The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

EXPLOITING MULTI-THREADED APPLICATION CHARACTERISTICS TO OPTIMIZE PERFORMANCE AND POWER OF CHIP-MULTIPROCESSORS

A Thesis in

Computer Science and Engineering

by

Chun Liu

© 2005 Chun Liu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2005

The thesis of Chun Liu has been reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Thesis Co-Adviser
Co-Chair of Committee

Mahmut Kandemir
Associate Professor of Computer Science and Engineering
Thesis Co-Adviser
Co-Chair of Committee

Mary Jane Irwin
Professor of Computer Science and Engineering

Natarajan Gautam
Associate Professor of Industrial Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Chip multiprocessors (CMPs) are becoming a popular way of exploiting the ever-increasing number of on-chip transistors. Multi-threaded applications aim to more efficiently utilize the raw power that CMPs provide than is currently possible. However, current multi-threaded applications exhibit load imbalances at various levels. The increasing capacity for on-chip storage and increasing costs of wire delays make the location of data on the chip very vital. Thus, it is important to place the data in the right location at the right time in the on-chip cache hierarchy. For the purposes of this study, we characterize the load imbalance at the barrier, imbalance amongst cache requests from different cores, and the demands on different blocks of the cache. Using the insights obtained from our characterization study, we then propose techniques that exploit such load imbalances to improve power and performance.

For the load imbalance problem at the barrier, we observe that the imbalances are quite predictable. Using an integrated hardware-software mechanism, we propose a novel technique for optimizing the power consumption of CMPs. By using a high-level synchronization construct called *barrier*, our technique tracks the idle times spent by a processor waiting for other processors to arrive at the same point in the program. Using this knowledge, the frequency of the processors can be modulated to reduce/eliminate idle time, thus providing power savings without compromising performance.

For the load imbalance problems imposed on the L2 cache by the different cores, we notice that the possible imbalance between the L2 demands across the cores favors a shared L2 organization, while the interference due to these demands favors a private L2 organization.

We propose a new architecture, called *Shared Processor-Based Split L2*. The new architecture captures the benefits of both types of organizations while avoiding many of their drawbacks.

We also study the demands on different blocks of the L2 cache, namely actively shared blocks and mostly privately accessed blocks. We show that, while there are a considerable number of L2 accesses to shared data, the actual volume of data is relatively low. Consequently, it is important to keep the shared data fairly close to all processor cores for both performance and power reasons. Motivated by this observation, we propose a small center cell cache residing in the middle of the processor cores which provides fast access to the cores' contents. We demonstrate that this cache organization can considerably lower the number of block migrations between the L2 portions that are closer to each core, thus providing better performance. Combined with sequential tag-data access, the power consumption of such a shared cache can be reduced further.

Table of Contents

List of Tables ix

List of Figures xi

Chapter 1. Introduction 1

1.1 The Case for CMP 4

Chapter 2. Multi-Threaded Applications and Load Imbalance 8

2.1 Background 8

2.1.1 Explicit Threads 9

2.1.2 Implicit Threads 9

2.1.3 Hybrid: OpenMP 10

2.2 Characterizing Parallel Applications 10

2.2.1 Load Imbalance due to Synchronization 11

2.2.2 Load Imbalance due to Competition of Shared Resources 12

2.2.2.1 Intra-Application Heterogeneity 12

2.2.2.2 Inter-Application Heterogeneity 13

2.2.2.3 Shared Blocks versus Privately Accessed Blocks 13

2.3 Related Work 14

Chapter 3. Synchronization Characterization and Barrier Optimization 16

3.1 Introduction 16

3.2 Related Work 19
3.2.1 Multi-Core Processors 19
3.3 DVS in Chip Multi-Processors 20
3.4 Our Approach 21
3.4.1 Barriers 21
3.4.2 Proposed Optimizations 22
3.5 Experimental Setup 25
3.6 Results 27
3.6.1 Disparity at Barrier Arrival Times 27
3.6.2 Idle Time Predictability 30
3.6.3 Energy and Performance Results 31
3.6.4 Using a More Powerful Predictor 33
3.6.5 Comparison with the Thrifty Barrier 34

Chapter 4. Load Characterization and Uniform-Cache Optimization 42
4.1 Background 42
4.2 L2 Organizations and Proposed Architecture 46
4.2.1 Proposed Architecture 46
4.2.2 Hardware Support 49
4.2.3 Exploiting the Proposed Mechanism 50
4.3 Experiments 52
4.3.1 Methodology and Experimental Setup 52
4.3.2 Base Results 55

4.3.2.1 Intra-Application Heterogeneity 60
4.3.2.2 Inter-Application Heterogeneity 61
4.3.3 Sensitivity Analysis 62
4.3.3.1 Impact of Larger L2 62
4.3.3.2 Impact of Memory Access Latency 64
4.3.3.3 Impact of Other Parameters 64
4.4 Related Work 64

Chapter 5. Access Characterization and Non-Uniform Cache Optimization 72
5.1 Introduction 72
5.2 Related Work 75
5.3 Motivation 77
5.3.1 Workloads 77
5.3.2 Characterization 80
5.4 Center-Cell based L2 Organization and Data Lookup 81
5.4.1 Placement 86
5.4.2 Migration and Eviction 86
5.4.3 Search 87
5.5 Results 90
5.5.1 Experimental Parameters 90
5.5.2 Reduction of Cache Block Migration 91
5.5.3 Profile of L2 Hits 92
5.5.4 L2 Hit Latency 93

5.5.5 Unnecessary Non-Preferred Lookups 95

5.5.6 Number of L2 Cells Accessed 95

5.5.7 Average Access Energy 96

5.5.8 Execution time 97

Chapter 6. Conclusion and Future Work 108

References 112

List of Tables

3.1 Base simulation parameters used in our experiments. 27

3.2 Frequencies and voltages used in the evaluation. 28

3.3 The benchmark codes used in this study. 29

3.4 Idle times at barrier averaged over all processors and iterations for selected loops from our benchmarks. 32

3.5 Normalized energy consumption and increase in execution time of our mechanism vs. Thrifty Barrier (TB) using the same Oracle and Last-Value Predictors (LVP) for both. 34

4.1 SSN configurations studied for the 2MB L2. Note that the splits are themselves of 128K each, an integral number of such splits — called a chunk — are allocated to a CPU. 53

4.2 Base simulation parameters used in our experiments. 55

4.3 The benchmark codes used in this study and their important characteristics for the private L2 case. 56

4.4 The IPC results for different worloads and different L2 management strategies. 56

4.5 The breakdown of L2 accesses for SSU. 61

4.6 The IPC results for different workloads and different L2 management strategies for a 4MB L2. 63

5.1 Our applications and important statistics. 79

5.2 Important parameters for a 3200um low-latency wire at 25nm. 85

5.3 Simulation parameters. 91

5.4 Schemes we compare in our experiments. Table shows the sizes of the center cell and preferred L2 regions of each core. It also shows the latency to these L2 portions from each core. 92

List of Figures

2.1 Three programming paradigms of parallelization 8

2.2 Active and stall times using a barrier that is being invoked iteratively. 11

3.1 Two dynamical voltage and frequency scaling approaches for CMPs. VC: Voltage Controller, PLL: Phase Locked Loop 36

3.2 Active and stall times using a barrier that is being invoked iteratively. 36

3.3 Stall times and execution times for some important parallel loops in our applications. An (x,y) point in the CPU0 curve indicates that fraction y of the time the last processor to arrive at the barrier spends x cycles or less executing the loop before coming to the barrier. For other CPUs, a point (x,y) says that for a fraction y of the time, the processor in question waits x cycles or less at the barrier point. 37

3.4 Predictability of active times. The y-axis histograms the percentage occurrence of the normalized estimation error (i.e. x-axis is $\frac{a_{pred}-a_{actual}}{a_{actual}}$). 38

3.5 Overall energy consumption with Oracle Predictor. 38

3.6 Overall energy consumption with the Last Value Predictor. 39

3.7 Percentage of the execution time spent in a particular frequency (8 processors). 39

3.8 Percentage Increase in execution time with Last-Value Predictor. 40

3.9 Normalized energy consumption for mgrid: Last-Value Predictor vs Markov Predictor. (*x*p means *x* processors are used.) 40

3.10 Percentage increase in execution time for mgrid: Last-Value Predictor vs. Markov Predictor. 41

3.11 Percentage of the execution time spent in a particular frequency for mgrid with the Markov Predictor (8 processors). 41

4.1 (a) Private L2 Organization. (b) Shared L2 Organization. 44

4.2 (a) The table structure that shows the processor-L2 split associations. (b) Addressing L2 splits in our proposal. 67

4.3 IPC and the number of L2 misses for specjbb. 68

4.4 Spatial Heterogeneity Factor (SHF) for each epoch. Note that the y-axes are on different scales. 69

4.5 Temporal Heterogeneity Factor (THF) for each CPU. Note that the y-axes are on different scales. 70

4.6 The L2 space allocation for ammp+apsi under SSN-152. 71

4.7 IPC increase over the P case. 71

5.1 Percentage of L2 blocks and accesses. ”s2”, ”s3” and ”s4” on X-axis indicate that the number of cores sharing/accessing the blocks is 2, 3 and 4 respectively. Each graph is drawn as a cumulative percentage starting from the ”Private” case. 82

5.2 Running distance distribution of the shared blocks in L2. X axis shows the running distance, and the Y axis shows the cumulative percentage of occurrences of that running distance. 83

5.3 Four-core 16MB L2 chip layout. 98

5.4 Layout of the preferred cells and access latencies as seen from a core's perspective. . . . 98

5.5 Cascade evictions of L2 blocks. . . . 99

5.6 Example migration and eviction. . . . 100

5.7 The search (probing) strategies employed by *CC.CT* and by different variants of *CC.DT* . . . 101

5.8 Fraction of L2 cache that can be accessed under a given number of cycles. . . . 102

5.9 Normalized cache block migration with *CC.DT* and *CC.Large* with respect to *M.DT*. . . . 102

5.10 L2 hit distribution. . . . 103

5.11 Average L2 hit latencies with the different schemes. . . . 104

5.12 Average L2 hit latencies for *CC.DT* under the different search options. . . . 104

5.13 Unnecessary non-preferred lookups under the aggressive and moderate schemes as a percentage of L2 accesses. . . . 105

5.14 Average number of cells probed for each L2 hit. . . . 105

5.15 Normalized average energy consumption per L2 access. . . . 106

5.16 Reduction in execution time with respect to *M.CT*. . . . 107

Chapter 1

Introduction

Following Moore's Law, the transistor count on a single chip has more than doubled every 18 months. Transistor count growth by decades has been exponential: from 10 thousand to 100 thousand during the 1970s, to a million in the 1980s, and up to 40 million in the 1990s. A single chip with more than a billion transistors is envisioned by the end of this decade. The increasing number of transistors has substantially improved the IPC (instruction per cycle) from 0.1 to roughly 3 today. The ability to pack billions of transistors on-chip has opened the door to another level of high-performance computer systems. But, such computational power does not come without cost; namely, as the technology scales, wire delay [47] and power consumption [11] increase. According to one study in [47], a signal can only reach 5 percent of the die length within a clock cycle (for the 60nm process to be realized around 2010). Moreover, studies in [11] indicate that power density is fast becoming a problem, but power density cannot be addressed by packaging alone.

Solving these issues efficiently and utilizing the chip's real estate effectively are important issues that need to be addressed in the near future. Several approaches to performance improvement have been proposed for the billion transistor challenge. Some of these approaches are summarized below:

- Wide-issue superscalar processors

  These contain a single 16- or 32-issue, out-of-order processing core with a large trace

cache and near perfect branch predictors to keep the pipeline filled to exploit ILP (instruction-level parallelism). The main issues with such a processor are the limitations of ILP [68], design complexities and power inefficiency.

- Superspeculative superscalar processors

  Due to a shortened clock cycle, deeper pipelines are often used in modern processors. To mitigate the loss caused by pipeline stalls, [42] propose a superspeculative superscalar architecture that relies heavily on data speculation to prevent performance loss. In such an architecture, aggressive speculations such as value predictions are widely used. Although the additional data speculation hardware alleviates the ILP limitation, superspeculative processors at least share, if not exacerbate the complexity and inefficiency problems of wide-issue processors.

- Simultaneous Multi-threaded (SMT) processors

  SMT techniques allow concurrent sharing of functional units that can increase utilization and throughput. This technique has been implemented in Intel's Pentium 4 [25] and IBM's Power5 [18] processors. However, such a technique is not a viable long-term solution due to bottlenecks imposed by memory ports and design level complexities.

- Chip Multiprocessors

  As mentioned above, wire delay constraints will limit the area that a signal can reach in a single cycle. Such a limitation will lead to a distributed hardware design that consists of several processor cores with supporting hardware that are connected by either synchronous or asynchronous interconnections. CMPs are commercially available in the high-performance server arena. For example, IBM's Power4/5 [18] and Sun's UltraSparc

IV are both dual-core processors. Intel's Itanium-2 [48] based CMP has 4 cores, and Compaq WRL's Piranha [8] is an 8-core design. Like the SMT approach, this technique was initially intended to improve throughput for multi-programmed workloads, and applications therefore need to be parallelized to gain speedups. Consequently, explicitly parallel tasks and/or good parallelizing compilers are the keys to multi-threaded application performance. Recent efforts such as [28, 17, 54] have proposed novel techniques to accelerate the performance of single-threaded applications using thread-level speculation, while [51] demonstrates a checker processor that can be used to build a more reliable system.

- Vector IRAM processors

  Intelligent RAM [55] integrates DRAM cells and combinational logic onto the same chip, thereby providing a higher bandwidth and lower latency for memory accesses. This integration can be quite beneficial for data-intensive applications. However, this type of architecture requires compiler support to vectorize the application code. This architecture can satisfy the requirements for data-intensive workloads due to the integrated RAM, but it is unlikely to be a successful substitute for general-purpose processors.

- Grid processors

  A grid processor [60, 66] is composed of a tightly coupled array of ALUs connected via a network, onto which large blocks of instructions are scheduled and mapped. Instruction results are forwarded directly from producing ALU to consuming ALU without returning to a register file. To mitigate on-chip communication delays, applications are scheduled so that their critical dataflow paths are placed along nearby ALUs. All management actions in Raw processor [66], including cache coherence, are performed by software. Such designs

expose the underlying hardware and communication latencies to the compiler, thus their performance heavily reply on the competence of the compilers. Currently, stream-based signal-processing computations can benefit the most from such an architecture, where the software can be easily mapped on the exposed parallel hardware. The main problem with using this technique to benefit general-purpose workloads, however, is that it lacks a good parallelizing compiler and software support.

1.1 The Case for CMP

As seen above, most recent advances (aggressive speculation and wide-issue superscalar) in computer micro-architecture attempt to improve aspects of single-threaded application performance, like execution time. As the chip real estate increases, such a trend leads to a less efficient design. On the other hand, attempts to expose the underlying hardware to the compiler lack software support. In comparison, the CMPs design, which targets the performance of multi-threaded applications, could ideally be both area- and energy-efficient, since it uses simpler, more efficient cores. Compatible with current software, the CMPs approach also provides smooth transition for those software.

- *Scalability*

  Wide-issue, super-speculative superscalar processors attempt to sustain the ILP parallelism through speculation. They do not, however, attempt to mitigate the wire delay or scalability issues since they tend to have long wires and centralized resources. Similarly, SMT processors share the same scalability issue, but they improve function unit utilization

by giving the wide-issue superscalar processors the capability of allowing multiple threads to execute simultaneously.

