ABSTRACT

PATSILARAS, GEORGE. Leveraging Coupled Cores for Single-Threaded Application Performance. (Under the direction of Dr. James Tuck.)

The shift to multicores has resulted in a focus on parallel applications. At the same time, not all applications can benefit from parallelization techniques for two reasons. The first reason is the increased complexity of the hardware, which makes it difficult for the programmer to optimize for performance due to variable communication latency between cores. The second reason is due to not all applications having code which can be parallelized. For these applications the focus has been to reduce the long memory latencies.

At the same time, the projected amount of dark silicon (the fraction of the chip powered off) on a chip is growing. Hence, Asymmetric Multicore Processors (AMP) offer a unique opportunity to integrate many types of cores, each powered at different times, in order to optimize for different regions of execution. We design an AMP with both a core specialized for Memory Level Parallelism (MLP) and one specialized for Instruction Level Parallelism (ILP). We propose a hardware-level, application steering mechanism called Symbiotic Core Execution (SCE). SCE detects MLP phases by monitoring the L2 miss rate of the application, and it uses that information to steer the application to the best core. Interestingly, we show that L2 miss rates are important for deciding when an MLP region begins and when it ends. As a program runs, its execution migrates to a core customized for MLP during regions of high MLP; when the region ends, it is re-scheduled on the core that fits the application characteristics. This way we can utilize the shift to multicores and improve single-threaded applications.

On the other hand, historically, high level languages allow programmers to implement complex code. At the same time, architectural innovations over the last two decades targeting instruction level parallelism guaranteed that code would run faster. Now, with the shift to multicores, there is an added dimension caused by thread interaction and coordination that must be managed to get high performance. However, these interactions are translated to instructions
in such a way that their meaning is hard to glean from the hardware at run time. Hence, there is a wide gap between the programmers understanding and the resulting behavior at run time. In order to give programmers higher performance, hardware can be designed to better match the programmers abstraction.

To address this problem we propose an extension to the hardware/software interface of multicore processors called abstract parallel operations (APOs). We describe their properties and a design template for their implementation. To provide additional insights, we implemented a single-producer/ single-consumer queue as an APO to demonstrate our case and illustrate possible solutions to common and recurring problems. Our abstract queue appears robust to system level events, self-managed, virtualized, and optimized for performance.
Leveraging Coupled Cores for Single-Threaded Application Performance

by

George Patsilaras

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the Degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Gregory Byrd

Dr. Vincent Freeh

Dr. Eric Rotenberg

Dr. James Tuck

Chair of Advisory Committee
DEDICATION

To my parents... grandparents... sister... and soon to be wife...
George Patsilaras was born in Athens, Greece in 1983 to Vasilis Patsilaras and Maryanne. He earned his Bachelor’s degree in Electrical and Computer Engineering from North Carolina State University, in 2005, and his Master’s degree in Computer Engineering 2007. After that, he began working on his doctoral degree under the guidance of Prof. James Tuck, working in the areas of high-performance architectures for single threaded applications. During his graduate studies, he had the opportunity to intern at IBM, Raleigh, a number of times.
ACKNOWLEDGEMENTS

As my ten-ure as a student, correction eleven-year, journey at North Carolina State University comes to an end, I would like to thank many people for their help in getting by the happy and difficult moments.

I would like to thank my advisor for his help and patience to all my crazy ideas. I am also appreciative of my PhD committee members, Dr. Byrd, Dr. Rotenberg, and Dr. Freeh for their valuable feedback and guidance. Dr. Viniotis for his graduate school advise.

I would like to thank my father, mother and sister for the tremendous support and reminding me to believe in myself and keep on working hard.

My soon to be wife, Teri, for her support and patience.

I also acknowledge Rajeshwar Vanka, Sanghoon Lee, Ganesh Balakrishnan, Siddhartha Chhabra, Ahmad Samih, Niket Choudhary, Elliot Forbes, Sandeep Navada, Muawya Al-Ottom, Karthik Sundaramoorthy. The hours of brainstorming on ideas or complaining about the conference submission process has been a fun time.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>List of Tables</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
</tbody>
</table>

## Chapter 1 Introduction

1.1 Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores for Single-Threaded Performance  
1.2 Improving Fine-Grain Thread Level Parallelism Using Abstract Queue Operations  
1.3 Published Work  
1.4 Organization of Document  

## Chapter 2 Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores for Single Threaded Performance

2.1 Background  
2.1.1 Application Limitations for High Performance  
2.1.2 Effects of a Core’s Width on Cycle Time  
2.1.3 Checkpointed L2 Miss Processing with Value Prediction: $CLP+VP$  
2.1.4 Complexity  
2.1.5 Core Customization  
2.2 Motivation  
2.2.1 Effects of MLP Techniques on Different Cores  
2.2.2 Exploring MLP on Different Core Widths  
2.3 Symbiotic Core Execution  
2.3.1 Overall Architecture  
2.3.2 SCE Mode Scheduling  
2.3.3 Preferred Core Scheduling  
2.3.4 MLP Region Detection  
2.3.5 MLP Epoch Scheduling  
2.4 Evaluation of SCE  
2.4.1 Experimental Setup  
2.4.2 Power Modeling  
2.4.3 Performance Compared to Symmetric Multicore Processors  
2.4.4 Power and Energy Delay$^2$  
2.4.5 Performance Compared to Asymmetric Multicore Processors  
2.4.6 SCE Execution Analysis  
2.4.7 Sensitivity to Scheduling Parameters  
2.4.8 MLP Region Scheduling Benefits on $CLP+VP$  
2.5 Related Work  
2.5.1 Memory Level Parallelism  
2.5.2 Asymmetric Multicore Processors  
2.6 Discussion and Summary  

v
LIST OF TABLES

Table 2.1 Asymmetric Core Configurations ................................................. 16
Table 2.2 Core Details. Cycle counts are in processor cycles .......................... 27
Table 2.3 Core Configurations ....................................................................... 28
Table 2.4 Characterizing SCE: Total overhead of switching over entire execution,
    scheduling accuracy and value predictor accuracy ................................. 35
Table 2.5 Characterizing Execution in SCE Mode: Percent of execution spent on
    Core4 and Core2+CLP+VP in SCE mode .................................................. 36
Table 3.1 Applying Architectural Design Pattern to Abstract Operations .......... 46
Table 3.2 AQO Actions on Specific Coherence Actions .................................... 52
Table 3.3 Architecture simulated ................................................................... 59
Table 3.4 System configurations with various queues ..................................... 60
Table 3.5 Sensitivity of AQO to hardware queue size for DSWP benchmarks .... 64
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Multicore Abstraction Gap</td>
</tr>
<tr>
<td>2.1</td>
<td>Function definition in mcf</td>
</tr>
<tr>
<td>2.2</td>
<td>Scaling of Issue Queue in terms of delay</td>
</tr>
<tr>
<td>2.3</td>
<td>MLP regions</td>
</tr>
<tr>
<td>2.4</td>
<td>Virtual instruction window of CLP+VP</td>
</tr>
<tr>
<td>2.5</td>
<td>Performance of different core widths normalized to Core4</td>
</tr>
<tr>
<td>2.6</td>
<td>Performance versus MLP potential</td>
</tr>
<tr>
<td>2.7</td>
<td>Percent of Total Execution Spent in Different L2 Bins</td>
</tr>
<tr>
<td>2.8</td>
<td>Architecture of Tile and AMP</td>
</tr>
<tr>
<td>2.9</td>
<td>SCE scheduling algorithm</td>
</tr>
<tr>
<td>2.10</td>
<td>Responsive Switching to MLP Core</td>
</tr>
<tr>
<td>2.11</td>
<td>Lazy Switching to Preferred Core</td>
</tr>
<tr>
<td>2.12</td>
<td>SCE scheduling for MLP being the preferred core</td>
</tr>
<tr>
<td>2.13</td>
<td>Power breakdown of each component in a multicore</td>
</tr>
<tr>
<td>2.14</td>
<td>Speedup for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4</td>
</tr>
<tr>
<td>2.15</td>
<td>Power for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4</td>
</tr>
<tr>
<td>2.16</td>
<td>Energy Delay^2 for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4</td>
</tr>
<tr>
<td>2.17</td>
<td>Speedup for SCE all, SCE none, SCE4, and SCE imm normalized to SCE none</td>
</tr>
<tr>
<td>2.18</td>
<td>Sensitivity to scheduling parameters</td>
</tr>
<tr>
<td>2.19</td>
<td>Increase in executed instructions from CLP+VP vs benefit of MLP region scheduling</td>
</tr>
<tr>
<td>3.1</td>
<td>Scaling of Issue Queue in terms of delay</td>
</tr>
<tr>
<td>3.2</td>
<td>APO Design Pattern</td>
</tr>
<tr>
<td>3.3</td>
<td>Architectural Overview and Semantics for Hardware Queue</td>
</tr>
<tr>
<td>3.4</td>
<td>Queue Data Structure and Operations</td>
</tr>
<tr>
<td>3.5</td>
<td>Coordinating actions with the hardware queue using Red Zone Dequeueing</td>
</tr>
<tr>
<td>3.6</td>
<td>Performance of AQO with Unidirectional Micro-Benchmark</td>
</tr>
<tr>
<td>3.7</td>
<td>Speedup of AQO using DSWP over HAQu</td>
</tr>
<tr>
<td>3.8</td>
<td>Normalized L2 Cache accesses</td>
</tr>
<tr>
<td>3.9</td>
<td>Normalized Energy consumption</td>
</tr>
<tr>
<td>3.10</td>
<td>Breakdown of enqueueing operations serviced</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The last two decades of single thread performance scaling have been a golden era for computer architecture. In this period, programmers focused on implementing correct programs using high level languages and relied heavily on advances at the architectural level and below to transparently speedup their code from one processor generation to the next. As frequency scaling has slowed down due to power constraints, so has the increase in single-threaded performance. Until now, clock frequency performance was predominantly increased linearly depending on the memory footprint of applications.

Chip Multicore Processors (CMPs) have superseded large monolithic processors due to the latter’s excessive design complexity and high power consumption. As a result, most CMP designs have focused on increasing the number of cores and on re-organizing the memory hierarchy. This reflects a bias toward parallel and multi-programmed workloads, and deemphasizes single-threaded performance concerns.

Despite the shift to parallel workloads, single-threaded performance is still important. As a result many techniques have focused on automatically parallelizing single-threaded applications or using hardware which is specialized and more power efficient to execute single-threaded applications. Decoupled Software Pipelining (DSWP)[38] is a technique to automatically parallelize single-threaded applications by creating threads with independent work and executing
them concurrently. The more threads that overlap, the larger the performance gains. However, not all single-threaded applications can be parallelized. For those applications that cannot be parallelized, specialized, asymmetric multicore processors (AMPs) can improve performance by executing different regions of code on the best suited cores.

DSWP attempts to exploit fine-grain parallelism. At the same time, performance challenges remain, primarily due to communication between cores being hampered by 10s of cycles of delay through the interconnect and cache coherence protocol. This problem increases as we add levels of cache to the cache hierarchy, despite processors being located side-by-side within a few clock cycles (of wire delay).

AMPs have been shown to improve power efficiency for single-threaded applications. However, performance improvements remain challenging, primarily due to two runtime decisions. The first decision is the need of AMPs to know which core is best for a certain region. The second decision is the need to know how to identify different phases to execute on the available cores.

This dissertation presents two hardware optimizations to improve performance and energy efficiency in current and future CMPs for single-threaded applications. The first optimization is Symbiotic Core Execution (SCE). The second optimization is Abstract Queues (AQ).

1.1 Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores for Single-Threaded Performance

As CMPs increase their number of cores, a large portion of the chip will have to remain powered off [13] at any given time, due to power limitations. This means that architects should not envision designs in which every transistor remains turned on permanently, but rather ones in which portions of the chip are judiciously turned on/off depending on the characteristics of a workload. For example, a specialized processing core could be off most of the time and only enabled when it is really needed. Hence, there is growing interest in Asymmetric Multi-
core Processors (AMPs). AMPs are typically composed of single-ISA, heterogeneous cores with varied microarchitectural features. AMPs offer a unique opportunity to speedup single threads by specializing cores for distinct phases of many different applications. This comes with some caveats which include: 1) how to select which core out of a selection is the highest performing core and 2) when to switch to this preferred core.

Memory level parallelism (MLP) has been proposed as a way to boost the performance of applications that stall frequently due to misses in the last level cache (LLC). Rather than waiting for one access at a time, the goal of MLP techniques is to exploit available memory bandwidth to request many memory accesses at once. A variety of hardware-only MLP enhancing techniques have been proposed [10, 32, 14, 34, 12, 5, 22, 56, 26, 49, 55]. These techniques are advantageous since they can transparently accelerate sequential codes, but they are limited by their high energy consumption. Some of these techniques leverage precomputation to issue loads in advance of when they are needed. Other MLP techniques leverage hardware within a single core to detect a long latency load, then speculate past the load to generate overlapping misses [14, 5, 22, 56]. Multi-threaded approaches have been considered which automatically construct prefetching threads [14], or tightly couple two cores to act as a large instruction window [55]. Given the importance of tolerating long latencies to memory within a single thread, future processors will likely incorporate techniques to overlap long latency misses.

AMPs are an interesting design space in which to consider the addition of such MLP techniques. Since many applications have instruction level parallelism (ILP) and MLP phases during their execution, cores on an AMP could be leveraged to exploit the behavioral differences among these phases. Furthermore, the power and design complexity of new architectural features to exploit MLP can be mitigated by designing a custom core for MLP.

