtractions and absolute value comparisons can each be performed in parallel and require a total of four rows. The accumulation of the 16 values requires an addition tree which needs four more rows to complete, resulting in a total of eight rows.

The base address for the 32 bytes of input data is located in register six. The input data are loaded via two quadword SPL load instructions separated by memory and input queue offsets of 16 bytes. The *dist1* function is then invoked using the `spl_init` instruction. The configuration ID associated with *dist1* passes row-by-row with the data. After the data is written into the output queue, it can pass in the next cycle to the Store Queue if the SPL store instruction has already issued – a likely scenario.

If the full input width (64 bytes) was required, four quadword loads would have been needed. To sustain the maximum SPL bandwidth over multiple iterations of this function, the load instructions in the core pipeline must overlap with processing of the `spl_init` instruction (extracting the configuration ID and placing it in the appropriate queue position). Otherwise, the loads for the next iteration would be delayed by the `spl_init` and associated store of the results of the previous function.

### 3.1.7 Sharing Policies

We explore hardware-level control policies for spatial and temporal SPL sharing. The advantage of spatial sharing is that each core is given dedicated resources and is guaranteed to make forward progress each cycle, although possibly at a degraded rate due to increased virtualization. In temporal sharing, all cores vie for the same SPL resources, but those resources are larger and so virtualization occurs less often, perhaps even less so than with private SPLs. We discuss each of these approaches in turn and then propose a hybrid approach that attempts to capture the benefits of both.

**Spatial Sharing Algorithm**

The first decision with spatial sharing is how finely to divide up the fabric. One can choose an equal number of rows based on the number of sharers, i.e., the SPL is split into $n$ equal sections if there are $n$ sharers, or split according to expected application usage. These approaches require a large number of intermediate muxes. We investigated a simpler alternative that splits by only powers of two; if, for example, there are 5-8 sharers, the SPL will be broken into eight partitions and some of these may lie unused.

To determine when to merge SPL partitions, per-core idle cycle counters and an idle count threshold value (1000 in our implementation) are associated with each shared SPL pool. When a core has no SPL instructions in-flight, its idle counter is reset. The counter counts up each cycle that the core does not request use of the SPL. Once the threshold is reached, the SPL checks to see if the number of threads now using the SPL falls within a different power of two partition. If so, it waits for all current in-flight functions to finish and then repartitions the SPL, allocating the core's partition to the other active sharers.

Figure 3.6 shows an example progression with spatial sharing. Initially, all SPL blocks are unpartitioned. When a core first requests the SPL, if it is not in use by any other core, the core is allocated the entire fabric (Figure 3.6(a)). If the SPL is already in use by another core or cores, and there is not a free partition, the SPL must be split. When the SPL controller detects a new sharer it stops issuing SPL instructions from cores currently sharing the SPL, waits for all current in-flight SPL instructions to complete and then splits the SPL into the appropriate number of new partitions (Figure 3.6(b)). Once split, all cores can begin issuing SPL instructions to their respective partitions (Figure 3.6(c)).

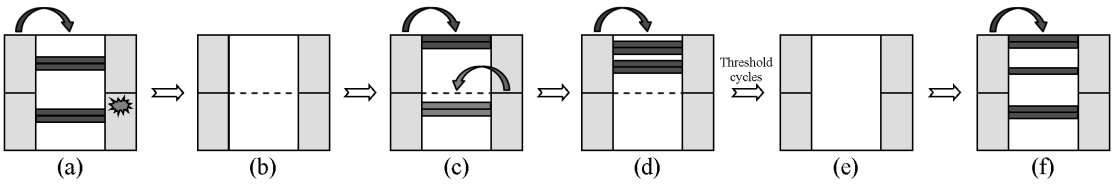



Figure 3.6: Example of spatial sharing. In (a) a single core is using the SPL when a second core makes a request. The current instructions complete and then the SPL is split (b). In (c) both cores issue SPL instructions to their partitions. After some time the second core stops issuing instructions and its partition empties (d). After the merge threshold has passed, the in-flight instructions complete and the SPL is merged (e). Finally in (f), the first core continues using the entire SPL.

At some point, one of the cores may stop issuing instructions to its partition (Figure 3.6(d)). After the partition has remained idle for a certain period the SPL is again drained and the partitions are merged (Figure 3.6(e)). Finally, once merged, the remaining threads can continue to use the SPL (Figure 3.6(f)).

### Temporal Sharing Algorithm

For temporal sharing, we share the fabric among the cores in a round-robin manner on an SPL cycle-by-cycle basis. This ensures that every processor makes progress but can require the fabric to swap configurations more frequently. An alternative policy could favor the thread that last used the SPL to reduce the possibility of configuration thrashing. Depending on how frequently a thread requests the SPL, however, this could lead to starvation of other processes. We chose round robin due to our good experience with this policy in other contexts.

Figure 3.7 shows two cores sharing an SPL using round robin temporal sharing. Initially, only one core is using the SPL and these instructions get issued immediately (Figure 3.7(a)). When the second core begins using the SPL, instructions are issued in a round robin manner (Figure 3.7(b-d)). The per-core

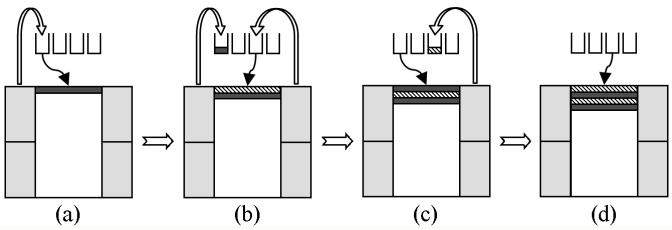



Figure 3.7: Example of temporal sharing. In (a) the first core issues an instruction which immediately gets issued to the SPL. In the next cycle (b), both cores issue SPL instructions. With round robin scheduling the instruction from the second core is issued and the other instruction is placed in the queue. The next cycle (c), the queued instruction from core 1 is issued and the request from core 2 is queued. Finally in (d), the queued instruction from core 2 is issued.

input queues permit each core to prepare multiple SPL instructions while the core awaits its turn in the rotation.

### Hybrid Sharing Policy

We also explore a hybrid combination of the previous two techniques. The policy uses round robin temporal sharing but splits the SPL spatially when the overhead due to temporal sharing becomes excessive. To determine when to spatially split or recombine the SPL we track the average amount of time a queued SPL instruction spends waiting to be issued.

To implement this policy each SPL input queue tracks over a fixed time interval (1,000,000 cycles in our implementation) the number of SPL instructions issued and the total queue wait time. At the end of the interval, the average wait time per SPL instruction is computed. If this value exceeds an upper threshold (4 in our implementation), the SPL is split. Likewise, if the average wait time is less than a second threshold (0.125), the two clusters are merged. This split/merge operation is performed in a hierarchical manner such that a cluster

Table 3.5: Architecture Parameters.

| Processor Configuration | IO | OOO1 | OOO2 | OOO4 |
|---|---|---|---|---|
| Fetch/Decode Width | 2 | 2 | 4 | 8 |
| Issue/Retire Width | 1 | 1 | 2 | 4 |
| Branch Predictor | gshare + bimodal | | | |
| BTB Size | 512B | 512B | 1KB | 2KB |
| RAS Entries | 32 | | | |
| Integer Registers | 32 | 64 | 64 | 64 |
| FP Registers | 32 | 64 | 64 | 64 |
| Integer Queue Entries | 32 | | | |
| FP Queue Entries | 16 | | | |
| ROB Entries | X | 64 | | 96 |
| Int ALUs | 1 | 1 | 2 | 3 |
| Branch Units | 1 | 1 | 2 | 3 |
| Int Mult/Div Units | 1/1 | | | |
| FP ALUs | 1 | | | |
| LD/ST Units | 1/1 | | | |
| L1I Cache | 8kB WT 2-way, 2-cycle access | | | |
| L1D Cache | 8kB WB 2-way, 2-cycle access | | | |
| L2 Cache | 1MB per core WB 8-way, 10-cycle access | | | |
| Main Memory Access Time | 100 ns | | | |

will always merge with the cluster it most recently split with before merging with any other cluster. Merges must also be performed on clusters of the same size. Although other split/merge options are possible, we chose these restrictions to reduce hardware and control overhead.

## 3.2 Evaluation Methodology

We use a highly modified version of SESC [71] to perform our evaluation. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. The major architectural parameters are shown in Table 3.5. We use Wattch and Cacti to model dynamic power and Cacti and HotLeakage to model leakage power.

### 3.2.1 Benchmarks

We investigate shared SPL under both coarse-grain multithreaded and parallel workloads. For our coarse-grain workload, we use one benchmark from SPEC 2006 [78], four benchmarks from MediaBench [51] and three benchmarks from MediaBench II [28]. For parallel workloads, we select two benchmarks from the ALPBench suite [52] and a version of the Java Grande [75] *crypt* benchmark ported to C++. We run the ALPBench version of *MPGdec* with two different command line parameters (-o0 and -o3) as they produce different execution characteristics. Specifically, the o3 version executes additional code that utilizes the SPL.

Since the SPL versions of each benchmark execute a significantly different number of instructions than the baseline, we compare execution times for complete program execution as IPC comparisons are meaningless. Due to the long runtime of *456.hmmer* under reference inputs we use Early SimPoints [58] to approximate whole program execution for this benchmark. We select two 250 million instruction SimPoints from the original source code (i.e., code not utilizing the SPL). We determine where each of the two SimPoints begin and end and augment the code to fast forward through all but these two intervals. In this way both the original and SPL versions of the code execute functionally equivalent amounts of code. We continuously respawn jobs that finish before the longest running thread in order to maintain a consistent SPL access pattern from all threads.

Benchmark statistics are given in Table 3.6. All but four of the benchmarks use at least 20 rows, but few of these occupy the SPL for any great length of

Table 3.6: Benchmark, number of SPL functions, maximum rows used by SPL functions, percentage of execution time of optimized regions, percentage of SPL instructions executed relative to total committed instructions, and percentage of time with at least one SPL instruction in flight for OOO1 cores.

| | SPL Functions | Max Rows | % Optimized Exec Time | % Dyn. SPL Insts | SPL Usage |
|---|---|---|---|---|---|
| cjpeg | 5 | 21 | 49.9% | 1.19% | 17.77% |
| djpeg | 3 | 23 | 61.9% | 0.84% | 9.72% |
| g721enc | 1 | 26 | 45.5% | 0.67% | 24.11% |
| mpeg2dec | 3 | 10 | 62.9% | 1.07% | 22.28% |
| gsm_toast | 2 | 16 | 54.2% | 2.83% | 28.92% |
| gsm_untoast | 1 | 22 | 75.8% | 2.18% | 36.55% |
| adpcmDec | 1 | 24 | 95.9% | 10.29% | 79.22% |
| 456.hmmer | 1 | 10 | 85.0% | 1.15% | 40.51% |
| MPGenc | 4 | 16 | 69.1% | 0.72% | 17.23% |
| MPGdec_o0 | 5 | 20 | 44.8% | 0.35% | 15.30% |
| MPGdec_o3 | 12 | 20 | 47.8% | 0.57% | 19.25% |
| crypt | 1 | 298 | 97.9% | 4.48% | 99.90% |

time. This indicates that for our workloads and architecture, there may be more opportunity for temporal rather than spatial sharing.

### 3.2.2 SPL Programming and Function Mapping

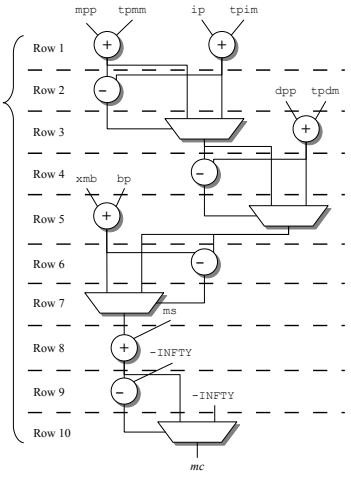
The SPL is used to accelerate a wide range of operations. We show one example function mapping from the SPEC 2006 application *456.hmmer*. We accelerate the `P7Viterbi` function, which accounts for 85% of the program execution time. The core loop of the function is shown in Figure 3.8(a). Figure 3.8(b) shows how the portion of the code that calculates `mc` is mapped to the SPL. In the optimized code, the core first loads the input values needed to compute `mc` into the fabric, the SPL computes the value of `mc`, and finally the core receives the result. After receiving `mc`, the core computes the values of `dc` and `ic` and repeats the loop.

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
  if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

(a) Source code




(b) Calculation of `mc` in SPL

Figure 3.8: Mapping SPEC2006 *456.hmmer* `P7Viterbi` to SPL.

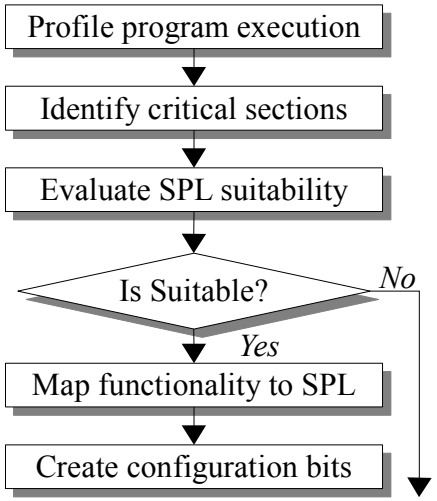



Figure 3.9: Procedure of mapping functions to SPL.

We modify our workloads by hand to create the SPL mappings. Previous work has shown that compilers can produce good mappings for reconfigurable architectures [4, 7, 35, 53, 94], and We believe our design could leverage this prior art in an actual implementation.

The procedure for identifying and mapping functionality to the SPL is shown in Figure 3.9. Each application is first profiled to identify the most important regions of execution. Each of these identified regions is then evaluated for SPL

suitability in terms of the number of inputs and outputs required, the type of required operations (e.g., integer addition/subtraction, Boolean operations, and conditional selection), and inter-operation dependencies. SPL mappings, similar to that discussed below, are then generated for the selected regions, and these mappings are transformed into SPL configuration bits.

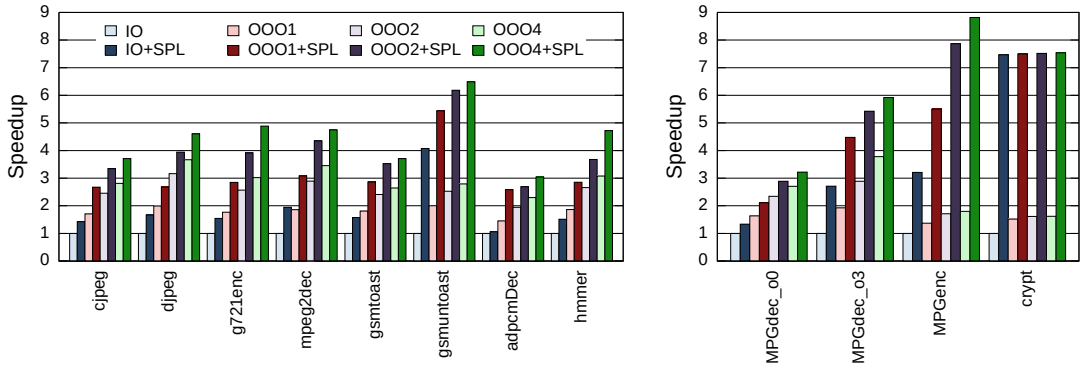
## 3.3 Results

Prior work has shown that SPL coupled with a standard processor can significantly speed up certain application classes, e.g., multimedia workloads. We confirm these results for our private SPL design. We further analyze energy consumption, evaluate parallel workloads, and address how well SPL augments cores of different complexity, important topics that have not been previously covered in detail. Using these results as a baseline we then evaluate the performance of shared SPL.

### 3.3.1 Characterization of CMPs with Private SPL

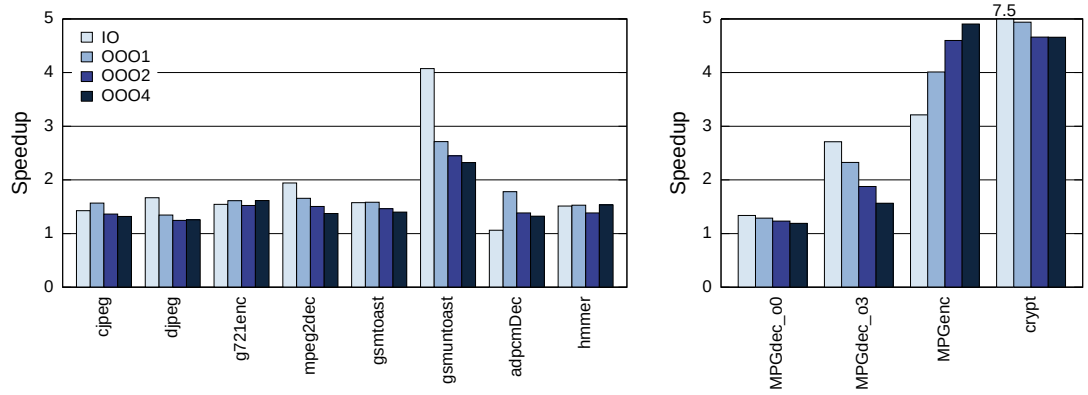
The performance of coarse-grained (showing individual benchmark performance) and parallel workloads on different complexity CMPs, with and without private SPL, is shown in Figure 3.10. For each benchmark, the values are normalized to the performance on an IO core without SPL.

In seven of the eight coarse grain benchmarks and three of the parallel benchmarks, a CMP with an n-way OOO core plus SPL outperforms the next larger




(a) (b)

Figure 3.10: Performance for (a) coarse-grain and (b) parallel workloads relative to IO cores without SPL.




(a) (b)

Figure 3.11: Performance for (a) coarse-grain and (b) parallel workloads relative to the same core type without SPL.

OOO core; in six of these cases the OOO1+SPL outperforms the OOO4 core, which consumes far more area and power.

Figure 3.11 shows SPL performance relative to the performance of the same base core without SPL. The relative speedup provided by adding SPL generally decreases as core complexity increases. This occurs due to the higher baseline performance of the more complex cores, and the limit of one SPL instruction each cycle independent of the issue width. In a number of cases, how-

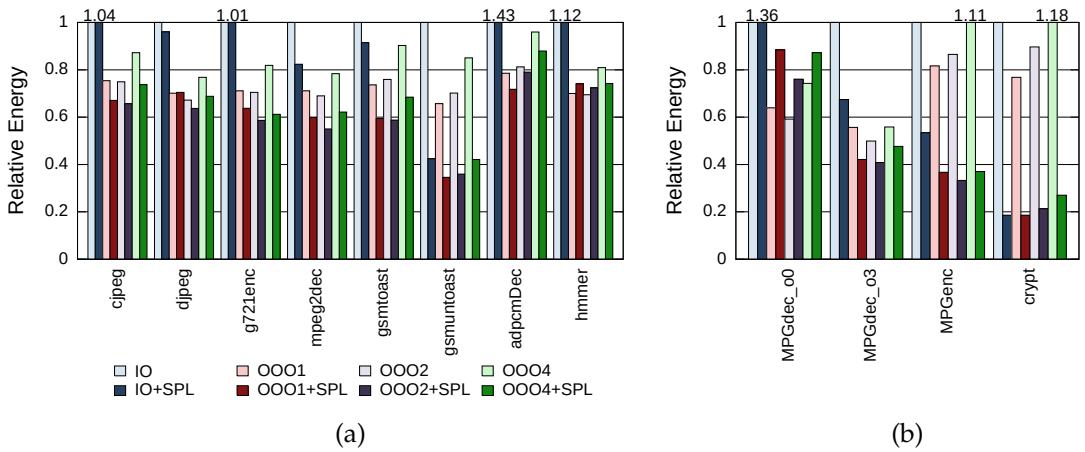



Figure 3.12: Energy consumption for (a) coarse-grain and (b) parallel workloads relative to IO core without SPL.

ever, adding the SPL to the single-issue OOO core provides larger benefits than adding it to the single-issue IO core. This is particularly pronounced in *adpcm* where adding SPL to the IO core provides only 6% benefit, while adding it to the OOO1 core yields a 78% performance improvement. This shows that, in certain cases, there are clear benefits from being able to reorder instructions when using the SPL. There are also a few instances where the four-way issue core outperforms the two-way issue core. This is due to the larger ROB, which allows additional SPL instructions to be processed while waiting for earlier SPL instructions to complete.

In terms of energy, adding SPL tends to decrease energy consumption despite the additional power consumed by the SPL. This reduction results from both decreased execution time, which reduces total leakage energy, and the elimination of many dynamic instructions, which reduces dynamic energy. When using the SPL, a single SPL instruction – albeit, an expensive one in terms of execution energy – replaces a very large number of conventional instructions, generally decreasing the total energy consumed. The payoff depends on the

complexity of the fabric mapping – with operations like multiplication being particularly costly – versus the number and type of conventional instructions that are replaced. Figure 3.12 shows the relative energy consumption for our coarse-grained and parallel workloads normalized to IO cores without SPL. An n-way OOO core plus SPL achieves a greater energy reduction than the next larger OOO core in 83% of the cases, and in all but one case OOO1+SPL consumes less energy than OOO4. Energy savings are so significant that, when considering energy$\times$delay$^2$ (ED$^2$), OOO1+SPL outperforms OOO4 in nine of the cases compared to only six when considering execution time alone, again with the aforementioned area savings.