On the other hand, while the IRAM and RAW processors are scalable and can handle the wire-delay issue well, the dramatic change in architecture necessitates efficient software/compiler support that has not yet been addressed. With scalable interconnects and a separate memory port, CMPs are considerably more scalable (for instance, IBM's Power5 [18] can accommodate 64 cores).

- *Efficiency*

  Rather than throwing all resources into a single processing core and making it very complex to design and verify, chip-multiprocessors (CMPs) consisting of several simpler processor cores can offer a more cost-effective and simpler way of exploiting these higher levels of integration. CMPs also offer a higher granularity (thread- or process-level) which allows parallelism in programs to be exploited by the compiler or the runtime environment. Without this higher level of granularity, the hardware would be forced to extract parallelism at the instruction level on a single, larger multiple-issue core.

- *Compatibility*

  Support for legacy software is more important for commercial applications than academic ones. In comparison to the IRAM and RAW processors, CMPs offer a gradual transition from single-thread architecture to multi-thread architecture with adequate parallelism and better area utilization.

- *Economy*

Building a CMP is relatively inexpensive compared to the other approaches. Typically, a state-of-the-art uniprocessor is duplicated and connected using a network, after which inter-node communication and peripheral hardware (i.e., a big shared cache) are integrated onto a single chip. Such an approach is less risky and incurs less NRE (non-recurring engineering) costs. There is clear evidence of a trend towards CMP designs in several commercial offerings and research projects [53, 39, 44, 52, 8].

All these compelling reasons motivate the trend toward CMP architectures. The wide deployment of CMP has been inhibited by three primary restricting factors: the lack of parallelized applications, synchronization overhead, and power density.

Most applications have not been explicitly parallelized, and there is inherent difficulty in building an efficient parallelizing compiler. However, we could argue that this is a chicken and egg problem. Once CMP hardware becomes more readily available and the rewards of parallelizing the task are more significant, we will see more development on the software front in the form of more parallel applications. To accelerate the adoption and deployment of parallel applications, several attempts to ease the burden of parallelizing applications have been proposed, implemented and adopted, such as OpenMP [22].

Another factor limiting adoption of CMP architectures is the synchronization overhead, which is caused by inter-thread dependencies, and can throttle the scalability (performance). This thesis identifies multi-threaded applications' micro-architectural characteristics and provides ways to improve their performance when run on CMPs. We will first investigate the inter-thread application interactions. Next, we will show how to exploit the design space of CMP inter-core communication and inter-core caching that can be used to meet the demands of the

multiple threads. Lastly, we study the increased effectiveness of data placement and movement that results from Non-Uniform Cache Architectures (NUCA).

The final limiting factor is the thermal envelope, which is the power budget for a single die. As the processing cores are getting bigger and more complex, more circuits and increased power budgets are being used to improve single-threaded application performance by speculative execution. Today's most demanding single-core processors have a power budget as high as 140 Watts. Certainly, putting two or more such cores onto a single die would further increase the power consumption, making the cooling of the running cores significantly more challenging.

This thesis aims to exploit the characteristic of applications to make them run faster and consume less power on CMPs. The subsequent chapters of this thesis are organized as follows. Chapter 2 discusses multi-threaded applications and the load imbalance. In Chapter 3 the load imbalance of multi-threaded applications at the barrier is investigated and a power-saving scheme is proposed and evaluated. Chapter 4 evaluates the imbalance of multi-threaded applications and proposes a novel uniform cache design. In Chapter 5, based on the observation of load imbalance between mostly shared blocks and mostly privately accessed blocks, the data placement and movement are evaluated in an original NUCA cache architecture. Finally, Chapter6 concludes the dissertation with a summary and suggestions for future work.

Chapter 2

Multi-Threaded Applications and Load Imbalance

In this chapter, we briefly describe the parallelization of applications and the characteristics of multi-threaded application execution, especially from the imbalanced load perspective.

2.1 Background

Applications running on CMPs can be expected to take advantage of the multiple processor cores by creating threads to run in parallel.

```
main ()
{
   for (i=1; i<num_proc;i++) {
      CREATE(SlaveStart);
   }
   SlaveStart();
   WAIT_FOR_END(num_proc-1);
}

void SlaveStart()
{
   for (j=0; j<3; j++)
      disp[disptplus][MyNum[j] += ...

}
```

(a) explicitly threaded

```
main()
{
   for (i=0; i<ARCHnodes; i++)
      for (j=0; j<3; j++)
      disp[disptplus][MyNum[j] += ...
}
```

(b) implicitly threaded

```
main()
{
#pragma omp parallel for private(j)
   for (i=0; i<ARCHnodes; i++)
      for (j=0; j<3; j++)
      disp[disptplus][MyNum[j] += ...
}
```

(c) openmp

Fig. 2.1. Three programming paradigms of parallelization

2.1.1 Explicit Threads

In an explicitly threaded paradigm, software explicitly creates and terminates threads. Data is partitioned in advance to avoid hazards or inter-thread primitives, such as locks or semaphores used to synchronize data accesses. Inter-thread dependencies are resolved using primitives such as barriers.

Figure 2.1(a) shows an example of explicit threads. Each thread except the main thread is forked explicitly, and a barrier is used to avoid data race.

2.1.2 Implicit Threads

Unlike explicit threads, an implicitly threaded paradigm does not require explicit thread control primitives. Such a paradigm assumes that the data and control flow dependencies are in sequential program order. The hardware has to infer such order to maintain those dependencies. Since the hardware gets no hint from the software, speculation techniques are used to ensure correctness and deliver performance.

An example of implicit threads is shown in Figure 2.1(b). Like the normal sequential loop, the iterations are duplicated and assigned to different implicit threads by a compiler, and the threads are dispatched at the same time. All the threads except the first iteration are marked ”speculative”. Before any speculative thread commits its computation, it must first check whether any data hazards have occurred. If hazards have occurred, the thread will be squashed and restarted.

2.1.3 Hybrid: OpenMP

To alleviate the burden of explicitly programming with threads while still maintaining the performance advantage of explicit threaded application, OpenMP uses compiler directives to instruct the compiler to partition the workload and library routines to support runtime dynamic parallelism. This approach combines the simplicity of implicit threads while still approaching the performance of explicit threads. OpenMP programs could use three programming flavors: Loop-level OpenMP, SPMP with OpenMP and nested parallelism with OpenMP, as described in [38]. Figure 2.1(c) shows an example of OpenMP threads.

Most of the applications used in decade-long studies are explicitly threaded applications. The rest of this thesis focuses on explicitly threaded applications and OpenMP applications. In particular, we choose the most popular parallel benchmarks for shared-memory systems: Splash-2 [71], the NAS Parallel Benchmark (NPB)[5], some commercial workloads (database and web server) [29, 7] and SpecOMP [2]. Among those well-studied parallel benchmarks, Splash-2, NPB, database and web server are explicitly threaded, while NPB-OpenMP and SpecOMP use OpenMP directives.

2.2 Characterizing Parallel Applications

As compared to executing single-threaded applications, executing multi-threaded applications poses new challenges. The threads tend to communicate with each other and compete for shared resources. A well-balanced execution of multi-threaded applications can maximize the efficiency of a CMP system. Balance in execution is not easy to achieve, however, as most of the real world multi-threaded applications require hand-tuning for desired performance and/or

efficiency. This section will try to identify the factors leading to the imbalanced execution of such applications.

2.2.1 Load Imbalance due to Synchronization

The parallel activities (processes/threads) of a parallel program often need to exchange information during their execution. Synchronization is used both to avoid race conditions and to ensure the validity of the data values that a processor uses in its computation. One commonly used synchronization construct is the *barrier*. A barrier construct is typically initialized with a count field, say x. A processor making a barrier call, is stalled, or blocked until x-number of processors have come to that barrier. This event is pictorially depicted in Figure 2.2 with four processors, where P3 gets to the barrier first and needs to wait for time $s3$, before the last processor, P0, also gets to the same barrier (whose stall time is 0). The other two processors incur stall times between 0 and $s3$.

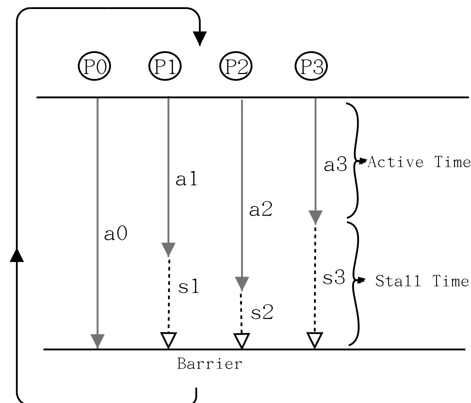


Fig. 2.2. Active and stall times using a barrier that is being invoked iteratively.

Reducing or eliminating the stall times (*s1* through *s3* in Figure 2.2) is a promising way to optimize the power consumption problem without affecting performance. We present the details in Chapter 3.

2.2.2 Load Imbalance due to Competition of Shared Resources

In addition to the imbalance in execution time, we also study load imbalance among different threads and the variation of load over time. What interests us the most is the imbalance of loads at the L2 level and how we could exploit such behavior to optimize performance and power.

2.2.2.1 Intra-Application Heterogeneity

In order to understand how an application's characteristics impose a heterogeneous load on the L2 cache, we next define two important metrics. We call these metrics the *Spatial Heterogeneity Factor (SHF)* and the *Temporal Heterogeneity Factor (THF),* and they are defined as follows:

$$SHF_{epoch} = \frac{\sigma_{cpu}(L1Misses)}{L1Accesses_{epoch}}$$

$$THF_{cpu} = \frac{\sigma_{epoch}(L1Misses)}{L1Accesses_{cpu}},$$

where SHF_{epoch} and THF_{cpu} correspond to the spatial heterogeneity factor for a short period of time across the CPUs and the temporal heterogeneity factor for a single CPU over the whole execution time, respectively. The $\sigma_{cpu}(L1Misses)$ represents the standard deviation of the L1 misses across the CPUs for a short period of time, while $\sigma_{epoch}(L1Misses)$ represents the

standard deviation of the L1 misses over the execution time for a single CPU. $L1Accesses_{epoch}$ and $L1Accesses_{cpu}$ give the number of L1 accesses within that period (across all processors) and the number of L1 accesses by a processor (over the execution time), respectively.

Essentially, these metrics try to capture the standard deviation (heterogeneity) of loads imposed on the L2 structure (the L1 misses) between the CPUs (spatial) over a short period of time (temporal). The metrics are weighted by the number of L1 accesses in order to accurately capture the resulting effect on the overall IPC. In other words, in an application where although the standard deviation of loads may be high, if overall accesses/misses are low, there is no significant impact on the IPC.

2.2.2.2 Inter-Application Heterogeneity

Inter-Application Heterogeneity refers to the load imbalance where multiple applications are running on CMPs. Each of these applications could be either single-threaded or multi-threaded. For example, a memory-intensive application can run well alongside a CPU-intensive application, since the two applications stress different parts of the system. We can also use the Spatial Heterogeneity Factor (SHF) and Temporal Heterogeneity Factor to describe their behaviors.

Both intra-application heterogeneity and inter-application heterogeneity are discussed and evaluated in detail in Chapter 4.

2.2.2.3 Shared Blocks versus Privately Accessed Blocks

Another interesting aspect of the application behavior is the sharing pattern, or the way cache blocks are accessed by different cores. This could direct us to optimize the latency to

access shared blocks. Intuitively, we would like to put actively shared blocks closer to all the cores, and put the exclusively accessed blocks closer to the core that is accessing it.

In Chapter 5, we define the *running distance* of cache block accesses, which is the number of times the same core accesses a particular block before other cores request the block The distribution of running distance shows the activeness of cache blocks being accessed by multiple cores. If the running distance is low, multiple cores will interactively access the block. Otherwise, it means one core will access the block for a significant period of time before another core begins accessing it.

The details are presented in Chapter 5, where we demonstrate that a running distance of 5 or more is quite uncommon for the studied benchmarks.

2.3 Related Work

[71] characterizes the SPLASH-2 programs in terms of fundamental properties and architectural interactions. The authors investigated computational load balancing, the communication-to-computation ratio, traffic needs, important working set sizes, and issues related to spatial locality. In [58], the performance of commercial workloads, namely online transaction processing (OLTP) and decision support system (DSS), was evaluated on a shared-memory multiprocessor system that consisted of out-of-order processors. The authors demonstrated how various ILP features, such as the size of instruction, issue and execution windows, could effectively improve performance. They collect traces of Oracle server processes on a 4-node Alpha server using ATOM, and subsequently feed those traces to a ccNUMA machine based on RSIM. The results presented focused more on instruction-level parallelism (ILP) than on thread-level parallelism (TLP). In [70], architectural requirements and scalability of NAS parallel benchmarks

were presented for two different platforms: a cluster of workstations and a ccNUMA system. [49] studied the TLP and ILP performance of three applications on an SMT system. The authors identified the bottlenecks caused by overloading of threads and suggested that ILP and TLP are not interchangeable and should be treated separately.

Several studies have been conducted that characterize benchmarks which use OpenMP programming models. [3] characterized the SpecOMP benchmark using hardware counters on a 4-node Sun SMP system. The authors presented detailed statistics of the pipeline (branch misprediction and FP dependency) and memory system stalls (I-cache misses and load dependency). Those statistics are then used to derive the instructions per cycle (IPC), communication-to-computation ratio, and memory-accesses-to computation ratio. [45] studies the cache-coherence traffic of a variety of OpenMP benchmarks on a 4-node SMP system. Their cache simulator is modeled after a cache-coherent shared-memory multiprocessor system. They instrument the binary to obtain the memory reference traces, and feed the traces to their cache simulator. They show that a significant number of misses are caused by invalidations, some of which are not caused by true-sharing. [27] presents the performance of the basic OpenMP constructs for the NAS Parallel Benchmark OpenMP version and for the SpecOMP 2001 application benchmarks. [46] proposes thread-level speculation to allow the waiting threads to execute beyond barriers. In this paper, the authors present data sharing and threads-waiting statistics, and show performance gain from using thread-level speculation. They model 16-node and 64-node ccNUMA machines with release memory consistency.

Chapter 3

Synchronization Characterization and Barrier Optimization

3.1 Introduction

While CMP design has clear advantages over other approaches as discussed in Chapter 1, the denser levels of circuit integration make power dissipation a serious concern in chip design. Slowing down the clock to accommodate lower supply voltage levels to reduce power consumption can have significant ramifications on the performance of applications. As a result, it is imperative to design techniques that can provide considerable power savings without having a significant impact on performance. A symbiotic interaction between the underlying hardware and the high level software running on it can go a long way towards meeting this goal. While the underlying hardware can provide several mechanisms for controlling power consumption, the software can use high-level workload information to modulate the hardware mechanisms effectively to maximize the rewards.

In this chapter, we present a novel example of this type of symbiotic relationship between a hardware power saving mechanism (dynamic voltage-frequency scaling (DVS) [14]) and the application software, wherein the latter exploits high-level workload information regarding work imbalance between the parallel threads to control operating frequencies/voltages.