We study the behavioral characteristics of high MLP code and determine that it can benefit from core customization. This leads to design a new system called Symbiotic Core Execution (SCE), in which at least one core is customized for the characteristics of code with high ILP and another core is customized for the characteristics of higher MLP. With the help of transparent
and judicious scheduling mechanisms in hardware, regions of high MLP can be exploited by the characteristics of the MLP-customized core; when ILP dominates, a preferred core is used.

The contributions of this work are:

- Characterization of the performance benefits of core customization on an AMP for MLP regions. Specifically, the analysis of L2 miss rates to compare program regions with similar behavior on different core designs. Contrary to conventional wisdom, even with frequent L2 misses, frequency and core width combined are an important design characteristic. In this space, a narrow core has definitive performance benefits over a wider core in sequential program regions.

- Symbiotic Core Execution (SCE), a technique which benefits from new opportunities created by an increasing fraction of dark silicon on future chips. By using an AMP with one core customized for ILP and one specialized for regions of high MLP, we are able to exploit the fine-grained phase behavior of applications while powering off the unused core.

- A hierarchical, hardware-level scheduling mechanism designed for SCE, which judiciously switches cores to exploit regions of high MLP on the customized MLP core without incurring a high switching overhead. We also detect when the MLP region ends, in order to switch back to the preferred core at an appropriate time. We show it is sufficient to look at L2 miss rates in order to identify MLP regions.

Our approach provides an average improvement of 5.3% and 6.6% for SPEC2000 and SPEC2006 applications, respectively, with a max speedup of 14.5%, compared to implementing the MLP technique on a single core. Our findings yield an important message for designing AMPs with specialized cores: core customization enables efficient exploitation of MLP, and application steering mechanisms for MLP are simple to implement and effective.
1.2 Improving Fine-Grain Thread Level Parallelism Using Abstract Queue Operations

In the past, the abstraction between language and architecture was good enough that programmers did not need to understand how deep pipelines with out-of-order execution worked to get better performance. With the shift to multicore, such transparent support for performance scaling is more challenging. Even if we look beyond the problem of parallelizing programs and assume that programs will be multi-threaded, serious challenges remain for providing transparent performance improvements.

In particular, there is an added dimension of difficulty caused by thread interaction and coordination that must be managed to achieve efficient execution. However, current multicores do not capture these interactions explicitly. Instead, they are decomposed into sequences of instructions whose meaning is tough to decipher at runtime. Consider the case of a queue, as shown in Figure 1.1(a); the abstract meaning to the programmer is clear, but the hardware sees a sequence of instructions involving accesses to shared variables, atomic operations, and possibly fence instructions [21, 25, 19, 27, 31]. Without a significant investment of effort, the hardware cannot understand the kind of interaction taking place.

Recent proposals have addressed some of these problems with efficient abstract operators in hardware [36, 37, 43, 46, 52]. Fig. 1.1(c) shows an example for hardware queues. However, such proposals are usually ad hoc solutions, and often do not provide an effective abstraction. For example, management of chip-level resources within a program or across multi-programmed workloads are left to the programmer to solve some other way; also, system level events, like context switches, are often unsupported or require OS changes. In these cases, the job of the system programmer gets harder instead of easier.

We refer to the discrepancy in abstraction as the multicore abstraction gap. The abstract understanding of the programmer, as it relates to critical multi-threading operations, is far removed from the hardware’s implementation of them. This abstraction gap leads to a couple
of problems. First, programmers have a difficult time designing algorithms that achieve high performance on a wide variety of multicore architectures and in a wide variety of applications. As cache hierarchies, instruction sets, and on-chip interconnects change, so will the ideal implementation of hardware. The second problem is faced by chip architects. Without a better way to proceed, chip designers are forced to optimize a set of hardware operations that are far removed from the programmer’s abstraction. While these changes are beneficial, such innovations may not lead to clear benefits for the programmer. Instead, software may need to be changed to fully exploit their potential. Therefore, we need to narrow the abstraction gap so that programmers can leverage an efficient and high performing abstraction and architects have a clear and direct framework for innovation.

In order to narrow the gap, we propose a general architectural design pattern for abstract parallel operations (APOs). We provide to the programmer an interface for an abstract parallel operation such as queues in the ISA, but at the same time hide implementation details as shown in Figure 1.1(b). Within this abstraction layer, our design pattern informs the
chip architect how to tolerate system level events (i.e. context switches), how to manage limited
hardware resources, and how to account for thread interactions. Our approach manages limited
chip resources by allowing instructions in the APO interface to execute in either hardware or
software. Robust support of system level events is provided by interpreting the meaning of
cache coherence operations on APO structures. Finally, it provides a coordinated strategy for
managing thread interaction by arbitrating critical resources transparently in hardware. With
this pattern in place, it is easier for architects to design robust APOs.

To illustrate our design pattern, a design of a single-producer/single-consumer hardware
queue is illustrated, which is called an abstract queue operation (AQO). From this detailed
design, we have discovered that the size, number, and topology of hardware queues can be
abstracted away from the programmer with very little overhead and with no OS level support.
This means that the programmer need not be aware that they are using hardware queues at all.
Therefore, the architect can directly improve the queue’s behavior because it is an architected
concept.

Overall, our work makes the following contributions:

• We are the first to propose a general architectural design pattern for integrating abstract
queue operations into multicore architectures. Also, we describe important features of our
design pattern, such as how to tolerate system level events (i.e. context switches), how to
manage limited hardware resources, and how to account for thread interactions.

• We explain how to apply our pattern to single-produce/single-consumer queues.

• Using our pattern and by building on top of a recently proposed hardware accelerated
queue (HAQu [28]), we implement an abstract hardware queue in detail; as a result, we
discover important novel features of abstract hardware queues.

We evaluate our design in the context of micro-benchmarks and automatic, fine-grained
parallelization using decoupled software pipelining(DSWP) on SPECint applications. Compared
to a hardware-accelerated queue communicating through memory, our design obtained speedups of 7.9× on our micro-benchmark and 11% on average for our DSWP parallelized codes.

1.3 Published Work

Aspects of this dissertation have been already published.

- **Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores in the Dark Silicon Era**, published in January 2012 in ACM Transactions of Architecture and Code Optimization (TACO) special issue on High-Performance and Embedded Architectures and compilers. This work focuses on using asymmetric core configuration to exploit memory level parallelism.

1.4 Organization of Document

The remainder of this dissertation is organized as follows. Chapter 2 presents how to efficiently exploiting memory level parallelism on asymmetric coupled cores for single threaded performance. Chapter 3 presents how to transparently improve fine-grain thread level parallelism using abstract queue operations. Finally, Chapter 4 concludes.
Chapter 2

Efficiently Exploiting Memory Level Parallelism on Asymmetric Coupled Cores for Single Threaded Performance

This chapter explores coupling two specialized cores – one for ILP, and one for MLP – to improve performance of single-threaded applications. The chapter is organized as follows. Section 2.1 gives some background on MLP techniques and AMP benefits. Section 2.2 presents a detailed study of different core designs for varying levels of MLP. Section 2.3 describes our Symbiotic Core Execution. Section 2.4 discusses our methodology and provides a detailed evaluation. Section 2.5 is devoted to related work, and Section 2.6 summarizes the chapters contributions.
... 
while( arcin ){ 
    tail = arcin->tail; 
    if( tail->time + arcin->org_cost > latest ) 
    { 
        arcin = (arc_t *)tail->mark; 
        continue; 
    } 
    red_cost = arc_cost - tail->potential + head_potential; 
    if( red_cost < 0 ){  
        ... 
    }  
    arcin = (arc_t *)tail->mark; 
} 

Figure 2.1: Function price_out_impl( network_t *net) in mcf

2.1 Background

2.1.1 Application Limitations for High Performance

Last level cache misses (called L2 misses here on) are an important factor affecting performance of applications. Pointer-chasing, large working sets, or frequent changes in phases can result in high L2 misses. Figure 2.1 shows a code fragment from mcf, an application with lots of L2 misses. This code represents a pointer-chasing loop dominated by long-latency memory operations. Typically, a large instruction window can help L2 misses, In this code, the instructions following the load depend upon it. Even though an out-of-order processor can continuing executing instructions while the miss is handled, the lack of independent instructions prevent it from doing so.
2.1.2 Effects of a Core’s Width on Cycle Time

A large instruction window along with a complex microarchitecture might enhance IPC, but at the same time it could increase the clock period. For instance, increasing the size of the issue window and issue width can boost IPC for applications with abundant ILP. However, that may not translate into higher performance because the clock period may increase to accommodate the larger content addressable memory and deeper select tree.

Fabscalar [9], is a state of the art tool that enables architects to synthesize customized designs and evaluate the effects of different designs in terms of frequency, area and power. Using Fabscalar, we can synthesize a Verilog model of an arbitrary superscalar processor and analyze how the size of key processor structures can affect frequency. Figure 2.2 shows the impact of increasing the issue window on the delay of the wakeup-select logic for different issue widths. These assume a 45nm technology and the same input voltage. As we can see from the graph, the smaller the width and issue queue size, the faster the clock frequency can be.

We used Fabscalar to perform a design space exploration on different key structures of processors (ROB size, LSQ Unit, Core Width etc). Our search focused on identifying cores with
the highest frequency with issue widths of one, two, and four. In our search, we assumed a pipelined architecture with a constant depth and fixed supply voltage, then we varied the issue width and all related microarchitectural structures. We found that the 4-wide core’s maximum frequency was 1.66GHz, the 2-wide core frequency was 2.22GHz and the 1-wide core had a maximum frequency of 2.5GHz (more architectural details can be found in Section 2.4.1). The design search considers all possible timing critical paths of a modern superscalar out-of-order processor, for example wake-select logic, rename logic, cache access time etc. [39], in order to synthesize a processor with a realistic clock frequency. Since we keep the total pipeline depth constant for our exploration, the synthesized core reflects the trade-off between the pipeline complexity and the propagation delay. As in any design space exploration, it is important to search for an appropriate set of designs. Fabscalar considers many circuit-level and architecture-level optimizations, although it is not exhaustive. Therefore, a design team could find other core designs our search did not consider. In general, however, attempts to increase the frequency through microarchitectural complexity do not always have the expected effect on performance, area, or power consumption. In the rest of the chapter, we use the core designs found by Fabscalar.

2.1.3 Checkpointed L2 Miss Processing with Value Prediction: CLP+VP

Another way to virtually increase the instruction window is by using lookahead-like paradigms, such as Runahead Execution [33], or CAVA [5], which try to speculatively extract high amounts of MLP. Our basic mechanism, shown in Figure 2.3(a) adopts some features from CAVA [5] and Clear [22]. Once an L2 miss reaches the head of the ROB, a checkpoint of the register file is recorded at the point just before the load. Then, we place a predicted value in the destination register of the load and continue execution as a speculative epoch. By retiring the load with a speculative value, forward progress can be made while waiting for the memory operation to complete. Once the L2 miss completes, the processor checks the actual value with the predicted one and exits the speculative epoch if the prediction was correct. If the load’s value
was incorrect, execution is restored to the checkpoint, the value predictor tables are updated, and the processor resumes from the mispredicted load instruction. In order to avoid recovery in the case of a successful prediction, the L1 cache buffers the speculative state [5]. We will refer to this basic strategy as Checkpointed L2 Miss Processing with Value Prediction (CLP+VP). CLP+VP supports making predictions on multiple outstanding loads, but only one checkpoint is kept, so any misprediction results in rolling back the entire speculative epoch back to the first predicted load value. Therefore, we adopt CAVA’s mechanisms to prevent aggressive speculative execution through loads which are value predicted with low confidence.

Figure 2.3(b) shows the CLP+VP architecture with the required Speculative Data Cache [5]. In addition, the OPB is the Outstanding Prediction Buffer. It tracks all outstanding predictions, and if the prediction does not match, the Recovery and Rollback logic is triggered.

2.1.4 Complexity

Design complexity is not easy to measure. Single core techniques are preferable for design complexity since modifications are constrained to a single core. However, even single core modifications are increasingly costly in terms of complexity, use of die area, and power as the transistor counts of cores continue to creep up. Previous research has estimated complexity[3] for different components of a processor.

A good approximation is the number of transistors. The more the transistors added for a
technique, the higher the design complexity will be. This assumes a linear relationship between verification time and transistor count. As such AMP cores offer the unique opportunity to reduce the complexity of a design by implementing a technique on a simpler/smaller core rather than a more complex core. For our cores studied in Section 2.1.2 we can assume that adding CLP+VP on a 2-wide smaller core will have a smaller design complexity than having to add the same technique with larger structures on a 4-wide core.

2.1.5 Core Customization

Core customization is an attractive way of utilizing the real estate of CMPs due to application diversity. Previous research has focused on exploiting applications’ diversity given different cores [23]; however, there have been few studies on core preference for a given architectural technique. Looking at problems such as pointer-chasing and solutions like CLP+VP a question someone might have is what impact this has on core selection for an AMP. There are many factors that make this analysis non-trivial and difficult to anticipate.

- The trade-off between core-width and frequency is largely driven by ILP. While MLP and ILP are not strictly correlated, conventional wisdom argues that MLP tends to come in regions of lower ILP. Hence, this suggests narrower cores may be better.

- Latencies to memory are fixed regardless of frequency. Therefore, the frequency of the core may not matter since the execution time of MLP regions may be ultimately dominated by latency to memory. Hence, the fixed latency to memory suggests that an average sized core — in other words, nothing too fast, and nothing too slow — will work just as well as anything else.

- Finally, the accuracy of techniques like CLP+VP to speculate over L2 misses using predicted values will determine how much MLP is exploited. If misspeculations are infrequent, CLP+VP behaves like a very large window, which could favor wider cores. However, if misspeculations are frequent or if regions executed speculatively tend to have lower ILP,
Through detailed analysis in the next section, we hope to better understand these factors.