### 3.3.2 Evaluation of Shared SPL

We now evaluate systems in which the SPL is shared among two, four, and eight cores. Using the results from the previous section, and relative area and power comparisons, we pare down the number of reasonable comparison points. We pair shared SPLs with OOO1 cores since this core plus six rows of SPL consumes similar area to OOO2 and, of the cores we investigate, OOO1 cores receive the most benefit (as quantified by energy$\times$delay$^2$) due to the addition of SPL. We compare the shared configurations to two versions with private SPL: one with the full 26 rows, and another with six rows, the same per core amount as the shared cases (Table 3.3). We also compare shared SPL to an alternative where the SPL has been replaced by additional cores. In our design, four OOO1 cores consume approximately the same area as the shared SPL, leading to a system with 12 OOO1 cores. For the coarse-grain multithreaded benchmarks we ideally scale performance by 50% to model the performance of a throughput ori-

ented system where multiple instances of the same application are running. For parallel applications we run twelve threaded versions of each benchmark.

For multithreaded workloads with two- and four-way shared SPL, we need to consider which applications should be scheduled together on the same SPL pool. Our algorithm statically schedules threads based on the average number of rows used by each application, with the objective of roughly equalizing total row usage. We schedule *g721enc*, the highest utilization thread with *mpeg2dec*, the lowest utilization one. We then pair the next highest with the next lowest on the next SPL pool, and so on until all threads are scheduled. Although this might seem to be the best possible assignment, thus possibly exaggerating the benefits of sharing in less ideal circumstances, we found that, although this produces a good schedule, it is not necessarily the best as other factors, such as wait time to access the fabric and the frequency of SPL instructions, also impact SPL sharing.

The performance and energy results for all benchmarks are shown in Figures 3.13 and 3.14, respectively. Sharing SPL pools reduces energy consumption by up to 34% overall compared to the use of private SPL, with little or no performance degradation with the proper sharing policy. The SPL area is also reduced by more than 4X.

The private SPL configuration with only six rows is not an acceptable alternative to fabric sharing. For four of the benchmarks – *gsmtoast*, *gsmuntoast*, *adpcmDec*, and *crypt* – the performance degradation is significant relative to the 26-row private SPL configuration. Spatial sharing improves slightly upon the 6-row private SPL organization for two of these four cases, but has the limited

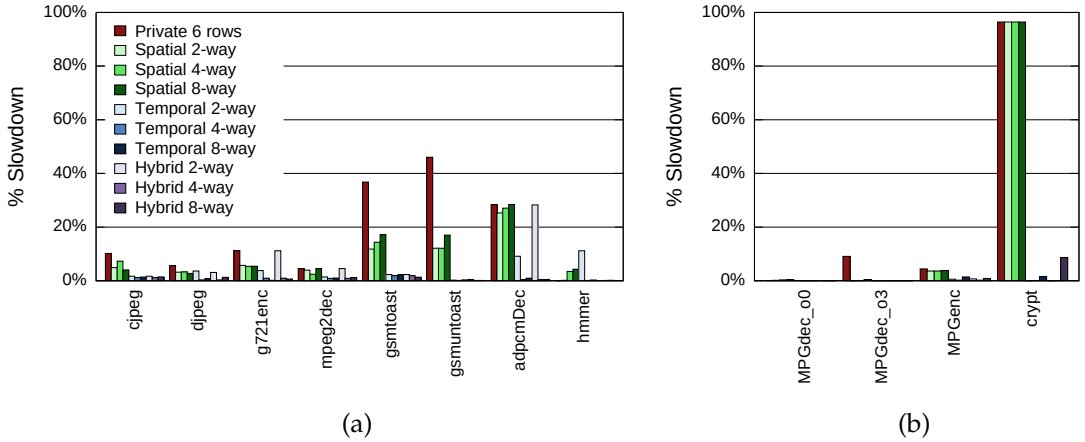



Figure 3.13: Performance for (a) coarse-grain and (b) parallel workloads relative to OOO1 + 26 row private SPL.

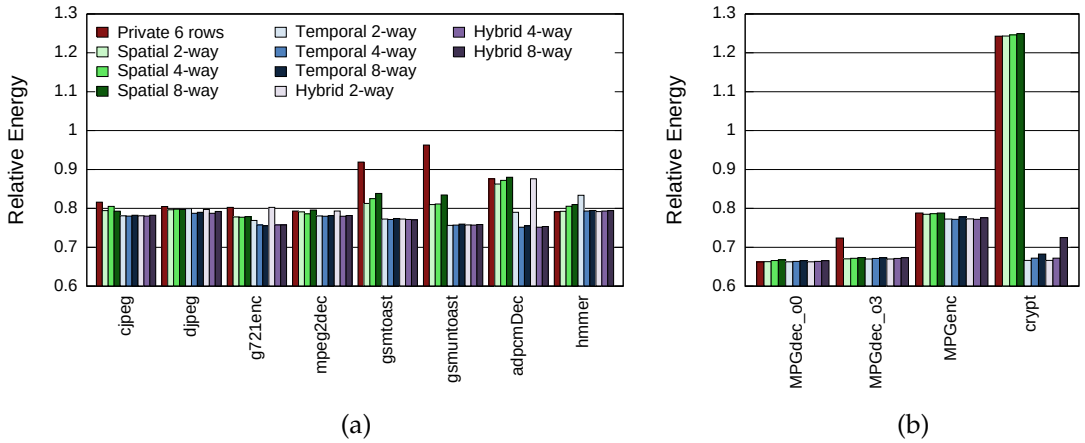



Figure 3.14: Energy consumption for (a) coarse-grain and (b) parallel workloads relative to OOO1 + 26 row private SPL.

benefit for *crypt* and *adpcmDec*. *Crypt*, for example, experiences almost a 100% performance slowdown with spatial sharing relative to 26-row private SPLs.

Temporal sharing, especially four- and eight-way sharing, outperforms spatial sharing overall for two primary reasons. First, all benchmarks but *crypt* need a maximum of 26 rows for all functions; thus, with four- and eight-way temporal sharing, virtualization is rarely required. Second, there are significant periods where the benchmarks access the SPL at a slow enough rate to

intersperse requests from different cores. With eight-way sharing, however, the performance of several benchmarks degrades due to an increase in input queue wait time due to increased SPL conflicts. This suggests that too high a degree of temporal sharing may not be desirable both due to increased wire delay to reach the SPL and due to increased contention to access the SPL.

Hybrid temporal/spatial sharing addresses the latter problem by adapting to different SPL usage phases and spatially splitting the SPL when temporal sharing induces high queue wait times. In this particular instance, an 8-way shared SPL with hybrid sharing performs 6% better than one using temporal sharing.

In the end, both four- and eight-way temporal and hybrid sharing achieve negligible performance loss, and significant energy savings, for every benchmark relative to the private SPL organization with far less area overhead. Hybrid sharing offers increased flexibility to adapt to different workloads at the cost of additional hardware to support spatial sharing and control hardware to determine when spatial sharing should be employed. With one exception, 4-way sharing provides the best performance, and as such serves as our assumed sharing degree in later sections.

Compared to replacing the SPL with additional cores, shared SPL provides better performance, 39% on average, in all but two instances, the exceptions being *djpeg* and *MPGdec-o0*, and provides equal or lower energy consumption in all cases (34% lower on average).

## 3.4 Conclusion

In this chapter, we propose a reconfigurable architecture that uses a shared fabric and control policies to reduce the costs of marrying reconfigurable logic and processor cores in future CMPs. We find that most applications do not continuously use the SPL during their execution in neither time nor space. Thus, providing each core with a private SPL is unnecessarily wasteful. With this in mind, we develop a shared SPL architecture and associated spatial and temporal control policies. Using intelligent sharing policies, our approach requires 4X less area and peak power than private SPL while achieving the same performance. Overall we find that four-way temporal or hybrid sharing provides the best compromise between decreased virtualization from larger SPL pools and increased contention due to larger degrees of sharing.

CHAPTER 4

# MANAGING MULTIPROGRAMMED WORKLOADS IN MULTIPLE SPL CLUSTERS

The last chapter showed that, while shared SPL provides significant area savings with little performance loss, the fabric should not be shared to any arbitrary degree. Since SPL sharing must be limited, and not all applications will be compiled to use the SPL, we envision multiple clusters on future large-scale CMPs, those containing only conventional processor cores and others containing a mix of processor cores and SPL (and potentially others still, although the overall mix of cluster types is not the focus of this work). Thus, at any given time of machine operation, there may potentially be many threads from sequential and parallel applications that are compiled to run on an SPL cluster. Like other shared resources in a CMP of SMT cores, where the partitioning of resources among the competing threads on a given SMT core and the co-scheduling of threads to multiple SMT cores significantly impact performance, the control of multiple multithreaded SPL clusters must be intelligently managed for good performance to be achieved.

We envision a *management layer* that dynamically optimizes the performance and power efficiency of these SPL-oriented threads in a RACM system with multiple SPL clusters. Specifically, the manager must make two interrelated decisions: (1) determine the best match of threads to clusters, considering the interplay of the different threads (including between those that frequently and infrequently virtualize the fabric [6, 34]); and (2) decide how best to employ the built-in SPL feature that permits spatially partitioning the fabric on-the-fly

to reduce contention among the threads, but at the potential cost of increased virtualization.

In this chapter, we explore a number of approaches to this complex management problem that range in sophistication from simple interval-based heuristic approaches to more advanced techniques that apply machine learning, multi-threaded phase-based optimization, and stability analysis. Our algorithms permit the use of very compact SPL fabrics that are performance competitive (on multiple mixed sequential and parallel workloads with high SPL demand – and thus many potential shared SPL conflicts) with large private SPL attached to each core, while consuming several times less die area and energy. Moreover, we show that replacing the SPL area with additional cores degrades performance by 62-143% for our workloads, demonstrating the benefit of dynamically managed clusters of shared SPL in future large-scale CMPs.

## 4.1 Large-Scale Cluster-Based CMPs

Figure 1.1(a) shows an overall depiction of an 18 core CMP with three clusters[1], with the external interface not shown for simplicity. Each of the two SPL clusters on the left hand side consists of four single issue out-of-order processor cores sharing a common pool of SPL. In the "conventional" cluster on the right hand side of Figure 1.1(a), each SPL has been replaced by one additional core, giving 10 cores in total. Applications that are not compiled to use the SPL run on this conventional cluster, while those that exploit the SPL run on one of the two left clusters. Of course, different mixes of SPL and conventional clusters (as well as

[1] Although relative sizes of the cores and SPL are accurate, this is not intended to represent an actual floorplan.

other cluster types) are possible, but this consideration is beyond the scope of this work.

Chapter 3 evaluated the use of SPL with a range of in-order and out-of-order core types and found that a simple out-of-order core coupled with SPL provided the best area-equivalent performance and power efficiency. Moreover, similar to SMT processors where adding additional contexts provides limited benefit beyond a certain point [86], sharing an SPL among four cores was shown to be the best trade-off between SPL fabric utilization and contention among competing threads. We confirm that this result holds true for our set of workloads by evaluating systems with both two 4-way and one 8-way shared SPL and find that in all cases two 4-way shared SPL clusters outperform a single 8-way shared SPL. Although we assume a 4-way shared SPL in the remainder of this work, the techniques presented apply to any degree of sharing.

Each SPL, shown in more detail in Figure 1.1(b), is a highly-pipelined row-based reprogrammable fabric that is temporally shared among four cores. Each of the two clusters incorporates hardware monitors that capture cycle-level event counts relevant to application characteristics and SPL usage. As is shown in Figure 1.1(d), the SPL Cluster Manager periodically reads the monitored information in order to assign threads to clusters, and to spatially partition each SPL as appropriate to optimize performance and power efficiency.

### 4.1.1 SPL Hardware Microarchitecture

In our SPL design (discussed in detail in Chapter 3), 16 8-bit wide reconfigurable cells are combined to form a 128-bit wide row. Rows are clocked at

Table 4.1: Relative area and power of eight single-issue out-of-order cores, eight private SPLs, and two four-way shared SPLs.

| | Rows/ SPL | Total Area | Peak Dynamic Power | Total Leakage Power |
|---|---|---|---|---|
| Eight Cores | N/A | 1.00 | 1.00 | 1.00 |
| Eight Private SPL | 12 | 0.97 | 0.29 | 1.32 |
| Two 4-way Shared SPL | 12 | 0.29 | 0.07 | 0.34 |

500MHz, one-quarter that of the processor core frequency of 2GHz (the same as the Pentium Core2 Duo [43] and the AMD X2 Dual-Core [1], both of which are implemented in the same 65 nm technology assumed for the SPL). For our benchmarks, 12 rows of private (per-core) SPL permits all but one of the configurations, the major loop within *crypt*, to achieve maximum performance[2]. Using the methodology of Section 3.1.4, we arrive at the area and power results for eight single issue out-of-order cores, eight private 12-row SPLs, and two four-way shared SPLs (as appears in Figure 1.1(a)) shown in Table 4.1. Although the area is prohibitive, we use this 12-row private SPL as the baseline against which we compare our dynamically managed shared SPL architecture. The latter is much more compact, requiring 4X less area than the private SPLs, and is much more power-efficient as well. While one might consider shrinking the private SPL, the previous chapter showed that this yields poor performance.

### 4.1.2 Temporal Sharing and Spatial Partitioning

The SPL is temporally shared in a time-multiplexed, round-robin fashion among the four cores sharing the fabric. The SPL also supports spatial partitioning, which permits private or semi-private operation by dividing the SPL into up

[2]The major loop in *crypt* requires nearly 300 rows and so achieves less than optimal speedup for any reasonably sized fabric.

to four virtual clusters. Spatial partitioning reduces contention from sharing threads, but also reduces the amount of resources available to each core, possibly leading to degraded throughput due to increased virtualization. Figure 1.1(b) shows the additional multiplexers and tristate drivers necessary to support both forms of sharing.

## 4.2 SPL Cluster Management

Having provided background on the fabric architecture, we now turn our attention to the dynamic runtime management of multiple clusters of shared, partionable fabrics. The objective of these policies is to achieve approximately the same performance of private (per-core) fabrics with the dramatically lower area and power consumption afforded by shared fabrics. When multiple sequential and parallel applications that are compiled to use the SPL are simultaneously executing, the SPL Cluster Manager (Figure 1.1(d)) optimizes overall performance through two inter-dependent mechanisms: (1) *thread assignment* among the clusters, and (2) *spatial partitioning* and *recombination* of the SPL within each cluster.

There are a number of different factors that contribute to the SPL usage characteristics of an application, including the frequency of SPL accesses and the number of rows needed by each optimized function. Experimentation revealed that the applications that are the biggest concern are those that either make frequent accesses to the SPL or require a large number of rows and therefore incur frequent virtualization. Applications that make frequent SPL accesses can be substantially impacted by poor scheduling, whereas applications with signifi-

Table 4.2: Management policy considerations.

| Consideration | Alternatives |
|---|---|
| Metrics | SPL Accesses, SPL Wait Time, Avg. Rows, Grad. Insts. |
| Spatial Partitioning | Yes, No |
| Granularity | Various fixed intervals, phase change |
| Algorithm | Split Assignment, Equalize Assignment, Hill Climbing, Hybrid |
| Stability | None, after n non-useful changes, when average degradation $<$ threshold, after n random intervals |
| Randomness | Swap SPL threads, swap any threads, swap with thread or empty core |

cant virtualization can substantially degrade the performance of other applications sharing the same cluster. We implemented and evaluated a wide range of management policies given the considerations listed in Table 4.2. We limit our discussion to four representative dynamic management algorithms.

Threads can also be statically assigned to a particular cluster on a CMP through the OS scheduler (in fact, we assume some initial static assignment for our dynamic policies). As we show later, the performance of static thread scheduling varies greatly, with slowdowns ranging from less than 1% to over 1028% compared to a 12-row private SPL. Moreover, static scheduling requires dependable *a priori* knowledge about the threads and their potential interactions. Finally, many programs go through different phases during execution and their use of the SPL can be different in each phase. Static scheduling cannot exploit this dynamic phase behavior.

### 4.2.1 Per Interval Thread Assignment Policies

We first investigated a number of policies that determine the assignment of threads to SPL clusters every interval based solely on the performance of the previous interval and make no use of the spatial partitioning capability of the SPL. From our experiments we found that, although all applications are impacted by poor scheduling choices, certain applications are impacted more than others. In particular, the largest performance losses occur when threads that require large amounts of virtualization share an SPL cluster with those that rely heavily on the SPL. Based on this insight, we explored a number of interval-based thread assignment policies, the best of which was *Average Row Assignment*. Average Row Assignment uses the average number of rows used by the functions of a particular thread as an indicator of its degree of virtualization. Functions that require a large number of rows on average will experience more virtualization, assuming the amount exceeds the number of physical rows available. Thread assignment based on the number of rows alone, however, is insufficient; the SPL access frequency should also be taken into account as an indication of how much each thread relies on the SPL. Threads that heavily utilize the SPL are more likely to be degraded by increased wait time to access the fabric.

Average Row Assignment allocates threads to clusters based on the ratio of the average number of rows used by the thread to the number of SPL accesses within the last interval. Threads with high access rates and low row usage will have small values while threads with infrequent accesses and high row usage will have high values. To assign threads to clusters we use a *split assignment* policy which aims to schedule threads with high and low values on different

clusters. The threads are sorted based on the given metric, the first $n/c$ threads are assigned to the first cluster, the next $n/c$ threads to the second, and so on, where $n$ is the number of threads and $c$ is the number of clusters.

In order to compute the decision metric, each core maintains counters to track the number of instructions issued in the last interval. The core also tracks the number of rows required by each SPL instruction. This latter information is stored in the configuration information for each SPL function and is therefore available to the SPL Cluster Manager.

### 4.2.2 Composite Thread Assignment/Partitioning Policies

Average Row Assignment only considers thread assignment. As described previously, each SPL can also be spatially partitioned. This can be useful if SPL instructions are queued for a long time due to virtualization or due to high SPL usage from the number of threads sharing the SPL. Spatial partitioning can reduce stalls due to either of these cases as it reduces the number of threads that share the same SPL partition. It must be applied intelligently, however, as it can also increase stall time due to increased virtualization.

When considering both thread assignment and spatial partitioning, the number of clusters is effectively dynamic, as each SPL can be divided in half, in quarters, or in one half and two quarters, and thread assignment must account for this cluster size variability. As before, per-core metrics are gathered to determine how to assign threads to however many clusters currently exist. Additionally, the SPL Cluster Manager must further determine when to split and merge SPL partitions in each cluster.

To determine thread-to-cluster assignments, this policy, henceforth referred to as Composite, uses the same SPL access to average row ratio with split assignment used by the Average Row scheduler. To determine when to split an SPL cluster, each core tracks the number of cycles an SPL instruction is stalled in the SPL queue and the number of its SPL instructions that are issued. If any thread spends too long on average waiting to issue an SPL instruction, i.e., the average wait time exceeds a threshold, then the SPL is split. Similarly, to determine when to merge, each cluster tracks the number of threads whose average wait time is less than a second threshold. If the sum of this value for the two clusters is greater than the current number of cores sharing a single cluster, then the two clusters are merged. Neither a split nor a merge will occur if both split and merge requests are received for the same cluster in a given interval.

### 4.2.3 Learning-Inspired SPL Cluster Management

The policies discussed thus far create their mappings of threads to clusters based solely on the relative ranking of some statistics for each thread during the last interval. Although this generally leads to good mappings, it may not produce the best possible mapping. Moreover, these policies make no attempt to learn from their previous actions. In an attempt to be more intelligent and achieve the best – or at least a better – mapping, we apply machine learning techniques to our cluster mapping problem.

### Hill Climbing

Since we desire a fast, purely online approach, we focus on hill climbing. Previous work by Choi and Yeung [13] investigated hill climbing for SMT resource allocation among concurrently running threads. Our scheduling problem is significantly different, and arguably harder, as we have to deal with both resource partitioning and determining which threads should share those partitions.

We investigate a number of variations on a stochastic hill climbing manager. At each scheduling interval the manager may perform one of the following actions to create a new assignment: (a) swap two threads from different SPL clusters; (b) split or merge an SPL cluster that is not already at its minimum or maximum size; (c) create a random number of partitions as well as a random mapping of threads to those partitions. The last option adds an element of stochasticity which aims to escape local minima. Each of these options is selected with a predefined probability. We investigate a variety of different restrictions on thread swapping, from allowing only threads using the SPL to be swapped, to allowing any threads to be swapped, to allowing a single thread to be swapped into a cluster with an unused core.

The new assignment is run for the next interval. At the end of the interval, the performance of that interval is compared to the performance of the best interval to date for the current phase. If the new mapping achieves better performance, then it is set as the new best mapping; otherwise, the mapping reverts to the previous best mapping. In either case, a new local search step is applied to the current best mapping. After some number of consecutive unbeneficial steps, the best schedule is assumed to have been found and the phase is declared stable. All future intervals in this phase use this stable mapping.