Applications running on CMPs can be expected to take advantage of the multiple processor cores by creating threads that run in parallel. Synchronization is needed to ensure that data and control dependencies are correctly enforced before a processor performs the next chunk

of work assigned to it. At such synchronization points, a processor may need to wait for one or more processors to arrive at a specific part of the code before all can proceed. As a result, processors can be forced to stall or wait for a considerable amount of time. We believe that one can exploit the processors idling at synchronization points to scale the frequencies/voltages appropriately. Doing so allows us to minimize the amount of idling without impacting the overall execution time, and lowering the frequency/voltage yields power savings.

One such synchronization construct that is popularly used in parallel programs is a *barrier*. When a processor comes to a barrier that is pre-set to a specified count *x*, it is stalled until *x-1* other processors also come to that barrier. All processors, except for the last one(s) to get to the barrier, therefore waste a considerable amount of time which could otherwise be devoted useful work. Elimination of this idleness will not contribute to lengthening the program execution. However, it can help in power optimization in two ways. One way to exploit the idleness is to transition the processor from a fully active state to a low power state when it arrives at the barrier, and re-initiate a fully active state when the last processor arrives at the barrier. This is the option that is explored in [41]. An alternate strategy, which is explored in this chapter, is to lower the frequency/voltage of processors that would have reached the barrier ahead of time, while they are executing useful work. In this way, we ensure that all processors reach the barrier at the same time. As long as we are able to predict the idleness at the barrier accurately, by applying DVS we can reduce power consumption during periods of useful work without affecting execution time.

The effectiveness of this approach depends largely on (i) the imbalance among the processors in the work assigned before getting to the barrier, and (ii) the ability to predict this

imbalance (i.e., the idleness of a processor at the barrier) accurately. Consequently, it is important to examine the execution characteristics of real applications with a detailed architectural model in order to evaluate the pros and cons of this approach. In this chapter, we investigate this issue using applications from the SpecOMP[2] suite. These are parallel applications written using OpenMP [22] directives, which are meant to be representative of realistic parallel workloads. We execute these applications on a detailed CMP platform that models multiple processor cores, their caches and an on-chip interconnect between the cores. We use the Simics [43] complete system simulator that models the operating system code as well.

From our simulations, we observe that there is considerable work imbalance between the processors, suggesting that there is significant scope for power savings using our technique. We also demonstrate that simple predictors based on prior history (in fact, the immediately previous value suffices in many cases) can provide an accurate estimate of processor idleness at the barriers. Based on these observations, our barrier-based frequency/voltage scaling strategy provides an average of 32% energy savings across the benchmarks, with only a 3.7% degradation in performance on the average.

The rest of this chapter is organized as follows. The next section discusses related work; Section 3.3 and 3.4 elaborate on our method; Section 3.5 gives our experimental setup and Section 3.6 presents the performance and power results.

3.2 Related Work

3.2.1 Multi-Core Processors

In [40], the authors discuss the possibility of using single-ISA heterogeneous cores to attack the power consumption problem. Since the previous generations of cores are much smaller and consume less power than the current generation does, dynamically switching between cores can increase energy efficiency. Due to the fact that only one core can be active at any time, such an architecture generally targets serial applications.

ACPI (Advanced Configuration and Power Interface)-type power management is exploited on barrier constructs in [41] to investigate the power efficiency of parallel applications. The authors argue that a core can be put into sleep mode when it reaches the barrier early. However, on-off control between sleep- and full-operation modes can be very costly. Furthermore, scaling of voltage during the execution, i.e., prior to arrival at the barrier, can possibly provide significant energy savings.

An earlier study [61] points out that the CMP can be an energy-efficient alternative to exploiting future billion transistor designs. The same study also mentions that voltage scaling can further complement this architecture, and the authors show from 9% to 15% power savings in multimedia applications that use independent threads. Our work is motivated by this observation and demonstrates a novel way of applying voltage scaling to show energy savings in the context of SpecOMP parallel applications.

3.3 DVS in Chip Multi-Processors

At least two alternatives are available for implementing DVS in CMPs. In the first one, depicted in 3.1(a), the scaling is applied to all processors uniformly. In this case only one supply voltage regulator and programmable PLL is needed. Since the processors are all running at the same clock rate and supply voltage, the interconnect between them can be synchronous and can run at the current clock rate.

In the second alternative, depicted in 3.1(b), each processor has its own supply voltage regulator and programmable PLL. Thus, each processor can be set to its ”ideal” clock rate with its supply voltage determined by its computational load. However, since the processors are no longer running in lock-step, an asynchronous design (multiple clock domain [62] or globally asynchronous locally synchronous (GALS) design [30]) will be required, incurring the additional overhead of request/acknowledge lines, circuit and timing overhead. Additionally, buffers may be required at the sender and/or receiver to accommodate processor speed mismatches.

The PLL and main driver accounts for about 10% of the clock energy [23], and this percentage increases as the technology scales. If we employ the second approach, the single PLL and main clock driver are replaced by local PLLs, and the main driver can be eliminated. Since the PLL itself does not consume much energy, we conservatively assume it uses 1% of the clock energy. Therefore, we could expect about 6% savings in clock system energy by employing asynchronous design for a four-core processor.

In [62], the frequency and voltage of different regions of a single core processor, namely front-end, integer unit, floating-point unit and load-store unit, are adjusted independently and dynamically. The study shows that 16.7% energy reduction could be achieved while incurring

a 3.2% performance loss. Similar results are shown in [32]. Considering that the parallelized applications tend to have partitioned work to achieve good speedup, especially with the MIMD (or SPMD) programming style, the interconnect between the cores do not need to be as closely coupled as the different units of a single core processor. Consequently, it could be more rewarding to bring multiple clock domains to different cores of a CMP than to the different regions of a single core processor.

Since our methodology exploits work imbalances across the cores, we would like to employ separate scaling for each core, and thus use the second configuration shown in Figure 3.1.

3.4 Our Approach

In this section, we give a quick overview of the barrier construct that is in wide use in parallel programs, following which we present our proposed optimizations.

3.4.1 Barriers

The parallel activities (processes/threads) of a parallel program often need to exchange information during their execution. Synchronization is used to avoid race conditions, and to ensure the validity of the data values that a processor uses in its computation. One commonly used synchronization construct is the *barrier*. A barrier construct is typically initialized with a count field, say x. A processor making a barrier call, is stalled/blocked until x processors in all have come to that barrier. This is depicted in Figure 3.2 with four processors, where P3 gets to the barrier first, and needs to wait for time $s3$, before the last processor, P0, also gets to the same barrier (whose stall time is 0). The other two processors incur stall times between 0 and $s3$.

This barrier construct is in wide use in parallel programs, since it is an easy way of demarcating phases of a program, or a way of ensuring each processor is done with one or more iterations of a loop, before proceeding. Consequently, it is extensively used at loop iteration boundaries (as in Figure 3.2) to ensure that the data values which every processor reads in later iterations are not stale. Many shared memory programming libraries support this barrier interface, including OpenMP [22] that is becoming a standard for shared memory parallel programs. Our experimental evaluations use parallel programs written using OpenMP, though the optimizations that we propose below are general enough to be useful across a wide class of parallel programs. Note that we are not proposing any new barrier construct itself. Rather, we use the existing barrier construct (implemented as busy-wait in OpenMP) and simply insert code around this construct as will be detailed shortly.

3.4.2 Proposed Optimizations

Reducing/eliminating the stall times (*s1* through *s3* in Figure 3.2) is a promising way to optimize the power consumption problem without affecting performance, as long as we do not affect the time taken by the last processor to get to the barrier. There are two ways of tackling this issue: (i) transiting the processor core to a low power mode over the duration of the stalls (as explored in [41]), or (ii) applying DVS during the work execution periods of P1, P2 and P3 (i.e., during times *a1, a2, a3*), such that we push *s1* through *s3* close to zero. We believe that the latter approach provides larger scope for power savings (since the work execution periods - referred to as active times - are typically much longer than the wait times, and a lower voltage/frequency during those periods can provide significant power savings), and the transitioning costs between voltage/frequency levels are also much lower (typically a few cycles as in [19]) than powering

up/down the core completely. As we will show later in our experimental results, our technique does provide higher power savings.

Our study mainly focuses on barrier constructs that are used in loops to avoid data hazards across different iterations. It is to be noted that loops constitute the bulk of the execution time of most applications, and any optimizations within loops can contribute to substantial overall savings. Consider the following program fragment where the application uses an OpenMP compiler directive (`#pragma omp parallel`) to parallelize a loop construct that is itself inside an outer loop:

```
for (i = 1 to niter) {
  #pragma omp parallel
  for (j = 1 to N) {
    ....
  }
}
```

The compiler translates this code fragment to the following form that is executed by each of the parallel threads (whose creation is not explicitly shown here), where `mystart` and `myend` delimit the iteration space for each thread.

```
for (i = 1 to niter) {
  for (j = mystart to myend) {
    ....
  }
            <-- (A) Arrive at Barrier
  barrier();  <- Inserted by compiler
            <-- (B) Leave Barrier
}
```

The barrier ensures that each processor is done with the inner loop before they all start the next iteration of the outer loop. The compiler analysis to figure out when and what kinds of synchronization to use is quite involved. In this study, we simply use an off-the-shelf OpenMP compiler that inserts these barrier calls, and additionally insert code at points (A) and (B) to perform certain functionalities as explained below. The overheads for these are captured in our experimental results.

Setting the frequencies/voltages to reduce the idleness at the barrier in the above example requires estimating the stall times of a processor at the barrier, and the time it spends in executing useful work itself (the inner loop) before it gets there. Let us say that we estimate the time to be spent in useful work for the next iteration of the outer loop to be a at the maximum clock frequency f_{max}, and the idle time at the barrier to be s. Then we can set the frequency (the compiler can generate code for doing this at point (B) in the program) for the next outer loop iteration as $f = \frac{a}{a+s} \times f_{max}$. We assume that the hardware supports an instruction (as in [19, 56]) to effect such frequency changes. Since the frequency levels that are offered by the hardware are discrete, once we calculate the target frequency using the above approach and find that it falls in-between two levels, we set it to the higher of the two levels in order to not penalize performance.

There are different techniques that can be used for predicting the active and idle times, and sophisticated predictors can be developed for doing so. For most of our experiments, we use a simple predictor – *last-value predictor* – wherein the current observation (for both a and s) is used as the estimate for the next iteration. We show that in most cases such a simple predictor suffices. In one application alone, we find a Markov predictor (as in [34]) can provide better estimates than the last value predictor. The last value, or the history of previous values (if a

more sophisticated scheme is needed), of active and idle times can be obtained by reading the processor clock at points (A) and (B) in the program, and the compiler emits code for doing so.

3.5 Experimental Setup

We implemented our approach using Simics [43] and performed extensive experiments with several parallel applications from the SpecOMP suite [2]. Simics is a platform for full system simulation that can run actual application code and completely unmodified kernel and driver code. We modified the simulator by enhancing it with accurate cache timing models and then captured the time differences between processors in arriving at barrier (synchronization) points as explained in the previous section.

The chip multiprocessor under consideration in this chapter is of the shared multiprocessor kind, where a certain number of CPUs share the memory address space. We assume that each CPU has its own private L1 instruction and data caches that it can access without going across a shared interconnect. Several proposed CMP designs from industry and academia already use such private L1-based configurations [8]. We keep the subsequent discussion simple by using a shared bus as the interconnect (though one could use higher bandwidth interconnects as well). We also use the MOESI [20] protocol (the choice is orthogonal to the focus of this chapter) to keep the caches coherent across the CPUs. We also assume that there is a shared L2 data cache. Finally, we assume the existence of hardware mechanisms for implementing DVS for each core independently.

The default simulation parameters used in our experiments are shown in Table 3.1. Unless stated otherwise, all experiments use these parameters. We performed experiments with two, four, and eight discrete voltage/frequency levels and the values used are listed in Table 3.2.

When we make experiments with two frequencies/voltages, we use 1000/1.30 and 533/0.95; and, when we make experiments with four voltage levels, we use 1000/1.30, 800/1.15, 533/0.95, and 300/0.80. These voltage/frequency values are similar to those employed by Transmeta's Crusoe [19]. The important characteristics of our benchmarks are given in Table 3.3. The second column gives a brief description of each benchmark and the third column gives the number of parallel loop nests in each benchmark. The next column lists the important loop nests, i.e., the loop nests that take bulk of execution time. Each loop is named after its host function and the line number of source code. For example, applu.rhs.34 refers to the loop located in function rhs at line 34 of application applu. Finally, the last column gives the number of instructions (in millions) simulated for each benchmark.

In this section, unless stated otherwise, the term "energy" is used for energy consumption in the data-paths of the processors. Voltage/frequency scaling is applied only to the data-path of the processors to save energy; the caches and system interconnect components operate with the highest available voltage/frequency. This is to ensure that we do not affect the snoop performance of the caches and interconnects, even when the data-path is operating at a lower frequency. The term "execution cycles" is used to denote the total number of cycles taken by the application when the number of instructions shown in the last column of Table 3.3 is simulated. In our results, we focus on two metrics: (i) *normalized energy consumption*, and (ii) *percentage increase in execution time*, and quantify these for the proposed optimization with respect to a system without these optimizations.

Parameter	Default Value
Number of Processors	4 and 8
L1 Size	8KB
L1 Line Size	32 bytes
L1 Associativity	4-way
L1 Latency	1 cycle
L2 Size (Total Capacity)	2MB
L2 Associativity	4-way
L2 Line Size	64 bytes
L2 Latency	10 cycles
Memory Access Latency	120 cycles
Bus Arbitration Delay	1 cycles
Replacement Policy	Strict LRU

Table 3.1. Base simulation parameters used in our experiments.

3.6 Results

We present our experimental analysis in several parts. First, we document results that illustrate the time variance between processors in getting to the barrier points. Following this, we discuss whether it is possible to predict the idle time a processor waits at a given barrier point. After that, we present our energy saving and performance results to give a picture of how our approach behaves. To explain our findings, we also give data showing how our approach utilizes available frequency levels. Finally, we present a comparison of our approach to the one proposed in [41].

3.6.1 Disparity at Barrier Arrival Times

Figure 3.3 gives CDF (cumulative distribution function) graphs that show the time difference (in terms of cycles) between the processors arriving at barrier points for the four processor case. Each graph corresponds to a single loop (with some representative loops of each application being shown) and plots four curves (one per processor). For each graph, the curve marked

Freq (MHz)	1000	900	800	677	533	433	300	150
Voltage (V)	1.30	1.25	1.15	1.05	0.95	0.875	0.80	0.75

Table 3.2. Frequencies and voltages used in the evaluation.

CPU0 corresponds to the processor which arrives last at the barrier point (i.e. its idle time is zero). We first describe its curve as it is interpreted differently from the remaining three curves. An (x,y) point in the CPU0 curve indicates that fraction y of the time (i.e., fraction y of total arrivals to this loop) the CPU0 spends x cycles or less executing the loop (the active time is the same as the total loop execution time in this case since there is no idling) before coming to the barrier. The remaining curves, on the other hand, reflect the time duration the corresponding processor waits at the barrier point. More specifically, a point (x,y) says that, for a fraction y of the time, the processor in question waits x cycles or less at the barrier point. Therefore, if the curves for CPU1 through CPU3 are close to the CPU0 curve, this indicates larger idle times for CPU1 through CPU3 (thus, more opportunity for our approach to save energy). Looking at these figures, we see that different loops (even those that belong to the same application code) exhibit different behaviors. For example, while in mgrid.zero3.15, there is not much opportunity to save power, applu.rhs.34 indicates that we can save substantially through DVS. In general, we find that there are several opportunities brought by barrier arrival time disparities that we can benefit from.