2.2 Motivation

2.2.1 Effects of MLP Techniques on Different Cores

One of the benefits of \textit{CLP+VP} is that it achieves a larger virtual instruction window by speculating over a L2 miss. Figure 2.4 shows the effect that this will have for a 4-wide core and a 2-wide core. The dark solid line cutting the instruction window shows the actual instruction window and how the virtual instruction window will be larger. Regardless of the width both cores will achieve a large virtual instruction window.

When comparing a 4-wide and a 2-wide core, we can assume that if there are no branches or dependent instructions the 4-wide core would achieve a larger virtual instruction window. On the other hand, due to instructions’ dependencies, the virtual instruction window of the 2-wide core can end up being larger due to its faster cycle time. As a result, a 2-wide core, besides issuing L2 misses earlier, can also issue more L2 instructions in the same time of execution. In the next section, we perform a study to determine which behavior is more prevalent for regions of high and low MLP.
Another benefit for a faster clock frequency is when CLP+VP mispredicts a value. In that case, the processor needs to rollback to the checkpointed state and re-execute the instructions. This is done more quickly on a 2-wide core giving a faster recovery time.

### 2.2.2 Exploring MLP on Different Core Widths

In this section, we investigate the behavior of different core designs, shown in Table 2.1, in code regions sensitive to MLP techniques. Our goal is to establish a relationship between core performance and suitability for exploiting MLP on the same code region. To establish enough data points to draw a strong conclusion about potential for MLP and core preference, we extract code regions of 10K instructions from a dynamic trace of the SPEC CPU 2000 applications and categorize these regions according to their L2 miss rate. Hence, for a wide range of applications, we can average out behavior based solely on the potential for MLP.

Figure 2.5 plots the performance of the Core2, Core1, Core4+CLP+VP, Core2+CLP+VP, and Core1+CLP+VP designs normalized to the Core4 design. The dashed lines show all the cores without CLP+VP, while solid lines are taking CLP+VP into account. Each bin on the x-axis indicates a different L2 miss rate (L2 misses/10K instructions). Each point on the y-axis is the speedup over Core4 for all code regions with a specific L2 miss rate. Since these cores have no additional MLP technique, they are only exploiting the MLP available from out-of-order execution.

In Figure 2.5 we can identify three regions. The first region, identified as ILP + low MLP,
has no to very few misses. Most high ILP applications spend the majority of their execution in this region (98\% for ammp). In this region 4-wide core designs have a clear advantage due the high ILP exploited by the width of the cores. The second region is identified as Medium ILP & MLP. In this region we see that ILP is diminished somewhat by the increasing number of L2 misses. Applications executing on the 4-wide cores (Core4 and Core4+CLP+VP) are faster than Core1, and Core1+CLP+VP. However, we also see that designs Core2, and Core2+CLP+VP are competitive. When comparing the 2-wide core designs to the 4-wide cores, we can consider the 2-wide cores a better design due to the power savings of a smaller core (for an explanation on how higher frequency can save on power see section 2.4).

The third region identified as MLP + Low ILP is the most interesting region. We see that the 2-wide core designs gain a slight advantage. In Figure 2.6b we show the actual IPC and instructions per second (IPS) for the 100 L2 miss bin. These regions have high amounts of L2 misses per sampled region. What is also interesting is that here we see clearly the core width versus core frequency trade-off.

For the third region, L2 miss rates are high. One might expect L2 misses to dominate the
execution time (regular instructions would complete while misses are being serviced), and since the time to access main memory is constant, the cores should perform the same. However, this is not the case. Figure 2.6a shows on the x-axis the distance between issued instructions which miss in the L2. On the y-axis we have the total percentage of load instructions which have an issue distance of 'x' or less from a previous L2 miss. The three lines represent different L2 miss rate bins from Figure 2.5. From this graph, it is clear that regardless of the amount of L2 misses in every 10K bin, L2 misses cluster close together. This clustering is important for two reasons. First, it explains why MLP techniques can benefit sequential performance. Second, clustering shows us that execution time for these bins is actually composed of memory access regions and non-memory access regions. The conventional wisdom fails to account for application and core-level effects in regions of high L2 miss rates, however, our analysis does account for them. Therefore, we see that the MLP+Low ILP region can exploit asymmetry.

Figure 2.7 shows the percent each benchmark spends in different L2 miss bins. As we can see high ILP applications such as bzip2 spend close to 99% of their execution with no misses or very few, while mcf spends nearly half of its execution (45%) in the 500 L2 miss bin. Overall we see that some bins might not have equal weight and could skew results, however as we can see in Section 2.4 our generalization achieves performance resulting from these trends.
2.3 Symbiotic Core Execution

The goal is to design an AMP which can identify MLP regions on the fly and schedule execution to the best available core. To do this, we propose Symbiotic Core Execution (SCE). The term symbiosis is borrowed from biology: two self sufficient organisms exist symbiotically when they survive better together than when alone. Symbiosis is a compelling description because it identifies the cores as being independent and capable of working alone, as is often needed in a multiprocessing environment. However, when necessary, they can be used together for a greater performance advantage.

SCE is based on AMP architectures and leverages a loose coupling of two asymmetric cores. In our design, one core will provide higher ILP, and the other core will be customized for MLP. Sequential performance is improved by assigning fine-grained phases of an application to the appropriate core. A key challenge of SCE is efficient fine-grained scheduling. In the rest of this section, we provide an overview of the SCE architecture and a description of our scheduling
2.3.1 Overall Architecture

SCE’s mechanism is implemented on a homogeneous ISA, heterogeneous tile CMP. Figure 2.8(a) shows an example SCE tile and Figure 2.8(b) shows the homogeneous-tile CMP implementation. Due to power constraints and the large areas of a chip that will have to remain powered off in the near future [13], we can power on only one core of the tile at a given time. We refer to this as SCE mode. In SCE mode, the OS sees one logical core per tile, and hardware can initiate fast context switching among the symbiotic cores. In the case where the number of threads is more than the number of cores per tile, we assume that SCE is disabled, and the chip can enable different combinations of types of cores to fit applications’ needs for a given power budget. Scheduling applications in this case would be similar to previous work such as [23].

Since SCE mode (one core per tile powered at a time) is our new support, we will primarily focus on how it works and will not discuss the different combinations of types of cores that could fit to a given power budget. In SCE mode, a scheduling algorithm designed to exploit the MLP core will determine when a core switch is needed. At that time, it stalls the fetch engine in preparation of collecting the program’s context. Once the L2 miss reaches the head of
the reorder buffer, the processor squashes the remaining instructions, and performs the context switch between the cores by copying the register file from one core to the next. We design the context switch to begin as soon as the head of the ROB stalls by a long latency instruction. That way, the context switch delay is overlapped by the servicing of an L2 miss. Once the context switch is finished, execution is resumed. If we switched because of a long latency instruction at the head of the re-order buffer, then that instruction is re-executed.

Our hardware level context switch happens only during SCE mode. Furthermore, the idle core will always be available, so our context switch mechanism can be fast. Even though we have hardware support for switching, many overheads may be incurred on a switch, such as cache warm-up penalty, cache invalidations, effects on the TLB, and poor branch predictor history. These effects are one reason why building an efficient SCE scheduler is important.

2.3.2 SCE Mode Scheduling

Our goal is for SCE mode scheduling to work well for a wide range of applications but especially for those that contain significant amounts of MLP. This is challenging because there are hierarchical, competing concerns. At the Core-level, we do not know a priori which node will usually provide the best performance. We must determine a preferred core on the fly. Furthermore, this preferred core can change throughout execution. As a result we must adapt to the coarse-grained phase of the running application. In addition, at the fine-granularity of MLP phases, we will switch to the MLP core to exploit its MLP technique, and we must switch back to the preferred core in regions of low ILP. Failure to identify a good start or end to the MLP region could take the application away from its preferred core for too long. Finally, even within MLP epochs, the MLP technique should be used only when it can actually provide a performance benefit. A good scheduling algorithm will manage all of these concerns.

In the rest of this section, we describe our hierarchical scheduling strategy in three parts: Preferred Core scheduling, MLP Region Detection, and MLP Epoch Scheduling.
2.3.3 Preferred Core Scheduling

Figure 2.9 shows pseudocode for our overall core scheduling strategy. Because we wish to choose the best core even when not in an MLP region, much of the code is concerned with picking the preferred core. Overall, the algorithm is divided into training mode and non-training mode.

Every 50 million cycles (25-30ms depending on the core), our algorithm enters training mode and picks a new preferred core. A typical operating system time slice would range from 5ms to 100ms [29]. Scheduling at the operating system time slice enables the scheduling mechanism to capture possible context switches and the effects they can have on performance. We also selected 50 million cycles in order to have a large enough period for our preferred core scheduling policy to not execute frequently while at the same time small enough to allow for many complex phase behaviors to be seen. Reducing the frequency would also have an impact on executing on an incorrect core for a period of time. An alternative to this scheduling mode could be using a phase change detection mechanism similar to [47].

The first if-condition controls the actions during training mode. For each core in the AMP tile, we run for a minimum of 100K cycles and record the instructions per second (IPS) in a hardware register. Note, we must use IPS to account for different operating frequencies. Once all cores in the tile have been trained, we select the core with the highest IPS. Note that we do not disable the MLP technique when training on the MLP core. While this may distort the IPS compared to running without the MLP technique, it usually does not alter the preferred cored decision.

When not in training mode, non-MLP regions will run on the preferred core by default until an MLP region is encountered. When we enter an MLP region, execution switches to the MLP core. As long as the MLP region is active, execution remains there. However, once the MLP region ends, execution returns again to the preferred core.

Overall, we expect this scheduling algorithm to work well for a wide range of programs. Since it retrains the preferred core on the AMP every 50 million cycles of execution, it is likely to adapt to changing program behavior whether or not MLP regions occur. Furthermore, if an
while(1) {
    if (in training period) {
        if (execution was on both cores) {
            set preferred core based on IPS
            exit training
        } else {
            force training period of 100K cycles on other core
        }
    } else {
        --------- MLP region Scheduling begin---------
        if (MLP region) {
            --------- MLP Epoch scheduling begin---------
            schedule on CLP+VP core
            --------- MLP Epoch scheduling end ---------
        } else {
            --------- Preferred Core scheduling begin--------
            schedule on preferred core
            --------- Preferred Core scheduling end---------
        }
        --------- MLP region Scheduling end---------
    }
    if (50 million cycles pass)
        enter training mode
}

Figure 2.9: SCE scheduling algorithm

application always benefits from being on the MLP core, this algorithm will ensure that it is
never scheduled off that core except for short regions of training. However, it critically relies
on judicious MLP Region detection. If MLP regions are triggered too frequently, it can add
significant overhead and hurt performance. In the next section, we consider the design of the
MLP Region Detection policy.

2.3.4 MLP Region Detection

Responsive switching to MLP core

The first goal of our region scheduling policy is to eagerly switch to the MLP core to exploit
regions of clustered misses. We leverage our analysis from Section 2.2 to determine the rate
of L2 misses needed for the MLP core to overtake the ILP core. According to Figure 2.5, on
average, we see the crossover point at 10 L2 misses per 10K instructions. So, as a heuristic, we assume that we need to observe misses at that rate in order for the MLP core to be profitable. We identify this rate as $r_{\text{miss}}$. We use $r_{\text{miss}}$ to identify the minimum number of L2 misses that should be observed in a region of $N_{\text{inst}}$ instructions in order to switch cores.

For each contiguous chunk of $N_{\text{inst}}$ instructions, our scheduler counts the number of misses. If fewer than $N_{\text{inst}} \times r_{\text{miss}}$ misses are observed at the end of the region, the counter is cleared. As soon as $N_{\text{inst}} \times r_{\text{miss}}$ misses are observed, the application is switched to the MLP core. Figure 2.10 illustrates this policy.

A key challenge is tuning $N_{\text{inst}}$. If it is too small, we will switch every time we see an L2 miss leading to significantly more overhead and loss of performance. However, if $N_{\text{inst}}$ is too large, it increases the likelihood that we wait too long to switch and miss good scheduling opportunities. After our experiments from Section 2.2 we found that in regions of 90 misses per 10K instructions our MLP core is better so we set $N_{\text{inst}}$ equal to the time it would take for our system to execute approximately enough instructions to observe this miss rate. After a series of experiments we concluded that $N_{\text{inst}} = 3000$ works well (approximately 9 misses $\times$ memory latency + the cycles to execute 1000 instructions at a CPI of 1). The hardware cost associated with the counters to measure L2 misses is very small and already present in current microprocessors.
Lazy Switching back to the Preferred Core

Once running on the MLP core, our scheduler evaluates a) if it was a correct decision to switch and b) if there is no longer a benefit from MLP. It is important to remain on Core2+CLP+VP long enough to exploit any MLP. However, it is desirable to correct erroneous decisions before too much time elapses.

Since L2 misses cluster, if the application is entering a region of clustered misses, we expect an elevated miss rate. We define $N_{stay}$ as the number of instructions that must be executed on the MLP core; during that region, an L2 miss must be observed. If no misses are observed, it is likely that clustering is not present (or no longer present) and execution should return to the preferred core. However, if a single miss is observed, we remain on the core for an extended execution, $N_{ext}$. At the end of the extended region, we perform the same evaluation again. Figure 2.11 illustrates this policy. We found that $N_{inst} = 3000$, $N_{stay} = 700$ and $N_{ext} = 3000$ work well. Note the rate of L2 misses required to switch back to the preferred core is lower than that required to switch away. Our scheduling policy is more sensitive to the $N_{inst}$ and $r_{miss}$ parameters.