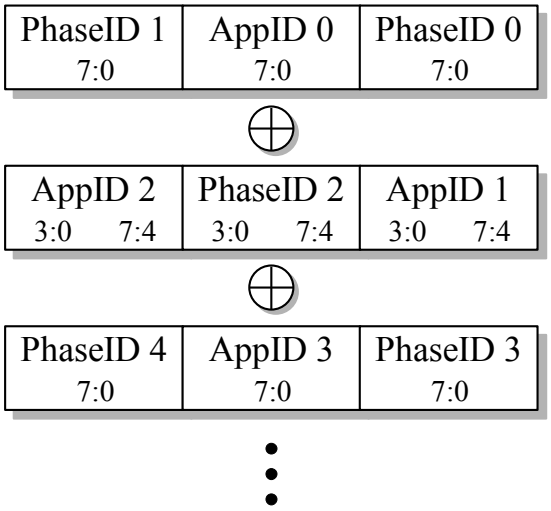



Figure 4.1: Hash function for phase IDs.

To identify phases, we develop a multithreaded/multiprogrammed workload phase tracker. We use the phase tracker of Sherwood et al. [73] to identify phases for each thread. The phase tracker reports the current phase for each running thread based on the mix of instructions executed during the last phase interval. This phase information is combined to index into a global management history table, which contains the best mapping executed so far for the given set of phases. In order to create a reasonably sized index to access the history table, we developed a hash function to map the application and phase IDs of all currently running threads to a reasonable number of bits. This function takes three byte groups of phase and application IDs (where each phase or application ID is one byte) and XORs them together as shown in Figure 4.1. The IDs are ordered by application ID. The IDs within every other group are rotated by four bytes to increase diversity. This hashing scheme produces less than a 3% average false match rate for our workloads, and more importantly, degrades performance over a perfect hashing scheme by less than 0.1%.

To determine the relative performance of different mappings, the manager tracks the peak number of instructions graduated by each thread on a per phase basis and calculates the performance degradation for the current phase relative to this peak performance. The performance degradation of all threads are averaged to produce the overall degradation for the interval. The peak instruction count is determined by averaging the five highest observed graduation rates in that phase.

As will be shown in Section 4.4, our best hill climbing algorithm is able to match the performance of the Composite management scheme, but rarely exceed it. This is due to the large performance degradation that can occur during some of the exploration intervals, and so any slight improvement in mapping over that found by the Composite algorithm is offset by the degradation incurred during the exploration period. Unlike typical pipeline resource allocation, small changes in the thread to SPL assignment can substantially change performance. This effect not only makes finding the optimal mapping difficult, it also significantly degrades throughput (by up to an order of magnitude) during intervals with poor mappings. If too many of these poor mappings are explored, performance degrades severely. Due to these large jumps, the exploration space is not necessarily nicely hill shaped; it is not only quite "bumpy" but there are likely to be numerous local minima that may be difficult to escape.

#### Hybrid Heuristic-Hill Climbing (H3C) Manager

Based on our experience with the previous management schemes, and additional experimentation, we devised a hybrid approach that addresses three main sources of performance degradation of the prior approaches. The first two

sources are present in the heuristic techniques and the last appears with Hill Climbing. First, most programs experience different phases in their execution, during which their use of the shared SPL may vary significantly. As such, the best mapping for one phase may be suboptimal for another, and reaching a new stable mapping may take multiple intervals using the aforementioned interval-based policies. Second, even within a phase there can be a small amount of variability in the performance of a thread. This variation can lead to a ping-ponging effect where threads are constantly being swapped between two clusters. This is especially true for multithreaded workloads where multiple threads can be performing the same task and performance can vary slightly depending on interactions with memory and other threads. This constant swapping can degrade performance due to the overhead for context switching threads. Finally, as mentioned previously, excessive exploration of the assignment space can degrade performance due to the significant performance degradation experienced in certain assignments. To address these issues, we devise a new algorithm that we call *Hybrid Heuristic-Hill Climbing (H3C)* that combines elements of the previous two approaches and incorporates a stability threshold such that further changes are not made if the performance is within some margin of "optimal."

H3C evaluates performance and maintains current assignments using the same phase-based approach as Hill Climbing. Unlike Hill Climbing, no change is made to the assignment for the next interval if the previous interval is determined to be stable. An interval is considered stable if the average performance degradation (as indicated by the graduation rate relative to peak, same as Hill Climbing) for all threads is less than some threshold.

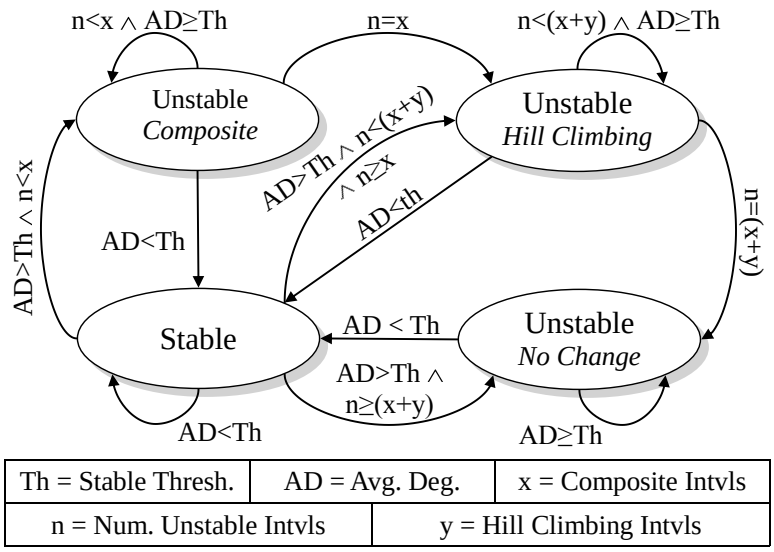



Figure 4.2: H3C Cluster Manager.

When not stable, the assignment for the next interval is determined by one of two methods. During the first $x$ intervals of a particular multithreaded/multiprogrammed phase the threads are assigned using the Composite algorithm from Section 4.2.2. The goal of this step is to create a generally good mapping that can be fine tuned in the next step. During the next $y$ intervals a learning-based local search like that described in Section 4.2.3 is used to try to improve upon the mapping produced by the Composite algorithm. After this step it is assumed that the "best" mapping has been found and no further exploration is performed for this phase, even if the average degradation is not less than the stable threshold in some future intervals. The complete set of H3C state transitions are shown in Figure 4.2. At each interval, the management table keeps track of the best mapping found so far and H3C reverts back to that mapping as a starting point for the next management interval if the previous interval did not improve upon the performance.

Table 4.3: Architecture parameters.

| | |
|---|---|
| Branch Predictor | gshare + bimodal |
| BTB Size | 512B |
| RAS Entries | 32 |
| Fetch/Decode/Rename Width | 2 |
| Issue/Retire Width | 1 |
| Integer Registers | 64 |
| FP Registers | 64 |
| Integer Queue Entries | 32 |
| FP Queue Entries | 16 |
| ROB Entries | 64 |
| Int ALUs | 1 |
| Branch Units | 1 |
| Int Mult/Div Units | 1 |
| FP ALU/Mult/Div Units | 1 |
| LD/ST Units | 1 |
| L1 Inst Cache | 8kB 2-way, 2-cycle access |
| L1 Data Cache | 8kB 2-way, 2-cycle access |
| L2 Cache | 1MB per core, 10-cycle access |
| Coherence Protocol | MESI |
| Main Memory Access Time (ns) | 100 |
| Phase History Entries | 256 |

## 4.3 Evaluation Methodology

We use a highly modified version of SESC [71] to evaluate our proposed shared SPL cluster management policies. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. The major architectural parameters are shown in Table 4.3. We use Wattch and Cacti to model dynamic power and Cacti and HotLeakage to model leakage power.

To model the overhead associated with performing thread management, instruction fetch for all cores is stopped for 1000 cycles at the end of each interval. This value was determined by executing code approximating the scheduling algorithms on our simulator to get an accurate cycle estimate. After this period, the instructions for any threads being migrated are drained and execution is

Table 4.4: Parameters for dynamic management policies.

| Scheduling Policy | Parameter | Value |
|---|---|---|
| Composite | Split threshold | 16×avg. rows used by core |
| Composite | Merge threshold | 2×avg. rows used by core |
| Hill Climbing | Probability of splitting SPL | 20% |
| Hill Climbing | Probability of merging SPL | 20% |
| Hill Climbing | Probability of random mapping | 10% |
| Hill Climbing | Threads to consider swapping | All |
| H3C | Intervals of Composite Scheduling | 5 |
| H3C | Intervals of Hill Climbing | 5 |
| H3C | Stability threshold | Avg. Deg. $< 4\%$ |
| All | Interval granularity | 100k cycles |

stopped for an additional 500 cycles (again determined by running the requisite code in the simulator) to model the time necessary to context switch all state – including internal SPL state – to the new core. Finally, the threads are started on their new cores, where warm-up of caches and TLBs is modeled. The processor undergoes a similar sequence when the SPL is spatially split or merged by the manager, although in this case the context switch and cache and TLB warm-up are not needed as threads continue to execute on the same core.

We investigated a number of management interval granularities ranging from 100k up to 20M cycles. We report results for the finest granularity of 100k cycles and discuss the effect of coarser intervals in Section 4.4.2.

We experimentally determined the best parameters for the dynamic policies, which are shown in Table 4.4.

### 4.3.1 Phase Tracking

We use the same parameters for our phase tracker as Sherwood et al. [73] with the exception of the phase interval length. We use a smaller 1 million instruction

interval due to the shorter phases of some of our applications. Given these parameters, we estimate that the tracker would require less than 1 kB of storage per core. Actual phase changes in the program as detected by the phase tracker may not exactly coincide with management interval boundaries as management intervals are based on *cycles* whereas phase tracking is based on *instructions*.

The global SPL Cluster Management history table contains 256 entries. Each entry contains the phase IDs for each thread, the mapping of threads to clusters, the size of each cluster, and the best performance seen to date for this phase. We estimate this table would require 4 kB worth of storage.

### 4.3.2 Benchmarks

We create four workload mixes to evaluate the performance and power efficiency of our approach. Each workload consists of a combination of parallel and sequential benchmarks. These mixes reflect the type of workloads systems are apt to see in the future as different applications are likely to be parallelized to different degrees. We choose three single threaded benchmarks from SPEC2006 [78], one from SPEC2000 [77], and one from MediaBench [51]. Our multithreaded workloads consist of two benchmarks from ALPBench [52] and a version of the JavaGrande [75] *crypt* benchmark ported to C++. We run the ALPBench version of *MPGdec* with two different command line parameters (-o0 and -o3) as they produce different execution characteristics. Specifically, the o3 version enables additional processing within the application which makes use of the SPL, leading to increased overall SPL usage. A complete list of the bench-

Table 4.5: Benchmark description, number of SPL functions, maximum number of rows used by SPL functions, percentage of execution time of SPL optimized regions, percentage of SPL instructions executed relative to total committed instructions, and percentage of time with at least one SPL instruction in flight. For MPGdec, o0 or o3 indicates which command line parameter is used in the run.

| | SPL Functions | Max Rows | % Exec Time Optimized | % Dyn. SPL Insts | SPL Usage |
|---|---|---|---|---|---|
| 300.twolf | 1 | 21 | 32.7% | 0.10% | 3.8% |
| 456.hmmer | 1 | 10 | 99.5% | 1.15% | 40.2% |
| 462.libquantum | 1 | 11 | 40.1% | 2.19% | 13.5% |
| 473.astar | 1 | 2 | 33.7% | 0.79% | 2.7% |
| cjpeg | 5 | 21 | 49.9% | 1.22% | 20.6% |
| MPGenc | 4 | 16 | 69.1% | 0.72% | 17.2% |
| MPGdec-o0 | 5 | 20 | 44.8% | 0.35% | 15.3% |
| MPGdec-o3 | 12 | 20 | 47.8% | 0.57% | 19.3% |
| crypt | 1 | 298 | 97.9% | 4.48% | 99.9% |

Table 4.6: Workload composition. For parallel workloads the number after the name indicates the number of threads spawned during the run.

| Name | Benchmarks |
|---|---|
| Mix A | MPGenc-2, MPGdec-o0-2, crypt-2, 456.hmmer, 473.astar |
| Mix B | MPGdec-o3-2, crypt-2, 456.hmmer, 300.twolf, 473.astar, 462.libquantum |
| Mix C | MPGdec-o3-4, crypt-2, 300.twolf, 462.libquantum |
| Mix D | MPGenc-4, crypt-2, cjpeg, 462.libquantum |

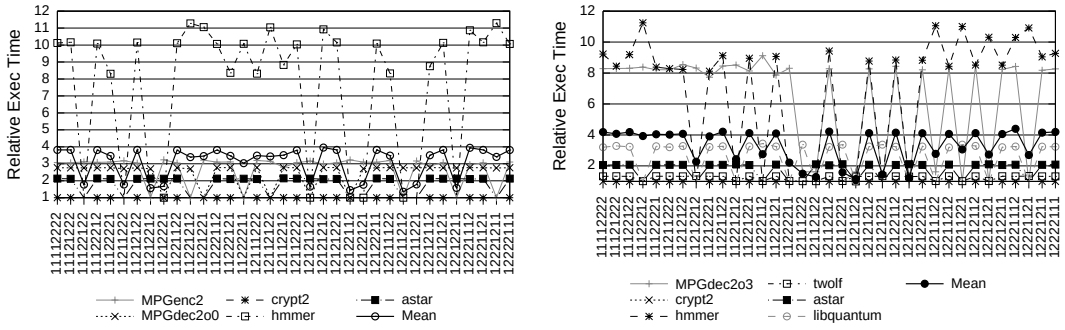
marks as well as the SPL usage characteristics of each can be found in Table 4.5. Table 4.6 lists the benchmarks in each workload mix.

In order to create SPL mappings, we profile each benchmark to determine which functions consume the largest portion of total execution time. Following this, we examine each of the functions in order to determine which ones can be efficiently mapped to our SPL fabric. Mappings, which include the number of rows needed, input values to be loaded, and results to be stored, are then created for those functions. Mappings are done by hand, although previous work has shown that compilers can produce good mappings for reconfigurable architectures [4, 7, 35, 53, 94].

Since dynamic thread scheduling is most useful when applications experience phase changes, we need to make sure we run our benchmarks long enough to witness these phase changes. The best option is to run benchmarks to completion. Due to the long running time of SPEC benchmarks with reference inputs, however, we are only able to run our non-SPEC benchmarks to completion. For our SPEC benchmarks we use Early SimPoints [58] to select two 250 million instruction SimPoints from the original source code (i.e., code not utilizing the SPL). Since using the SPL changes the number of instructions executed, we determine where each of the two SimPoints begin and end and augment the code to fast forward through all but these two intervals. In this way both the original and SPL versions of the code will execute functionally equivalent amounts of code. We select relatively long intervals to capture phase changes within an interval. In order to make our comparison fair, we continuously respawn threads that finish early so that longer running threads still experience contention for the SPL due to the shorter running threads. We stop the simulation when the longest running benchmark completes and report the execution time for each benchmark averaged over all completed runs.

## 4.4 Results

We first motivate the need for dynamic thread assignment and spatial partitioning by showing the varied performance achieved with static thread assignment relative to large private SPLs. We then compare dynamic management to the best, worst, and median-case static assignments. We also compare our approach with the performance and energy$\times$delay$^2$ that would be ideally achieved by replacing the SPL with additional cores.



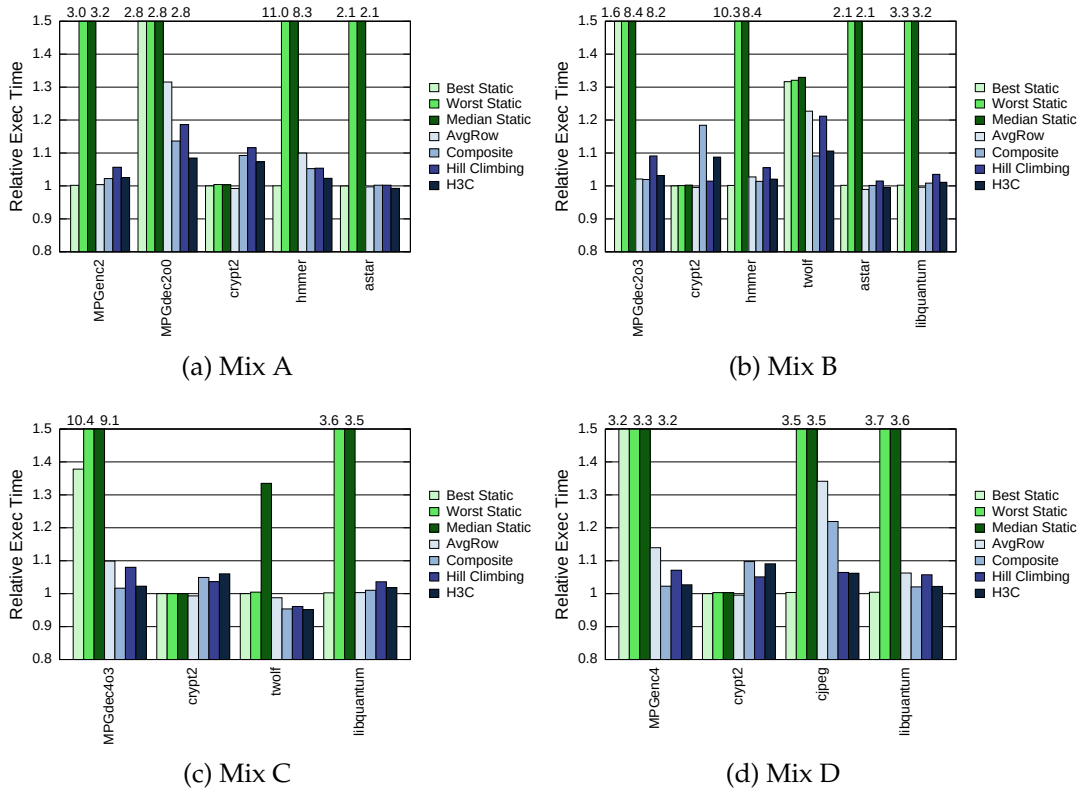

(a) Mix A (b) Mix B

Figure 4.3: Performance of (a) Mix A and (b) Mix B for shared SPL with all possible static schedules relative to private 12-row SPL.

### 4.4.1 Static Assignment Performance

The OS scheduler could assign threads to clusters statically (i.e., maintain the schedule throughout execution) using information regarding expected SPL usage gleaned from the compiler. For each workload, we simulate all 35 possible static assignments, and extract the best, worst, and median static assignments based on the mean of the relative execution time of all benchmarks. This information tells us what an oracle static scheduler could achieve, the worst performance that could occur if SPL usage is not taken into account by the scheduler at all, and the margin for error, i.e., whether most schedules are closer to the best or the worst schedule.

Figure 4.3 shows the performance of each benchmark for two workloads for all static assignments (results for the other two workloads show similar overall trends). The labels on the x-axis indicate the cluster, 1 or 2, to which each thread is assigned. A label of 12112212, for example, indicates the first spawned thread is assigned to cluster 1, the second thread to cluster 2, the third thread to cluster 1, etc.




(a) Mix A (b) Mix B

(c) Mix C (d) Mix D

Figure 4.4: Performance of shared SPL with dynamic scheduling algorithms and best, worst, and median static schedules relative to private 12-row SPL.

Performance with static assignment is highly variable, varying by as much as 1028% between the best and worst static schedules for some benchmarks. The best, worst, and median static schedule performance for each benchmark in each workload relative to the 12-row private SPL baseline is shown in Figure 4.4 (first, second, and third bars)[3]. The mean performance for the four workloads is shown in Figure 4.5. While in some cases the best (oracle) static scheduler performs reasonably well, the results for the worst schedule indicate that a static scheduler that is oblivious to SPL usage may perform poorly relative to the private 12-row SPL organization. Moreover, Figure 4.5 shows that the median schedule is much closer to the worst case schedule than the best case schedule.

[3]Note that the best static schedule is based on mean performance, and therefore might not be best for an individual benchmark.

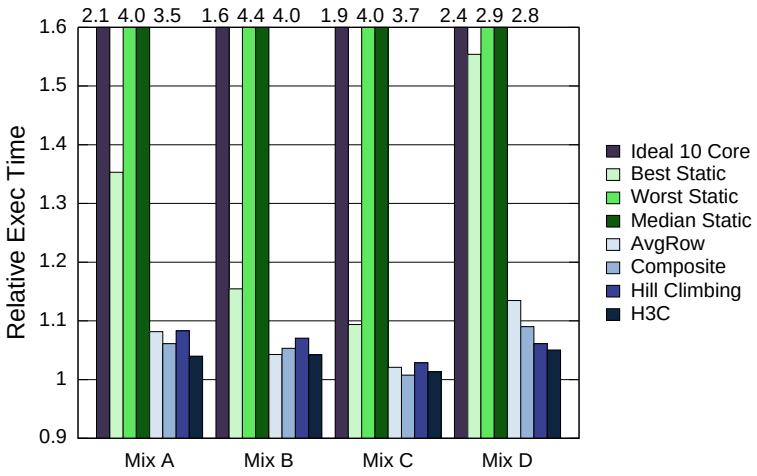



Figure 4.5: Average execution time for each workload relative to 12-row private SPL.

Thus, there is little margin for error in static scheduling; such errors could easily arise due to the lack of static information regarding the fabric contention among applications.