Still, even a large disparity between processor arrival times may not necessarily indicate that we can predict the idleness. In order to exploit this idleness through DVS, we need to be able to predict it. The next subsection discusses this point in detail.

Benchmark	Description	Important Loops	Number of Parallel Loops	Simulated Inst. (in millions)
applu	solution of five coupled nonlinear PDE's	rhs.34, rhs.56, rhs.178	15	1,770,699
apsi	Lake environment model operates on a 112x112x112 area array of data and iterates over 70 time steps	all except advc.1330, advt.1461, advu.1675, advv.1847	21	93,630
galgel	numerical analysis of oscillatory instability of convection	dswap.4222, dgemm.435, dger.5067, dtrsm.8220	21	111,324
mgrid	demonstrates the capabilities of a simple multi-grid solver in computing a three dimensional potential field.	zero3.15, comm3.176	3	125,887
swim	weather prediction operates on a 1335x1335 area array of data and iterates over 512 time steps	shalow.116, calc1.276, calc2.331 calc3z.381	8	39,272

Table 3.3. The benchmark codes used in this study.

3.6.2 Idle Time Predictability

As noted earlier, we can track the time that a processor arrives at the barrier and the time it leaves the barrier. The idleness at the barrier can be calculated using the difference between these two (discounting costs for the implementation of the barrier itself), and the active times can be calculated by taking the difference between the current arrival at the barrier and the previous departure from the barrier. In order to scale the voltage, we need to predict the idle and active times for the next iteration (to set the frequency to $\frac{a}{a+s} \times f_{max}$) upon leaving the barrier. Our default predictor is a history-based one, which predicts that the next idle and active times (for the next iteration) will be the same as that for the current iteration. In the rest of this chapter, this predictor is referred to as the *last-value predictor*. The curves in Figure 3.4 show, for each benchmark code, the accuracy of the distribution of predictions of a, summed over all processors and all loops in the benchmark. The x-axis shows the difference between the predicted loop execution a_{pred} and the real a_{actual} in terms of $\frac{a_{pred}-a_{actual}}{a_{actual}}$. Consequently, a negative value corresponds to underestimation, whereas a positive value implies overestimation. The y-axis gives the percentage of time a particular accuracy (on the x-axis) is achieved. Obviously, a large concentration around 0 on the x-axis indicates good predictability. We see from these graphs that, except for mgrid, the predictability of the remaining four benchmarks is quite good. In mgrid, we mostly underestimate the active times, and as will be explained shortly this forces our approach to operate with lower frequencies/voltages that can hurt performance. As for the other benchmark codes, we can expect energy savings through the last-value predictor-based DVS strategy without impacting performance significantly.

3.6.3 Energy and Performance Results

We present energy and performance data for two idleness predictors. The first one is the last-value predictor described above. The second one is an *oracle predictor*, which performs perfect prediction for each processor at each loop arrival. Clearly, the oracle is not implementable; the reason that we give its results here is to check how much inaccuracy the imperfect prediction brings and what its consequences are from both performance and energy angles.

For each application in Figure 3.5, the bars marked 2Levels, 4Levels and 8Levels correspond to two, four, and eight voltage/frequency levels using the oracle predictor with four and eight processors. One can see from these results that, except swim, our benchmarks take advantage of increased number of voltage/frequency levels. The reason that swim does not generate good results (savings) is because it has little opportunity for scaling voltage/frequency, as illustrated for some of its loops in Figure 3.3. When we look at the eight processor results with the oracle predictor, we see similar trends. In fact, in most cases, the results with 8 processors show better energy savings than those with 4 processors. The main reason is that the work imbalance increases with a larger number of processors, generally resulting in longer idleness as can be seen from the average idle times for the 4 and 8 processor configurations for several loops in Table 3.4. Consequently, the importance of exploiting idleness grows with increasing number of processors.

We now focus on the last-value predictor and present its energy results in Figure 3.6 for the four and eight processor cases. We observe similar results as for the oracle predictor suggesting that our predictor is doing a good job. We note that the last-value predictor results show better savings for mgrid with 4 processors. The main reason for this because we underestimate

Loop Name	4 Procs	8 Procs
applu.rhs.34	31.4%	29.7%
applu.rhs.56	25.1%	29.0%
applu.rhs.178	21.5%	37.2%
galgel.dswap.4222	0.551%	47.1%
galgel.dger.5067	59.3%	64.5%
galgel.dtrsm.8220	2.11%	8.04%
mgrid.zero3.15	33.2%	40.4%
mgrid.comm3.176	33.2%	40.4%
swim.shalow.116	1.21%	0.96%
swim.calc1.276	1.47%	2.13%
swim.calc2.331	2.27%	1.33%
swim.calc3z.381	2.61%	2.67%

Table 3.4. Idle times at barrier averaged over all processors and iterations for selected loops from our benchmarks.

the active time in this application (see Figure 3.4), causing more operation at lower frequencies than with the oracle predictor.

To further gain insight on these results, we give the frequency distribution for the last value predictor and oracle predictors for two of our benchmarks under 8 voltage levels: mgrid and galgel in Figure 3.7. A bar in these graphs gives the percentage of time the corresponding frequency level (and the associated voltage) is exercised. We see that the frequency distributions of the last-value predictor and the oracle predictor in galgel are very similar to each other, which explains the similar energy trends as well. In mgrid, the last-value predictor (which is not very accurate) puts the processors at lower frequencies, which gives power savings at the expense of performance loss.

It must be noted that the oracle predictor does not have any performance penalty since it selects the most suitable voltage/frequency for the idleness in question. However, the last-value predictor can lead to performance degradation when it underestimates the active times.

The reason is that underestimation of a leads to selection of lower voltage/frequency than the optimum one, and this can increase the execution time of the parallel loop under consideration. The percentage increases in execution time of our benchmarks are given in Figure 3.8. We see that the performance degradation in most benchmarks is less than 10% for both four processor and eight processor cases. In particular, the performance penalty is really low in galgel since our predictor is very accurate (Figure 3.4). The reason for the negligible degradation in swim is that we do not have much opportunity in this benchmark for voltage/frequency scaling at all. The largest degradation occurs with mgrid as a result of underestimates of a as explained earlier, and we look to address this problem in the next subsection.

3.6.4 Using a More Powerful Predictor

We further studied the behavior of mgrid and found that the behavior exhibited by its idleness pattern cannot be captured adequately by the last-value predictor. However, when we examined the time series patterns of active and idle times, we found a pattern that repeats itself, suggesting that a Markov predictor could be very useful in this case. We use a simple Markov predictor (described in [34]) that uses a small table indexed by the last value to give the next estimate. The normalized energy consumptions for mgrid with the Markov predictor are given in Figure 3.9. For ease of comparison, we also reproduce the energy results with the last-value predictor. We see that, for both 4 and 8 processors, the Markov predictor leads to less energy savings than the last value predictor. However, when we look at the performance overhead results (shown in Figure 3.10), we see that the performance penalty is considerably lower. Note that this simple Markov predictor also subsumes a last-value prediction mechanism, and can be a more generic way of implementing active/idle time prediction. When we compare the running

frequencies of mgrid in Figure 3.7 and in Figure 3.11, it is clear that the Markov predictor is able to reduce the underestimates of active times, thus putting the processors more frequently at the higher frequencies.

3.6.5 Comparison with the Thrifty Barrier

Benchmark		**4P** applu	apsi	galgel	mgrid	swim
Energy Consumption	TB-Oracle	80.7%	84.2%	99.7%	94.2%	98.3%
	Ours-Oracle	70.5%	74.6%	76.1%	77.9%	99.4%
	TB-LVP	73.7%	80.8%	99.3%	79.0%	98.2%
	Ours-LVP	67.5%	74.1%	77.1%	57.5%	99.1%
Performance	TB-Time Increase	8.31%	6.38%	0.07%	9.8%	1.40%
	Ours-Time Increase	10.48%	6.95%	3.04%	1.6%	0.36%
Benchmark		**8P** applu	apsi	galgel	mgrid	swim
Energy Consumption	TB-Oracle	66.4%	73.4%	70.2%	71.1%	98.4%
	Ours-Oracle	55.5%	63.9%	66.7%	68.4%	99.1%
	TB-LVP	67.8%	72.2%	68.7%	61.0%	98.5%
	Ours-LVP	57.0%	62.6%	64.7%	56.8%	99.4%
Performance	TB-Time Increase	7.22%	3.97%	0.31%	9.07%	1.16%
	Ours-Time Increase	7.73%	4.61%	4.53%	1.5%	0.18%

Table 3.5. Normalized energy consumption and increase in execution time of our mechanism vs. Thrifty Barrier (TB) using the same Oracle and Last-Value Predictors (LVP) for both.

As observed earlier, there are two ways of exploiting the idleness at barriers for power savings. While our approach uses voltage scaling to reduce the power during periods of active execution, a related study [41] has used the idleness to transition the processor to a low power mode (called the *thrifty barrier* approach). In order to compare our savings with those of [41], we show in Table 3.5 the energy savings of our approach (using the oracle and our prediction

mechanisms) along the savings obtained with the thrifty barrier. In the interest of fairness, we use the same oracle and the same prediction mechanisms to estimate the idleness in both the approaches.

As can be observed, whether it be perfect knowledge, or with the predicted idleness, our approach provides better energy savings than the thrifty barrier in nearly all cases (except swim where there is really no scope for power savings anyway). The main reason for those results is that the idleness durations may not be large enough to accommodate the cost of turning down the processor and bringing it back up fully for the thrifty barrier. On the other hand, our mechanism can still scale down the frequency/voltage even while idleness is not very significant. Our mechanism is able to provide these energy savings, while being comparable in performance to the thrifty barrier.

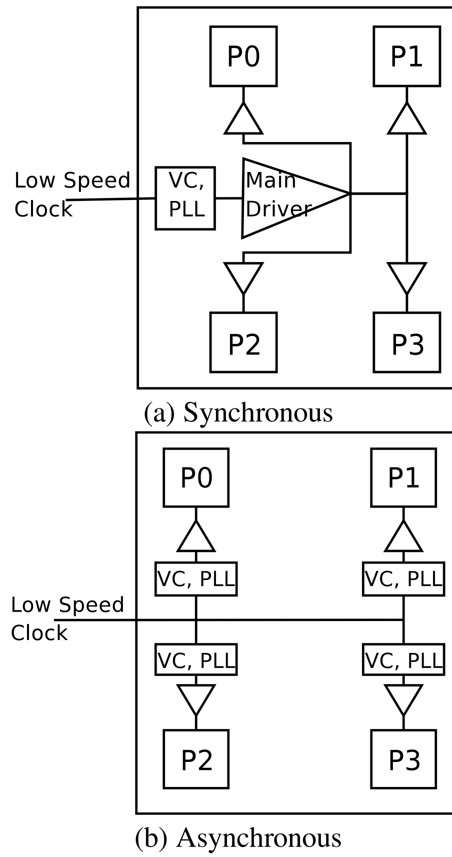


Fig. 3.1. Two dynamical voltage and frequency scaling approaches for CMPs. VC: Voltage Controller, PLL: Phase Locked Loop

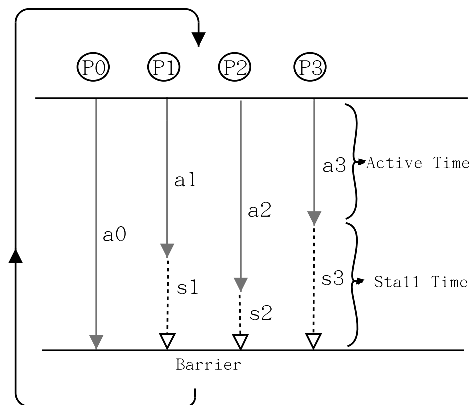


Fig. 3.2. Active and stall times using a barrier that is being invoked iteratively.

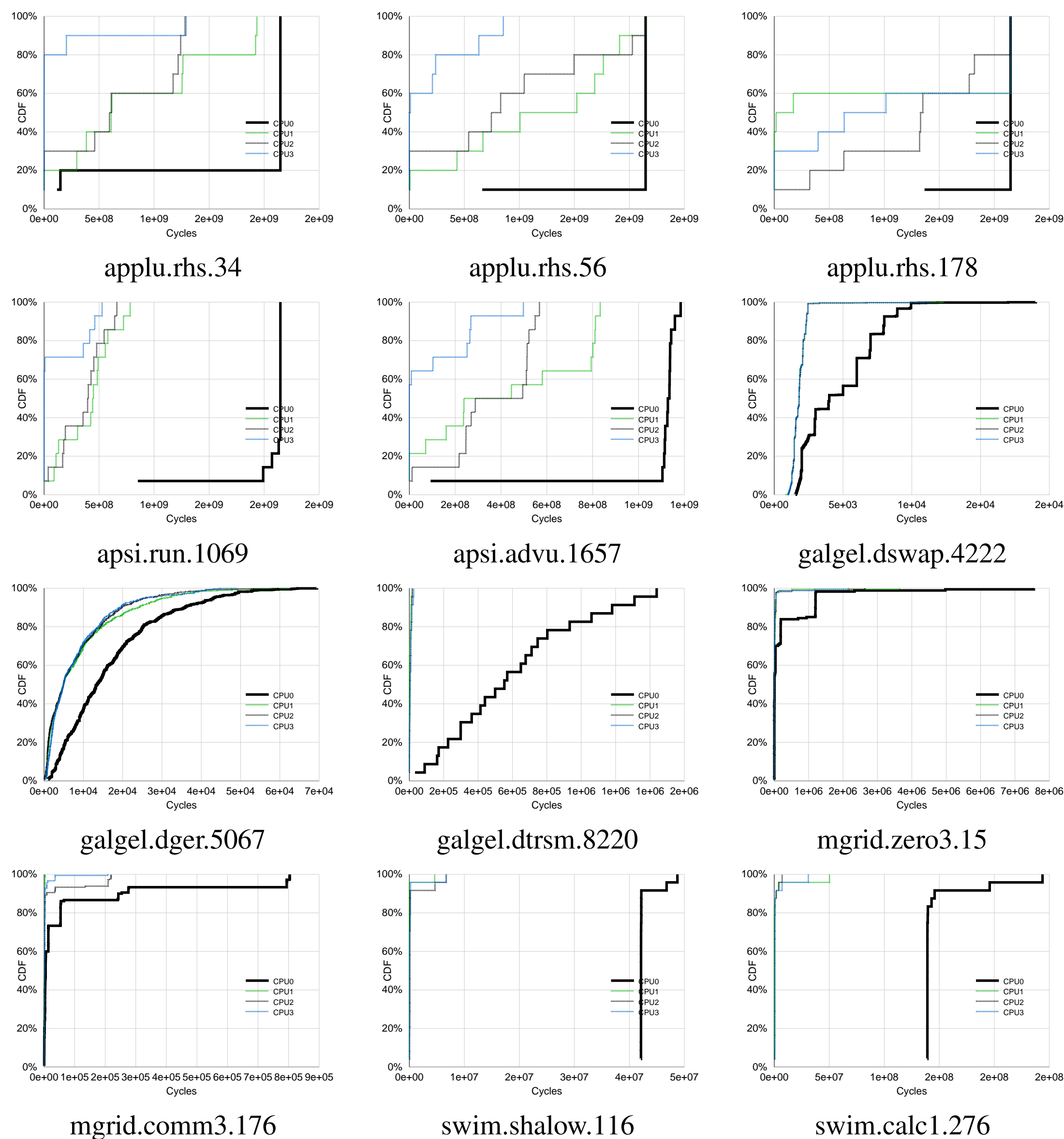


Fig. 3.3. Stall times and execution times for some important parallel loops in our applications. An (x,y) point in the CPU0 curve indicates that fraction y of the time the last processor to arrive at the barrier spends x cycles or less executing the loop before coming to the barrier. For other CPUs, a point (x,y) says that for a fraction y of the time, the processor in question waits x cycles or less at the barrier point.