MLP Core is Preferred Core

There is a high change the MLP core can be the preferred core. In this case as shown in Figure 2.12 we show the different behavior. As we reach a execution mode boundary either that is to switch to an MLP region or to an ILP region we would reset the counters and continue with using the new phase counters. This prevents stalling the processor and makes the migration
between modes seamless.

### 2.3.5 MLP Epoch Scheduling

Previous work has shown that MLP techniques need to be smart on how speculation is performed on L2 misses [34, 5]. In order for CLP+VP to be effective, the value prediction must be accurate in order to avoid polluting the cache and fetching instructions down the wrong path. We use a last-value predictor indexed by a hash of the PC and the branch history and we use a confidence estimator with a two-bit saturation counter indexed by the PC, just like in [5]. If the confidence on a prediction is too low, we do not speculate. Furthermore, we geometrically accumulate the confidence estimate for all the speculative load values created during an epoch. Once the accumulated confidence drops below a threshold, we stop speculating.

Another source of inefficiency is speculating over the shadow of an L2 miss that overlaps with no other L2 misses. If running on the ILP core (when it is the preferred core), this inefficiency is prevented because we are unlikely to switch to the MLP core for isolated L2 misses. When we are on the the MLP core, we avoid speculation because our value prediction confidence estimator will eliminate solitary misses with low prediction accuracy.

### 2.4 Evaluation of SCE

#### 2.4.1 Experimental Setup

To evaluate our AMP proposal we used SESC [44], a detailed, microarchitectural simulator, and compiled SPEC2000 and SPEC2006 applications using a MIPS cross compiler built from
Table 2.2: Core Details. Cycle counts are in processor cycles

<table>
<thead>
<tr>
<th>Core Setup</th>
<th>Multicore FabScalar 1.66 GHz</th>
<th>Single Core FabScalar 1.68 GHz</th>
<th>Multicore FabScalar 2.22 GHz</th>
<th>Single Core FabScalar 2.31 GHz</th>
<th>Multicore FabScalar 2.30 GHz</th>
<th>Single Core FabScalar 2.5 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>Fetch Rate</td>
<td>Issue Rate</td>
<td>Retire Rate</td>
<td>ROB</td>
<td>LD/ST Queue</td>
<td>Mem/Int/Fp Units</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>128</td>
<td>54/38</td>
<td>2/3/2</td>
</tr>
<tr>
<td>Frequency</td>
<td>1.66 GHz</td>
<td>1.68 GHz</td>
<td>2.22 GHz</td>
<td>2.31 GHz</td>
<td>2.30 GHz</td>
<td>2.5 GHz</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>96</td>
<td>42/38</td>
<td>2/2/2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>36/24</td>
<td>36/24</td>
<td>1/1/1</td>
</tr>
<tr>
<td></td>
<td>246 cycles</td>
<td>256 cycles</td>
<td>255 cycles</td>
<td>277 cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Memory RI</td>
<td>37 cycles</td>
<td>50 cycles</td>
<td>55 cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area(mm²)</td>
<td>1.636</td>
<td>1.118</td>
<td>0.718</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pipeline:</td>
<td>3-cycle fetch, 1-cycle decode, 1-cycle Rename, 1-cycle dispatch, 2-cycle issue, 2-cycle issue, 1-cycle RegRead, 2-cycle Mem/Int/Fp Units, 1-cycle DataCache, 1-cycle writeback, 1-cycle retire</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core4 Cache:</td>
<td>IL1 and DL1: Size=32KB; Assoc=4-way; Line size=64B; RT=3 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core2 Cache:</td>
<td>IL1 and DL1: Size=16KB; Assoc=4-way; Line size=64B; RT=3 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared L2 Cache:</td>
<td>Size=2MB; Assoc=8-way; Line size=64B; RT=10 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BTB: size=2K; Assoc=2-way, Supply Voltage=1.1V, Tech=45nm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch predictor:</td>
<td>bimodal size=16K; gshare-11 size=16K; Branch Mispred. Pen.=14 cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H/W Pref.:</td>
<td>16-stream stride prefetcher; hit delay=8 cycles; buffer size=16KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCE param.:</td>
<td>Nstay=700cycles, rmiss=0.001 and Nnext=3000cycles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLP+VP configuration:</td>
<td>OPB = 128 entries Max. Outs./Instr.=612/3072</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BHLV+GLV predictor, table size=4096 entries each</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GCC 4.4 [16] with -O3. Our simulations use the ref inputs set. We skip the initialization stage of applications and simulate at least 600 million correct instructions for SPEC2000 and 300 million for SPEC2006. We used as many of the SPEC2000 and SPEC2006 as we could to guarantee a mix of applications with both low-MLP and high-MLP behavior; also, some were excluded due to limitations of the cross-compiler or simulator.

We also used the FabScalar framework [9] to weigh the cost of different superscalar designs in terms of clock period, area, and power (static and dynamic). The FabScalar framework synthesizes Verilog models of arbitrary superscalar processors, where each superscalar processor can be customized in terms of pipeline ways (width of the processor) and sizes of the memory structures within a stage.

Since a superscalar processor makes use of many specialized and highly-ported RAMs/CAMs/FIFOs (e.g., physical register file, rename map table, issue queue, load-store queue, active-list etc.), we are also using the register file compiler from the FabScalar framework. The register file compiler uses custom layouts of multi-ported bit-cells and peripheral circuits to generate memory structures and characterize their access times and power consump-
Table 2.3: Core Configurations

<table>
<thead>
<tr>
<th>Label</th>
<th>Symmetry</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core4</td>
<td>Symmetric</td>
<td>4-wide 1.68GHz core (Base)</td>
</tr>
<tr>
<td>Core2</td>
<td>Symmetric</td>
<td>2-wide 2.31GHz core</td>
</tr>
<tr>
<td>Core2+CLP+VP</td>
<td>Symmetric</td>
<td>2-wide 2.31GHz CLP+VP core</td>
</tr>
<tr>
<td>Core4+CLP+VP</td>
<td>Symmetric</td>
<td>4-wide 1.68GHz CLP+VP core</td>
</tr>
<tr>
<td>SCE CLP+VP</td>
<td>Asymmetric</td>
<td>4-wide 1.66GHz core+2-wide 2.30GHz CLP+VP core</td>
</tr>
<tr>
<td>SCE none</td>
<td>Asymmetric</td>
<td>4-wide 1.66GHz core+2-wide 2.30GHz core (Base AMP)</td>
</tr>
<tr>
<td>SCE All</td>
<td>Asymmetric</td>
<td>4-wide 1.66GHz CLP+VP core +2-wide 2.30GHz CLP+VP core</td>
</tr>
<tr>
<td>SCE4</td>
<td>Asymmetric</td>
<td>4-wide 1.66GHz Core+4-wide 1.66GHz CLP+VP core</td>
</tr>
<tr>
<td>SCE imm</td>
<td>Asymmetric</td>
<td>4-wide 1.66GHz Core+2-wide 2.30GHz CLP+VP core with imm. switch</td>
</tr>
</tbody>
</table>

We used Synopsis Design Compiler C2005.09-SP3 and Cadence SoC Encounter V7.1, using the FreePDK OpenAccess 45nm Standard Cell Library [50] to synthesize our different designs in order to estimate timing, power and area. SESC supports power modeling and we feed the power characteristics of each structure with data from Fabscalar. To keep the design space within limits, we do not modify the voltage supply, which could affect frequency, and restricted the total pipeline depth to 14 (from Fetch to Commit) for all compositions of the processors used in the evaluation. We believe that a 14-deep pipeline is appropriate, since it can also be found in contemporary high performance microprocessors [53].

2.4.2 Power Modeling

Since we allow different core designs to operate at different frequencies, it is important to carefully consider trade-offs in terms of power and performance in our evaluation. In this section, we describe the methodology for calculating the operating frequencies of each core in all simulated configurations.

The processors evaluated in this study have different peak power characteristics. To see why, we will briefly discuss the overall power model for a processor. Power consumption of a processor consists of the static and dynamic component ($P_{\text{Total}} = P_{\text{Dynamic}} + P_{\text{Static}}$). Dynamic power consumption on a chip is given by equation 2.1, where $P$ is the power, $C$ is the effective load, $\alpha$ is the switching activity factor, $V$ is the input voltage and $F$ is the processor frequency.
The peak power is calculated much like \( P_{\text{Total}} \) except that an average switching factor is replaced with the worst case dynamic load for the chip. When running on an SCE configuration, the peak power is:

\[
P_{\text{PeakSCE}} = \max(P_{\text{DynPeakCore2}}, P_{\text{DynPeakCore4}}) + P_{\text{StaticCore2}} + P_{\text{StaticCore4}} + P_{\text{PeakAllElse}}
\]  \hspace{1cm} (2.2)

Since it is the case that \( P_{\text{DynPeakCore2}} < P_{\text{DynPeakCore4}} \), the equation can be simplified to only consider the peak dynamic power of Core4. Also, \( P_{\text{PeakAllElse}} \) is the remaining dynamic and static power from interconnect and cache. Compared to a single Core4 design, we have added extra leakage power.

In order to make a fair comparison, we set the peak power of the chip to that of the SCE configuration. Since the single core configurations cannot reach this peak power, we use the Fabscalar tool to identify a scaled voltage and frequency that boosts their power consumption to match the peak power of the SCE configuration. Interestingly, we use this same technique to provide our own design an advantage. When running on Core2, the peak power is overestimated by the difference in \( P_{\text{DynPeakCore2}} \) and \( P_{\text{DynPeakCore4}} \). Hence, SCE can also exploit scaled voltage and frequency when running on the smaller core.

We have performed a careful analysis of the peak power for each evaluated processor using the Fabscalar framework. Figure 2.13 shows the breakdown for each component out of \( P_{\text{PeakSCE}} \). Note, since we are comparing peak power not average power, the fraction of leakage power is relatively small. In Fig. 2.13a, we can see the power due to Core2 (Leakage Core2+L1, Dynamic 16K L1) contributes only about 1.66% to the peak power. This amount shows up in Fig. 2.13c as Unused Peak Power when evaluating a single Core4. Hence, Core4 can scale its voltage and frequency to consume 1.66% more dynamic power. As shown in Fig. 2.13b, the difference in
peak power when executing on Core4 versus Core2 on SCE is 13.01%; this is primarily due to the difference between the dynamic power for a 4-wide core compared to a 2-wide core. Hence, when executing on Core2 during SCE, the voltage and frequency can be scaled to consume 13.01% more dynamic power.

Table 2.2 shows the resulting frequencies in our experiments. The Fabscalar columns show the frequency reported by the Fabscalar tool. This is used for Core4 in SCE mode. The Single Core Scaled Peak column shows the adjustment to frequency when running on a single core (no SCE). Finally, the Multicore Scaled Peak column shows the adjustment to frequency when execution is on Core2 in SCE mode.

Table 2.2 shows the SESC configuration parameters used in our experiments. Labels used in the graphs are explained in Table 2.3 along with the configurations executed.

2.4.3 Performance Compared to Symmetric Multicore Processors

Figure 2.14 shows the speedups of SCE over Core4, our base case. We see that SCE with CLP+VP delivers an overall speedup of 1.14 for SPEC2000 applications and 1.10 for SPEC2006 over Core4. Looking at individual benchmarks we see that 19 out of 25 benefit from SCE. For benchmarks with no benefits, we see that they are characterized by low L2 miss rates. Finally, applu, mesa, mygrid, vortex, gromacs, and dealII, have a slight degradation due to regions of low
When compared to a single core with the CLP+VP technique, we see that our design is better. SCE CLP+VP is better than Core4 +CLPVP by 5.3% and 6.6% for SPEC2000 and SPEC2006 applications, respectively, with a maximum of 14.5%. The benefits come from the faster 2-wide CLP+VP core achieving better performance than the 4-wide CLP+VP for high MLP regions and the use of Core4 to complement 2-wide CLP+VP core’s for regions of low to no MLP.

Although not included here for brevity, we also performed an experiment in which the 2-wide core in SCE CLP+VP does not use the Scaled Peak frequency, rather it uses the Fab-scalar reported frequency. In this configuration, SCE CLP+VP performs marginally worse than the Scaled Peak SCE CLP+VP configuration (roughly 1.5% worse on SPEC2000 and 1% on SPEC2006), but the overall performance trend remains.

### 2.4.4 Power and Energy Delay

Figure 2.15 shows the power consumption for each design we evaluated. This includes the static and dynamic power for the occupied core, and cache hierarchy as well as the static power for the core not used. Caches remain active for the entire execution due to cache coherence, and so we modeled the static and dynamic power during the entire execution for regular accesses.
and invalidations triggered by the coherence protocol. The graphs show, our SCE CLP+VP design consumes less power compared to Core4+CLP+VP by 18.9% and 15.6% for SPEC2000 and SPEC2006, respectively. SCE, does require more power than Core2+CLP+VP, but this is a direct consequence of using Core4 to accelerate some non MLP regions. For benchmarks with high MLP regions such as mcf and equake, we see that average power is higher on the Core2+CLP+VP than Core4 due to speculation and re-executed instructions.

Figure 2.16 displays the energy delay$^2$ for each configuration. Overall, SCE CLP+VP reduces the energy delay$^2$ by 20.1% and 23.1% for SPEC2000 and SPEC2006 benchmarks over Core4. When compared to the Core4+CLP+VP design, the energy delay$^2$ savings are 18.3% and 21.1% for SPEC2000 and SPEC2006, respectively. Note that equake, mcf, swim and gcc are particular advantageous when considering energy delay$^2$ because their high performance offsets their high power consumption. An interesting observation here is that on average 56% of the average power is from dynamic power while the remaining 44% is due to static power.