### 4.4.2 Performance of Dynamic SPL Cluster Management

The individual benchmark and average workload performance of the four representative management policies presented in Section 4.2 relative to the performance of private 12-row SPL is shown in Figures 4.4 and 4.5, respectively.

#### Average Row and Composite Policies

Overall, the Average Row and Composite policies outperform the best possible static assignment by 21.9% and 23.6%, respectively. The benefits of permitting the manager to control spatial partitioning as done in the Composite policy are demonstrated in the overall results for both policies (Figure 4.5). Compared

with the much higher overhead private 12-row SPL organization, the Average Row approach experiences a 7.0% slowdown and the Composite policy experiences a 5.3% slowdown on average.

For a few of the benchmarks, the dynamic algorithms occasionally improve performance relative to the 12-row private SPL. This occurs because we simulate private L2 caches. When scheduled on several different cores, threads may make use of multiple L2 caches. Thus, on an L2 cache miss, the data might be sourced from another L2 cache rather than the slower main memory. To ensure that this effect is not the primary reason for the improvement of our policies, we run tests where all L2 misses are forced to access main memory. We find that the cache "sharing" effect on performance is negligible in comparison to the effect of the Cluster Manager.

**H3C Policy**

As mentioned previously, and shown in the results, the Hill Climbing manager typically does not outperform the simpler Composite approach due to the performance loss incurred during exploration.

The H3C manager achieves the best all around performance, outperforming all other options in all but one case. In the one exception, Mix C under the Composite manager, the performance with H3C is less than 1% worse than the Composite scheduler. Overall the H3C policy achieves 25.3% better performance than the best static schedule. Compared to the 12-row private SPL, the H3C management approach experiences only a 3.6% slowdown while consuming 4X less area.

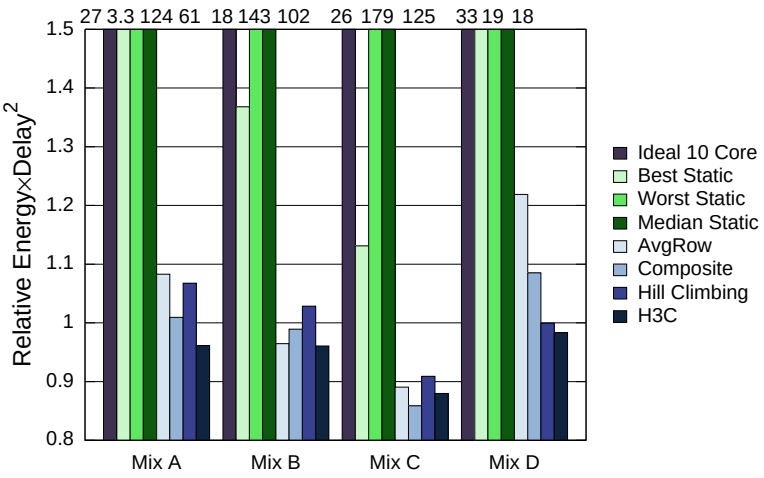



Figure 4.6: Average energy×delay$^2$ for each workload relative to 12-row private SPL.

When energy×delay$^2$ (ED$^2$) is considered (Figure 4.6), the benefits of shared SPLs incorporating both scheduling and spatial partitioning are further accentuated. The H3C manager achieves 5.4% better ED$^2$ than the 12-row private baseline on average (again with a 4X lower area cost due to the shared fabrics). By contrast, the best static schedule experiences an average 179% *worse* ED$^2$ than the 12-row private SPL. H3C is the only approach that provides better ED$^2$ than the 12-row baseline for all four workloads.

Another benefit of the dynamic policies is fairness. For most of the best static schedules, some of the threads achieve near optimal performance while others experience significant slowdown. With the dynamic policies, the performance impact is quite uniform across the threads.

**H3C Component Analysis** In Section 4.2.3 we detailed a number of factors that limit the performance of the Composite and Hill Climbing managers and how the H3C manager incorporates techniques to address these issues. To assess the importance of each, and also confirm that the proposed solutions

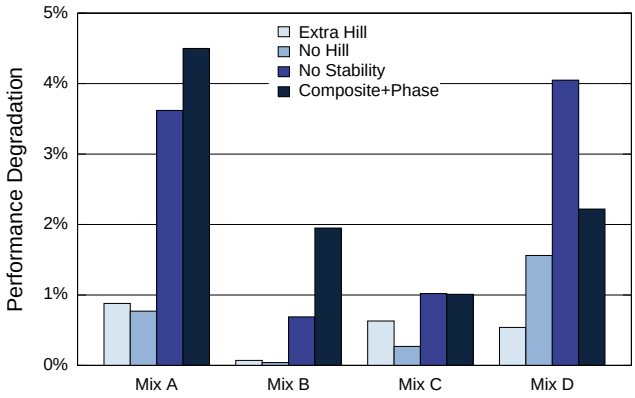



Figure 4.7: Performance degradation relative to H3C.

achieve their stated goals, we run simulations where one or more of the features of the H3C manager are modified. In particular we look at cases where the stability threshold is eliminated (No Stability), where hill climbing is eliminated (No Hill), and where additional hill climbing is performed (Extra Hill). We also look at a case where Composite scheduling is performed at every interval (essentially adding phase information to the base Composite manager) (Composite+Phase).

Figure 4.7 shows the performance loss of each case relative to the H3C manager. The H3C manager outperforms all of these alternatives in every instance. This confirms that hill climbing, stability detection, and phase analysis are all important and that eliminating any one of them degrades the quality of the manager. The most important of the factors is the stability threshold, without which performance loss increases by 2.3% on average. Continuously running the Composite scheduler with phase information leads to similar losses. Both indicate the benefits of limiting unnecessary exploration.

**Phase Analysis** One of the key features of our management schemes is their ability to dynamically adapt to different application phases. Figure 4.8 shows

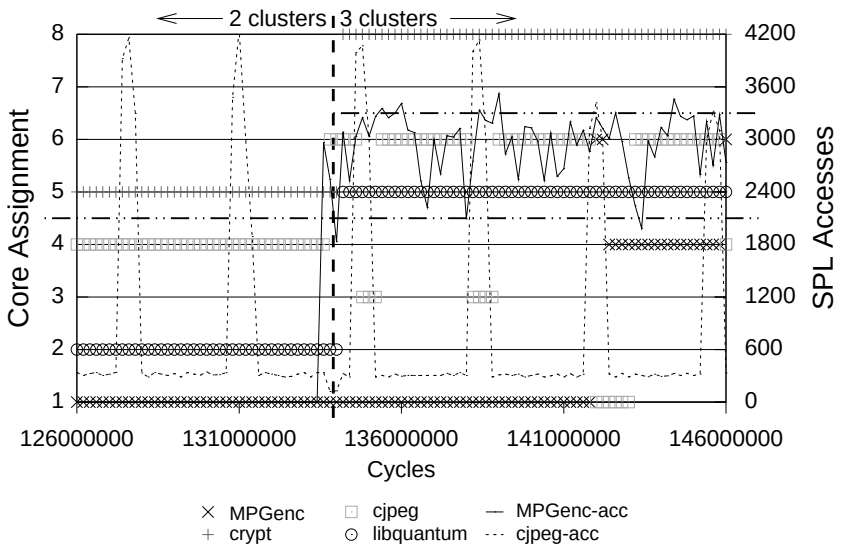



Figure 4.8: Thread-to-core assignment and SPL accesses for Mix D with H3C.

an example of the thread scheduling and cluster partitioning for a section of Mix D with H3C management. The graph shows the thread-to-core assignment for the four main threads along with the SPL access patterns of the two threads that change phases during the given period. The horizontal dotted lines in the graph show which cores share a SPL partition and the vertical line indicates when one of the clusters is partitioned.

At the start of the example, four threads are actively using the SPL. The two *crypt* threads share one cluster and the two single threaded applications share the other in order to minimize conflicts. Around 134M cycles, *MPGenc* starts a section that uses the SPL. The H3C manager monitors SPL usage and determines how to schedule threads and partition the fabric to adapt to this change. In particular, one of the clusters is divided so that *crypt* still has its own partition, and the assignment of threads to clusters is rearranged based on current usage statistics.

The figure also shows how the manager can adapt to the changing access pattern of *cjpeg*. During phases when its access rate increases, *cjpeg* is rescheduled on the larger cluster to achieve better performance. Unlike the Composite manager, however, which always makes the same change, we can see in the last two access peaks for *cjpeg* that the H3C manager explores other options in an attempt to find an even better mapping.

**Replacing the SPL with Additional Cores**

Figures 4.5 and 4.6 also show results for each workload in which each shared SPL is replaced by one additional single issue core; in other words, the workloads are run on the conventional cluster on the right side of the chip diagram of Figure 1.1(a). These results were obtained by simulating a given workload using the original benchmarks (no SPL code) with eight cores (one per thread), and then ideally scaling the results to 10 cores. This ideal scaling is achieved by linearly reducing the execution time by 1/5, but optimistically increasing the energy by only 12.5% (even though there are now 25% more cores).

A comparison of the Ideal 10 Core and all of the dynamic scheduling results in these graphs substantiates previous work that demonstrated significant benefits with SPL on particular applications. Performance degrades by 62-143% when the workloads are run on the 10-core cluster rather than the two with shared SPL, and the $ED^2$ differences are even more pronounced: up to 34X worse $ED^2$ for Ideal 10 Core. We emphasize again that on a large-scale CMP, those applications that are not compiled to use the SPL can be scheduled on a non-SPL cluster. For those threads that are compiled to use the SPL, effective assignment of threads to clusters, coupled with judicious dynamic spatial parti-

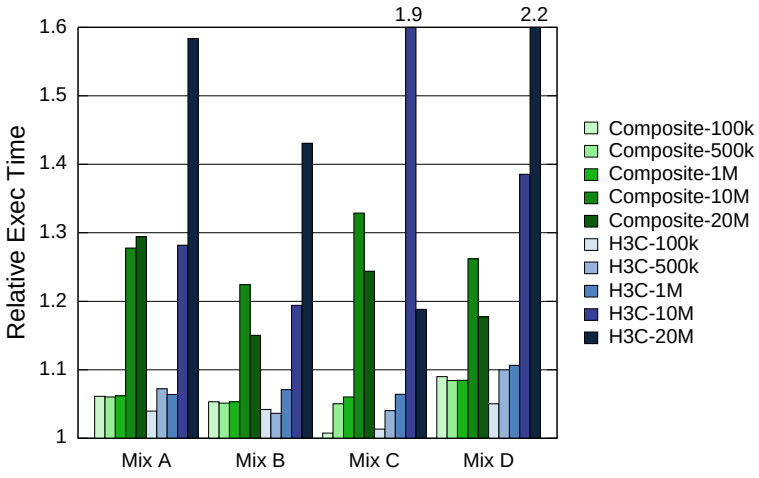



Figure 4.9: Average performance of Composite and H3C scheduling algorithms with different scheduling intervals for all workloads relative to private 12-row SPL.

tioning of the SPLs, is crucial to achieving good performance, power efficiency, and area efficiency with integrated shared SPL on large-scale CMPs.

### Coarser-grain Dynamic Management

Our results thus far assume that SPL management occurs at a 100k cycle granularity. We also investigated larger management granularities, which might be more amenable to implementation at the OS level. Figure 4.9 shows the average execution time for the Composite and H3C policies for intervals of 100k, 500k, 1M, 10M, and 20M cycles. As would be expected, the relative execution time tends to increase as the management granularity increases. The degradation with the 500k and 1M cycle intervals is relatively small, but increases substantially for the longer 10M and 20M cycle intervals. With longer intervals the SPL manager cannot respond as quickly to application phase changes. This can lead to spending more time in suboptimal cluster mappings, which can lead to significant performance loss. The H3C policy can be further im-

pacted by longer intervals do to its hill climbing phase. While performing hill climbing the manager makes random changes to the schedule in the pursuit of better performance. These random changes can sometimes yield a significantly worse mapping. The longer the management interval, the longer the applications spend in this bad mapping, and the more performance degrades. For the longest intervals, the simpler Composite scheme actually performs better on average because it avoids these bad mappings.

In the end, for a 20 million cycle (10 ms) interval we find that the Composite policy experiences a 21.6% slowdown relative to the large private SPL baseline. While this is still significantly smaller than the 252% degradation of the median static schedule, it is notably larger than the 3.6% degradation produced by H3C with a 100k cycle interval.

## 4.5 Conclusion

In this chapter, we propose dynamic SPL cluster management policies for future large-scale CMPs with integrated reconfigurable fabrics. We examine a range of policies that vary in their approach to mapping threads to clusters, as well as how they exploit the ability to spatially partition each SPL to mitigate inter-thread conflicts.

Of the four representative approaches that we present, our best policy judiciously combines elements of machine learning, phase-based analysis, and stability detection to assign threads to clusters and spatially partition the SPLs on-the-fly. This approach outperforms an oracle static scheduler, and experiences only a small slowdown compared with much larger private SPLs dedicated to

each core. We also show dramatic improvements over allocating the SPL area to additional cores. Overall, we demonstrate that sharing reconfigurable fabrics and managing their resources on-the-fly are key to making reconfigurable fabrics an attractive, cost-effective, option for future CMPs.

CHAPTER 5

# FINE-GRAINED COMMUNICATION IN A HETEROGENEOUS CMP

As already discussed, incorporating reconfigurable fabrics into commodity CMPs is more cost-effective than past monolithic proposals. This is especially true in light of the fact that future large-scale CMPs are likely to be heterogeneous in nature, with different areas of the die dedicated to accelerating particular types of applications. The amount of die area dedicated to reconfigurable fabrics may be sized in proportion to the expected proportion of applications that will benefit. As the industry moves to more cores on a die, the proportional cost of incorporating a reconfigurable fabric decreases, as does the proportion of applications needed to justify the presence of the fabric. Cost-effective integration is further enhanced by sharing the fabric among multiple cores, amortizing the area of power costs of the fabric and forming a cluster of cores+fabric. With the intelligent fabric management policies previously described, such sharing can increase fabric utilization and reduce overall fabric area and power costs, while achieving nearly the same performance as providing each core with its own, much larger, private fabric.

Sharing the reconfigurable fabric among multiple cores also creates optimization opportunities not possible with per-core private fabrics. In particular, shared fabric clusters – in addition to amortizing the fabric area and increasing power efficiency – can be organized on-the-fly in multiple ways to accelerate both multithreaded applications and the sequential applications that have traditionally been the focus of reconfigurable architectures. Figure 5.1 depicts a portion of a heterogeneous CMP and provides a simplified view of the three ways that the RACM architecture is dynamically organized to accelerate multi-

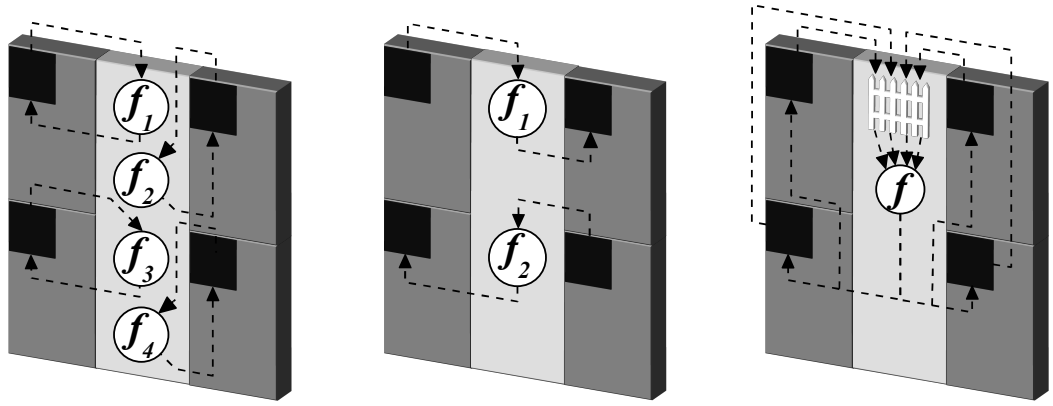



Figure 5.1: Shared SPL being used for (a) individual computation, (b) producer-consumer communication with computation, and (c) barrier synchronization with computation.

ple threads. Figure 5.1(a) depicts four threads, each of which is independently performing a function (each function $f_i$ may be unique or identical) within the fabric without communication. In Figure 5.1(b), the fabric is being used for two instances of fine-grain producer-consumer communication with integrated customized computation. In each instance, the producer thread feeds inputs into the fabric; the inputs pass through the fabric to perform the function; and the function output, which may be queued using any remaining fabric resources if necessary, is then passed to the consumer thread. Finally, Figure 5.1(c) depicts four threads synchronizing at a barrier within the fabric with a global function, e.g., a global minimum, computed in the fabric after the synchronization point. Synchronization for more than four threads is supported through an inter-cluster network.

While all of these cases show the individual threads or producer-consumer thread pairs temporally sharing the fabric, if necessary due to high fabric contention, the fabric manager can dynamically partition the fabric, giving each

thread a smaller private fabric[1]. Dynamic fabric partitioning can also serve to simultaneously configure the fabric for multiple purposes, e.g., for independent use by one set of threads in one partition, and producer-consumer communication in another.

Unlike previous proposals [3, 9, 36, 62, 67], RACM supports multiple communication models and also provides the ability to perform customized computation on communicated data. The latter provides performance improvements beyond what is possible with previous communication-only options and traditional fixed computation alternatives.

## 5.1 RACM Architecture

As described previously, RACM pairs a specially designed Specialized Programmable Logic (SPL) fabric with multiple cores of a CMP. An example RACM heterogeneous CMP with integrated SPL is depicted in Figure 1.1(a). The figure shows a 20 core RACM CMP with two SPL clusters on the left. Each cluster consists of four single issue out-of-order processor cores sharing a SPL fabric, which is shown at a high level in Figure 1.1(b) and explained in more detail in the next section. The fabric is temporally shared in a round-robin fashion among the cores in the same cluster and can be spatially partitioned as needed to reduce contention among the threads. Contention is further reduced by limiting the degree of fabric sharing, which also limits the maximum wire delay. In this particular example, the proportion of applications that benefit from the fabric is such that two shared fabric clusters are implemented. In a large-scale heterogeneous CMP with many tens or hundreds of cores, there may be several

[1] As we explained in Section 3.1.5, virtualization of the fabric makes this dynamic division of the fabric transparent to software.

Table 5.1: Relative area and power of four single-issue out-of-order cores and four-way shared RACM fabric.

| | SPL Rows | Total Area | Peak Dyn. Power | Total Leak. Power |
|---|---|---|---|---|
| Four Cores | N/A | 1.00 | 1.00 | 1.00 |
| 4-way Shared SPL | 24 | 0.51 | 0.14 | 0.67 |

SPL clusters as well as many other different cluster types, such as the traditional many-core cluster shown on the right hand side of Figure 1.1(a), on the die. Applications are mapped to a SPL cluster during phases that use the fabric and are mapped to other clusters during other phases in order to obtain the best overall performance.

### 5.1.1 SPL Organization

The computational substrate of RACM is the highly-pipelined, row-based SPL described in Section 3.1. The SPL is composed of 24 rows, in which each row contains 16 cells and each cell computes 8 bits of data. The SPL is clocked at a fixed 500 MHz. This is one-quarter the 2 GHz core frequency (the same as the Pentium Core2 Duo [43] and the AMD X2 Dual-Core [1], both of which are implemented in the same 65nm technology assumed for RACM) and allows each row to complete the longest permissible computation in a single cycle. Table 5.1 shows the relative area and power consumption of the SPL and associated single-issue cores.

The SPL is integrated with the processor core as a reconfigurable functional unit and interfaces to the memory system via a queue-based decoupled architecture as shown in Figure 1.1(b). Special SPL load instructions place values into the input queue at a particular data alignment. For output, the SPL simi-

larly writes to a local output queue that is then written out to the Store Queue using special SPL store instructions.

The SPL is temporally shared in a time-multiplexed, round-robin fashion among the cores sharing the fabric. The SPL also supports spatial partitioning where the fabric is divided into up to four virtual clusters. Spatial partitioning reduces contention from sharing threads, but also reduces the amount of resources available to each core, possibly leading to degraded throughput due to increased virtualization. Figure 1.1(b) shows the additional multiplexers and tristate drivers necessary to support both forms of sharing. Figure 1.1 also shows the barrier and thread-to-core tables, input queue valid bits, and row destination IDs added to support interthread and barrier communication, which are discussed in the upcoming sections.

### 5.1.2 RACM Communication

Most multithreaded applications use some form of communication to coordinate their activity. At a high level, communication requires the exchange of information between threads, be it a notification that a thread has arrived at a barrier, a notification that a thread is acquiring or releasing a lock, or a producing thread passing results to a consuming thread. We design RACM to facilitate efficient communication among those threads sharing the fabric, focusing on fine-grained interthread communication and fine-grained barrier synchronization.

### Fine-Grained Interthread Communication+Computation

Fine-grained interthread communication enables threads to communicate with each other much more frequently than would be possible using the traditional memory system. Such fine-grained communication is typically targeted at pipelined/streaming applications [9, 62]. To perform this type of communication, a queue, either in software or hardware, is established between the two communicating threads. The producing thread places data into the queue and the consuming thread reads data from the queue. Unless the queue is full/empty, the two threads can continue to produce/consume data without concern for how the other thread is progressing.