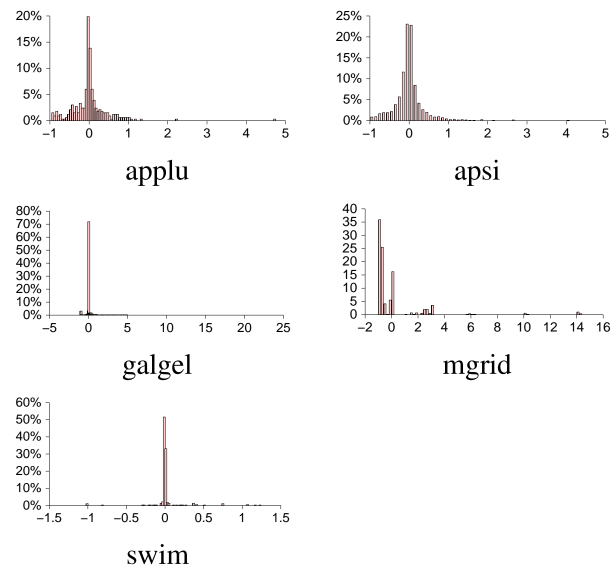


Fig. 3.4. Predictability of active times. The y-axis histograms the percentage occurrence of the normalized estimation error (i.e. x-axis is $\frac{a_{pred}-a_{actual}}{a_{actual}}$).

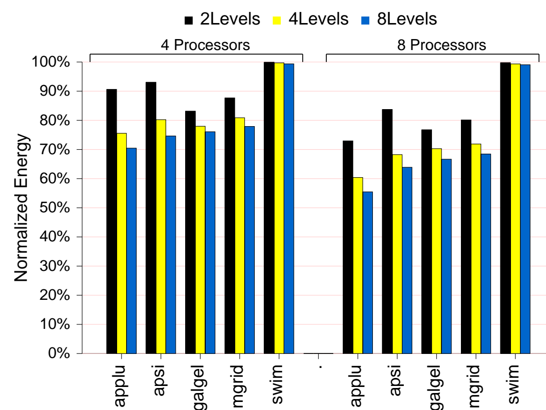


Fig. 3.5. Overall energy consumption with Oracle Predictor.

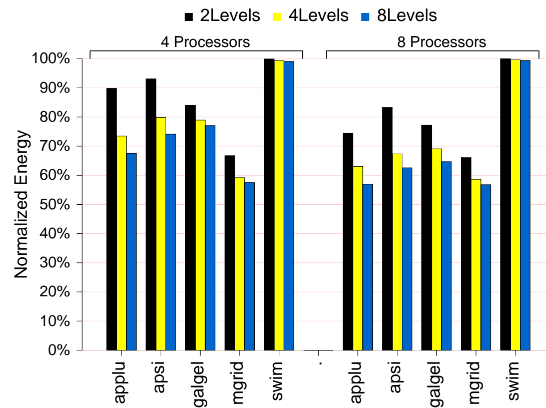


Fig. 3.6. Overall energy consumption with the Last Value Predictor.

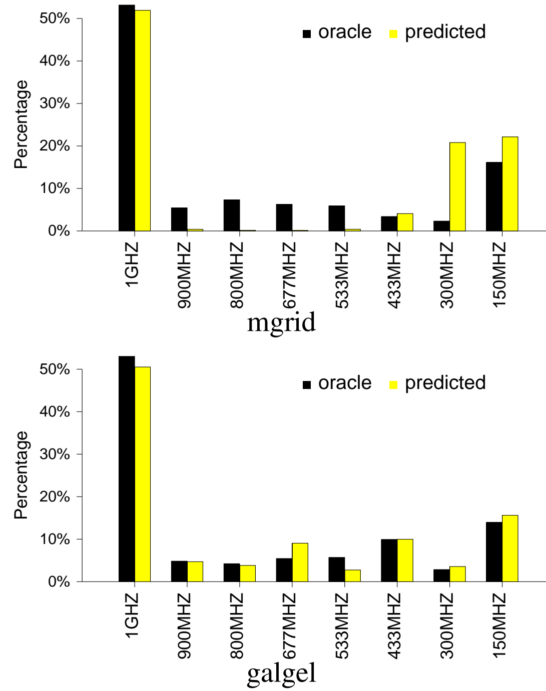


Fig. 3.7. Percentage of the execution time spent in a particular frequency (8 processors).

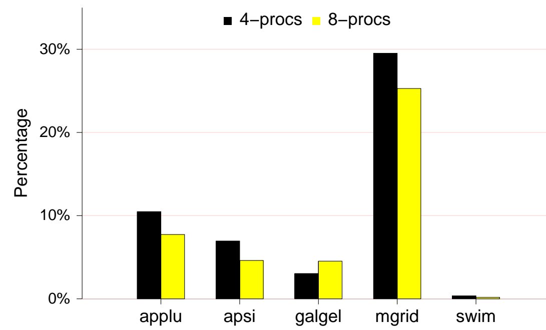


Fig. 3.8. Percentage Increase in execution time with Last-Value Predictor.

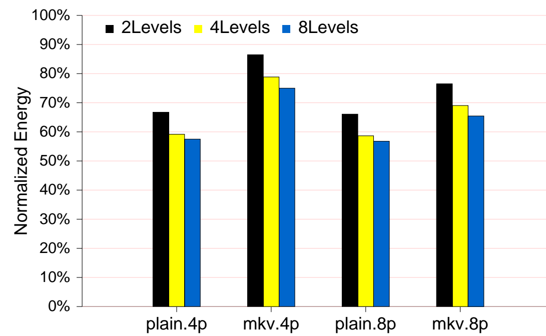


Fig. 3.9. Normalized energy consumption for mgrid: Last-Value Predictor vs Markov Predictor. (xp means x processors are used.)

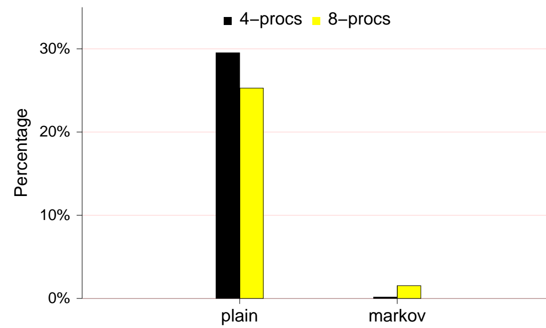


Fig. 3.10. Percentage increase in execution time for mgrid: Last-Value Predictor vs. Markov Predictor.

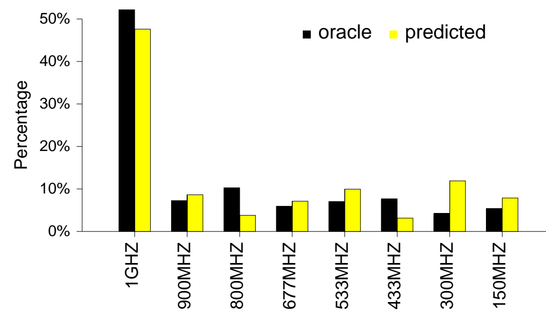


Fig. 3.11. Percentage of the execution time spent in a particular frequency for mgrid with the Markov Predictor (8 processors).

Chapter 4

Load Characterization and Uniform-Cache Optimization

As we can see in Chapter 3, the imbalances of execution time for different threads are quite predictable and such imbalances can be exploited to apply dynamic voltage/frequency scaling onto corresponding cores to reduce the power consumption. We would like to study the cause of such load imbalance to further improve performance in addition to conserving energy. Following the discussion about L2 cache demands in Chapter 2, this chapter presents the load imbalance of L2 demands across different cores. We evaluate two possible L2 organizations, private L2 and shared L2 and propose a shared processor-based split L2 cache design to accommodate the load difference without the drawbacks of private L2 and shared L2.

The chapter is organized as follows. The next section presents the CMP architecture under consideration, the existing L2 cache organizations and the mechanisms needed to implement our shared processor-based split organization. The details of the workloads, our simulation platform, the methodology for the evaluation and our experimental results are given in Section 4.3. A discussion of related work is presented in Section 4.4.

4.1 Background

The CMP under consideration in this chapter is of the shared multiprocessor kind, where a certain number of CPUs (of the order of 4-16) share the memory address space. We assume L2 to be the last line of defense before going off-chip, and consequently each CPU has its

own private L1 that it can access without crossing a shared interconnect. It is also possible that a data block can be duplicated across these private L1s. Several proposed CMP designs from industry and academia already use such private L1-based configurations. We keep the subsequent discussion simple by using a shared bus as the interconnect (though one could use fancier or higher bandwidth interconnects as well). We also use the MOESI [20] protocol (the choice is orthogonal to the focus of this chapter) to keep the caches coherent across the CPUs.

For the L2 organization, there are two possibilities. The first option is to place the L2 units on the CPU side of the interconnect (the L2 is also private) as is shown in Figure 4.1(a). These private L2 units are of equal size, and can directly serve the misses coming from L1, that is, if the misses can be satisfied from the private L2's local storage. If not, the misses needs to arbitrate for the bus and place the request on the bus to get a reply either from another cache or from off-chip memory. The other L2 caches need to snoop on the bus to check if any transaction pertains to them so that they can take appropriate action (responses for read/write requests or invalidates for upgrade requests). In this case, the coherence actions take place between the L2 caches, and not all bus transactions may need to permeate to L1 because of filtering effects by L2 (as noted in [20]). The private nature of L2 units can cause a data block to come into multiple L2 units based on the application access/sharing patterns, thereby reducing the aggregate L2 size.

The other option is to move the L2 to the opposite side of the bus. Thus it becomes a shared unit that is equally accessible across the shared interconnect to all the CPUs with equal cost (see Figure 4.1(b)). Note that while in this case the L2 appears as one big monolithic unit from the viewpoint of the CPUs, it is typically banked (divided into subunits/splits) to reduce the per-access dynamic power. Data blocks are placed in these banks based on their addresses. Consecutive blocks could either be interleaved consecutive blocks on successive banks, or one

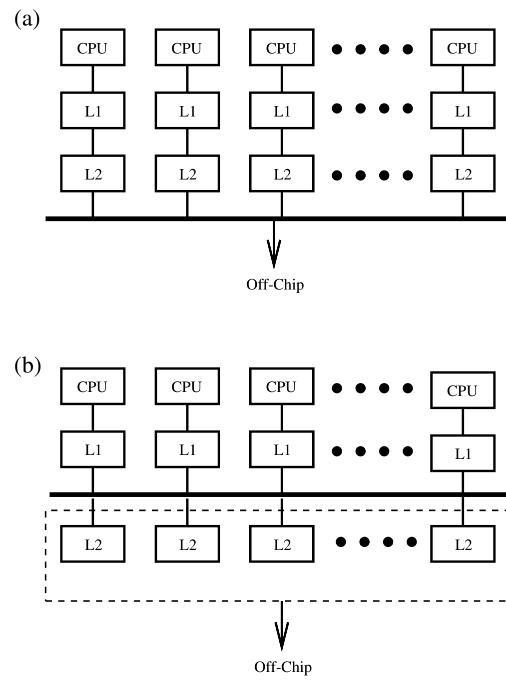


Fig. 4.1. (a) Private L2 Organization. (b) Shared L2 Organization.

bank could be filled first with consecutive addresses before going to another bank. In our study it does not matter which of these two options is chosen: we simply use the first option in the experimental evaluation. With this shared L2 organization (SI), there is at most one copy of a block within this structure, thereby utilizing the space more effectively (note that banking the L2 into multiple units has the same hit/miss behavior as a single monolithic non-banked structure). The downside of this approach is that upon an L1 miss, the requests need to go on the shared bus to get the data either from another L1, L2, or off-chip memory. In SI, the coherence protocol is performed between the L1 caches.

4.2 L2 Organizations and Proposed Architecture

4.2.1 Proposed Architecture

As observed, the advantages of private L2 are in reducing on-chip interconnect traffic (latency and contention). The downside is the possible increase in off-chip memory accesses because of reduced aggregate L2 capacity (due to duplication). The other problem may be due to the fact that the load induced on the L2 units may be different across the CPUs (i.e., one may have higher locality while another may exhibit poor locality). Consequently, we expect private L2 to perform better when (i) the sharing is not high across the CPUs (so that fewer duplicates reside across the L2 units) and (ii) the locality behavior is balanced across the CPUs.

The shared L2 organization described above does not have these two drawbacks. Despite the level of sharing across the CPUs, there is at most one copy of a data block that can reside in L2. Further, since all of L2 (and all its banks) are equally usable by any of the CPUs (the banking is done by addresses rather than by the CPUs using them), the imbalance of the locality behavior across the CPUs can be addressed by letting one CPU use up more space compared to another. The downside of the shared L2 organization is that (i) the load on the on-chip interconnect may become higher and (ii) there could be interference between the CPUs when utilizing L2 (one could evict the blocks used by another).

Having looked at the pros and cons of both these approaches, we next present our *Shared Processor-Based Split* L2 cache organization. The main goal here is to try to reduce off-chip accesses since we are at the last line of defense. It may be easier to provision additional hardware for on-chip components, e.g., wider buses, multiple buses, or fancier interconnects, and the more serious issue is to avoid the memory wall, i.e. *avoid going off-chip as much as possible*. This cost

is eventually going to be the limiting factor on performance (and perhaps even power) with the ever-growing application datasets, and the widening gap between processor and memory speeds.

Consequently, our solution strategy is to:

- Try to approach the behavior of private L2s when the sharing is not very significant and when the load is more evenly balanced. On the other hand, we would like to approach the behavior of shared L2 for higher sharing behavior and when the load is less balanced, while still insulating one CPU from another.
- Possibly pay a few extra cycles on-chip (perhaps not in the common case) if doing so will reduce the off-chip accesses.

Our shared processor-based split L2 uses the underlying organization of the Shared L2 described above, i.e., the L2 is on the other side of the bus, so that we can accommodate high data sharing whenever needed. However, in order to reduce the interference between the CPUs on this shared structure, our splits are based on the *CPU ids* rather than memory addresses, i.e., each CPUs is assigned one or more units/splits of the L2 structure. A request coming from an L1 miss goes across the shared bus, and checks the split(s) assigned to the corresponding CPU (note that the other L1s may snoop as usual). If the request can be satisfied by the split(s) being accessed — it is to be noted that as in the shared L2 case, there are no duplicates for a block in L2 — then the data is returned and the execution proceeds as usual. However, when the data does not reside in any of these split(s), instead of immediately going off-chip, the other (i.e., the ones that are not assigned to this CPU) splits are consulted to see if the requested block resides on any of those splits. If it does, then the data block is returned to the requesting L1 to continue

with the execution. Only when the block is not found in any of the splits is an off-chip memory access needed.