2.4.5 Performance Compared to Asymmetric Multicore Processors

The benefits of SCE also exist when compared to other AMP configurations which vary in terms of frequency or MLP techniques. Table 2.3 shows the configurations we compare against. SCE All, has the CLP+VP added to the 4-wide core as well as the 2-wide core. In this design, CLP+VP may be used at any time. SCE none has a 2-wide 2.30GHz frequency and a 4-wide
Figure 2.16: Energy Delay$^2$ for SCE CLP+VP, Core2, Core2 CLP+VP, and Core4 CLP+VP normalized to Core4

1.66GHz frequency neither of which have CLP+VP; the 2-wide is used for MLP regions. SCE4 has a 4-wide base core along with a 4-wide CLP+VP core for the MLP regions. Finally, SCE imm is our SCE core configuration with a scheduler which switches to the MLP core at the first L2 miss (as opposed to using L2 miss rate like SCE CLP+VP does).

Figure 2.17 shows the speedup of each design compared to the Base AMP design, shown as SCE none. Overall, our design is 4.7% and 3.3% better for SPEC2000 and SPEC2006 when compared to the base AMP with no MLP technique implemented. Our SCE design is also better than SCE4 by 4.8% and 8% for SPEC2000 and SPEC2006. When comparing against SCE imm we see that SCE CLP+VP’s performance is better by 6.3% and 9.9% over SPEC2000 and SPEC2006. This is an important indicator to why our scheduling mechanism is effective. It shows how the miss rate is more accurate for detecting MLP regions, than making scheduling decisions solely based on a single L2 miss.

When compared to SCE All we achieve identical performance, which indicates that our scheduler hardly misses any MLP regions. An added benefit of our scheme is the reduction in the instructions added due to value misprediction and using MLP techniques on low MLP regions which can’t be exploited (6.8% and 5.4% for SPEC2000 and SPEC2006). This reduction in instructions also translates to a slight reduction in power (about 1% across benchmarks). This result yields an important message. If an MLP technique is expensive to implement, then
select a customized core and only implement it there. On the other hand, if an MLP technique is cheap, add it everywhere. In either case, you get the best performance due to our effective hardware scheduling mechanism which exploits core-level differences.

2.4.6 SCE Execution Analysis

Table 2.4 displays details on the switching overhead for our SCE proposal. The Overhead column indicates the overhead due to switching cores. Our scheduling mechanism can overlap most of the context switch with L2 misses when switching between Core4 to CLP+VP. This explains why, out of the total overhead, the migration to the MLP core contributes an average of 24%. One more source of overhead is the invalidation of shared cache lines that are a result of switching cores. We have not measured this, but we modeled it in our simulations.

We can study how sensitive our design is to switching cost by varying the penalty associated with switching cores. We re-ran our experiments using a switching overhead 2.5x and 5x bigger than our original penalty (as shown in Table 2.2). We observed a degradation in performance of 1.11% and 2.9%, respectively, for SPEC2000. For SPEC2006, we observed a 1.4% and 3.7% degradation on average, respectively. This shows that although our scheduler is able to switch cores aggressively this doesn’t occur continuously. The applications which get penalized the most are the ones with the higher total overhead in Table 2.4.
Table 2.4: Characterizing SCE: Total overhead of switching over entire execution, scheduling accuracy and value predictor accuracy

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total Overhead</th>
<th>Sched. Acc.</th>
<th>Value Pred Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>0.31%</td>
<td>96.13%</td>
<td>24.75%</td>
</tr>
<tr>
<td>applu</td>
<td>0.02%</td>
<td>82.43%</td>
<td>34.76%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.02%</td>
<td>86.67%</td>
<td>9.94%</td>
</tr>
<tr>
<td>crafty</td>
<td>0.51%</td>
<td>82.87%</td>
<td>78.55%</td>
</tr>
<tr>
<td>equake</td>
<td>1.13%</td>
<td>96.80%</td>
<td>47.10%</td>
</tr>
<tr>
<td>gap</td>
<td>1.12%</td>
<td>95.78%</td>
<td>90.91%</td>
</tr>
<tr>
<td>gzip</td>
<td>0.07%</td>
<td>96.85%</td>
<td>4.04%</td>
</tr>
<tr>
<td>mcf</td>
<td>1.07%</td>
<td>99.78%</td>
<td>49.09%</td>
</tr>
<tr>
<td>mesa</td>
<td>1.42%</td>
<td>89.86%</td>
<td>88.01%</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.02%</td>
<td>74.08%</td>
<td>15.13%</td>
</tr>
<tr>
<td>parser</td>
<td>1.98%</td>
<td>93.83%</td>
<td>74.88%</td>
</tr>
<tr>
<td>swim</td>
<td>0.05%</td>
<td>91.45%</td>
<td>3.70%</td>
</tr>
<tr>
<td>twolf</td>
<td>0.74%</td>
<td>94.43%</td>
<td>44.31%</td>
</tr>
<tr>
<td>vortex</td>
<td>0.36%</td>
<td>87.98%</td>
<td>59.88%</td>
</tr>
<tr>
<td>vpr</td>
<td>0.91%</td>
<td>95.95%</td>
<td>22.22%</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.78%</td>
<td>85.00%</td>
<td>44.00%</td>
</tr>
<tr>
<td>perlbench</td>
<td>0.02%</td>
<td>90.00%</td>
<td>12.58%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.97%</td>
<td>94.51%</td>
<td>48.84%</td>
</tr>
<tr>
<td>milc</td>
<td>1.43%</td>
<td>93.34%</td>
<td>57.47%</td>
</tr>
<tr>
<td>gromacs</td>
<td>0.28%</td>
<td>83.10%</td>
<td>46.81%</td>
</tr>
<tr>
<td>gobmk</td>
<td>0.86%</td>
<td>91.43%</td>
<td>68.33%</td>
</tr>
<tr>
<td>namd</td>
<td>0.35%</td>
<td>89.86%</td>
<td>44.40%</td>
</tr>
<tr>
<td>dealII</td>
<td>1.89%</td>
<td>77.69%</td>
<td>31.33%</td>
</tr>
<tr>
<td>soplex</td>
<td>0.02%</td>
<td>97.78%</td>
<td>60.56%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>0.37%</td>
<td>91.20%</td>
<td>19.09%</td>
</tr>
</tbody>
</table>
Table 2.5: Characterizing Execution in SCE Mode: Percent of execution spent on Core4 and Core2+CLP+VP in SCE mode

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Core4</th>
<th>Core2+CLP+VP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>32%</td>
<td>68%</td>
</tr>
<tr>
<td>applu</td>
<td>78%</td>
<td>22%</td>
</tr>
<tr>
<td>bzip2</td>
<td>92%</td>
<td>8%</td>
</tr>
<tr>
<td>crafty</td>
<td>78%</td>
<td>22%</td>
</tr>
<tr>
<td>equake</td>
<td>07%</td>
<td>93%</td>
</tr>
<tr>
<td>gap</td>
<td>23%</td>
<td>77%</td>
</tr>
<tr>
<td>gzip</td>
<td>06%</td>
<td>94%</td>
</tr>
<tr>
<td>mcf</td>
<td>05%</td>
<td>95%</td>
</tr>
<tr>
<td>mesa</td>
<td>83%</td>
<td>17%</td>
</tr>
<tr>
<td>mgrid</td>
<td>72%</td>
<td>28%</td>
</tr>
<tr>
<td>parser</td>
<td>48%</td>
<td>52%</td>
</tr>
<tr>
<td>swim</td>
<td>12%</td>
<td>88%</td>
</tr>
<tr>
<td>twolf</td>
<td>11%</td>
<td>89%</td>
</tr>
<tr>
<td>vortex</td>
<td>83%</td>
<td>17%</td>
</tr>
<tr>
<td>vpr</td>
<td>90%</td>
<td>10%</td>
</tr>
<tr>
<td>wupwise</td>
<td>11%</td>
<td>89%</td>
</tr>
<tr>
<td>perlbench</td>
<td>47%</td>
<td>53%</td>
</tr>
<tr>
<td>gcc</td>
<td>12%</td>
<td>88%</td>
</tr>
<tr>
<td>milc</td>
<td>26%</td>
<td>74%</td>
</tr>
<tr>
<td>gromacs</td>
<td>58%</td>
<td>42%</td>
</tr>
<tr>
<td>gobmk</td>
<td>81%</td>
<td>19%</td>
</tr>
<tr>
<td>namd</td>
<td>69%</td>
<td>31%</td>
</tr>
<tr>
<td>dealII</td>
<td>92%</td>
<td>8%</td>
</tr>
<tr>
<td>soplex</td>
<td>24%</td>
<td>76%</td>
</tr>
<tr>
<td>omnetpp</td>
<td>46%</td>
<td>54%</td>
</tr>
</tbody>
</table>
Table 2.4 also displays the accuracy of our scheduler and the value predictor for our CLP+VP. Overall our scheduling mechanism’s accuracy is high. Value prediction accuracy is also interesting since low accuracy does not necessarily translate to low performance. Even for high MLP regions, when the value prediction is wrong, useful prefetches can generate enough MLP to improve performance during the re-execution of instructions. Finally, Table 2.5 shows the time spent on Core4 and Core2+CLP+VP in our SCE design. As we can see here, benchmarks with huge gains spend the most time on the MLP core.

2.4.7 Sensitivity to Scheduling Parameters

In Figure 2.18 we see the sensitivity of our scheduling parameters. We evaluated our scheduling parameters only using SPEC2000 benchmarks. The most important factor is \( r_{\text{miss}} \) which degrades performance by a geometric mean of 4.5%, when changing the miss rate to 0.06. \( r_{\text{miss}} \) performance degradation can increase more as we increase the miss rate. This is due to applications with medium MLP+ILP not meeting the \( r_{\text{miss}} \) rate to migrate to the MLP core, or once migrated to the MLP core not sustaining the minimum miss rate. \( N_{\text{stay}} \) can also influence our scheduling mechanism. As we decrease \( N_{\text{stay}} \) we observe that the time applications have to
meet the minimum \( r_{\text{miss}} \) rate decreases and forces applications to migrate back to the preferred core sooner. The performance can degrade across benchmarks by 1.9% for an \( N_{\text{stay}} \) of 300. We found that our scheduling mechanism is not very sensitive to \( N_{\text{ext}} \). We see that reducing it by 3 resulted in a decrease across benchmarks of 0.7%.

### 2.4.8 MLP Region Scheduling Benefits on \( CLP+VP \)

MLP region scheduling improves the performance of \( CLP+VP \) itself. We performed a study comparing a single core with two operation modes (MLP and non-MLP) which uses the MLP region scheduling to switch between the two modes. Overall, SCE scheduling provides a 0.6% performance improvement over the unmodified \( CLP+VP \) proposal. Furthermore, the increase in executed instructions due to \( CLP+VP \) is shown in Figure 2.19. As we can see, the number of increased instructions executed is decreased by 2.9% and 2.8% across SPEC2000, and SPEC2006 respectively improving the energy delay\(^2\). This decrease comes from turning off MLP speculation in regions of little to no MLP. This decrease in executed instructions benefits applications with low prediction accuracy (by up to 10%). On the other hand it also reduces performance (by up to 2%) for applications with high prediction accuracy since we speculate less.

![Figure 2.19: Increase in executed instructions from \( CLP+VP \) vs benefit of MLP region scheduling](image)

SPEC 2000
- Core4+CLP+VP
- Core4+CLP+VP+SCE Scheduling

SPEC 2006
2.5 Related Work

2.5.1 Memory Level Parallelism

Although we are in the era of CMPs, sequential performance is important. Lots of research has focused on designing processors that attack the memory wall by increasing the program’s MLP. Prefetching in the form of helper threads is a technique used to extract MLP [10, 32, 14]. Execution of the helper thread is done in parallel on a SMT, or on a separate core for CMP architectures. Other techniques such as WIB[26], CFP[49], KILO[11], and BOLT[20] focus on increasing the window size by unblocking the pipeline on cache misses, thereby increasing the MLP. In these techniques long-latency operations (and dependent instructions) are removed from the scheduling window and inserted to buffers, thus freeing resources. When the latency is resolved the instructions are reinserted into the scheduling window. Runahead execution [34, 12], CAVA [5], and Clear [22] tolerate the long latency of L2 misses by retiring the load instruction when it reaches the head of the ROB and continuing execution despite the fact that the load has not completed. When the memory request returns it re-executes the instructions or if the value was correct, it continues execution. Chou et al. [8] evaluates the effectiveness of out-of-order execution on MLP compared to in-order processors as well as the effectiveness of value predictors, branch predictors and runahead execution in extracting more MLP. Our work contributes to how an MLP technique could be added to an AMP.

For our MLP core we implement a technique which is like CAVA along with optimizations proposed for Runahead. We picked a runahead like MLP technique because it does not require any modification to the binary. The hardware modifications also require significantly less design effort (modify cache modules) over a technique like CFP[49] which modifies the processor pipeline. We picked a CAVA-like implementation instead of Runahead due to the benefits of Value Prediction on the load miss. This can avoid rollback and provide power savings.
2.5.2 Asymmetric Multicore Processors

Asymmetric Multicore Processor designs have been proposed as a solution to achieve higher performance per watt ratio for executing a wider range of applications [23, 4, 51]. This does not always result in improved performance over a homogeneous system for single threaded applications. Given the same area, previous proposals [51, 4, 23, 45] achieve performance improvements when scheduling between cores at the multi-application or multi-thread level. No previous proposal has suggested that fine grain scheduling can achieve performance benefits, while only using one core at a time and running one version of the application, which is what our scheme provides.