Since the SPL is shared between multiple cores, sending data to a different core simply requires writing the data to the output queue of the consuming core. The input and output queues provide queuing slots and the pipelined fabric serves as both a computational substrate and as additional *on-demand* queue slots. The baseline design already takes care of stalling the producer thread or fabric if slots are not available in either the fabric or output queue, respectively, and also stalls the consumer if the output queue is empty. If desired, the output can be sent to multiple cores by specifying multiple destinations in the configuration.

Figure 5.2 details the steps involved in interthread communication. First, the producing thread loads data to send to the consumer into its input queue (Figure 5.2(a)). Once all of the necessary data is loaded, the producer issues the SPL instruction (Figure 5.2(b)). The data progresses through the fabric to perform the computation programmed into the fabric. Once any computation is complete, the results are bypassed to the output queue of the consuming core

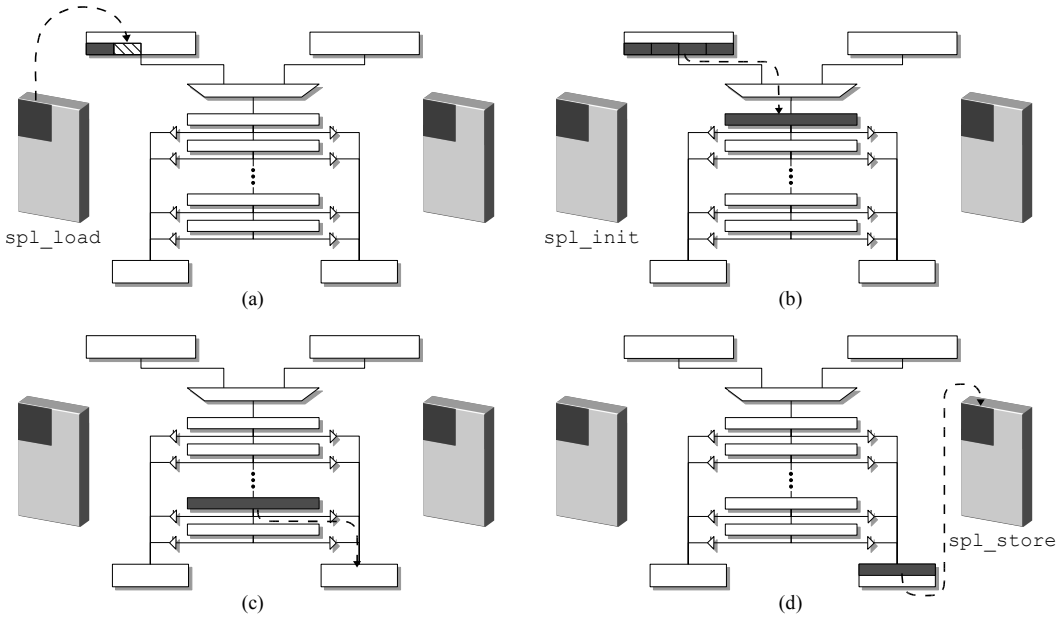



Figure 5.2: Walk through of intercore communication.

(Figure 5.2(c)). Finally, the consuming core stores the data from the queue to memory (i.e., cache) (Figure 5.2(d)).

Two features ease intercore communication via RACM. First, in order to fully utilize the queuing capacity of the fabric, instructions that have completed their computation but cannot yet be allocated an output queue slot continue to progress through the rows of the fabric, simply passing their output data through to the next row. This continues until either an output queue slot becomes available, at which point the data is immediately written to the output queue (bypassing any remaining rows in the fabric) or the instruction reaches the end of the fabric, at which point it stalls. When the fabric is stalled, instructions immediately following the stalled instruction stall as well. Bubbles in the SPL pipeline, however, are allowed to collapse and so some progress may continue to be made even if the head instruction is stalled. If the entire fabric is full, then the producing thread will stall if it attempts to issue additional SPL instructions.

The second feature is a small table to maintain a mapping of threads to cores in order to virtualize the selection of the destination core (see Figure 1.1(b)). If the thread to core mapping was known *a priori*, then the consuming core could simply be hard coded into the SPL configuration. Since this is unlikely in any real system, a dynamic method is needed to obtain the current location of the receiving thread. To achieve this, each SPL is augmented with a small table that provides the current mapping of threads to cores for that cluster. In our proposed 4-way shared fabric, each table has four entries. Each entry contains the thread and application ID currently running on each core as well as a count of the number of in-flight instructions destined for that core (the need for which will be described shortly). Assuming a limit of 256 thread and application IDs and a maximum of 24 in flight instructions (as the fabric has 24 rows), each per-SPL table requires a 11.5B CAM (16 bits for IDs, 5 bits for number of in flight instructions, and 2 bits for hard coded core ID). When an SPL instruction is issued, it obtains the core currently assigned to its destination thread (which may be either itself or another thread) from the table and stores its results to the appropriate output queue upon completion.

A side benefit of this table based approach is that instructions will not issue to the fabric if the destination thread is not available (i.e., not present in the table). This prevents the producing thread from filling up the fabric if the consumer is not even present, which could impact other threads sharing the fabric. If both threads are present but not well balanced, it is possible the fabric could still be full most of the time. However, assuming that the program is even remotely well written, the consumer would still be consuming values, even if at a slow rate, and, because of the SPL's round robin issue policy, other threads would continue to be able to utilize the fabric, albeit at a possibly slower rate.

Even with the table, however, SPL instructions could accumulate in the fabric if the consumer thread is switched out while data is in flight to it, which would require the consumer to be switched back into the same core to receive the values. To prevent this situation, the thread-to-core mapping table maintains a count of the number of in-flight SPL instructions destined for each core. On a request to switch out, the consumer checks for in-flight SPL instructions bound for that core. If these exist, the fabric is blocked from accepting any new instructions destined for the consumer and the consumer continues to execute until the in-flight counter reaches zero. At this point the consumer can be switched out and the fabric unblocked.

**Barrier Synchronization+Computation**

Barriers are one of the most frequent synchronization operations. However, with a typical memory-based implementation, the overhead of executing a barrier can be significant, especially as the number of threads increases. This overhead prevents the use of barriers at fine granularities. To address this drawback, various proposals [3, 5, 67, 72] have suggested dedicated mechanisms to reduce this overhead, thereby allowing parallelization of applications that would not otherwise be possible.

To implement barriers in RACM, SPL barrier instructions (indicated by a flag in the configuration information for the SPL function), must not be allowed to issue to the fabric until all participating cores have arrived at the barrier. To achieve this, each core participating in the barrier loads some value(s) into its SPL input queue. Once the loads from all of the cores have reached the head of their respective input queues and all threads have indicated arrival at the

barrier by executing the SPL initiate instruction, the loaded values from each core are passed into the fabric. The valid bits associated with every byte in the input queues are used to determine which values from each core should be loaded into the fabric. Once any computation is complete, results are placed into the output queue of each participating processor and the processor stores the data as appropriate. Participating cores are indicated by flags that travel through the fabric with the data (see top right of Figure 1.1(c)). A memory fence is executed following the stores to ensure that no subsequent memory operations are performed until the barrier is complete.

To determine that all threads have arrived, each SPL cluster maintains a table with information related to each active barrier. Each table (see Figure 1.1(b)) contains as many entries as cores attached to a RACM cluster, as each could be participating in a different barrier. Each entry contains the barrier ID, the current number of arrived threads, flags indicating which processors are participating in the barrier, IDs of participating threads, and the total number of threads involved in the barrier (stored as part of the configuration). A table entry is switched out when all threads associated with the barrier are switched out.

Figure 5.3 illustrates the steps involved in a RACM barrier. To start, cores load data into the input queues as normal (Figure 5.3(a)❶). When an SPL barrier instruction is issued, RACM checks the barrier table for an associated entry. If no entry is present, a new entry is allocated, the arrived count is set to one (Figure 5.3(a)❷), and the destination flag and thread ID for the participating core is set ❸. The former indicates to which output queues results are written when the function completes. The latter is used to make sure that all participating

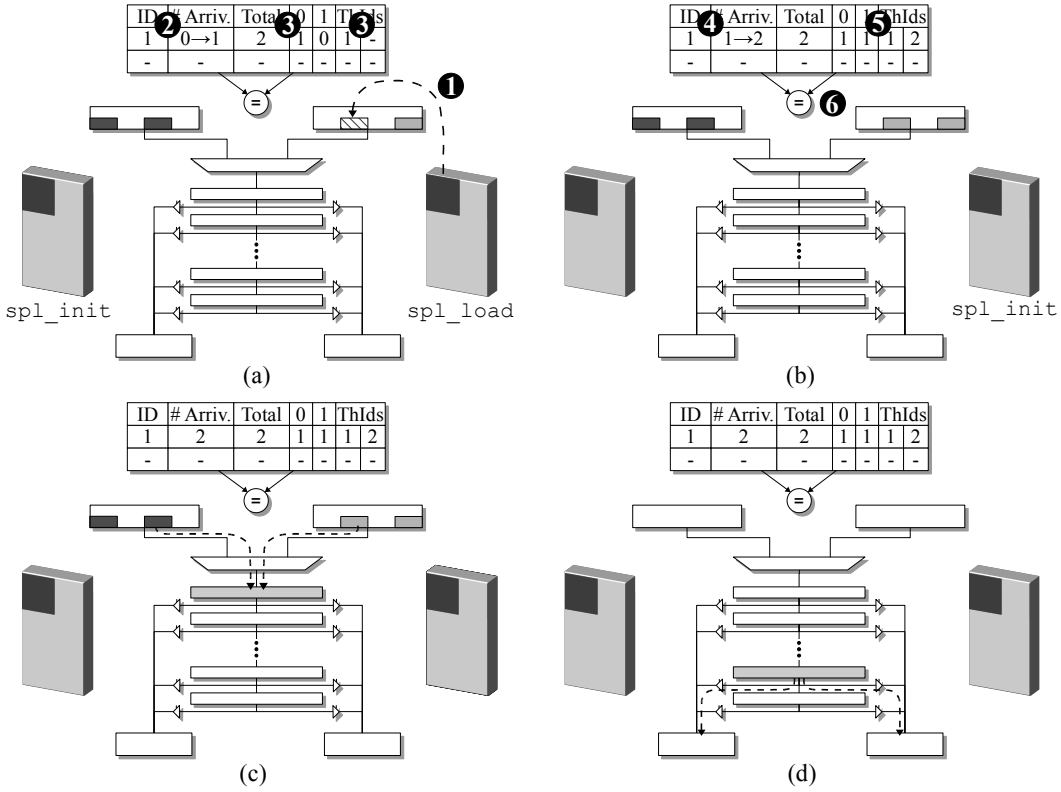



Figure 5.3: Walk through of barrier synchronization.

threads are running when the barrier is released. If an entry already exists for the barrier then the number of arrived threads is incremented (Figure 5.3(b)❹). The appropriate destination flag and thread ID in the barrier table is set ❺ and the arrived and total number of threads are compared ❻. If the arrived number now equals the total, the SPL controller issues an instruction to the fabric, passing in the data from the participants' input queues (Figure 5.3(c)). Once any computation is complete, the results are written to the output queues of all participating cores (Figure 5.3(d)) as indicated by the destination core bits in the barrier table. All participating output queues must have a slot available before the results will be output. Finally, each core stores data from the queue to cache memory (similar to Figure 5.2(d)).

In a system with multiple SPL clusters, each cluster communicates updates on the number of arrived threads with all other clusters (even though other clusters may not have participating threads). An alternative is to have clusters only monitor those barriers in which they have threads actively participating. This, however, requires that clusters obtain the number of currently arrived threads from another cluster each time a locally new barrier arrives, which increases the complexity of the intercluster network. Since the table is localized and is small in either case, whereas the increase in interconnect complexity has global impact, we choose to track all active barriers in every cluster to reduce interconnect overhead at the cost of an increase in table size.

A dedicated bus communicates barrier updates among clusters. The bus transmits the barrier ID as well as the associated application ID (as different applications might use the same barrier ID). With a limit of 256 IDs, the shared bus requires only 16 data lines plus control. Each table entry requires 8 bytes: 16 bits for IDs; 4 for number of arrived threads; 4 for total number of threads; 4 to indicate participating cores; 32 for participating thread IDs; and 4 to indicate if each participating thread is currently active. In a 16 core system, this requires a 128B table for each cluster.

Since the SPL input and output queues are saved on a context switch, all threads participating in a barrier must be actively running in order for all input data to be available. Each table entry maintains a list of the IDs of the local threads that are participating in the barrier as well as a bit indicating if they are actively running. If a barrier is ready to be released but not all participating threads are active, the RACM controller triggers an exception to switch the missing threads back in. Once all threads are available, the barrier can proceed. Since

RACM barriers are primarily intended for fine grain synchronization, switching out a thread that arrives early should be avoided in any event for performance reasons.

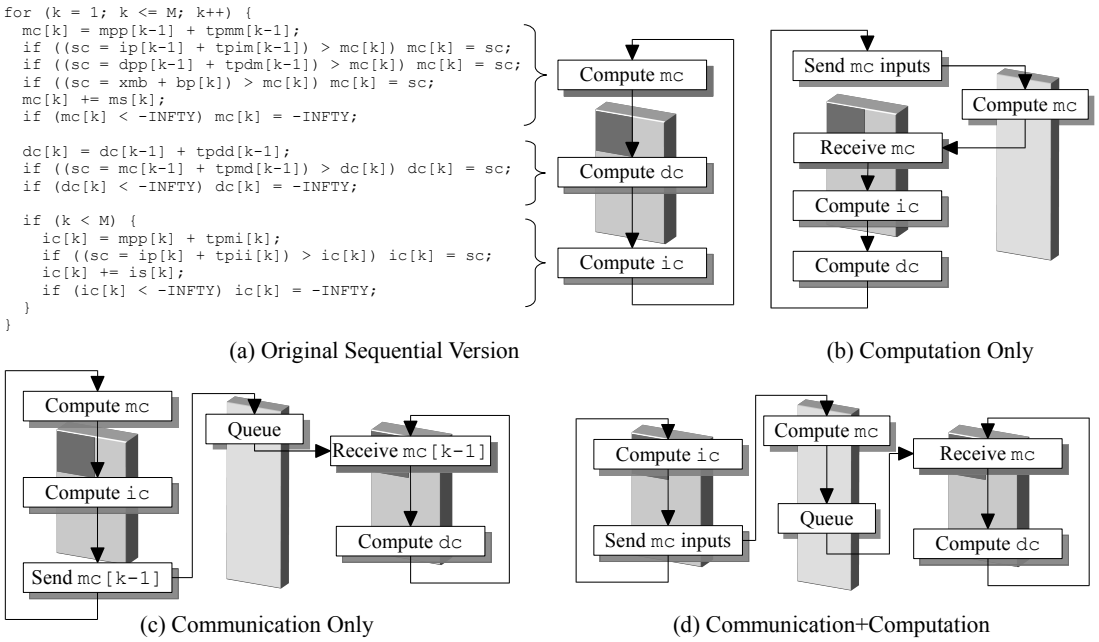
## 5.2 Communication Examples

We propose using the SPL to perform both fine-grained interthread communication and fine-grained barrier synchronization. In this section we show example applications that benefit not only from the enhanced communication, but also receive additional benefits due to the computational power of the SPL that could not be achieved with communication alone.

### 5.2.1 Interthread Communication+Computation Example

To illustrate interthread communication, we show an example parallelization of the SPEC2006 application *456.hmmer*. We optimize the inner loop of the `P7Viterbi` function, which implements the dynamic programming Viterbi algorithm. The original code for the optimized section is shown in Figure 5.4(a) along with a flow chart summarizing the computation being performed. This high level description will be used to show how the function is optimized for computation alone, communication alone, and for the combined computation+communication case.

We first look at how the SPL can be used to accelerate a portion of the computation, specifically the calculation of `mc`. As shown in Figure 5.4(b), the core loads the input values needed to compute `mc` into the fabric, the SPL computes




(a) Original Sequential Version

(b) Computation Only

(c) Communication Only

(d) Communication+Computation

Figure 5.4: Parallelization of SPEC 2006 *456.hmmer* `P7Viterbi`.

the value of `mc`, and the core receives the result. After receiving `mc`, the core computes the values of `dc` and `ic` and repeats the loop. Figure 5.5 shows the general functionality performed within each row of the SPL for the optimized section.

The next implementation creates a producer/consumer thread pair that uses the SPL solely for communication (Figure 5.4(c)). The producer thread is responsible for calculating the values of `mc` and `ic` and sending the value of `mc` from the previous iteration to the consumer through the SPL. The consumer receives this value and uses it to compute `dc`.

Finally, Figure 5.4(d) shows how computation and communication can be combined in the SPL. The producer thread computes `ic` and loads the inputs needed to compute `mc`. The SPL computes the value of `mc` and sends it to the consumer. The consumer receives this value and uses the value of `mc` from the previous iteration to compute `dc`. Computing `mc` in the fabric reduces the

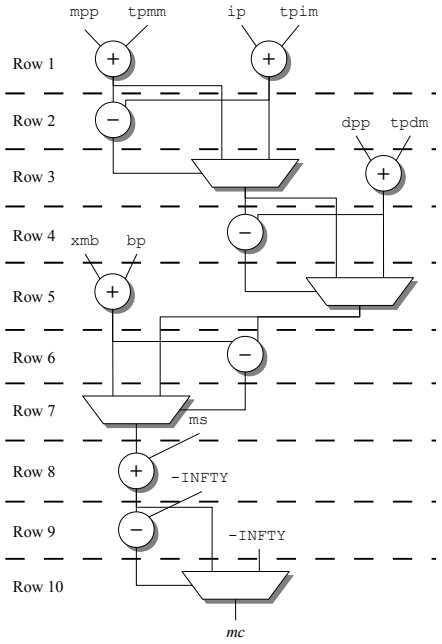



Figure 5.5: SPL `mc` calculation mapping.

amount of work for the producer, which better balances the threads and further improves the performance of the parallelization (see Section 5.4.1).

### 5.2.2 Barrier Synchronization+Computation Example

To show the operation of RACM barrier synchronization, we consider a parallel version of Dijkstra's Shortest Path Algorithm. Parallel versions of Dijkstra's Algorithm have previously been proposed. These algorithms, however, tend to provide limited or no speedups for small to moderate graph sizes. By using the SPL to perform the barrier synchronization, we can improve the synchronization while also using the fabric to perform computation during the barriers to further improve performance.

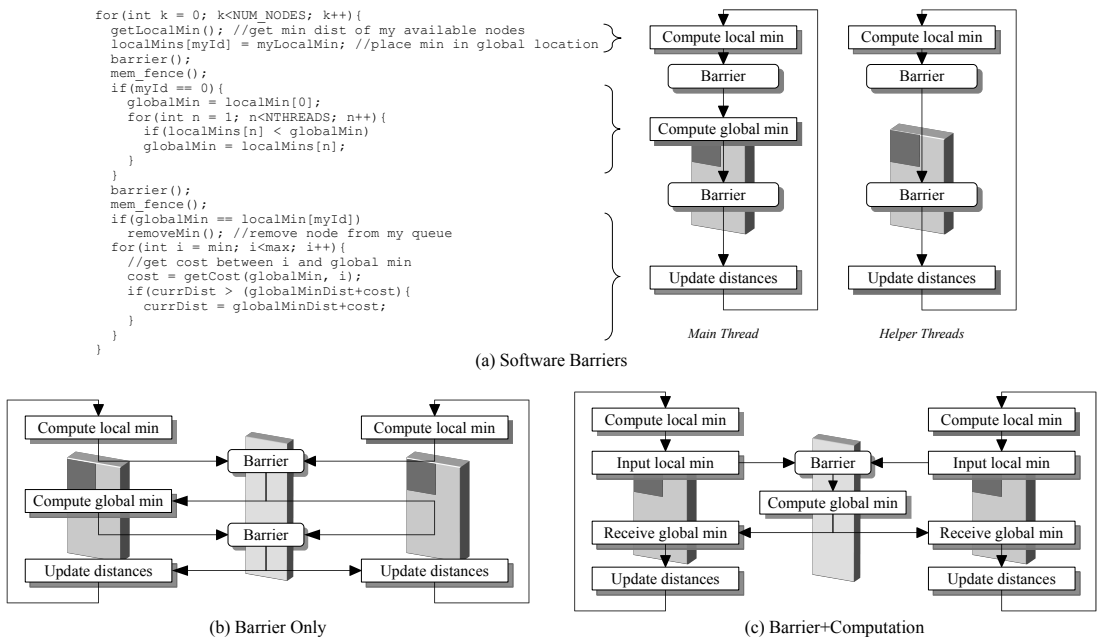



Figure 5.6: Parallelization of Dijkstra's Shortest Path Algorithm.

In the parallel version of Dijkstra's Algorithm, each thread is given a portion of the entire graph to maintain. Figure 5.6(a) shows pseudocode of the basic parallel algorithm and the high level flow of the main and helper threads. The code consists of three sections, delineated by code before, between, and after the two barriers. In the first section, each thread determines the minimum value of all unvisited nodes among its subset and places this value in a global location. In the next section, the main thread computes the global minimum from these local minimum values and makes this value globally available. Finally, each thread reads the global minimum and updates the distances for all of its nodes.