Note that there are three scenarios for an L2 lookup in our proposal:

- *Local Hit:* When the lookup finds the data block in the L2 split(s) assigned to the CPU issuing the request, all the splits assigned to that CPU are accessed in parallel and the performance cost is the cost of looking up any one split (taking say a cycles).

- *Remote Hit:* In this case, the lookup amongst its assigned split(s) fails, but the data block is found in another split that is not assigned to it. Consequently, the cost of servicing the request becomes $2a$ cycles.

- *L2 Miss:* In this case, the data block is not in any of the splits, and requires an off-chip access. The cost in this case will involve the cost of the off-chip access, in addition to the $2a$ cycles.

One may have opted to remove the *Remote Hit* case by looking up all the L2 splits (ignoring the CPU id) in parallel. Though performance efficient, the reason we refrain from doing this is due to the dynamic power issue, i.e., the dynamic power in this case is not going to be any lower than an unbanked monolithic L2 structure, which we discarded in the first place for this reason.

Having looked at the costs of lookup and servicing the requests, we next need to examine how the data blocks get placed in the different splits. Since we perform placement at the time of a miss, we need to consider the following cases:

- *Local Miss* + *Remote Miss*: Since the block is not in L2 (any of its splits), we assume that the CPU issuing the request is the one that will really need it. We randomly pick one of

the splits assigned to the requesting CPU, and then place the block within that split based on its address (as is usually the case).

- *Local Miss + Remote Hit*: In this case, we assume that the CPU issuing the latest request needs the block more than the CPU currently associated with the split where the block resides (i.e., we expect temporal locality). Consequently, we move the block over from the remote split to one of the splits (chosen randomly) for the requesting processor.

4.2.2 Hardware Support

We present a simple table-based mechanism for implementing the shared processor-based split L2 proposal. We propose to maintain a table in the L2 controller as shown in Figure 4.2(a) for an architecture with 4 processors and 8 L2 splits. The columns denote the splits of the L2 and the rows denote the CPUs, and an ”X” in the table indicates that the corresponding split is assigned to the specified CPU. When an L1 miss is incurred, the bus carries not just the address of the block being referenced, but also the id (which is usually the case) of the CPU initiating the request. The L2 controller (see Figure 4.2(b)) has already been given (downloaded to it in software) this table a priori, and it simply needs to lookup the table based on this CPU id, and can generate the chip selects (CS) for the corresponding L2 splits that need to be looked up. If these lookups fail (which will be sensed by the Hit Line in the figure), then the chip select is sent to the other splits (that were not looked up in the previous try). Only when this lookup fails as well do we need to go off-chip.

We would like to point out that this mechanism can easily fit into many asynchronous bus designs, since the interface to L2 is not very different from the external viewpoint (for the bus or

the CPUs). Further, the coherence mechanism that takes place between L1s is not affected with this enhancement (it is not any different from a monolithic or an address-banked L2).

We propose the following simple extensions to the operating system (OS) to work with this hardware: (i) a system call that specifies how the ”X”s for the CPUs should be allocated to the application, which can be invoked at any point during its execution, (ii) the operating system updating the table on the L2 controller appropriately during this system call, as well as updating its (shadow) software copy of this table that it maintains for each application, and (iii) the context switch mechanism at each CPU looking up the corresponding row of the OS shadow copy and updating the L2 hardware table accordingly. Note that protection is not really compromised, and if ever the OS wants to disallow a process from being allocated too many splits (or a specific split) the access control can be performed at the time of the system call. In addition, there could be OS issues in figuring out how to partition the L2 space between applications over the course of execution, which is beyond the scope of this chapter.

4.2.3 Exploiting the Proposed Mechanism

There are several benefits that we would like to point out with this implementation:

- There could be more than one ”X” in each column of the table, meaning that a split could be shared by different CPUs. Consequently, if we know that some CPUs have a high degree of data sharing, then we could assign them the same splits so that the local hit case is optimized.

- There could be more than one ”X” in each row, and we do not necessarily have to assign an equal number of ”X”s across the rows. This allows assigning more than one split

to a processor, based on its L2 space demand, and also optimizing for a non-uniform (heterogeneous) assignment (i.e., giving different number of splits to different processors) if the L2 locality behavior is different across the behaviors at any one time.

- We can disallow two CPUs interfering with each other, i.e., one evicting the blocks of another, by not giving them an ”X” on the same column.
- This table can be *dynamically loaded* to the L2 controller (possibly by memory mapping in some registers) using an operating system call, during the course of execution. This allows the ability to convey application-level information dynamically to the L2 hardware for the best performance-power trade-offs at any instant.

One can always opt for filling in the table to provision an equal allocation of splits to the CPUs — we refer to this as the *Shared Split Uniform (SSU)* case. This is the simplest of the options, where one may not want to analyze high-level workload behavior to further refine the table settings. However, our table-based implementation of the shared processor-based split L2 mechanism allows scope for non-uniform allocation of splits to the CPUs, i.e., different CPUs can get different number of L2 splits, and we refer to this as the *Shared Split Non-Uniform (SSN)* case in this chapter. In general, this software controlled table-based mechanism allows us to exploit the following workload characteristics:

- *Intra-Application Heterogeneity:* This is the L2 load imbalance within an application, and can be further classified as:
  - *Spatial Heterogeneity:* If the L2 localities of different CPUs are different within a single application, we can opt to allocate non-uniform splits to better meet those needs.

  - *Temporal Heterogeneity:* During the course of execution of an application, we can dynamically change the number of splits allocated to a CPU at different points of time, depending on how its dynamic locality behavior changes.

- *Inter-Application Heterogeneity:* Applications can have entirely different L2 behaviors. Whether they are running one after another, or even if they are running at the same time (on different CPUs or when they are time-sliced on the same CPUs), our mechanism provides a way of controlling the L2 split allocation according to their specific demands.

There are several techniques/algorithms that one can employ to fill the table values at possibly every instant of time. Such a detailed evaluation of all possible techniques is beyond the scope of this work. Rather, our objective here is to show the flexibility of our mechanisms, and we illustrate/use a simple *profile-driven approach* to set the table values to benefit from the uniformity/heterogeneity of the L2 locality/sharing behavior.

4.3 Experiments

4.3.1 Methodology and Experimental Setup

The design space of L2 configurations for comparison is rather extensive to do full justice in terms of evaluation within this chapter. Consequently, in our experiments where we compare the four approaches — Private (P), Shared Interleaved (SI), Shared Split Uniform (SSU) and Shared Split Non-uniform (SSN) — we set some of the parameters as follows. In the Private case, the number of L2 units has to obviously match the number of CPUs, say p. In the experiments for SI, we use p banks as well, with the address based interleaving of blocks. In the SSU case, we use p splits each with the same size as the private case. Note that in the case of SI

and SSU, we are not restricted by p, i.e. we could have more (of smaller size) or less (of larger size) banks/splits, while keeping the overall L2 capacity the same as the private case. In our experiments, we simply set the number of splits/banks to p, since it is closest to the private case (in terms of access times). In the SSN case, the splits are themselves all of the same size. The way that we provide different cache capacities for different CPUs is by giving different number of splits to the different CPUs. In order to examine the benefits of such heterogeneous allocations, our experiments use more than p splits. More specifically, our default experiments use a 8 processor configuration with 2 MB overall L2 capacity as in shown in Table 4.2, and we use 16 splits of 128K each in the SSN case.

SSN Configuration	**Allocation Chunks**
SSN-152	1*512K, 5*256K, 2*128K
SSN-224	2*512K, 2*256K, 4*128K
SSN-304	3*512K, 4*128K

Table 4.1. SSN configurations studied for the 2MB L2. Note that the splits are themselves of 128K each, an integral number of such splits — called a chunk — are allocated to a CPU.

There are several techniques/algorithms that one can employ to fill the table values at possibly every instant of time to accommodate the different kinds of application/spatial/temporal heterogeneity explained in the previous section for SSN. Such a detailed evaluation of all possible techniques is beyond the scope of this work. Rather, our objective here is to show the flexibility of our mechanisms, and we use a rather simple scheme. Specifically, we categorize CPUs into three groups based on the load that they impose on L2 — high, medium, and low —

and accordingly give them 512K, 256K, and 128K of L2 cache space, respectively. For instance, if a CPU is categorized at some instant as high L2 load, then it would be allocated 512K, i.e., 4 splits of 128K each. The SSN schemes that we consider (for the 2MB L2) are given in Table 4.1. For example, SSN-152 denotes, that the L2 space is divided into one 512K (i.e., 4 splits of 128K) chunk, five 256K chunks (i.e. each of 2 splits of 128K), and two 128K chunks (i.e., one split each). Note that the total size of the chunks matches the total L2 capacity, which is 2 MB in this case. A CPU, over the course of the execution, can move from one chunk to another. However, we do not vary the number of chunks or the size of the chunks over the course of execution (though our table-based mechanism allows that as well).

The allocation of chunks to CPUs is performed using a profile-based approach in this study. Specifically, we divide the execution into a certain number of epochs (256 in this case), and for each epoch we sort the CPUs in decreasing order of L2 miss rates, and then allocate the largest chunk to the CPU with the highest miss rate, the second largest chunk to the CPU with the second highest miss rate, and so on. This is a rather simple scheme that can use profile information to make reasonable allocations during the run. It is conceivable that future research can develop much more sophisticated (and dynamic) split allocation algorithms, and our table-based mechanism allows that. To conduct our experiments, we modified Simics [43] (using Solaris 9 as the operating system) and implemented the different L2 organizations. The default configuration parameters are given in Table 4.2, and these are the values that are used unless explicitly stated/varied in the sensitivity experiments.

In this study, we evaluated nine applications (eight from SPEComp [2] and Specjbb. The important characteristics of these applications are given in Table 4.3. The second column in this table gives the number of L1 misses (averaged over all processors) and the third column

Parameter	Default Value
Number of Processors	8
L1 Size	8KB
L1 Line Size	32 bytes
L1 Associativity	4-way
L1 Latency	1 cycle
L2 Size (Total Capacity)	2MB
L2 Associativity	4-way
L2 Line Size	64 bytes
L2 Latency	10 cycles
No. of L2 Splits in SI, SSU	8
No. of L2 Splits in SSN	16
Memory Access Latency	120 cycles
Bus Arbitration Delay	5 cycles
Replacement Policy	Strict LRU

Table 4.2. Base simulation parameters used in our experiments.

the L1 miss rate (averaged over all processors) when the P version is used. The corresponding L2 statistics are given in columns four and five. The last column shows the total number of instructions simulated. All benchmarks have been simulated for 4 billion cycles and use all 8 processors in our base configuration, unless stated otherwise.

While many of the experiments use one application running on Simics where spatial and temporal heterogeneity during its execution may arise, we also conduct experiments with running two applications concurrently to stress application heterogeneity. We use four processors for running each application, i.e., we use space sharing rather than time slicing for multiprogramming.

4.3.2 Base Results

We first summarize the overall results to illustrate the benefits of shared processor-based split L2, by comparing its IPC results with those for the two other organizations (P and SI)

Benchmark	L1		L2		Number of
	Number of Misses	Miss Rate	Number of Misses	Miss Rate	Instructions (in millions)
ammp	53176353	0.007	2015486	0.062	25,528
art_m	66061168	0.009	25712966	0.507	22,967
galgel	111400050	0.014	10683462	0.127	24,051
swim	261622475	0.111	95881598	0.296	7,761
apsi	378868309	0.117	27170810	0.083	15,713
fma3d	18853410	0.002	6199405	0.239	26,189
mgrid	333243418	0.153	68292105	0.185	10,294
applu	111232574	0.009	26477050	0.168	21,519
specjbb	828522034	0.353	22689664	0.083	9,413

Table 4.3. The benchmark codes used in this study and their important characteristics for the private L2 case.

Workload	P	SI	SSU	SSN-152	SSN-224	SSN-304	Best IPC	Best Version
ammp	5.967	6.026	6.049	6.020	6.010	6.006	6.049 (1.4%)	SSU
art_m	5.368	5.470	5.529	5.261	5.459	5.555	5.555 (3.5%)	SSN-304
galgel	5.622	5.583	5.618	5.724	5.731	5.727	5.731 (1.9%)	SSN-224
swim	1.821	2.462	2.399	2.488	2.543	2.596	2.596 (42.5%)	SSN-304
apsi	3.687	2.680	3.812	3.703	3.587	3.507	3.812 (3.4%)	SSU
fma3d	6.122	6.131	6.135	6.181	6.171	6.159	6.181 (1.0%)	SSU-152
mgrid	2.406	3.050	3.022	3.086	3.101	3.150	3.150 (30.9%)	SSN-304
applu	5.030	4.894	4.937	4.947	4.949	4.944	4.949 (-1.6%)	P
specjbb	2.200	2.547	2.890	2.606	2.709	2.810	2.890 (31.3%)	SSU
ammp+apsi	4.169	4.104	4.292	5.436	4.803	4.401	5.436 (30.4%)	SSN-152
ammp+fma3d	5.897	5.949	5.977	5.936	5.900	5.895	5.977 (1.4%)	SSN-152
swim+apsi	2.509	1.776	2.018	2.050	2.007	2.006	2.050 (-18.3%)	P
swim+mgrid	2.513	2.851	3.067	2.698	2.470	2.401	3.067 (22.1%)	SSU
Average:							(11.52%)	

Table 4.4. The IPC results for different worloads and different L2 management strategies.

which are the current state-of-the-art (see Table 4.4). The last two columns in Table 4.4 give the best IPC value for a given workload across all the L2 configurations considered (and —within the parentheses— the percentage improvement it brings over the private L2 case), as well as the version that gives this best IPC, respectively. These results are given for each application running individually, as well as some multiprogrammed workloads with two applications A and B running concurrently on four different processors each (specified as A+B).

We can make several observations from these results. First, we find that except in two workloads (applu and swim+apsi), the shared L2 organization based on processor-based split (our SSU or SSN proposal) does the best. In workloads such as swim, mgrid and specjbb, with higher L1 miss rates (which in turn exerts a higher pressure on L2), we find dramatic improvements in IPC ranging from 30.9% to 42.5% in some configurations (SSU or SSN). Even if we consider the average across these fourteen workloads, we get an average 11.52% IPC improvement over the private case. Let us examine these results in detail in the following discussion.

We first attempt to show why private L2 does better in the two workloads, and not as well in the rest. There are primarily two factors affecting L2 performance: (i) the degree of sharing which can result in duplication between the private L2 units or cause repeated non-local split references in SSU or SSN, and (ii) the imbalance of the load that is imposed on the L2 units by the different processors. In our experiments, we tracked the number of blocks that were residing in multiple private L2 units at any instant, which is an indication of the level of data sharing. In the case of applu, only around 12% of the blocks are shared at any time and even these are mainly shared between 2 CPUs. Consequently, there is not much duplication for this workload (applu) to affect the private L2 case. In addition, applu does not exhibit much spatial or temporal heterogeneity as will be shown later, making this workload more suitable for private L2. In the

case of swim+apsi, there is no sharing across the CPUs running these different applications, and they also have similar L1 miss behavior (shown in Table 4.3), which is an indication of the load on L2.