Recent work on AMPs designed for MLP is presented by Pericas et al. [40]. In this work, an AMP design is composed of a fast cache core and a small in-order memory core to exploit high and low locality code respectively. By coupling the cores together an increased instruction execution window is created. Our approach is different in that we use fully functional cores which can work independently if needed. Execution is also always on one core rather than spread across cores.

Other recent AMP proposals use a slipstream[35, 55] inspired design to speedup sequential performance. These techniques however require execution of the same application on multiple cores concurrently, or require recompilation of the applications to create a reduced version of the program that will work as a perfect predictor or prefetcher [15].

2.6 Discussion and Summary

Due to power limitations, chips are having larger portions of the chip powered off. Asymmetric Multicore Processors(AMPs) offer a unique opportunity to utilize this powered off chip real estate and customize cores for specific regions of execution. In this study, we focus on regions of execution dominated by high MLP. Using a detailed model of cores accurate enough to calculate detailed timing and power characteristics, we determined that narrower cores, in our case a 2-
wide issue width, were more effective at exploiting Checkpointed L2 Miss Processing than a wider 4-wide issue core — providing better performance and energy efficiency across the MLP continuum. Therefore, we conclude that customization for MLP is important on an AMP.

We leveraged this finding to support Symbiotic Core Execution on an AMP. SCE is an effective scheduling mechanism because it allows MLP regions to exploit the higher performance and better power efficiency of the customized core while still leveraging the benefits of other cores during regions with little to no MLP. Using SCE, we achieve performance improvements of 5.3% and 6.6% for SPEC2000 and SPEC2006, respectively, with a maximum speedup of 14.5%. For the same study, it achieves a 18.3% and 21.1% energy delay$^2$ reduction for SPEC2000 and SPEC2006, respectively.

Our findings yield an important message for hardware designers. Core customization enables efficient exploitation of MLP. Even if our MLP technique is cheap enough to add to all cores, all of the performance can be obtained on a single customized core. Also, customization leads to power efficiency. Finally, application steering mechanisms for MLP are simple and effective. As we enter the era of dark silicon and consider techniques that exploit customization to boost single thread performance, better exploitation of MLP is feasible and beneficial.
Chapter 3

Improving Fine-Grain Thread Level Parallelism Using Abstract Queue Operations on Multicores

This chapter presents abstract queues and their use to improve fine-grain thread level parallelism. It is organized as follows: Section 3.1 presents our abstract operations properties; Section 3.2 describes the architecture of AQO. Section 3.3 provides the experimental setup and the full evaluation in the context of micro-benchmarks and DSWP. Section 3.4 discusses related work, and Section 3.5 summarizes the chapters contributions.

3.1 Bridging the Multicore Abstraction Gap

3.1.1 Properties of Abstract Parallel Operations

An Abstract Parallel Operation (APO) is much like an abstract data type [41]. It has a data structure in memory that aggregates the information needed to carry out the operation, and it has an interface that operates on the data structure. The interface consists of a set of instructions or library calls that are part of an application’s programmer interface (API). The data structure
is abstract, hence the organization of the structure is hidden from the programmer.

An APO provides an operation meaningful to parallel programmers. The interface of the APO must have clear and well defined semantics, both in terms of its functional meaning but also as it relates to ordering of events in the processor. Since programmers often deal directly with the ordering of references to variables, APOs should have clearly stated semantics with respect to loads and stores. Ideally, the APO will adopt rules consistent with the memory consistency model of the chip.

Finally, for all instructions in the APO interface, we assume that a software equivalent is available. We do not assume it must be efficient, but it must correctly implement the abstraction. Since most APOs were first developed in software, it is likely that they both exist and have efficient implementations for some usage scenario. We use this requirement to make software and hardware work interchangeably on the APO. Note, this allows us to begin an APO operation in hardware and finish it in software if interrupted in the middle by a system level event (e.g. context switch).

Figure 3.1 illustrates properties of the APO. The interface is shown on the left and the opaque data structure at the bottom. Programmers only need to understand the interface,
while the details of the implementation are hidden inside the architecture. During execution, instructions in the interface can execute either in custom logic or in a sequence of micro-ops that are equivalent to the software algorithm. However, both implementations use the data structure to carry out their operation.

3.1.2 An Architectural Design Pattern for Abstract Parallel Operations

Our architectural design pattern for APOs provides a framework in which architects can directly target APOs and improve their efficiency. It does this through the following three items. (1) Virtualizes operations by allowing them to occur in either hardware or software. In this way, limited or unavailable resources do not create a deadlock. (2) Hardware performs arbitration/selection of limited/critical resources. Since hardware is never infinite, single programs and multi-programmed workloads must arbitrate for the available hardware. An arbiter is part of the pattern. (3) All hardware operations must work robustly in the presence of system level events. Therefore, the pattern requires a means to take action on a context switches, page flush, or other system-level event which affects the APO. While many strategies are possible, including ones that modify the OS, we prefer architectural solutions which do not require OS modifica-
tion. To accomplish this goal, we leverage cache coherence explicitly, since it has already been
designed to work seamlessly with virtual memory and all OS events.

**Architectural Description**

Now we consider the architectural components for the pattern, as shown in Figure 3.2. While not
a physical architectural component, the APO Data Structure (DS) is the connection between
the software and hardware. This data structure is allocated in memory by the programmer
(e.g. using malloc) and initialized to contain essential information needed for the hardware
to implement the APO. An API exposed to the programmer can see to its allocation and
initialization. The hardware is designed with innate knowledge of the structure and how to
operate on it.

Any hardware implementation of the APO must use this data structure and keep it cache
coherent. To do so, an APO Data Structure Cache (DS$) is added to each core. This cache
stores the relevant information from the APO DS close to the core and maintains coherence
on the lines. Therefore, it is important that the DS be created in such a way that it is cache
friendly. The DS may be specific to a single kind of APO in order for architects to be able to
control the design of the structure and tailor it to a specific chip’s architecture under-the-hood.
Since the DS$ remains coherent, we have no concern about its contents being lost or becoming
stale due to system level events. However, an APO may have to interpret coherence events on
the DS$ in a special way. We will discuss this more later (Section 3.2.2).

The APO Functional Unit (APO FU) added to each core adds necessary support for the
operation. Depending on the APO and the bottleneck being solved, this may be more or less
extensive and integrated throughout the core (making it more that just a functional unit). Fur-
thermore, the core must contain support for switching to software-based execution if necessary.
Architecturally, this is supported through interaction with the Arbiter and branch misspecu-
lation support. If the Arbiter fails to acquire the needed hardware resources, then when the
instruction reaches the head of the re-order buffer, the core rolls back to re-execute the instruc-
Table 3.1: Applying Architectural Design Pattern to Abstract Operations.

<table>
<thead>
<tr>
<th>Abstract Operation</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>setup(Qptr,Buffer,Size)</td>
</tr>
<tr>
<td>Programming</td>
<td>enq{b,h,w} Rs, QPtr</td>
</tr>
<tr>
<td>Interface</td>
<td>deq{b,h,w} Rd, QPtr</td>
</tr>
<tr>
<td>Hardware</td>
<td>Hardware Queues between cores or through memory [28]</td>
</tr>
<tr>
<td>Implementation</td>
<td>Identify core-core communication and map to queue or memory</td>
</tr>
<tr>
<td>Arbiter</td>
<td></td>
</tr>
</tbody>
</table>

tion using a software version held in micro-code (or it could use custom logic that was always available without arbitration).

Finally, an Arbiter/Selector interfaces with the DS$ and APO FU to choose a hardware resource for the operation. Since APOs involve multiple threads, arbitration and selection logic must consider the behavior of all threads participating in the APO. For a Single-Producer/Single-Consumer (SPSC) queue, this would only consist of two threads.

### 3.1.3 Examples

Table 3.1 illustrates how we envision the API, hardware implementation and arbitration of our design pattern for APOs. This example is meant to show the generality of our design pattern. Just to reiterate, we believe having such a common framework is important for enabling innovation on multicores.

Queues are a common parallel programming primitive. Ideally, cores sitting side-by-side communicating through a queue should be able to pass data very quickly instead of using loads and stores. Therefore, we propose an abstract queue implemented using a fast hardware buffer between two cores. The interface consists of two instructions and a function call: enqueue, dequeue, and setup. Enqueue and dequeue work as expected. Setup allows the programmer to allocate a queue of a user-specified size. Then our arbitration logic detects if two threads communicating through a common queue are located near each other; if so, it has them use hardware
3.2 Architecting a Single-Producer/Single-Consumer Queue as an Abstract Parallel Operation

3.2.1 Overview of Architecture

Using our pattern, we support a hardware queue for fast communication between cores. We call this hardware queue, Abstract Queue Operations (AQOs). To keep our design simple and cost effective, we only place the queues between neighboring cores and we only implement a single-producer/single-consumer queue. The overall architecture is shown in Figure 3.3(a). We will provide an overview by way of an example.

The DS contains fields typical for basic queue operations in software: for example, an `enq`
pointer where the last enqueue stored its data, a \texttt{deq} pointer where the next dequeue will load from, a pointer to the start of the data buffer, and the size of the data buffer. The DS shown in Figure 3.3(b) is not yet optimized (it will be Section 3.2.2) but it contains all necessary fields. This data structure is allocated by the programmer and initialized using an API call (Step 1 in Fig. 3.3(a)). The programmer can create as many queues as needed for all the communication in the program.

The FU and DS will provide support for efficient operations on the hardware queue. The FU is inspired by a recent proposal [28], but we modify it to support operations on a hardware queue. When a queue operation first occurs, for example the dequeue in Step 2 in Fig. 3.3(a), the FU will load the relevant details about the operation from the DS in memory into the DS (Step 3). This will provide the FU access to important information, like the \texttt{enq}, \texttt{deq}, and size of the queue. If the queue is not empty, then the actual dequeue can proceed. A similar operations occur for enqueue as well (Steps 4 and 5).

At this point, the Arbiter observes the enqueue and dequeue operation and determines they are occurring on the same queue data structure in memory. Ideally, communication should occur through a hardware queue. So, the arbiter determines if a hardware queue is available. If one is, the DS cache is updated to bind operations on the specified queue with operations through a specific hardware queue (Step 6). On the other hand, if a hardware queue is not available either because of topology or because they are all being used, then the operation proceeds using custom logic or micro-code.

Assuming a hardware queue is available and arbitrated, the enqueue will happen in the hardware structure(7). Also, the dequeue will occur through the hardware (Step 8). Since one enqueue happened, the DS line for that queue will be marked dirty and the state of the data structure in memory will eventually be updated to reflect the changes.

We have indicated that the DS allows our design to work in the presence of system level events. We achieve this by monitoring coherence events on cached lines in the DS. If another core tries to perform an enqueue on the same queue instance, then an invalidate message will
be sent to the current owner of the line. This occurs naturally because a new enqueuer must
update fields in the DS. If an AQO is in progress at the current owner, the invalidate operation
will force a writeback of dirty state and notify the arbiter that the queue is no longer in use.
Some actions need to be taken to ensure that no data in the queue is lost. We discuss this more
in Sections 3.2.1 and 3.2.2.

**Important Architectural Design Decisions (Hidden from the Programmer’s View)**

Our pattern places several requirements on the design of any AQO. The two relevant ones in
this case are: (i) it must have both a hardware and software implementation that can be used
interchangeably, and (ii) it must be robust with respect to system level events and requirements.
Therefore, we will operate the queue in much the same way a software queue works and support
the hardware queue as a form of acceleration. To help explain this behavior, Figure 3.3(c-d)
shows the semantics of enqueue and dequeue. In part (c), we show a snapshot of the queue’s
buffer in memory and in hardware both containing the same state. The \texttt{deq} and \texttt{enq} pointers
indicate what is in the queue. In part (c), we show what happens in both memory and in the
HWQueue while operating. On the far left, we show an empty queue. After Enqueue(a), both
memory and HWQueue have an element. Next, after Enqueue(b), both have two items. Finally
after the Dequeue at the far right, the dequeue would occur through hardware, and the \texttt{deq}
pointer in memory is updated to reflect the operation.

Another challenge is that some data may be in the hardware queue when a context switch
occurs. If the consumer, for example, is switched out for a long time, it may be advantageous
to free up the hardware resource for another pair of threads. But, if it contains data that is not
stored in memory, it would have to be saved. This poses unnecessary complexity in our design.
Therefore, we assume that all enqueues write their data to memory to ensure a permanent
record, even if a hardware queue is being used. In that way, if we need to relinquish a hardware
queue for any reason, the state can be dumped. While it may seem inefficient, this policy gives
the architecture, specifically the arbiter, considerable flexibility and freedom in implementation.
Surprisingly, it does not affect performance very much (see Section 3.3).

Our implementation also allows the size of the hardware queue to be different from the size of the queue in memory. The hardware queue can be small to accommodate the amount of data in the queue on average, whereas the queue in memory is sized based on the worst case needs of the program\(^1\). If sized smaller, the hardware queue may become full before the software queue.

In this case, enqueue can simply write its data directly to memory or wait for a free entry in the queue. However, if we drop an item from the hardware buffer while it is full and then resume enqueuing a few items later, there is a \textit{hole} in the hardware buffer where the items should be.

In order to keep the logic simple and detect such a hole, we store both the address and the data in the hardware queue, as shown in Fig. 3.4(c). (Uses of this case are discussed more in Section 3.2.4.)

### 3.2.2 Queue Data Structure and DS for Enqueue/Dequeue Operations

**Layout.** Our Queue Data Structure (QDS), shown in Figure 3.4(a), is designed to support software queue and hardware queue execution. Our data structure is not much different than conventional designs [27, 21]. We have: a \textit{buffer} where data is written; an \textit{enq} pointer that holds the last location enqueued; a \textit{deq} pointer that holds the next location to dequeue; and,

---

\(^1\)For example, streaming applications with feedback paths may have a minimum queue size requirement.
the size of the buffer. We also replicate this information for the producer and consumer on separate cache lines; they will primarily use these copies to eliminate the possibility of false sharing.