The first optimization that can be made is to replace the software barriers with RACM barriers, as shown in Figure 5.6(b). As with previous dedicated barrier techniques [3, 72], replacing the software barriers with RACM barriers provides significant performance improvements. Performance can be further improved beyond that possible with previous techniques by using the compu-

tational power of the SPL to compute the global minimum within the fabric. Figure 5.6(c) shows this optimization for the case where all threads share a single SPL cluster. Each thread computes its local minimum as before and then loads this value into the SPL. While performing the barrier, the SPL computes the minimum of the input values. Each participating core receives the global minimum from the SPL and updates the distances for its nodes. Since the SPL outputs the global minimum directly, one of the barriers is eliminated.

If the threads are spread across multiple clusters, the fabric still helps compute the minimum; however, this operation is performed in multiple stages and requires an extra barrier to ensure proper execution. The first stage computes regional minimum values (minimum values of all cores in a single cluster). The second barrier ensures that all clusters have finished storing these results. At the final barrier each cluster loads the regional minimum values and the fabric computes the final global minimum. Despite the extra barrier, performance is still improved over using the SPL for communication only (see Section 5.4.3).

## 5.3 Evaluation Methodology

We use a modified version of SESC [71] to evaluate our proposed communication schemes. We assume processors implemented in 65 nm technology running at 2.0 GHz with a 1.1V supply voltage. The major architectural parameters are shown in Table 5.2. We use Wattch and Cacti to model dynamic power and Cacti and HotLeakage to model leakage power.

Table 5.2: Architecture parameters.

| | |
|---|---|
| Fetch/Decode/Rename Width | 2 |
| Issue/Retire Width | 1 |
| Branch Predictor | gshare + bimodal |
| RAS Entries | 32 |
| BTB Size | 512B |
| Integer/FP Registers | 64/64 |
| Integer/FP Queue Entries | 32/16 |
| ROB Entries | 64 |
| Int/FP ALUs | 1/1 |
| Branch Units | 1 |
| LD/ST Units | 1 |
| L1 Inst Cache | 8kB 2-way, 2-cycle access |
| L1 Data Cache | 8kB 2-way, 2-cycle access |
| L2 Cache | 1MB per core, 10-cycle access |
| Coherence Protocol | MESI |
| Main Memory Access Time (ns) | 100 |

### 5.3.1 Benchmarks

We use benchmarks from the SPEC2006 [78], SPEC2000 [77], MediaBench [51], MiBench [30], and Livermore Loops [55] suites along with the Unix utility *wc* to analyze the three usage modes from Figure 5.1. A complete list of the benchmarks used for each operation mode, the functions we optimize in each, and the percentage of total program execution time consumed by the functions are listed in Table 5.3. *Cjpeg* makes use of two operation modes, computation-only and computation+communication, and is evaluated with other communicating workloads. We execute two 250 million instruction Early SimPoints [58] for SPEC workloads with reference inputs and run all other workloads to completion.

To evaluate barrier synchronization we use parallel versions of Livermore Loops 2, 3, and 6 and Dijkstra's Algorithm, the latter using inputs from MiDataSets [30]. Two of the benchmarks, specifically Livermore Loop 3 (*LL3*),

Table 5.3: Benchmark details.

| Benchmark | Functions Optimized | % Exec Time |
|---|---|---|
| Computation Only | | |
| g721dec | `fmult` | 48% |
| g721enc | `fmult` | 46% |
| mpeg2dec | `store_ppm_tga`, `conv422to444`, `conv420to422` | 63% |
| mpeg2enc | `dist1` | 70% |
| gsmtoast | `Calculation_of_the_LTP_parameters`, `Weighting_filter` | 54% |
| gsmuntoast | `Short_term_synthesis_filtering` | 76% |
| 462.libquantum | `quantum_toffoli`, `quantum_cnot` | 40% |
| Communication+Computation | | |
| wc | `wc` | 100% |
| unepic | `read_and_huffman_decode` | 22% |
| cjpeg | `rgb_ycc_convert`, `jpeg_fdct_islow` | 50% |
| adpcm | `adpcm_decoder` | 99% |
| 300.twolf | `new_dbox_a` | 30% |
| 456.hmmer | `P7Viterbi` | 85% |
| 473.astar | `regwayobj::makebound2` | 33% |
| Barrier Synchronization | | |
| Livermore Loop 2 (LL2) | | 100% |
| Livermore Loop 3 (LL3) | | 100% |
| Livermore Loop 6 (LL6) | | 100% |
| Dikjstra's Algorithm | | 100% |

which is transformed to operate on integers, and Dijkstra's algorithm, include computation in the fabric after synchronization. In Dijkstra's Algorithm, computation is performed during the synchronization operation (as in Figure 5.1(c)). *LL3* makes use of two RACM modes of operation: performing computation on the data within the loop (Figure 5.1(a)), and using the SPL to accelerate synchronization between iterations (Figure 5.1(c)).

### 5.3.2 RACM Programming

We modify our workloads by hand to create the producer/consumer pairs and SPL mappings. Previous work has shown that compilers can produce good mappings for reconfigurable architectures [4, 7, 35, 94] and good partitionings for pipelined applications [36, 57]. We believe our design could leverage this prior art in an actual implementation.

## 5.4 Results

We first evaluate RACM in the context of a heterogeneous CMP executing entire programs and then look at the performance of the optimized regions to show the source of the improvements.

### 5.4.1 RACM in a Heterogeneous CMP

To show the benefit of including SPL clusters in a heterogeneous CMP, we compare the performance of a RACM system to a system with larger cores and dedicated communication hardware, while executing entire applications. The RACM system is composed of clusters of four OOO1 cores plus a 24-row SPL coupled with clusters of 2-way issue out-of-order (OOO2) cores (the organization shown in Figure 1.1(a)). The second system is composed of clusters of OOO2 cores with a dedicated communication network, similar to previous proposals [9, 62]. Assuming zero hardware cost for the communication network, four OOO2 cores with this idealized communication support (*OOO2+Comm*)

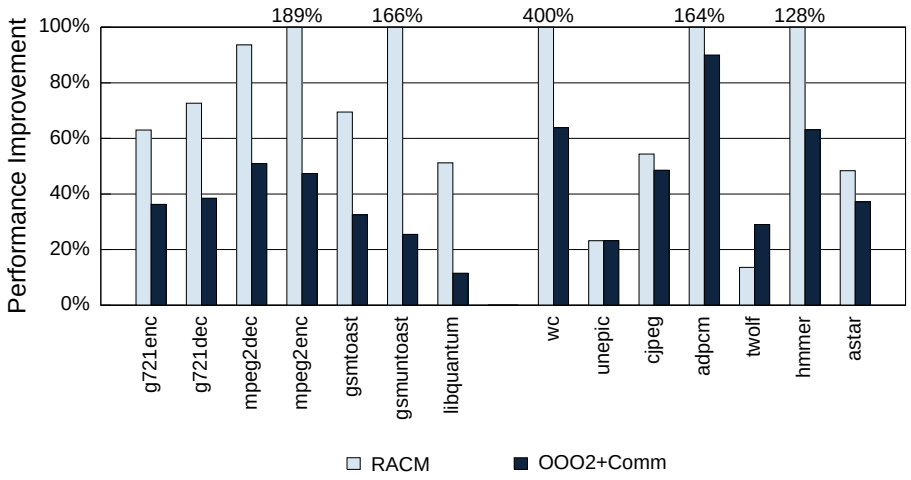



Figure 5.7: Performance relative to single threaded baseline.

consumes approximately the same area as an SPL cluster. In the RACM configuration, regions of code that utilize the SPL are run on the SPL cluster while other regions are run on an OOO2 core. The migration overhead is accounted for by draining in-flight instructions and stopping execution for 500 cycles (determined by running the requisite code in the simulator) to model the time necessary to context switch all state to the new core.

We compare the two schemes with workloads that use the SPL for computation alone and workloads that use the SPL for computation+communication (results for barrier synchronization are discussed separately in Section 5.4.3). Computation-only workloads are run concurrently with other computation-only workloads to include appropriate SPL contention. Communicating workloads are run separately, but are given access to only half of the SPL, making it area equivalent to the *OOO2+Comm* alternative (the SPL is assumed to be partitioned and the other half is in use by other threads). The performance improvement of the two configurations relative to executing the original sequential code on a single-issue core is shown in Figure 5.7. Computation-only benchmarks are

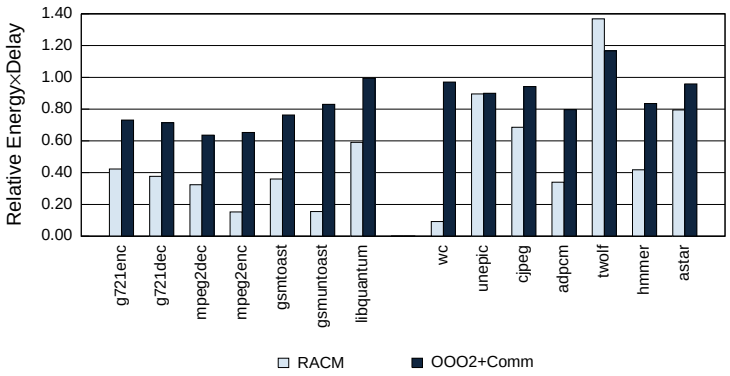



Figure 5.8: Energy×delay relative to single threaded baseline.

shown on the left side of the figure and computation+communication benchmarks are on the right side.

RACM performs as well or better than the alternative in all but one case. On average it provides 49% better performance than *OOO2+Comm* for computation-only workloads and 41% better performance for communicating workloads. In the one exception, *twolf*, the time between sequential and parallel regions is short enough to prevent migration to the 2-way issue core during the sequential regions. The benefit of executing on the 2-way issue core during the sequential regions is enough to outweigh the performance benefit of executing on the SPL during the parallel sections.

While adding RACM communication and computation improves performance relative to a single threaded implementation, energy efficiency may degrade given the energy costs of the extra core and SPL. Figure 5.8 shows energy×delay (ED) for the two configurations relative to the single threaded baseline. RACM provides better (i.e., lower) ED than both the baseline and *OOO2+Comm* configurations in all but one case. The one exception is again *twolf*, where both alternatives achieve worse ED than the baseline, indicating that *twolf* should be run as a single thread on a simple core if energy is a sig-

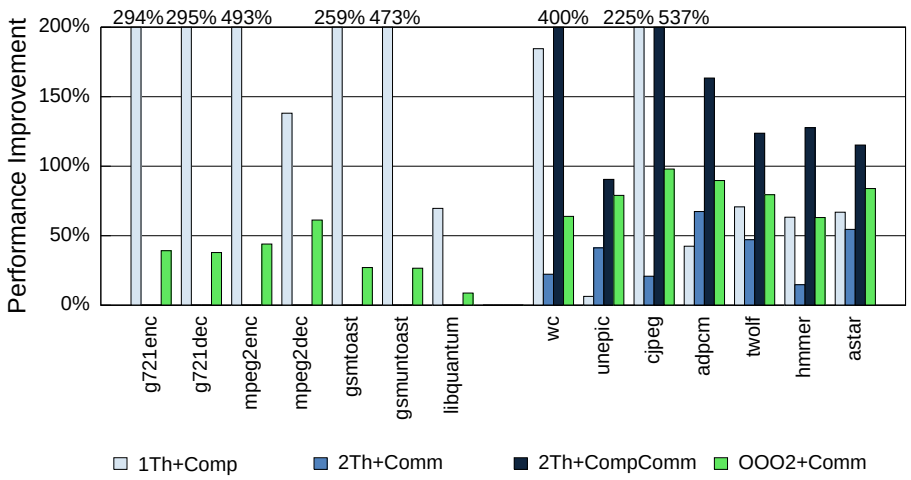



Figure 5.9: Performance improvement of optimized functions relative to performance of single threaded baseline.

nificant concern. *OOO2+Comm* also generally provides better ED than the baseline, but only marginally so in many cases (such as *cjpeg*, *astar*, and *libquantum*). With the exception of *twolf*, RACM provides both better performance (45% better on average) and lower energy consumption (35% less on average) than *OOO2+Comm* for all benchmarks.

### 5.4.2 Analysis of Optimized Regions

We now look at the performance of just the code regions optimized for RACM to see the source of the above improvements. Figure 5.9 shows the performance improvements relative to the single threaded baseline of a single thread using the SPL for computation (*1Th+Comp*) and (where appropriate) dual threads with the SPL used for communication (*2Th+Comm*) and dual threads with the SPL used for computation+communication (*2Th+CompComm*). We also show the performance of running on the system of OOO2 cores with idealized communication hardware (*OOO2+Comm*). Using the SPL for computation only

(*1Th+Comp*) provides significant performance improvements (289% and 105% on average for computation-only and communicating workloads, respectively).

Focusing on the workloads employing communication, using the SPL for producer-consumer communication alone provides a 38% improvement in performance for the optimized region relative to the single core baseline. By adding the speedups obtained from communication and computation alone, we would expect to achieve an average speedup around 143% when the two techniques are combined. The results for *2Th+CompComm*, however, show an average improvement of 223%. The reasons for this behavior will be discussed in the next section.

It is only with the combination of computation and communication that RACM outperforms the *OOO2+Comm* alternative in all cases (by 79% on average), showing the clear benefit of integrating SPL computation and communication. The results for *twolf* also show that computation+communication does indeed perform better *OOO2+Comm* for the optimized region, confirming the earlier statement that the performance loss is due to RACM's lower performance compared to *OOO2+Comm* on the sequential regions between the optimized sections.

To confirm the need for hardware-based communication, we also ran the benchmarks with software queues, with and without SPL computation, and found that these experience more than a 180% slowdown on average relative to the *single threaded* baseline.

**Contributing Factors**

We analyzed the benchmarks to identify the factors that contribute to the performance improvements for combined SPL communication+computation. Primary among these factors is that the combination of SPL computation and communication reduces the amount of time between successive SPL requests relative to using either technique in isolation, often by 2X or more. In *wc*, for example, the average time between SPL requests drops from 62 and 19 cycles with communication and computation in isolation, respectively, to 12 cycles when the two techniques are combined. This increased access rate improves performance by increasing the amount of concurrent processing in the SPL.

Relative to the single threaded case, splitting the application into a producer/consumer pair allows us to place sections of code with poor branch or load performance in their own thread to reduce or eliminate their impact on performance. In *unepic*, for example, the consumer if responsible for a section of code with both an unpredictable branch as well as a pointer chasing load. By placing just this code in the consumer and the rest in the producer, the consumer can start processing these unpredictable instructions earlier. This reduces the impact of the unpredictability of these instructions and improves performance. With RACM communication we can also perform computation during the communication, meaning that each core is now responsible for approximately half of the SPL instructions (either the loads or the stores). This reduces the number of instructions that both threads need to process, which can lead to reduced pressure on the ROB and other related structures. This leads to fewer pipeline stalls and therefore better performance.

Compared to just communicating data, performing computation on the data while in flight to the consumer provides multiple sources of improvement. For one, the SPL computation removes instructions from one or both threads. This can help to better balance the work done by both threads and allow for more efficient pipelining. Both *astar* and *adpcm*, for example, are consumer bound with just communication. By performing computation in the SPL, computation previously performed by the consumer is now performed in the SPL, leading to more balanced producer/consumer threads. For these two cases, of the 12 slots available, the average number of occupied queue slots decreases from 11.5 and 10.9 to 2.5 and 3.1 for *astar* and *adpcm*, respectively. These more balanced threads spend less time waiting on a full/empty SPL queue, which improves performance. Removing instructions from one or both of the threads can also reduce pressure on the ROB and related structures, again improving performance. *Cjpeg* and *unepic* are two examples that see reduced ROB stall time with integrated computation, with ROB full time decreasing from 27% and 30% to 11% and 0%, respectively. Finally, moving computation inside the SPL can improve branch prediction in one or both threads by moving conditional operations into the SPL. *Adpcm* and *wc* are two cases that see such a reduction in misprediction rate with the prediction rate improving from 85% to 97% for *adpcm* and from 80% to 99.9% for *wc*. The improved branch prediction improves processor efficiency which again improves performance.

**Energy Efficiency Results**

Figure 5.10 shows the ED of the three SPL implementations and the *OOO2+Comm* alternative relative to the single threaded baseline without SPL.

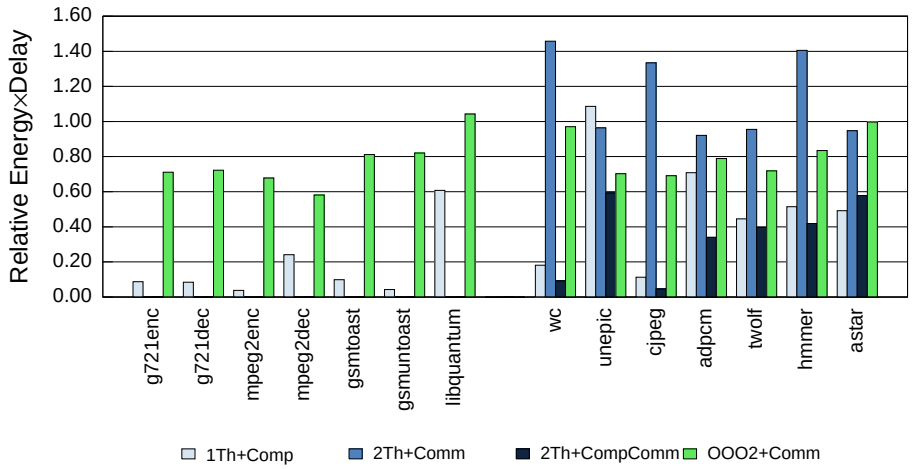



Figure 5.10: Energy×delay relative to single threaded baseline.

While adding computation or communication in isolation reduces ED in many cases, neither is able to provide enough performance benefit to overcome the added power consumed by the extra core and/or SPL in all cases. RACM communication+computation always improves performance and reduces energy consumption compared to *OOO2+Comm* and is the only option to provide better ED than the the single threaded baseline in all cases.

### 5.4.3 Fine-Grained Barrier Synchronization

We evaluate the performance of software (SW) versus RACM barriers for our four barrier applications when executing 2, 4, 8, and 16 threads. Figure 5.11 shows the performance for SW and RACM barriers (with and without computation where appropriate) for the 8 and 16 threaded cases (2 and 4 threads show similar trends and are omitted for graph clarity).

Similar to other fine-grained synchronization techniques [3, 67, 72], performing barriers via RACM significantly improves performance over SW barriers.

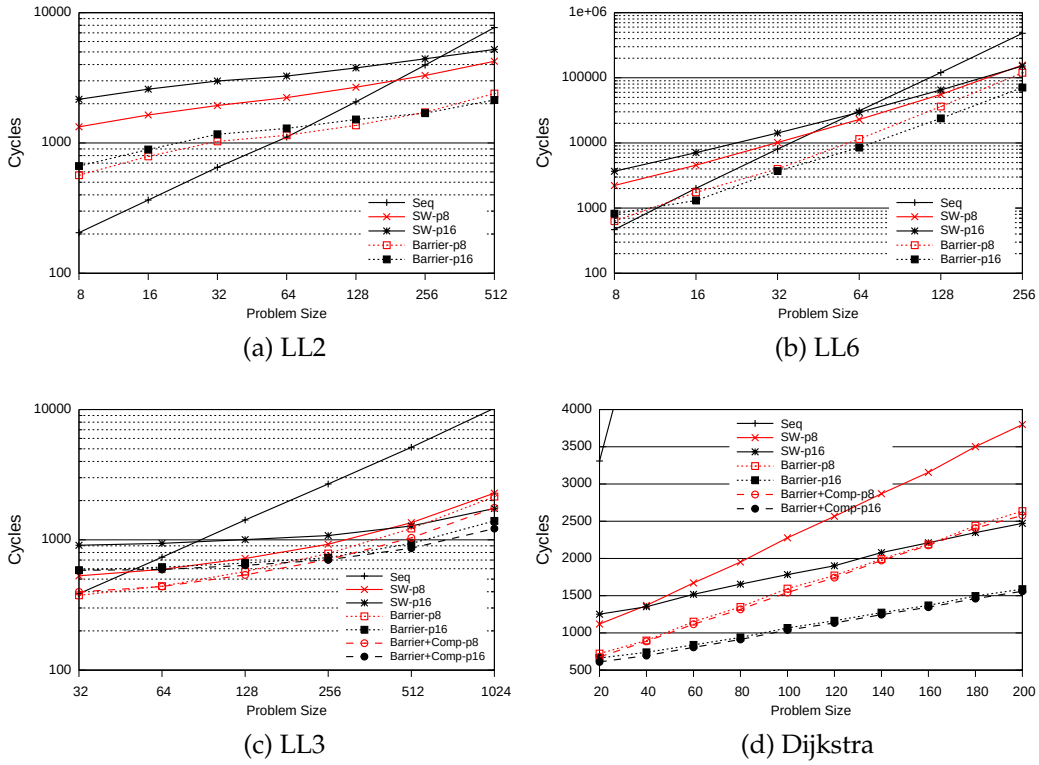
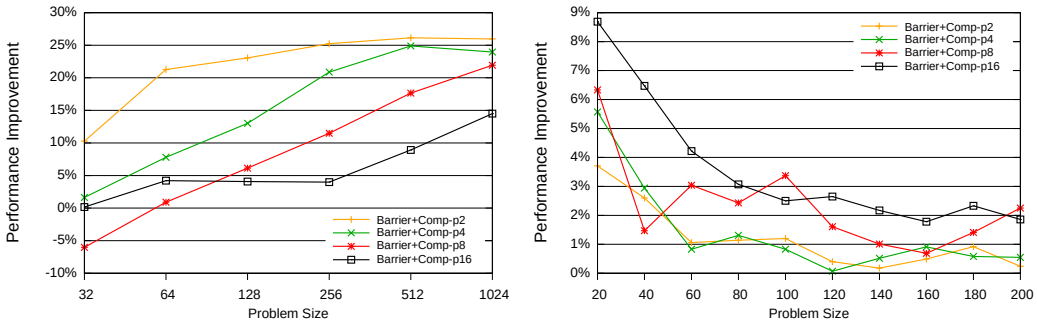



Figure 5.11: Per iteration execution time for Livermore loops (a) 2, (b) 6, and (c) 3 and (d) Dijkstra's Algorithm.