On the other hand, in most of the other applications, the data sharing and/or the load imbalance issues make the shared L2 perform better than the private L2 case. For instance, when we consider specjbb, SSU is over 31% better than the private L2, due to both sharing effects (we find around 35% of the blocks are being shared on the average when considering entire execution), as well as the imbalance caused by spatial and temporal heterogeneity (to be shown shortly). While Table 4.4 summarizes the overall results, we also collected detailed statistics over the course of execution, and present some of them (the IPC and L2 misses) for each epoch in the execution for both the P and SSU cases in Figure 4.3 for specjbb. One can see that there is a direct correlation between the L2 misses and the corresponding IPC values, and we find that SSU is able to lower the L2 misses, thereby improving the overall IPC. Specifically, while the IPC in the P case never exceeds 2.5, we find IPCs over 3.0 during several epochs in the execution for the SSU case. The less frequent accesses to the off-chip memory with SSU provide this IPC improvement. Although not explicitly quantified in this study, this can also reduce the main memory energy consumption. We will get further into the correlations between application characteristics and L2 behavior later in this section.

Of the two problems — duplication due to sharing and load imbalance across the CPUs — SI mainly addresses the sharing issue, i.e., only one copy of a block resides in all of L2 regardless of how many CPUs access it. While it can accommodate some amount of imbalance/heterogeneity by letting one or more CPUs occupy more L2 space than others, this can also lead to interference between the CPUs (eviction of one by another). We find that SI brings

some improvement over the private case in some of the workloads (e.g., art_m, swim, mgrid, swim+mgrid). However, providing a shared L2 alone is not sufficient in many others, or is not necessarily the best option even in these cases. Note that SI represents the current state-of-the-art for the shared L2 organization. On the other hand, when we go to SSU and SSN, in addition to data sharing, we also find the insulation of L2 space of one CPU from another, and the possible spatio-temporal heterogeneity in the workloads can help these schemes over and beyond what SI can provide. In nearly all the cases where SI does better than the private case, SSU or SSN performs even better, and our two schemes do better than the private case even when SI is not a good option.

Our shared processor-based split L2 organizations (both SSU and SSN) are not only able to address the sharing issue (perhaps not as effectively as SI since there may be an additional cost — though not as expensive as going off-chip — to go to a split not allocated to that processor), but are able to reduce the interference between the CPUs, and possibly allocate more/less space to a CPU as needed. This helps reduce L2 misses and the associated off-chip memory access costs. These advantages make these schemes provide much better IPC characteristics compared to either the P or the SI cases as can be observed in Table 4.4.

The previous results largely depend on application characteristics, both in terms of the degree of sharing and in the heterogeneity of the load that the CPUs exercise on the L2 cache. We found that the latter effect seems to have more consequence on the results presented above. For instance, in mgrid, on the average at any instant, less than 3% of the blocks were shared across the CPUs (i.e., they were present in more than one L2 at any time in the private case). On the other hand, when we move to the shared L2 configurations for this application, we get more than 25% savings compared to the private case. Consequently, in the rest of this discussion, we

examine the issue of heterogeneity of L2 load in greater depth, and look at this heterogeneity at an intra-application (finer) and an inter-application (coarser) granularity.

4.3.2.1 Intra-Application Heterogeneity

We plot SHF_{epoch} and THF_{cpu} for each epoch and each processor, respectively, in Figures 4.4 and 4.5 for the nine individual application executions. We find a direct correlation between the cases where our split shared L2 organizations does better (in Table 4.3) and the cases where the heterogeneity factors are high in these graphs. Specifically, we find the heterogeneity — both spatial and temporal — is much higher in specjbb, mgrid, swim and to some extent in apsi, compared to the other five. Note that these are also the applications where we find significant improvements for SSU or SSN compared to the P or SI cases.

When we compare SSU and SSN, we find that though the latter gives slightly better results (in five of our nine individual application executions), the difference between the two schemes is not very significant. There are several reasons for this behavior. First, the allocation of L2 units to the CPUs in SSN is not necessarily the most efficient. For instance, if we consider SSN-152, only when there is one CPU that dominates on the L2 load, with five others in-between the extremes right through the execution, would this be an ideal choice. If the application characteristic does not match this static choice of different chunk sizes, then the performance may not be very good. It should be emphasized that this is a problem of the specific implementation evaluated here, rather than a problem of our table-based mechanism, and future work — using our table-based mechanism — can possibly develop fine-grain dynamic split allocation strategies based on L2 behavior. Second, in addition to misses, the other performance advantage that can come for SSN compared to SSU is in moving more of the remote hits (of SSU) to the local

hit side. Because of non-uniform allocations, it is possible that a CPU with a higher L2 load may find more blocks within its allocation rather than in someone else's allocation (which is still a hit but incurs a higher access latency). However, if we look at Table 4.5 which shows the local and remote hit (and L2 misses) fractions for SSU, we see that the contribution of remote hits is not very high, and it is the effect of the misses that is more important.

Workload	Local Hit	Remote Hit	Miss
ammp	94.2%	2.9%	2.9%
art_m	47.3%	11.1%	41.6%
galgel	80.4%	10.4%	9.2%
swim	69.5%	0.7%	29.8%
apsi	89.7%	4.2%	6.2%
fma3d	76.1%	1.7%	22.2%
mgrid	81.6%	0.3%	18.1%
applu	76.6%	10.0%	13.4%
specjbb	75.1%	17.1%	7.8%
ammp+apsi	91.8%	2.2%	6.0%
ammp+fma3d	90.9%	2.2%	6.9%
swim+apsi	76.8%	8.5%	14.6%
swim+mgrid	79.0%	3.8%	17.2%

Table 4.5. The breakdown of L2 accesses for SSU.

4.3.2.2 Inter-Application Heterogeneity

In terms of inter-application heterogeneity, our four workloads in Table 4.4 — ammp+apsi, ammp+fma3d, swim+apsi, swim+mgrid — capture different scenarios of L2 load. In ammp+fma3d, the load introduced on L2 (see the miss rates of these two applications in Table 4.3) by both applications is rather low, thus not showing significant difference across the schemes. In swim+apsi

and swim+mgrid, the L2 load by both applications is rather high (and balanced), with the balance being higher in the former (see the heterogeneity graphs for apsi and mgrid, where the latter shows higher heterogeneity) making the private case a fairly good choice. Still, the load across applications is more or less balanced, thus making the schemes again comparable. On the other hand, when we consider ammp+apsi, we have the first with a rather low load, and the second with a rather high load. With an unequal allocation to this space-shared multiprogrammed workload, we can give different amounts of cache space to these individual applications so that we can get the best overall IPC. For instance, with SSN-152, we can give 1.25MB (of the total 2MB L2) to apsi and the other 0.75MB to ammp, though this partitioning can change from epoch to epoch based on the dynamics of the execution (see Figure 4.6 to see how L2 space is allocated to the two applications for the duration of execution which more of less tracks this 5:3 proportion). Consequently, this can provide a lower miss rate for apsi without really affecting the miss rate of ammp, to provide a better overall IPC value.

4.3.3 Sensitivity Analysis

Having considered one set of hardware configurations, we have also studied the L2 organization issues with other important parameters and the results from some of those studies are given below.

4.3.3.1 Impact of Larger L2

The experiments until now used a total L2 size of 2 MB (with the number of splits being equal to the number of processors for SI and SSU, and being 16 for SSN). We now present results with a 4 MB L2 cache space, in which we again have 8 (number of CPUs) splits for SI

and SSU, and have 32 splits for SSN. We conduct experiments with four different chunk sizes of these splits for SSN, namely 1 MB, 512K, 256K, and 128K. As with the earlier nomenclature, we consider the following SSN configurations with these chunks: SSN-1520, SSN-2240, SSN-2312, SSN-3032, and SSN-3104. The corresponding IPC results are given in Table 4.6 for four different workloads.

Scenario	**P**	**SI**	**SSU**	**SSN-1520**	**SSN-2240**	**SSN-2312**	**SSN-3032**	**SSN-3104**	**Best IPC**	**Version**
swim	1.838	2.367	2.493	2.488	2.543	2.550	2.533	2.541	2.550 (38.77%)	SSN-2312
applu	5.044	4.926	4.938	4.947	4.949	4.940	4.944	4.948	4.949 (-1.89%)	P
specjbb	2.518	2.762	2.834	2.606	2.709	2.846	2.868	2.817	2.868 (13.92%)	SSN-3032
ammp+fma3d	5.910	5.942	5.946	5.936	5.900	5.884	5.872	5.882	5.946 (0.60%)	SSU

Table 4.6. The IPC results for different workloads and different L2 management strategies for a 4MB L2.

As we provide more L2 capacity, we find two factors emerging. First, the miss rates go down overall, and the differences arising from miss rates diminish. However, we find that the shared processor based split schemes still provide considerable savings (e.g. swim). It is to be noted that along with L2 capacities, application datasets are also expected to get larger in the future, thus emphasizing the importance of this novel organization. The second observation that we note is that a larger capacity, also allows more splits for SSN, thus giving a finer granularity of allocation across the CPUs. We find this effect helping SSN a little (e.g. note that in this case, SSN gives the best performance for specjbb compared to SSU in the earlier results).

4.3.3.2 Impact of Memory Access Latency

As we move into the future, the gap between off-chip memory accesses and processing speeds is likely to grow. Consequently, it becomes even more critical to reduce off-chip accesses. Having considered results for a memory access latency of 120 cycles until now, we next present the results for a 240 cycle latency experiment.

Figure 4.7 shows the IPC improvement over the private case for SI and SSU with both 120 cycles and 240 cycles. We find that the benefits of the shared processor-based split caches are amplified not only over the private case (in fact, in applu SSU starts doing better than P which was not the case earlier), but also over the SI organization.

4.3.3.3 Impact of Other Parameters

We have also studied the impact of other parameters - the number of CPUs, and L1 sizes - on the results, and found that while the absolute IPCs/savings vary the overall trends mentioned above hold.

4.4 Related Work

There has been a considerable amount of previous research in designing memory hierarchies and cache coherence protocols for SMP (multi-chip multiprocessors) systems. It is well beyond the scope of this chapter to cover all such related work, and the reader is referred to [20] for an in-depth treatment of this area. At the same time, prior studies have also looked at the capacity and organization issues for L3 shared caches [24] and the characteristics of the workloads affecting memory hierarchy design [59, 7], again in the context of SMP systems. Our work targets single chip multiprocessors, where an off-chip access to the memory hierarchy can incur

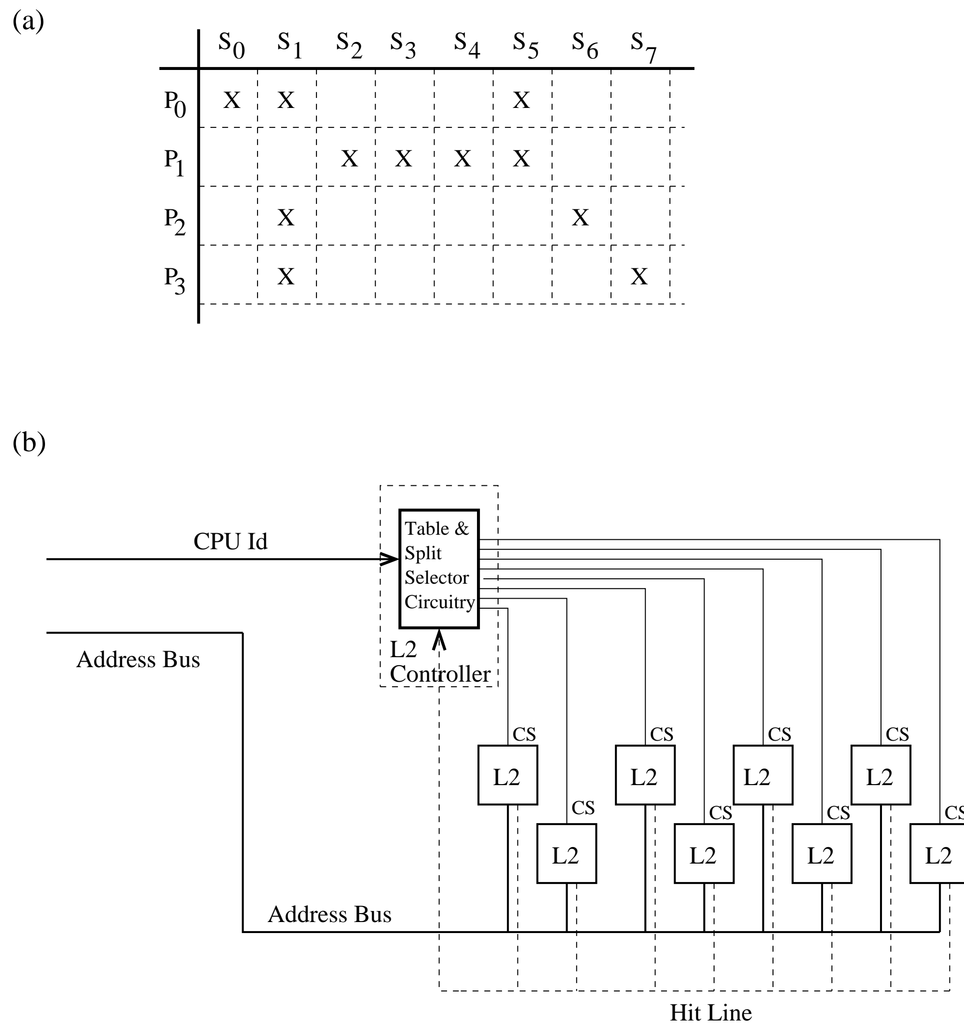
a much higher cost than exchanging information on the shared interconnect between the cores on a CMP. Consequently, it becomes more important to reduce off-chip accesses than to save a few cycles within the chip. There has been no prior in-depth comparison of the pros and cons of private vs. shared organizations for the on-chip last line of defense to the memory wall for CMPs.

One advantage of our implementation of the Shared Processor-based Split Cache design is the adaptability and morphability to application characteristics. It becomes possible to give the right core, the right L2 space at the right time and to change the allocation whenever needed. Prior work has recognized the importance of adaptive caches for different purposes. Two works in particular [72, 1] look at adjusting cache sizes and/or other parameters, primarily in the context of uniprocessors, in order to save dynamic/leakage power. Other work [57] on this topic dynamically adjusts the cache for performance benefits. In the design automation area, there have been efforts to design application-specific memory hierarchies based on software-managed components [13]. Nevertheless, none of these prior studies have analyzed the benefits of the malleability of L2 organization for CMPs.

Other related work on the topic of adjusting cache space to application characteristics has been completed by [65, 64]. In this study, the authors partition the cache space between multiple processes and activities, executing on one processor in order to reduce interference between time slices. Cache space usage across multiple threads has also been studied in the context of SMT processors [67].