Enqueue and dequeue operations always specify a queue instance on which to operate. This instance amounts to a pointer to a QDS in memory. If that instance is not yet in the DS$, then the QDS is loaded. Since the DS$ is aware of the QDS layout, it can save the data compactly in a single entry without regard to padding, shown in Figure 3.4(b). Also, there are extra fields for the arbiter, such as the mapping to a hardware queue.

**Operation.** Figure 3.4(c) and (d) show the enqueue and dequeue operation, including support for the hardware queue. There functionality was inspired by [28, 27, 21]. Note, these operations are implemented in hardware, but they use conventional data paths whenever possible. References to DS$[q] mean that the DS$ is accessed with the address, q. Now, if no hardware queue has been reserved for the operation, then all operations occur through memory. If a hardware queue is available, then as described in Section 3.2.1, data is stored in the buffer on every enqueue. Likewise, for dequeue, data can be retrieved from the hardware queue if it is being used and the queue is not empty.

There are a few important aspects of these operations worth pointing out. First, the enqueue primarily uses information stored in the Producer’s line; likewise, the dequeue operation primarily uses information stored in the Consumer’s line. So, most of the time, they work independently. But, when the producer needs to update its pDeq value because it has become stale, it updates it using deq rather than cDeq. This is better because we avoid sharing the Consumer’s line. But, the dequeue must occasionally update deq to keep it up to date (shown in the last if-statement). So, every CHUNK_SIZE dequeues, deq is updated. CHUNK_SIZE is an architecturally dependent constant value that is large enough for at least one full cache line to be consumed before updating deq. Note, similar behavior applies to the cEnq and enq variable.

**Cache Coherence Interaction.** The DS$ line will hold information about a queue as long as no coherence actions invalidate the entry. This has a few important implications. First,
Table 3.2: AQO Actions on Specific Coherence Actions

<table>
<thead>
<tr>
<th>Line</th>
<th>Coherence Event</th>
<th>Description/Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Invalidate</td>
<td>Resizing of the data structure. Write back dirty data, invalidate, and free HWQ. Next operation on this queue must reload entry.</td>
</tr>
<tr>
<td>Second</td>
<td>Invalidate</td>
<td>Producer notifies consumer of writes. Mark second line as invalid (State[1]). Must reload enq on next access.</td>
</tr>
<tr>
<td>Third</td>
<td>Invalidate</td>
<td>Consumer notifies producer of reads. Mark third line as invalid (State[2]). Must reload deq on next access.</td>
</tr>
<tr>
<td>Producer’s Line</td>
<td>Invalidate</td>
<td>Another thread or core is enqueueing. The current core is no longer producer. Write back dirty content, invalidate full entry, and free HWQ.</td>
</tr>
<tr>
<td>Producer’s Line</td>
<td>Invalidate</td>
<td>Another thread or core is dequeueing. The current core is no longer consumer. Write back dirty content, invalidate full entry, and free HWQ.</td>
</tr>
</tbody>
</table>

the DS$ should only track lines that are critical for performing the requested operation. For example, a producer only needs the Producer line most of the time; likewise the consumer thread only needs the Consumer line. Second, if another core starts modifying part of the cached QDS, the entry must be evicted from the DS$ and if dirty written back. Third, if an evicted entry is using a hardware queue, the DS$ must notify the arbiter that the hardware queue is no longer needed. Fourth, even if a thread using a hardware queue is switched out, as long as its QDS entry is valid, the hardware queue may remain allocated for it. This means that a consumer could keep dequeueing entries from the hardware queue even after a producer is switched out. Of course, once the queue is empty the consumer may be forced to stall if it is waiting for data from a switched out thread.

In order to make this behavior work, we need to carefully interpret the coherence actions on different lines of the QDS. Table 3.2 describes the actions taken for invalidate events on each line of the QDS and their meaning. Note, there are three cases that we want to happen infrequently: invalidates on Line 1, the Producer line, and the Consumer line. The latter two events mean that the queue is being used by different threads or that the original thread was rescheduled on a different core. If this happens frequently, we will lose efficiency of the hardware queue since all of its state will be discarded. The queue will essentially just operate through memory. It’s advantageous if threads remain on the same cores for a long time and tend to communicate using the same QDS instance (more on this in Section 3.2.5). Updates to Line 1
mean that the internals of the queue are being modified. This makes sense if the AQO supports resizing. In this case, the only thing to do is evict the entry from the DS$. Such an operation should be quite rare.

Note, downgrades from owner to sharer are also possible. To handle them, the data should be written back and marked as shared, but it need not be evicted. Of course, any entry that is downgraded must re-obtain exclusive access before updating the line.

3.2.3 Resource Arbitration

The Arbiter serves a critical role for the AQO. It identifies when communication is occurring through queues, determines if the communication can be mapped to a hardware queue, and reserves a hardware queue if one is available. To accomplish these actions, the Arbiter works closely with the DS$ to monitor on-going communication through queues. Since queues can be uniquely identified by the physical address of the QDS, the arbiter can identify inter-thread communication by monitoring the enqueue and dequeue operation stream at each processor. When a match is detected, the arbiter looks for an available hardware queue. In our design, we only place hardware queues between adjacent cores for simplicity and cost effectiveness. Therefore, the arbiter must decide if the two communicating cores are adjacent; if so, it can reserve an available hardware queue. When it makes its choice, it updates the DS$ entry to indicate that future enqueues and dequeues can proceed through a specific hardware queue.

In many programs, threads are balanced, which means enqueue and dequeue operations will happen close together in time. This means that communication can be established quickly. However, the producer and consumer are not guaranteed to arrive together. The arbiter may need to track active queues over a period of time to determine which cores are communicating. All in all, the logic can be implemented by using a small CAM to remember on-going queue operations and checking new operations against the CAM for a match.

To keep things simple, our experiments assume that the arbiter can establish a mapping as soon as both an enqueue and dequeue have occurred on a given queue but not before that.
Once a mapping is established, it remains until a coherence action invalidates it.

### 3.2.4 Performance Optimizations

With our implementation so far, the hardware queues can be used instead of communicating through memory. Now, we consider two optimizations that allow our design to better exploit the potential of our new hardware. One is focused on using the hardware queue more effectively, and the other is focused on scheduling.

**Red-Zone Dequeueing**

As described in Section 3.2.1, the consumer must wait to dequeue until `enq` and `deq` indicate that data is available. However, because they are only updated sporadically (every `CHUNK_SIZE` enqueues), the data will likely be in the hardware queue long before `enq` is updated. Rather than wait for the update, we would like the consumer to go ahead and read data out of the hardware queue. However, a naive approach to this can jeopardize correctness. To see why, consider the example in Fig. 3.5(a) and (b). Note, in each part of the figure, we replicate the state of the same queue to show the different perspectives of the Producer, the Consumer, and the hardware queue. In part (a), the consumer sees an empty queue, but items are available in the HWQ. This is because the Producer has enqueued but not updated the shared `enq` variable. So, in part (b), the Consumer proceeds with the dequeue from HWQ and updates `cDeq`, since it must remember that it read the corresponding item. Now, the state of the queue in (b) is incorrect: it was an empty queue, but now it appears to be a mostly full queue, quite erroneously. (Even if we update `cEnq` at the same time, we would just introduce other data races.) One way out of this predicament is to make the queue data structure aware of the hardware queue. Since architects get to design the structure, hardware specific details can be added.

Dequeue operations from the hardware queue are only a problem when the queue appears empty according to memory state. This region of dequeueing from an apparently empty queue will be referred to as operating in the *Red Zone*. To support dequeueing in the Red Zone, we
Figure 3.5: Coordinating actions with the hardware queue using Red Zone Dequeueing.

need an extra read pointer in the queue data structure (cHWDeq) to track dequeues from HWQ. This is added to the Consumer’s line in the QDS and is already shown in Fig. 3.4(a)-(b).

When not in the Red Zone, cHWDeq is the same as cDeq. But, when the Red Zone is entered, cHWDeq advances independently to track those dequeues. As long as data is available in the HWQ, dequeues can proceed and cHWDeq is incremented. Figure 3.5(c) shows the correct state after dequeuing from HWQ; cHWDeq is incremented to reflect the recent dequeues. The consumer can operate in the Red Zone as long data is available in the queue.

Eventually, the consumer may reach a condition in which it can no longer dequeue from HWQ, but data may be available in memory. So, it must update its cEnq pointer to detect if additional data is in the buffer in memory. However, there are two possibilities for the location of cEnq after the update, as shown in Figure 3.5(d). It may precede or equal cHWDeq, or it may fall after it. Where it will land is dependent on when the last update to enq occurred, and there is no way to know when or where that was. So, we consider what to do in both cases. In Figure 3.5(e), the update moves cDeq after cHWDeq since there is data waiting to be dequeued from the buffer in memory. cDeq is made equal to cHWRead, and, eventually, deq is updated with the new cDeq as well. Note that the consumer is no longer in the Red Zone.

Figure 3.5(f) considers the case that enq precedes cHWDeq\(^2\). In this case, cDeq is made equal

\(^2\)This is not strictly a less-than operation. It must consider wrap around within the buffer. cDeq serves as a fixed reference point for this calculation.
to \texttt{cEnq} because all items available in the memory buffer have already been dequeued. The consumer is still in the Red Zone, and there is nothing yet available to dequeue.

Note, it is critical that \texttt{cDeq} remain a serialization point between the Producer and Consumer. In this way, we can be sure that the Producer does not wrap around and overwrite data not yet read from memory. Also, it guarantees that we can distinguish case (e) and (f) in the figure.

When the Red Zone algorithm is in effect, the hardware queue will tend to remain empty, since dequeues will quickly consume enqueued data. In this case, the consumer will never need to access the data located at \texttt{cDeq}. This will reduce the number of cache accesses and allow the producer to keep the line cached. As a result, enqueues become more efficient since their stores are unlikely to generate a coherence cache miss. Hence, we have avoided moving cache lines from one core to the other.

\textbf{Optimizing Throughput By Inserting Stalls}

Once two threads are communicating through a hardware queue, they may not always be in perfect sync. For example, the producer may be able to work much faster than the consumer leading quickly to a full hardware queue, or a momentary switch out may result in the consumer falling far behind. The producer must decide to either (i) stall until the consumer frees up space or (ii) skip the hardware queue and spill to memory.

There are important implications for either policy. If the Producer stalls, it may wait uselessly while the consumer performs other work. On the other hand, if the Producer does not wait, then it will start enqueuing to memory instead of the hardware queue. Since the Consumer will be forced to get the data from memory, it may slow down even more. If it is possible to sync up the Producer and Consumer, that will yield the best result. However, if it is not possible, it is better to let the Producer just enqueue to memory.

Our solution is to add a watchdog timer. When the hardware queue is full, every subsequent enqueue is stalled up to $t_{\text{wait}}$ cycles. Once that time expires we proceed with the enqueue by
storing it to memory. This way we do not wait too long and we also ensure that the Producer and Consumer will be given the chance to sync back up. Using our experimental infrastructure in Section 3.3, we have studied how long $t_{wait}$ has to be for a producer and consumer thread operating at maximum throughput (two enqueues/cycle) and determined empirically that 108 cycles is sufficient.

3.2.5 Miscellany

Interaction with System Level Events

Because our AQO leverages cache coherence explicitly, no OS support is needed for it. However, other OS interactions still need to be considered for efficiency.

**Thread Scheduling.** Since our implementation is abstract, the programmer is unaware of the underlying architecture, and it is the architect’s responsibility to provide high performance for all communication through queues. But, due to implementation trade-offs, hardware queues may be placed such that they only speed up communication between some cores. Hence, thread placement would affect performance. To help solve this problem, the programmer may need a rule of thumb for optimizing thread scheduling. For example, the rule may be: *nearby cores communicate faster than distant cores*. This reveals no significant implementation details and is actionable. One way to do this is with system utilities or libraries that allow programmers to bind threads to a given core and to get core locality information. An example is the numactl utility, which allows the user to set a specific NUMA scheduling or memory placement policy without having to modify an application [1].

**Virtual Memory.** To keep the implementation of AQOs simple, they must be friendly users of virtual memory. As pointed out in [28], queue operations may access multiple pages in virtual memory if care is not taken. Typically, processors allow at most two page faults for a given instruction due to unaligned accesses. While this could be remedied in a variety of ways, it is reasonable to function as much like existing instructions as possible.

Also, the DS$^+$ and hardware queue contain state that may need to be flushed on a page
replacement. Logic which handles a page eviction on the L1 caches will need to be extended to evict relevant data from the DS$ and hardware queue as well.

**Implementation Details**

A recent proposal called, HAQu [28], described in detail how to implement efficient queueing operations in the microarchitecture of a processor. However, they do not support a queue in hardware. So, we adopt some of their implementation within the processor core, but we make changes for compatibility with our system. Because we implement on top of HAQu, we do not need micro-code support for queue operations; rather, we use the HAQu logic to update the buffer in memory.

In HAQu, all queue operations proceed through memory, so they are essentially loads (for dequeues) or stores (for enqueues). We extend their support for operating on a queue in memory with operations on a hardware queue as well. For an enqueue using the hardware queue, it inserts data into the hardware queue at retirement. A dequeue is modified somewhat more than enqueue. According to HAQu, on a typical dequeue, the dequeue is added to the Load-Store Queue (LSQ), since in their architecture it is basically a load. We do the same thing, but if data is available in a hardware queue, the dequeue obtains its data from the hardware queue as soon as it is available subject to our Red Zone Queueing algorithm. If the dequeue occurs through hardware, we mark the address of the load in the LSQ as known and completed in the same cycle. However, if the data is not found in the hardware queue, the load’s address is updated in the LSQ and it must obtain the data from memory, just like in HAQu.