For the Livermore Loops, the RACM versions start outperforming the sequential code for much smaller vector lengths. For instance, in *LL6* with 16 threads, RACM barriers start outperforming the sequential case at a problem size between 8 and 16 whereas SW barriers only start outperforming the sequential case at a size of 64, demonstrating the benefits of finer-grain synchronization. Fine-grained synchronization also makes larger thread counts useful for smaller problem sizes. In *LL2*, for example, the 16 threaded version starts outperforming the 8 threaded version at a problem size of 256, but the 16 threaded SW version never outperforms the 8 threaded case for the problem sizes we investigate. In *dijkstra*, RACM barriers not only outperform software barriers with the same number of threads but also outperform software barriers with two or four times the number of threads in some cases.




(a) LL3 (b) Dijkstra

Figure 5.12: Performance improvement of barriers + computation over barriers alone for (a) *LL3* and (b) *dijkstra*.

### Fine-Grain Barrier Synchronization with SPL Computation

Certain parallel benchmarks, such as *LL3* and *dijkstra*, also benefit from the computational capabilities of the SPL. This computation is either performed as part of the barrier operation, as is done in *dijkstra* (see the discussion in Section 5.2.2), or in a separate SPL function that only performs computation, as in *LL3*. The execution time and performance improvement relative to barriers alone of barriers+computation for the two benchmarks are shown in Figures 5.11(c-d) and 5.12, respectively.

For *dijkstra*, where the computation is performed as part of the barrier, the benefits of adding computation are most pronounced as the number of threads increases and at finer synchronization granularities. This is due to the fact that thread synchronization, which is the portion of code accelerated by RACM computation, consumes more time with smaller problem sizes and as the number of threads increases. In the 16 threaded case, adding computation provides up to a 9% improvement versus hardware barriers alone.

In *LL3*, on the other hand, where the computation is a separate function, the most benefit is received with a smaller number of threads and/or coarser synchronization granularities. In either of these cases, each thread has more work to do between barriers. This means that the computation section makes up a larger percentage of the execution time and so speeding it up provides greater relative benefit. When there are an extremely small number of loop iterations per thread, the Barrier+Comp case can actually perform worse than synchronization alone as there are not enough SPL instructions to take advantage of the pipelined nature of the fabric. This can be seen in Figure 5.12(a) for small problem sizes and large thread counts. In each of these cases each thread has only 2 or 4 iterations to perform and so little pipelining occurs. For the larger problem sizes, however, the performance improvement is significant, ranging from 15-26%.

**Energy Efficiency Results**

Figure 5.13 shows energy×delay (ED) results for the four synchronization workloads relative to the single threaded case. In general, the break even point for ED – the point at which the ED of the parallel case drops below the sequential case – for both SW and RACM barriers requires a larger problem size (coarser grained synchronization) than the performance break even point. This occurs since, especially at very fine granularities, the performance improvement achieved by increasing the number of threads is not ideal (i.e., doubling the number of threads does not halve the run time). For 16 threaded *LL2* and *LL6*, SW barriers *never* break even for the problem sizes we investigate. RACM barriers always

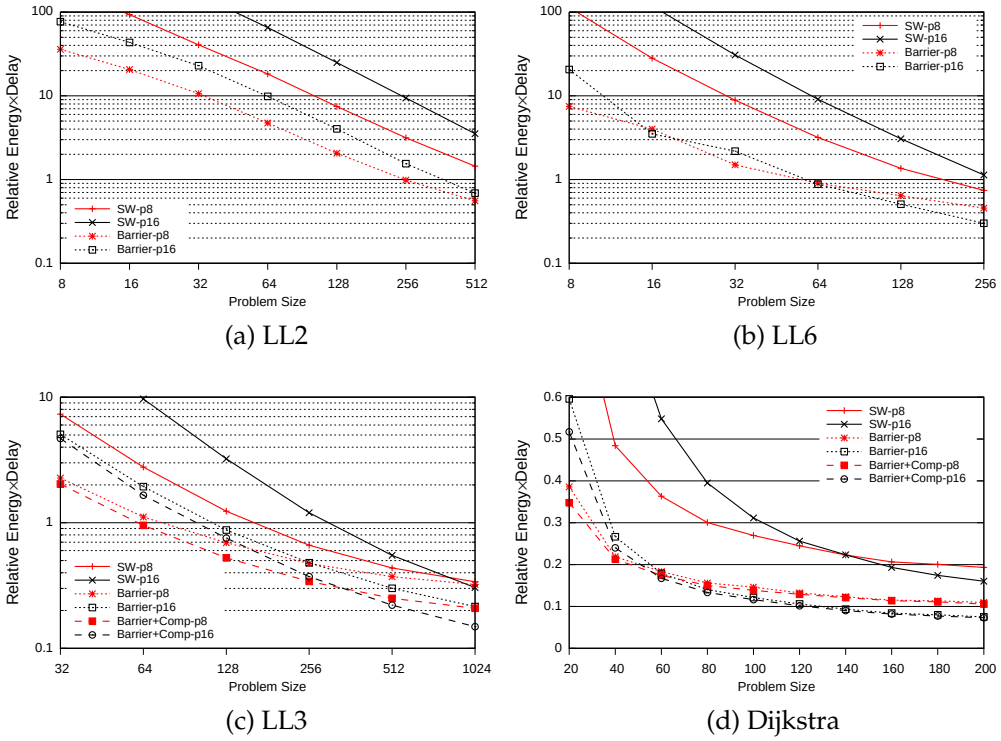



Figure 5.13: Energy×delay for Livermore loops (a) 2, (b) 6, and (c) 3 and (d) Dijkstra's Algorithm relative to sequential execution.

achieve better ED than their SW counterparts, despite the additional energy consumed by the SPL which is not present with SW barriers.

We also evaluate the performance achieved by replacing the SPL with additional cores and dedicated fine-grain barrier support [3, 72]. Since the SPL consumes as much area as two single-issue cores, we simulate a system where each SPL is replaced by two additional cores (yielding a total of 24 cores for the case that originally had 16 cores+SPL) and the cores are connected with a dedicated barrier network that incurs no hardware cost. We find that, compared to such a homogeneous cluster, RACM barriers+computation achieves up to 25.9% and 62.5% lower ED for *dijkstra* and *LL3*, respectively, demonstrating the benefits of RACM custom computation with fine-grain barrier synchronization.

## 5.5 Conclusion

This chapter proposes using the SPL to perform multiple forms of fine-grained communication in a heterogeneous CMP. In addition to accelerating computation like traditional reconfigurable fabrics, RACM can be configured to facilitate multiple forms of fine-grained communication. In contrast to previous fine-grain communication approaches, RACM enables custom computation to be integrated with communication. Combining these multiple modes leads to a significant 45% performance improvement relative to what could be achieved with an area equivalent system with larger cores and free hardware communication. Similarly, we also show that RACM provides better energy efficiency than can be achieved by using the area consumed by the SPL for either additional or more powerful cores, providing a 44% reduction in ED. These results demonstrate the significant advantages of incorporating reconfigurability into future heterogeneous CMPs.

# CHAPTER 6

# CONCLUSIONS AND FUTURE WORK

This thesis investigates reconfigurable architectures for chip multiprocessors (RACM). In particular, it focuses on how the recent shift to multicore processors allows more efficient integration of reconfigurable logic on-chip than possible with previous monolithic single-core designs. In addition to exploring the benefits of shared reconfigurable logic for sequential applications, which were the almost exclusive focus of previous reconfigurable works, we also evaluate the advantages of reconfigurable logic for parallel applications and for parallelizing otherwise sequential applications.

This work makes the following contributions:

- *Shared Reconfigurable Fabric for CMPs* – We design a tightly integrated, row-based reconfigurable fabric that is specially tailored for sharing among multiple cores of a chip multiprocessor. This Specialized Programmable Logic (SPL) fabric is shown to provide significant performance an energy benefits compared to what could be achieved by dedicating the fabric area to more powerful cores;
- *Multiple SPL Sharing Schemes* – We propose mechanisms to temporally and spatially share the fabric between multiple cores. It is shown that, with proper sharing, the size of the SPL can be drastically reduced while still achieving the same performance as much larger per-core private fabrics;
- *Dynamic Cluster Management Policies* – Shared SPLs must be intelligently managed if they are to achieve optimal performance. We create multiple dynamic cluster management policies to manage the assignment of

threads to cores and the partitioning of the SPL. The best of these, the Hybrid Heuristic-Hill Climbing (H3C) manager, achieves significantly better performance than the oracle best static schedule and consumes less energy, with only minor performance degradations, relative to large private fabrics;

- *Fine-grained Intercore Communication* – We extend the SPL to facilitate fine-grained intercore communication, by which one thread can send data to one or more receiving threads. This fine-grained communication allows parallelization of applications that would not be possible with typically memory based communication;
- *Fine-grained Barrier Synchronization* – We augment the SPL to support fine-grained barrier synchronization. By performing barriers through the SPL, threads can synchronize much more frequently than possible via memory, allowing new, and more efficient, parallelization of algorithms;
- *Integrated Computation during Communication* – RACM interthread communication and barrier synchronization both allow computation to be performed on the data while it is in flight the the receiving cores. This provides optimization opportunities not possible with previous hardware communication techniques and can provide superlinear speedups compared to performing either computation or communication in isolation.

Overall, we show that, in the context of a heterogeneous CMP, RACM provides significant performance and energy benefits relative to what can be achieved by allocating the SPL area to either more or more powerful cores for a range of sequential and parallel applications. This work demonstrates that, with

the integration of ever more cores on die, integrating reconfigurability within some portion of the chip is indeed a worthwhile consideration.

## 6.1 Future Work

The analysis in this work focuses on a single design point in the reconfigurable computing space: a pipelined, row-based reconfigurable functional unit. Numerous other proposals [6, 20, 22, 34, 39, 64, 65, 68, 91, 92] have investigated various other points in the space. Each of these designs has their own set of advantages and disadvantages. Surveys such as [15, 37, 85] have provided a qualitative analysis of the trade-offs involved in the different design points. To our knowledge, however, no work has performed a unified, *quantitative* analysis of the design space in an attempt to analyze the strengths and weaknesses of the different designs relative to each other.

If reconfigurability is every to make its way into mainstream microprocessors, work is needed that quantitatively compares of a range of reconfigurable architectures, considering factors such as area, power, and breadth of applicability. This exploration should take into account such issues as

- Computation granularity - bit-, subword-, or word-level computation;
- Level of integration - functional unit, coprocessor, external coprocessor;
- Fabric width and depth - single word wide or multiple words wide, how many concurrent and sequential operations are supported;
- Number of fabric inputs and outputs;

- Can the fabric aid more than just pure computation - communication, fault tolerance, profiling, etc.;
- Amount of on-chip configuration storage;
- How the fabric is controlled - fine-grain direction by core or self control.

It is unclear whether it makes more sense to first perform the exploration for private fabrics and then expand the exploration to shared fabrics or to start immediately with shared fabrics. Different design points may require different, and possibly yet unexplored, styles of sharing. Coprocessor style reconfigurable fabrics, for example, which typically have their own control and often perform streaming type operations, may have to be shared at a coarser granularity than investigated for the SPL. The relative merits of different designs may shift when considering private versus shared usage. A design may have many advantages in a private context, but may offer limited sharing opportunities. Knowledge of these trade-offs will be important when determining what type of reconfigurability to include in future CMPs.

APPENDIX A

# DYNAMIC PARTITIONING MANAGER

In addition to the dynamic partitioning algorithms described in Chapter 4, we also investigated a Dynamic Partitioning Manager (DPM) that aims to find the best partitioning of a single SPL while minimizing energy consumption. To minimize energy consumption, the DPM power gates sections of the SPL so long as performance is not significantly degraded. In the end, the management policies from Chapter 4, the best of which was H3C, take a more holistic system view and so we concentrated on them. H3C and related policies, however, do not include any power reduction techniques, and so the power management portion of the DPM could provide a simple extension to something like H3C. The DPM would attempt to power down sections of the SPL once H3C had reached its final decision for a given phase. In this situation the DPM would only explore power gating the already established clusters and would not deal with partitioning the fabric any further.

## A.1 Dynamic Spatial Partitioning for Performance and Power

As with other shared resources, such as caches, sharing the SPL among multiple applications can lead to performance degradation. When two dual-threaded applications employing interthread communication share a fabric, one application may be producer-bound while the other is consumer-bound. In such a case, messages to the consumer of the latter thread could fill up the fabric as they wait for the slower consumer to process them. This would impede the progress of the producer-bound thread as it could not issue messages as quickly as if run in isolation due to frequent stalls waiting for queue slots to become available.

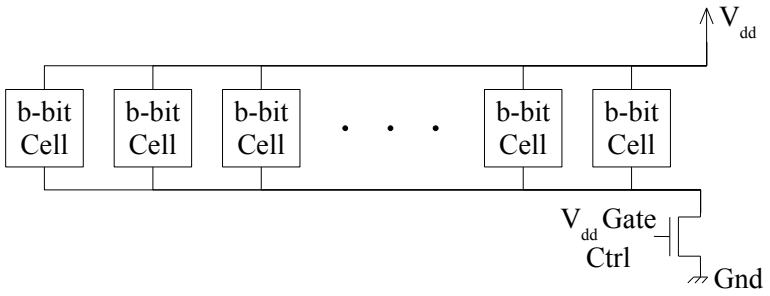



Figure A.1: SPL row with power gating support.

RACM addresses this shortcoming via spatial partitioning (described in Section 3.1.7) to create two virtual fabrics, one for each thread, to separate the consumer- and producer-bound threads. Splitting the fabric, however, is not always beneficial as it reduces the number of rows and therefore the number of queue slots available to each application, and may increase the amount of virtualization during computation.

Power gating is already in use in today's microprocessors [33, 66], and in many cases it can virtually eliminate the static power consumed by the gated circuit with minimal impact on active cycle time [59]. Figure A.1 shows a single row of the SPL with an NMOS gated-$V_{dd}$ transistor [59] that allows power gating the entire row. Since the DPM is responsible for both spatial partitioning and row power gating, we perform power-gating at the spatial partitioning granularity, i.e., one-quarter (6 rows), one-half (12 rows), or three-quarters (18 rows) of the SPL can be powered off.

### A.1.1 DPM Design

The DPM controls spatial partitioning and row power gating to minimize energy×delay (ED) while keeping the performance loss within some bound.

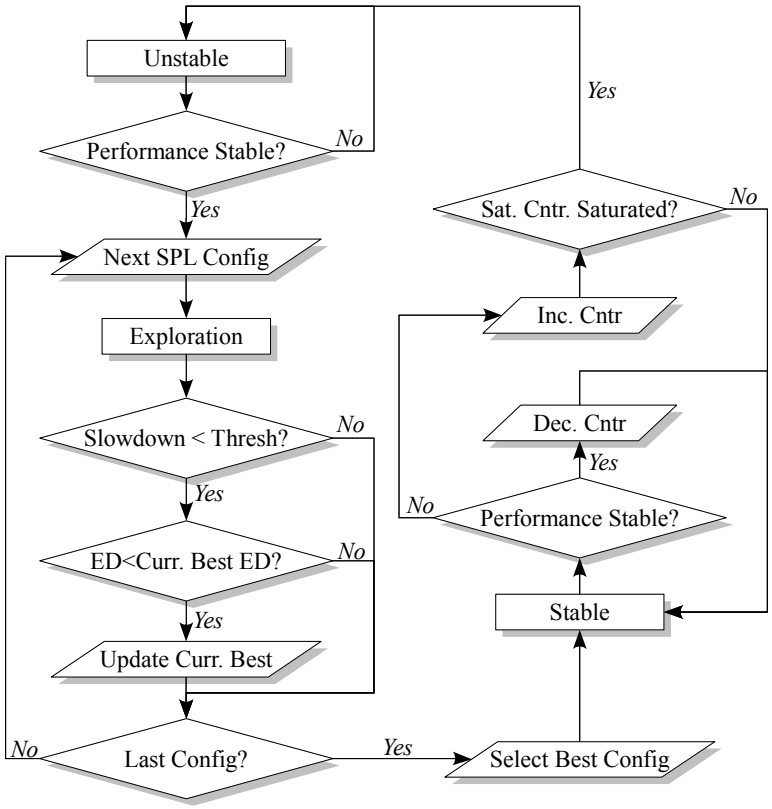



Figure A.2: Operation of the Dynamic Partitioning Manager.

With a 4-way shared fabric and four single-threaded applications, there are 28 possible configurations when fabric partitioning, power gating, and assignment of threads to partitions are taken into account. Although we focus on two dual-threaded applications in this section, where there are only seven possible configurations, we design the DPM for the general case of many more combinations.

The operation of the DPM is shown in Figure A.2. The DPM starts in the *Unstable* state waiting for the performance, as indicated by the number of instructions graduated during a fixed interval (one million cycles in our case), of all threads using the fabric to stabilize. Performance is considered stable if the relative number of instructions graduated between two successive intervals is within some percent, in our case 10%. Once performance is stable, the manager

moves into the *Exploration* state and begins exploring the different possible fabric configurations. Each configuration is run for a single interval and at the end of the interval the performance and ED are compared to the best configuration found so far. If the average slowdown relative to the baseline is less than some threshold (2% in our case), and the ED is better than the ED of the current best configuration, then the current configuration is set as the new best.

From the viewpoint of an entire program run, energy consumption typically increases when performance decreases as the program will execute longer. Within a fixed interval, however, energy consumption often decreases as performance decreases because less work is being done. To address this, we use power per instruction as a proxy for energy in our ED calculations. We assume that power can be estimated using performance counters, similar to what has been proposed in [44].

Due to the relatively large number of possible configurations, a full search of the configuration space during the exploration phase is undesirable. To reduce the number of explored configurations, we perform the exploration in two steps. In the first step, all spatial partitionings and thread assignments are explored. For four single-threaded workloads there are 11 possibilities (for two dual-threaded workloads there are only 2 possibilities, shared or split). At the end of this step, the configuration that produces the best ED is selected. In the second step, all possible power gatings of this partitioning are explored, for a maximum of three additional configurations. This yields 14 total intervals, or 5 with two dual-threaded workloads.

Once all configurations have been explored, the DPM configures the SPL to the best configuration and enters the *Stable* state. While in the stable state

no changes are made to the SPL configuration. To allow adaptation to phase changes, the DPM continues to monitor the relative performance between successive intervals. If the difference between two intervals exceeds the aforementioned stability threshold, then a 3-bit saturating counter is incremented and the last interval performance values are not updated. If the average performance falls within the stable range, the saturating counter is decremented. If the counter saturates, the DPM returns to the *Unstable* state and starts the process anew.

To split, merge or power on/off the SPL, instruction issue to the fabric is halted and all in-flight instructions are allowed to complete. The appropriate change to the fabric configuration is made and instructions are again allowed to issue.

## A.2 Results

Figure A.3 shows the energy consumption of *2Th+CompComm* (CC) with and without the DPM enabled relative to the single threaded baseline and breaks the energy into the energy consumed by the SPL and the energy consumed by the processor core (including L1 caches). Enabling the DPM (CC+DPM in the figure) reduces SPL energy consumption by more than 50% on average with no performance loss. As a whole, this leads to a 6.6% reduction in total power dissipation compared to the *2Th+CompComm* case without the DPM.

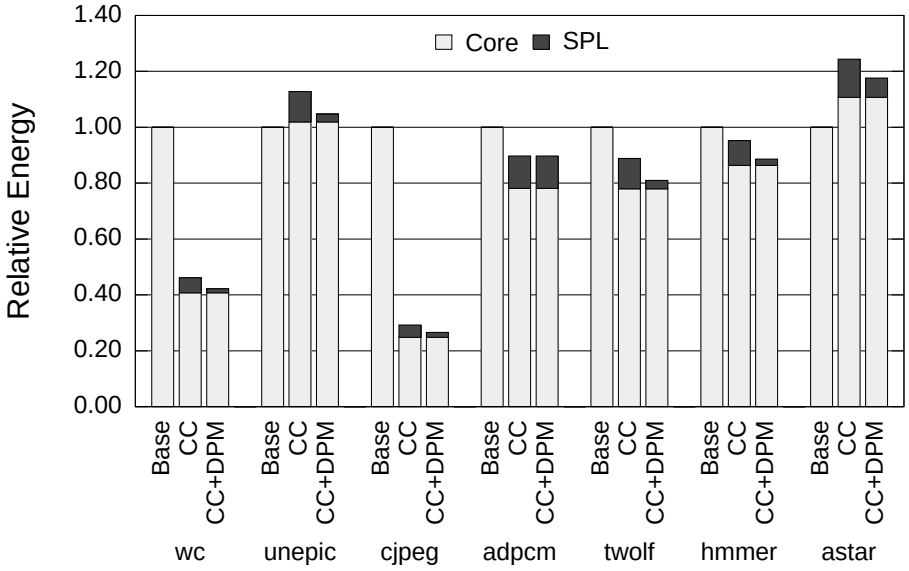



Figure A.3: Energy consumption relative to single threaded baseline.