Also, to handle speculative execution, all dequeues copy their data into a replay buffer. In that way, rolled-back dequeue operations can re-execute using data in the replay buffer, rather than going to memory.
### Table 3.3: Architecture simulated.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No. of Cores:</strong> 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Frequency 3.2 GHz</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fetch width 6</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Issue width 3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Retire width 3</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Branch predictor:</strong>  Hybrid GShare &amp; Local</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mispred. Penalty 14 cycles</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BTB 2K, 2-way</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private L1 Cache:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Size 64KB, assoc 4</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Line 64B</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Latency 2 cycles</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private L2 Cache:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Size 2MB, assoc 8</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Latency 9 cycles</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Line 64B</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Protocol MESI</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Memory:</strong> Latency 300 cycles, Bandwidth 25.6 GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ring interconnection:</strong> Link Bwd (8 data bytes/cycle): 25.6GB/s</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AQO:</strong> size: 16 entries, 2-ports, Acc. Latency: 2cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>(t_{\text{wait}}): 108 cycles</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 3.3 Evaluation

To evaluate AQO we have integrated our design into SESC [44], an event-driven performance simulator, that accurately models CMP architectures. Table 3.3 shows the details of the simulated architecture. The core, L1, and L2 cache are common for all architectures we evaluated. The core is an aggressive 3-issue superscalar processor. We model a cache coherent CMP design with private L1 and L2 caches. Coherence is supported at the L2 cache over a ring interconnect. To support AQO, two hardware queues (one for each direction) of 16 4-byte entries are added between neighboring cores.

We will compare AQO to HAQu (like AQO but it always communicates through memory) and an aggressive hardware queue(IdealQu). IdealQu uses AQO’s instructions but maps them into a high speed buffer connecting two cores. We assume 1 cycle to read/write from/to the queue, and it has a large 4 KB hardware buffer (the same size as the queue in memory for HAQu). Table 3.4 shows the labels used in the graphs along with their description.

We used benchmarks from SPEC2000 and MiBench [18] as well as a micro-benchmark to stress test our system. For all of our application binaries, we used GCC 4.5 targeted to
Table 3.4: System configurations with various queues.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HAQu</td>
<td>DSWP with HAQu</td>
</tr>
<tr>
<td>NoRedZone</td>
<td>DSWP on AQO with no red zone</td>
</tr>
<tr>
<td>AQO</td>
<td>DSWP with AQO</td>
</tr>
<tr>
<td>IdealQu</td>
<td>DSWP with Aggressive Hardware Queue</td>
</tr>
</tbody>
</table>

the Mips32 architecture. After skipping the initialization (typically 1-6 billion instructions), a certain number of markers are executed, so that the baseline binary graduates from 100 to 500 million instructions in the Baseline binary.

3.3.1 Micro-Benchmark

We use a micro-benchmark to evaluate the performance of AQO. This benchmark is designed to stress test the queues by communicating at maximum throughput. A producer thread executes an unrolled loop full of enqueues, and a consumer thread executes an unrolled loop full of dequeues. More specifically in each run, enqueue and dequeue operations are executed 64 times in a loop with 10000 iterations, and we used 4-byte elements for each queue operation. In total, we transport 2,560,000 bytes. There is very little branching overhead, so the dominant behavior is that of the queue operations. For each micro-benchmark, simulation begins after the producer and consumer reach a barrier at the beginning of the loop.

Performance

Figure 3.6 shows the performance speedup of AQO and HAQu normalized to the execution time of HAQu with a software queue size of 16 entries (1 cache line). Along the x-axis, the results of two different experiments are shown. On the left hand side, we vary the Software Queue Entries while keeping the hardware queue size constant. On the right hand side of the graph, we keep the Software Queue Entries at 1K and vary the hardware queue size. Overall AQO achieves a speedup of $7.9 \times$ over a competitive implementation of HAQu with 1K entries. This
is calculated by re-normalizing to the 1K entry queue in memory. This is from communicating using hardware queues instead of the cache hierarchy.

In the left part of Figure 3.6, we study the effect of a fixed-size hardware queue with different software queue sizes. We do this to examine how well we abstract away the underlying implementation of our hardware queue. We see that when the software queue size is smaller or equal to the cache line size, our gains are smaller than our best speedup ($7.9 \times$) but still significant compared to HAQu. The reduced performance occurs because we cannot enqueue/dequeue in/from the hardware queue when the queue in memory is full. Hence, a small queue in memory is inefficient; but once it has 128 entries (equivalent to 4 cache lines), it achieves high throughput. Also, for AQO with NoRedZone, we see that the performance is only $1.3 \times$ better than HAQu. Because the consumer must wait for shared variable updates, the full advantage of the hardware queue cannot be extracted.

We also examined how sensitive our design is to the hardware queue size. On the right side of Figure 3.6, we kept the software queue fixed at 1K entries and varied the hardware queue size. Interestingly, a hardware queue size of 4 entries was enough to provide most of the performance, and by 16 entries there is no additional performance to be attained. Hence, the size of the hardware queue can be kept quite small with little performance impact.
Based on these experiments, we adopt a software queue size of 1K entries and a hardware queue of 16 entries.

### 3.3.2 Decoupled Software Pipelining

In this section, we evaluate AQO compared to HAQu and IdealQu in the context of DSWP [37]. Our DSWP algorithm is very similar to [37]. The only difference is that we do not choose loops that include any function call. Also, all communication between two stages uses the same queue, ensuring communication occurs through one queue and will be quickly detected by the AQO. Our performance measurements are over the entire simulated region, not just the loops parallelized by DSWP. As a result we see smaller speedups compared to our micro-benchmarks.

Figure 3.7 shows speedups of each queue normalized to HAQu. In the figure, G.M. means the geometric mean of all benchmarks. Overall, AQO is better than HAQu, on average, by 11%. There is no degradation in performance to any benchmark compared to HAQu, and we see a maximum speedup of 38%. Compared to the IdealQu implementation, AQO achieves an overall slowdown of 3.7%. IdealQu, which is an ideal case, achieves a 14.9% speedup over HAQu. AQO is slower than IdealQu due to stores at the producer and updates to shared state. IdealQu also is given the advantage of faster enqueue/dequeue operations (one cycle each.)

Figure 3.7 also shows the importance of red zone dequeueing. Having AQO without our red
zone optimization results in a speedup of 2% compared to HAQu. Opportunities to leverage the hardware queue are rare without this support.

Figure 3.8 shows L2 cache accesses for each application. As described in Section 3.3.1, due to the utilization of the hardware queue, AQO avoids sending dequeue requests to the cache hierarchy and memory. The geometric mean across all benchmarks is 6.2% with a maximum of 16.8%. As a result, AQO accesses the L2 cache less frequently than HAQu. An important effect of the reduction of cache accesses are its energy savings. The energy savings seen in Figure 3.9 come from enqueue/dequeue operations accessing the hardware queue structure rather than accessing the cache hierarchies. The geometric mean of energy savings is 3.2% with a maximum of 16.3%.
In Figure 3.10, we see the ratio of hardware queue access versus memory queue access for each DSWP application. We see that some applications used the hardware queue more effectively than others. Applications which use hardware queues the most tend to have balanced communication and computation. The other applications tend to use memory more due to imbalance among the producer and consumer. We have discovered that our DSWP algorithm tends to create larger consumers than producers when a good balance is not possible. Improvements in DSWP task selection may help remedy the problem and improve performance in some cases. However, even with the imbalance, gzip is able to achieve higher performance by occasionally using the hardware queue.

In Table 3.5 we see the sensitivity of AQO to different hardware queue sizes in the context of DSWP. We ran the same DSWP benchmarks and show the geometric mean of all benchmarks. We observe that a 16-entry hardware queue is sufficient to achieve the majority of the speedup.
3.4 Related Work

**Abstract Operations Design Pattern.** While the architecture community has focused on accelerating common operations such as locks, the solutions have been *ad hoc*. Therefore, we have defined abstract operators and focused on the design of a pattern that can support a wide variety of them. Our pattern provides a conceptual and architectural framework for future innovation. We know of no other work with this broad and ambitious goal.

**Queueing.** There have been many proposals for efficient synchronization in hardware. In the 1980s, various fast inter-core communications were proposed [48, 2, 17, 30]. However, their intent was different than ours; they hoped to provide fast hardware primitives not abstract away queue operations. Recently, in the context of fine grained parallelism, a variety of dedicated hardware queues have been proposed to accelerate communication between cores [42, 37, 24, 46, 28, 52, 6, 7, 54]. However, none of these proposals achieve the same degree of abstraction for the programmer as our queue achieves. Our performance even may be less than what their specialized architecture can achieve for some workloads. For example, ReMap [52] is a shared reconfigurable architecture to accelerate fine grained point-to-point communication in a heterogeneous CMP; it can use its reconfigurable fabric for operations in addition to communication. On the downside, it requires OS support. An interesting possibility is to use a ReMap fabric to support multiple abstract operators using our design pattern.

The Hardware Accelerated Queue (HAQu) [28] shares many of the same ambitions as our AQO. They start with a software implementation of a single-producer/single-consumer queue and speedup the features that are critical for faster and more efficient fine-grained threading. However, HAQu does not provide an explicit hardware queue between cores. As a result, the latency from an enqueue to a dequeue of a given data item can be quite large. For fine-grained parallelism that is sensitive to this communication delay, HAQu alone cannot provide high performance. Also, HAQu does address system level issues, but it needs explicit fence operations to maintain consistency across system level events. Our design requires no such support, making it a more robust design for a wider range of systems.
3.5 Discussion and Summary

In this proposal, we proposed Abstract Parallel Operators (APOs) as a way of bridging the abstraction gap on current multicore architectures. We have described an architectural design pattern that explains how to design APOs that are robust, efficient, and operate transparently with respect to the operating system. This can be used as the building block for other abstract parallel operations such as barriers. Here, we summarize these details to draw some final conclusions.

If a designer wishes to implement a new APO similar to our Abstract Queue Operation (AQO), there are a few design decisions and architectural components to build. First, the abstract operation should have a clearly defined interface and software implementations for each instruction in the interface. These operations are based on a cache friendly data structure whose details are hidden from the programmer. Then, for the critical bottlenecks related to the abstract operation, custom logic is added to the processor; and, there may be multiple implementations depending on which bottleneck(s) matter(s) for performance or efficiency. An arbiter should be designed to manage resources and select the best resources to improve application behavior. Finally, coherence actions on the data structure may need to be mapped to arbiter actions on the custom logic to ensure robust system operation. While these steps are not trivial to construct or develop for an abstract operation, they provide an effective framework to encourage innovation at the hardware/software interface.

We have shown how to use this pattern to design an AQO. To achieve the best performance, novel extensions to conventional algorithms are needed, for example the red zone dequeueing algorithm for our abstract queue. Furthermore, we were able to find good architectural trade-offs in our design. As a result, we obtained speedups of $7.9 \times$ for our micro-benchmark and 11% on DSWP parallelized codes over a hardware accelerated queue. We also demonstrated other benefits, like reduction in L2 accesses and improved power efficiency.

We see great potential for our design pattern to provide significant benefits in both performance and power efficiency. Furthermore, because architects are fully in control of their
implementation and will work hard to achieve high performance and efficiency, programmer’s
can trust them to become more efficient with each new processor generation.
Chapter 4

Conclusion

Off-chip and on-chip communication latencies are important areas of focus for high performance computing. This dissertation considers two shortcomings in state-of-the-art CMP designs. The bias toward multi-threaded and multi-programmed workloads, which deemphasizes single-threaded performance, and the design strategy which has not paid sufficient attention to the diverse memory behaviors of applications running on these platforms. We believe Asymmetric Multi-core Processors (AMPs) and Abstract Queue Operations (AQOs) offer a unique opportunity to improve inter and intra-chip communication latency, due to different memory and communication behaviors exhibited by single-threaded and multi-threaded applications.

In our first study, we observe that narrower cores, in our case a 2-wide issue width, were more effective at exploiting Memory Level Parallelism (MLP) than a wider 4-wide issue core providing better performance and energy efficiency across the MLP continuum. We use AMPs to exploit this finding to support Symbiotic Core Execution. SCE is an effective scheduling mechanism because it allows MLP regions to exploit the higher performance and better power efficiency of the customized core while still leveraging the benefits of other cores during regions with little to no MLP. Using SCE, we achieve performance improvements of 5.3% and 6.6% for SPEC2000 and SPEC2006, respectively, with a maximum speedup of 14.5%. For the same study, it achieves a 18.3% and 21.1% energy delay reduction for SPEC2000 and SPEC2006,
respectively.

In our second study, we hope to reduce the communication latency in fine-grain thread level parallelism by using Abstract Queue Operations. We target producer-consumer applications. Our design operates transparently, delegating the decision to the hardware which determines when to use software or hardware queues. The hardware implementation is abstracted from the programmer relieving him from having to manage limited hardware queue. To achieve the best performance, novel extensions to conventional algorithms are needed, for instance the red zone dequeueing algorithm for our abstract queue. Furthermore, we were able to find good architectural trade-offs in our design. As a result, we obtained speedups of $7.9 \times$ for our micro-benchmark and 11% on DSWP parallelized codes over a hardware accelerated queue. We also demonstrated other benefits, like reduction in L2 accesses and improved power efficiency.
REFERENCES


73