### A.2.1 Multiapplication Workloads

With only a single application running, the DPM only has to consider power gating the SPL. When multiple applications concurrently share an SPL, the DPM must consider both fabric partitioning and power gating.

To analyze the performance of the DPM under multiapplication workloads we pair every communicating application from Section 5.3.1 with every other one, for a total of 21 different workload combinations. We run each pairing under three different configurations. In the first, the applications share the entire 24-row fabric. In the second, the fabric is statically partitioned and each application has access to 12 rows of private fabric. The final option employs the DPM to dynamically determine the best configuration to minimize energy while still obtaining good performance. The performance results for all pairings of *2Th+CompComm* (CC) workloads are shown in Figure A.4. The performance

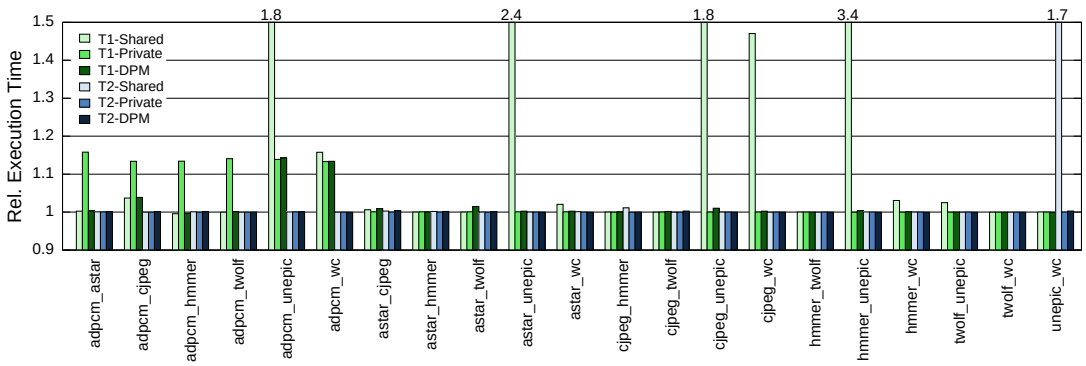



Figure A.4: Performance of two-application interthread communication+computation workloads with 24-row shared, 12-row private, and 24-row dynamically managed SPL, relative to 24-row private SPL per application. The first three bars refer to the first application in the pair and the second three bars refer to the second application.

of each application is normalized to the performance of that application running on a private 24-row SPL.

In almost a third of the pairings, one of the two threads experiences a significant slowdown (>40%) with no DPM. The DPM reduces the degradation relative to large private fabrics to less than 14% in all cases and less than 1% overall. In all cases, the dynamic scheme closely matches the performance of the better of the two static configurations, Shared or Private, with a performance degradation of less than 0.2% on average and a maximum degradation of 1.5%.

When two dual-threaded workloads share the fabric, there is typically less opportunity for power gating due to higher fabric demand. In cases where spatial partitioning is required to achieve good performance, the maximum possible fabric leakage reduction is 50% as at least two SPL sections must be left on, one for each partition. Despite this limitation, with DPM, SPL power consumption decreases by 27% on average, and up to 62%, over the best static configuration per pairing. This translates to total energy savings of 3% on average, and up to 7.9%, for the whole system (see Figure A.5 for details) with the aforemen-

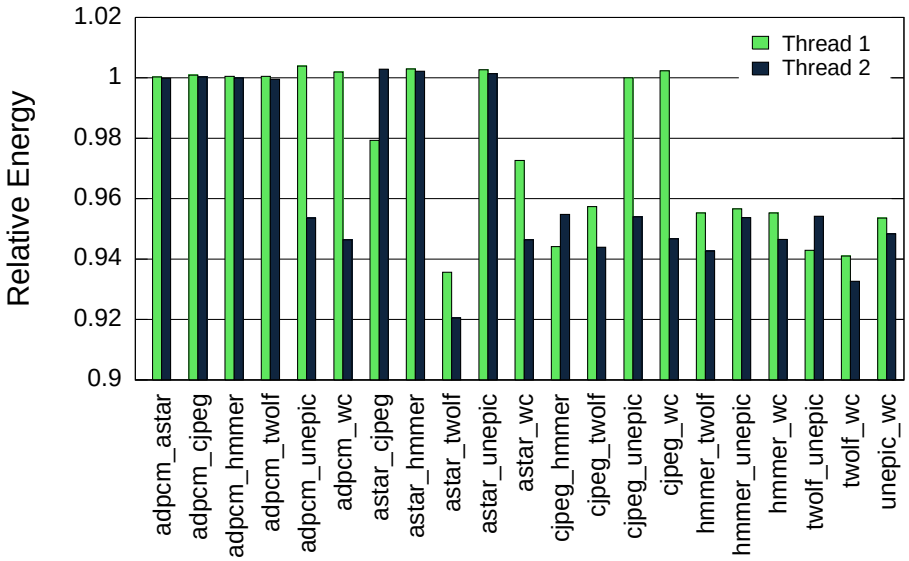



Figure A.5: Total energy consumption with DPM relative to best performing static partitioning per workload.

tioned negligible 0.2% average performance degradation. In some cases, fabric power savings with DPM exceeds 50%. The primary reason is that when the fabric is not partitioned, up to 75% of the fabric can be powered down. Also, by turning off rows, queued instructions have fewer queue slots to pass through, which saves dynamic energy.

## BIBLIOGRAPHY


[1] Advanced Micro Devices. AMD Athlon X2 Dual-Core Details. `http://www.amdcompare.com/us-en/desktop/details.aspx?opn=ADH2350IAA5DD`, 2007.

[2] A. Ahmadinia, C. Bobda, D. Koch, M. Majer, and J. Teich. Task Scheduling for Heterogeneous Reconfigurable Computers. *17th Symposium on Integrated Circuits and Systems Design*, pages 22–27, 7-11 Sept. 2004.

[3] Carl J. Beckmann and Constantine D. Polychronopoulos. Fast Barrier Synchronization Hardware. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 180–189, 1990.

[4] Mihai Budiu and Seth Copen Goldstein. Fast Compilation for Pipelined Reconfigurable Fabrics. In *Proc. 1999 ACM/SIGDA 7th Int'l Symposium on Field Programmable Gate Arrays*, pages 195–205, February 1999.

[5] C. Caşcaval, J.G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J.E. Moreira, K. Strauss, and H.S. Warren Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proc. 8th IEEE Symposium on High Performance Computer Architecture*, pages 311–321, 2002.

[6] S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas. Managing Pipeline–Reconfigurable FPGAs. In *Proc. 1998 ACM/SIGDA 6th Int'l Symposium on Field Programmable Gate Arrays*, pages 55–64, February 1998.

[7] T. Callahan, J.R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *Computer*, 33:62–69, April 2000.

[8] Jorge Carrillo and Paul Chow. The Effect of Reconfigurable Units in Superscalar Processors. In *Proc. 2001 ACM/SIGDA 9th Int'l Symposium on Field Programmable Gate Arrays*, pages 141–150, 2001.

[9] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *Proceedings of the 10th Int'l Workshop on Field-Programmable Logic and Applications*, pages 605–614, August 2000.

[10] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for

Constructive Cache Sharing on CMPs. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 105–115, 2007.

[11] Zhimin Chen, Richard Neil Pittman, and Alessandro Forin. Combining Multicore and Reconfigurable Instruction Set Extensions. In *Proc. 18th ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, pages 33–36, 2010.

[12] Sangyeun Cho and Lei Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 455–468, December 2006.

[13] Seungryul Choi and Donald Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proc. 33rd IEEE/ACM Int'l Symposium on Computer Architecture*, pages 239–251, 2006.

[14] Yuan Chou, Pazhani Pillai, Herman Schmit, and John Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Proc. IEEE/ACM 33nd Int'l Symposium on Microarchitecture*, pages 147–158, 2000.

[15] Katherine Compton and Scott Hauck. Reconfigurable Computing: a Survey of Systems and Software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[16] T. Constantinou, Y. Sazeides, P. Michaud, B. Fetis, and A. Seznec. Performance Implications of Single Thread Migration on a Chip Multi-Core. *SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[17] Convey Computer. The Convey HC-1 Computer. White Paper, November 2008.

[18] M. Dales. The Proteus Processor – A Conventional CPU with Reconfigurable Functionality. *Lecture Notes in Computer Science*, 1673:431–437, 1999.

[19] M. Dales. Initial Analysis of the Proteus Architecture. *Lecture Notes in Computer Science*, 2147:623–627, 2001.

[20] M. Dales. Managing a Reconfigurable Processor in a General Purpose Workstation Environment. In *Proc. of the Design, Automation, and Test in Europe Converence and Exhibition*, pages 980–985, 2003.

[21] Andre DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 31–39, 1994.

[22] Andre DeHon. DPGA Utilization and Application. In *Proc. 1996 ACM/SIGDA 4th Int'l Symposium on Field Programmable Gate Arrays*, pages 115–121, 1996.

[23] Ashutosh S. Dhodapkar and James E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proc. 29th IEEE/ACM Int'l Symposium on Computer Architecture*, volume 30, pages 233–244, 2002.

[24] A. El-Moursy, R. Garg, D.H. Albonesi, and S. Dwarkadas. Compatible Phase Co-Scheduling on a CMP of Multi-threaded Processors. In *Proc. of the 20th Int'l Parallel and Distributed Processing Symposium*, 2006.

[25] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX 2005 Annual Technical Conference*, pages 395–398, Berkeley, CA, 2005.

[26] Michael Feldman. FPGA Acceleration Gets a Boost from HP, Intel. *HPCWire*, September 2007.

[27] Michael Feldman. Reconfigurable Computing Prospects on the Rise. *HPCWire*, December 2008.

[28] Jason Fritts, Frederick Steiling, and Joseph Tucek. MediaBench II Video: Expediting the Next Generation of Video Systems Research. In *Proc. of The Int'l Society for Optical Engineering*, volume 5683, pages 79–93, 2005.

[29] W. Fu and K. Compton. An Execution Environment for Reconfigurable Computing. *Proc. 2005 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 149–158, April 2005.

[30] Grigori Fursin, John Cavazos, Michael O'Boyle, and Livier Temam. MiDataSets: Creating the Conditions for a More Realistic Evaluation of Iterative Optimization. In *Proceedings of the 2nd Int'l Conference on High Performance Embedded Architectures and Compilers*, pages 245–260, 2007.

[31] P. Garcia and K. Compton. A Reconfigurable Hardware Interface for a Modern Computing System. In *Proc. 2007 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 73–84, April 2007.

[32] Philip Garcia and Katherine Compton. Kernel Sharing on Reconfigurable Multiprocessor Systems. In *Int'l Conference on Field Programmable Technology*, pages 225–232, 2008.

[33] G. Gerosa, S. Curtis, M. D'Addeo, Bo Jiang, B. Kuttanna, F. Merchant, B. Patel, M. Taufique, and H. Samarchi. A Sub-1W to 2W Low-Power IA Processor for Mobile Internet Devices and Ultra-Mobile PCs in 45nm Hi-K Metal Gate CMOS. In *IEEE International Solid-State Circuits Conference, 2008. Digest of Technical Papers*, pages 256–611, February 2008.

[34] S. Goldstein, H. Schmit, M. Moe, Midhai Budiu, and Srihari Cadambi. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *Proc. 26th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 28–39, May 1999.

[35] Seth Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and R. Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *Computer*, 33:70–77, 2000.

[36] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proc. 10th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, October 2002.

[37] R. Hartenstein. A Decade of Reconfigurable Computing: A Visionary Retrospective. In *Proc. of the Conference on Design, Automation, and Test in Europe*, pages 642–649, 2001.

[38] S. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessor. In *Proc. 1998 ACM/SIGDA 6th Int'l Symposium on Field Programmable Gate Arrays*, pages 65–74, 1998.

[39] S. Hauck, T. W. Fry, M. M. Holser, and J. P. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12:207–217, February 2004.

[40] Scott Hauck, Matthew Hosler, and Thomas Fry. High-Performance Carry Chains for FPGA's. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8:138–147, 2000.

[41] John Hauser and John Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *Proc. 1996 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 12–21, 1997.

[42] Seongmoo Heo and Kriste Asanovic. Replacing Global Wires with an On-Chip Network: A Power Analysis. In *Proc. of the 2005 Int'l Symposium on Low Power Electronics and Design*, pages 369–374, 2005.

[43] *Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 Sequences*, 2007. Intel Datasheet: 313278-004.

[44] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proc. IEEE/ACM 36th Int'l Symposium on Microarchitecture*, pages 93–104, December 2003.

[45] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting Fine-Grain Thread Level Parallelism on the MIT Multi-ALU Processor. In *Proc. 25th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 306–317, June 1998.

[46] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. 13th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[47] P. Koka and M. H. Lipasti. Opportunities for Cache Friendly Process Scheduling. In *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[48] Rakesh Kumar, Norman Jouppi, and Dean Tullsen. Conjoined-core Chip Multiprocessing. In *Proc. IEEE/ACM 37th Annual Int'l Symposium on Microarchitecture*, pages 195–206, 2004.

[49] Rakesh Kumar, Dean Tullsen, and Norman Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proc. 15th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 23–32, 2006.

[50] I. Kuon and J. Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer–Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007.

[51] Chunho Lee, Miodrag Potkonjak, and William Mangione-Smith. MediaBench: A Tool for Evaluation and Synthesizing Multimedia and Communications Systems. In *Proc. IEEE/ACM 30th Int'l Symposium on Microarchitecture*, pages 330–335, 1997.

[52] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Deves. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Proc. of the IEEE Int'l Symposium on Workload Characterization*, pages 34–45, 2005.

[53] Yanbing Li, Tim Callahan, Ervan Darnell, Randolph Harr, Uday Kurkure, and Jon Stockwood. Hardware-software co-design of embedded reconfigurable architectures. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 507–512, 2000.

[54] Zhiyuan Li and Scott Hauck. Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. In *Proc. 2002 ACM/SIGDA 10th Int'l Symposium on Field Programmable Gate Arrays*, pages 187–195, 2002.

[55] Livermore Loops Coded in C. `http://www.netlib.org/benchmark/livermorec`, 2009.

[56] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth C. Goldstein, and Mihai Budiu. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proc. 12th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 163–174, 2006.

[57] G. Ottoni, R. Rangan, A. Stoler, and D.I. August. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proc. IEEE/ACM 38th Annual Int'l Symposium on Microarchitecture*, pages 105–118, November 2005.

[58] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proc. 12th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 244–255, Sept 2003.

[59] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and T. N. Vijaykumar. Gated-Vdd: a Circuit Technique to Reduce Leakage in Deep-

Submicron Cache Memories. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 90–95, 2000.

[60] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-Stage Decoupled Software Pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 114–123, April 2008.

[61] R. Rangan, N. Vachharajani, A. Stoler, G. Ottoni, D.I. August, and G.Z.N. Cai. Support for High-Frequency Streaming in CMPs. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 259–272, December 2006.

[62] R. Rangan, N. Vachharajani, M. Vachharajani, and D.I. August. Decoupled Software Pipelining with the Synchronization Array. In *Proc. 13th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 177–188, October 2004.

[63] R. Razdan, K.S. Brace, and M.D. Smith. PRISC Software Acceleration Techniques. In *Proc. of the 1994 IEEE Conference on Computer Design*, pages 145–149, October 1994.

[64] Rahul Razdan and Michael Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proc. IEEE/ACM 27th Int'l Symposium on Microarchitecture*, pages 172–180, 1994.

[65] Charle Rupp, Mark Landguth, Tim Garverick, Edson Gomersall, and Harry Holt. The NAPA Adaptive Processing Architecture. In *Proc. 1998 IEEE Sym. on Field–Programmable Custom Computing Machines*, pages 28–37, 1998.

[66] N. Sakran, M. Yuffe, M. Mehalel, J. Doweck, E. Knoll, and A. Kovacs. The Implementation of the 65nm Dual-Core 64b Merom Processor. In *IEEE International Solid-State Circuits Conference, 2007. Digest of Technical Papers*, pages 106–590, February 2007.

[67] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norman P. Jouppi, Mike Schlansker, and Brad Calder. Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers. In *Proc. IEEE/ACM 39th Annual Int'l Symposium on Microarchitecture*, pages 235–246, December 2006.

[68] S. Sawitzki, A. Gratz, and R. Spallek. CoMPARE: A Simple Reconfigurable Processor Architecture Exploiting Instruction Level Parallelism. In *The 5th Australasian Conference on Parallel and Real-Time Systems*, pages 213–224, 1998.

[69] Herman Schmit, David Whelihan, Andrew Tsai, Mathew Moe, Benjamin Levine, and R. Taylor. PipeRench: A Virtualized Programmable Datapath in 0.18 Micron Technology. In *Proc. of the IEEE 2002 Custom Integrated Circuits Conference*, pages 185–192, 2002.

[70] Semiconductor Industry Association. Int'l Technology Roadmap for Semiconductors, 2007.

[71] SESC Architectural Simulator. `http://sourceforge.net/projects/sesc`, 2007.

[72] Shisheng Shang and Kai Hwang. Distributed Hardwired Barrier Synchronization for Scalable Multiprocessor Clusters. *IEEE Trans. Parallel Distrib. Syst.*, 6(6):591–605, 1995.

[73] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase Tracking and Prediction. In *Proc. 30th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 336–349, June 2003.

[74] Steven Sinha, Peter Kamarchik, and Seth Goldstein. Tunable Fault Tolerance for Runtime Reconfigurable Architectures. In *Proc. 2000 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 185–192, 2000.

[75] L.A. Smith, J.M. Bull, and J. Obdrzálek. A Parallel Java Grande Benchmark Suite. In *Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, pages 8–17, 2001.

[76] Allan Snavely and Dean M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *Proc. 9th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 234–244, 2000.

[77] Standard Performance Evaluation Corporation. SPEC CPU Benchmark Suite. `http://www.specbench.org/cpu2000/`, 2000.

[78] Standard Performance Evaluation Corporation. SPEC CPU Benchmark Suite. `http://www.specbench.org/cpu2006/`, 2006.

[79] Stretch Inc. *The S6000 Family of Processors*, 2007.

[80] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *J. Supercomputing*, 28(1):7–26, 2004.

[81] Sun Microsystems, Inc. *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005*, March 2006.

[82] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the 2007 Conference on EuroSys*, pages 47–58, 2007.

[83] David Tarjan, Shyamkumar Thoziyoor, and Norman Jouppi. CACTI 4.0. *HP Technical Report*, 2006.

[84] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002.

[85] Timothy Todman, George Constantinides, Steve Wilton, Peter Cheung, Wayne Luk, and Oskar Mencer. Reconfigurable Computing: Architectures and Design Methods. *IEE Proc. – Computers and Digital Techniques*, 152(2):193–205, March 2005.

[86] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. 23rd IEEE/ACM Int'l Symposium on Computer Architecture*, pages 191–202, 1996.

[87] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative Decoupled Software Pipelining. In *Proc. 16th IEEE/ACM Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 49–59, September 2007.

[88] Srinivas N. Vadlamani and Stephen F. Jenks. The Synchronized Pipelined Parallelism Model. In *Proc. of the Parallel and Distributed Computer Systems Symposium*, 2004.

[89] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik". Orion: A Power-Performance Simulator for Interconnection Networks. In *Proc. IEEE/ACM 35th Int'l Symposium on Microarchitecture*, pages 294–305, November 2002.

[90] M. Wirthlin and B. Hutchins. A Dynamic Instruction Set Computer. In *Proc. 1995 IEEE Symposium on Field–Programmable Custom Computing Machines*, pages 99–107, 1995.

[91] M. Wirthlin and B. Hutchins. Sequencing Run-Time Reconfigured Hardware with Software. In *Proc. 1996 ACM/SIGDA 4th Int'l Symposium on Field Programmable Gate Arrays*, pages 122–128, April 1996.

[92] Ralph Wittig and Paul Chow. OneChip: An FPGA Processor With Reconfigurable Logic. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 126–135, 1996.

[93] Zhi Ye, Andreas Moshovos, Scott Hauck, and Prithviraj Banerjee. CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit. In *Proc. 27th IEEE/ACM Int'l Symposium on Computer Architecture*, pages 225–235, June 2000.

[94] Zhi Alex Ye, Nagaraj Shenoy, and Prithviraj Banerjee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proc. 2000 ACM/SIGDA 8th Int'l Symposium on Field Programmable Gate Arrays*, pages 95–100, February 2000.

[95] Sami Yehia, Nathan Clark, Scott Mahlke, and Krisztiàn Flautner. Exploring the Design Space of LUT-based Transparent Accelerators. In *Proceedings of the 2005 Int'l Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 11–21, 2005.