The popularity of CMPs is clearly evident from the different commercial developments and research projects [21, 53, 39, 35, 44, 50, 52, 8] on this topic. Within these studies, primary issues concern the design of the interconnect, and the design of the data-path for effective on-chip

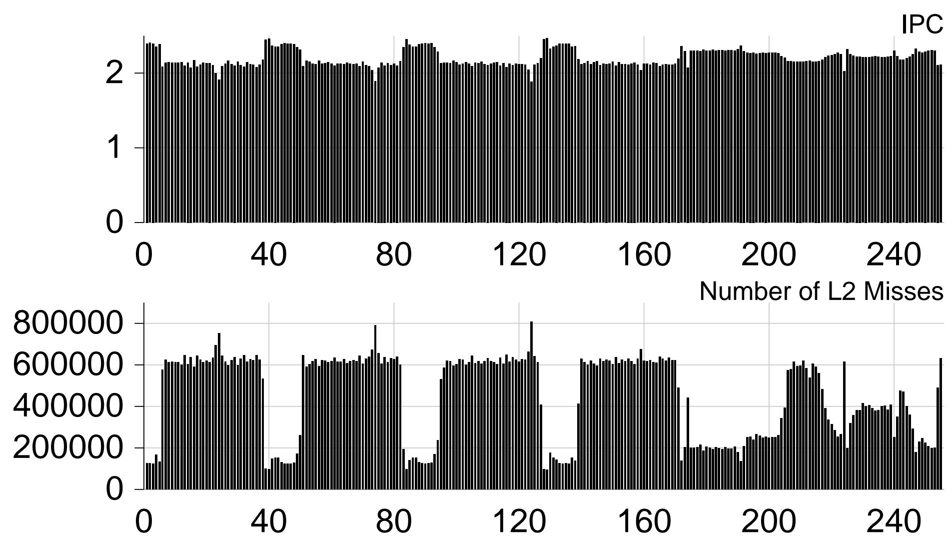
parallelism. To our knowledge, this is the first work that has examined different L2 organizations and proposed a new one in order to reduce off-chip memory accesses.



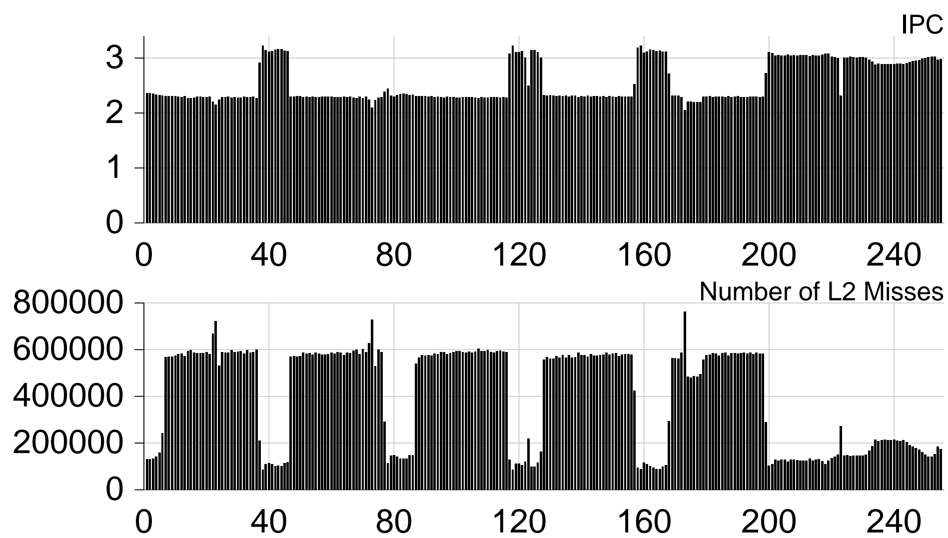
(a)

	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
P_0	X	X				X		
P_1			X	X	X	X		
P_2		X					X	
P_3		X						X

Fig. 4.2. (a) The table structure that shows the processor-L2 split associations. (b) Addressing L2 splits in our proposal.



(a)SpecJBB (the P case)



(b)SpecJBB (the SSU case)

Fig. 4.3. IPC and the number of L2 misses for specjbb.

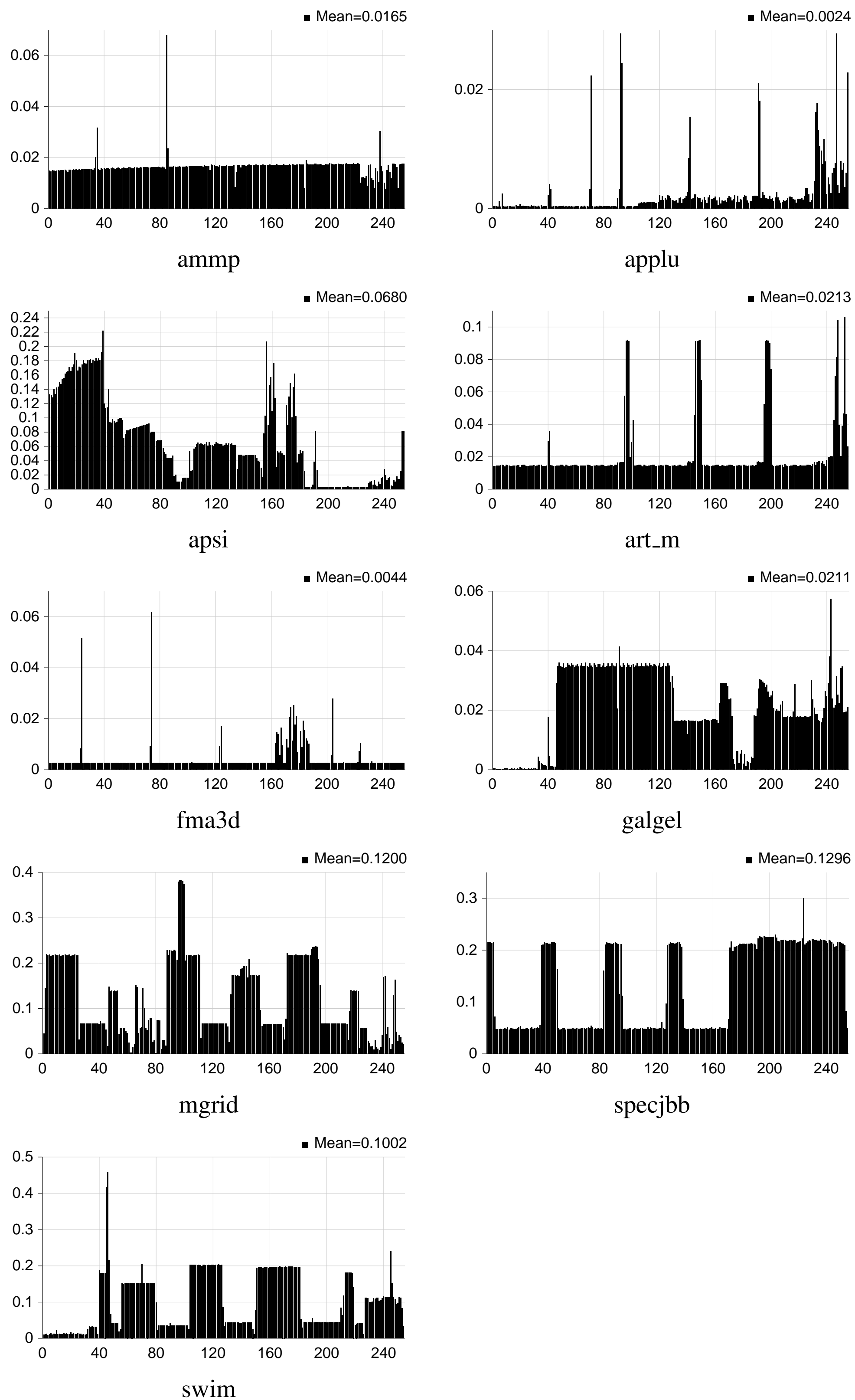


Fig. 4.4. Spatial Heterogeneity Factor (SHF) for each epoch. Note that the y-axes are on different scales.

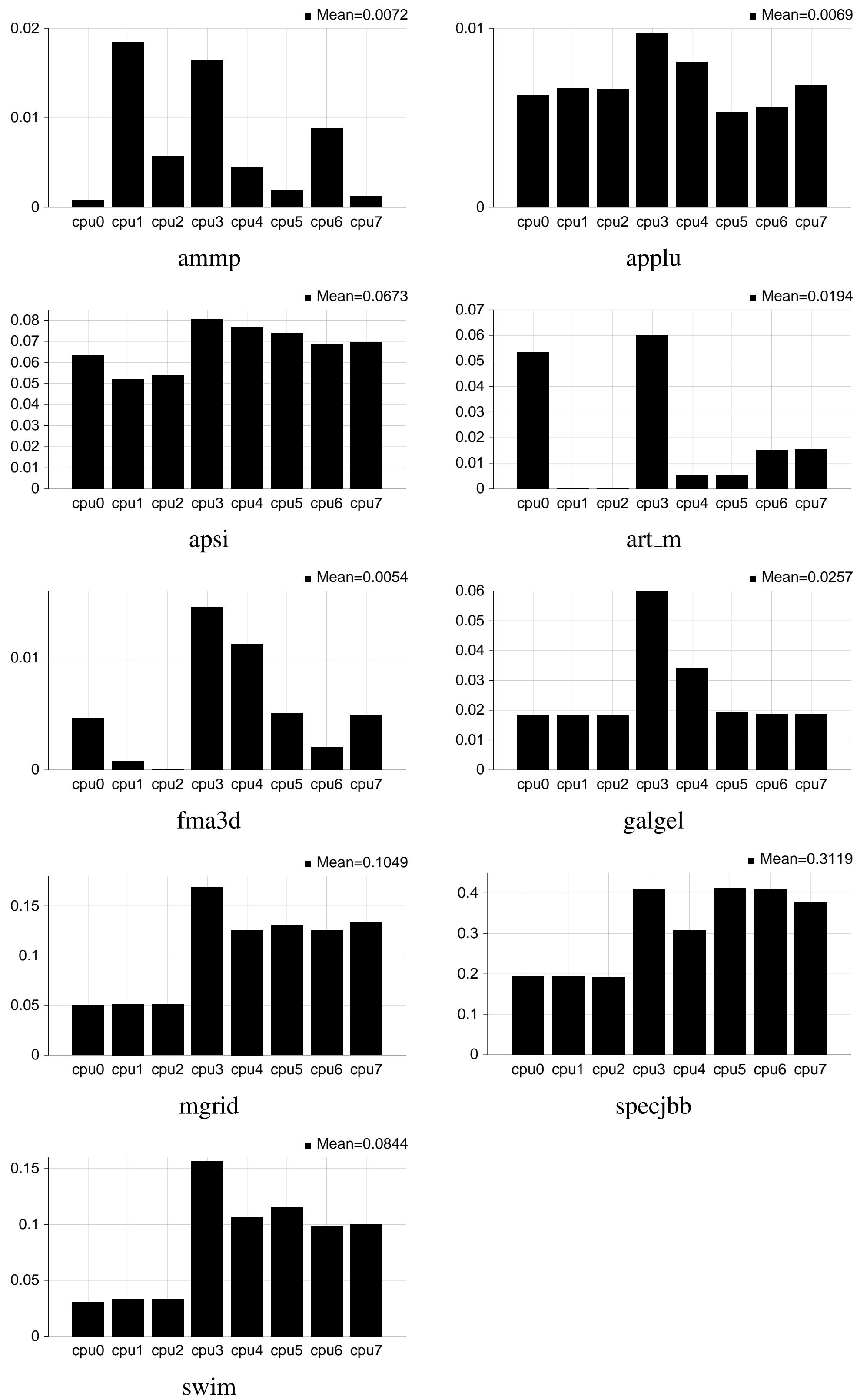


Fig. 4.5. Temporal Heterogeneity Factor (THF) for each CPU. Note that the y-axes are on different scales.

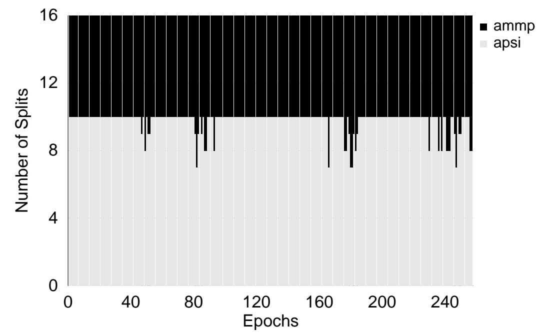


Fig. 4.6. The L2 space allocation for ammp+apsi under SSN-152.

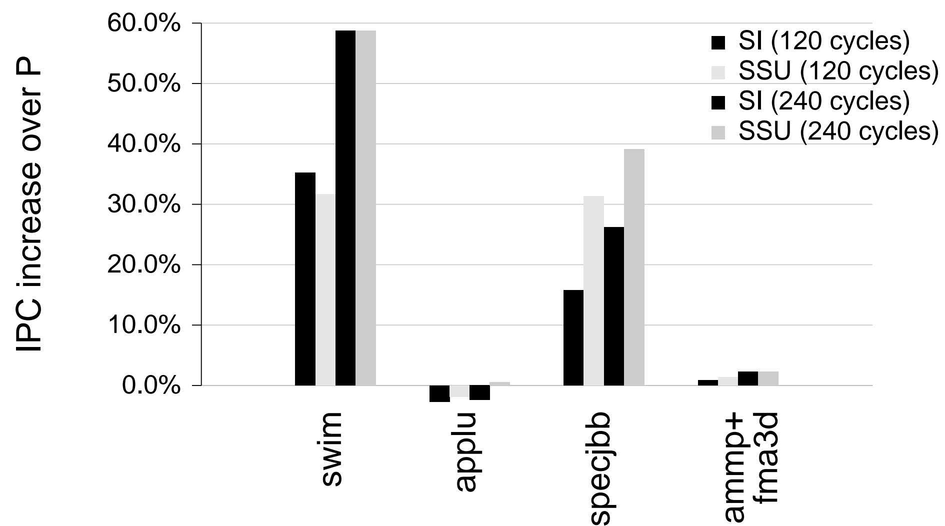


Fig. 4.7. IPC increase over the P case.

Chapter 5

Access Characterization and Non-Uniform Cache Optimization

5.1 Introduction

Chapter 4 presents an original way to partition the shared L2 cache according to the load demands from different cores. In that chapter, we assume the accesses to different blocks from different cores have the same latency, i.e., the shared on-chip cache is uniformly accessible to all the cores. With deeper levels of integration, the relatively high cost of going to off-chip memory is forcing chip designers to provision large on-chip cache structures. For instance, the IBM Power5 [18] has 1.9 MB L2 cache, and Intel's Itanium2 [48] is projected to have 24MB of on-chip L3 cache. With such large caches, the uniformity of cache access latency, which we assumed in Chapter 4, breaks down. The signals have to traverse longer wires [47], making some parts of the cache more expensive to access than others. Exposing this non-uniformity of cache accesses to the micro-architecture is becoming increasingly important, so that one can spatially place the data items into these structures based on application access patterns. This has led to recent investigations into optimizing data placement for large Non Uniform Cache Architectures (NUCA) [60].

Chip-Multiprocessors, which provide an efficient way to exploit the growing transistors on a single die, demand even more on-chip cache real estate to fuel their processing needs. However, non-uniformity in this paradigm becomes inescapable–not only is accessing different portions of the cache from a single core non-uniform, but so also is the access time to a single

location from different cores. This makes the design of the L2 cache (where we allocate the cache space on chip and how the cores and cache space should be relatively located), and its usage (where a data item should be placed and how we move data items across this space spatially and temporally) extremely critical in CMPs.

The issue of data placement to locate frequently accessed items in the cache cells with lower access times has been explored in [60]. More recently, there have been proposals to exploit NUCA for CMPs. Chishti et al. [16] extend their earlier idea [15] by replicating tags to facilitate look up. However, a protocol is required to maintain coherence between these replicas, and the global wires running these protocols will be long. Beckmann and Wood [10] propose a layout for cache space organization where different portions are closer to different cores, and suggest schemes for migrating the blocks between these cores. While this scheme does not replicate the blocks (and does not require coherence maintenance at the L2 level), the center portion, which is expected to eventually contain the shared blocks (across the cores), is equally far from all the cores. As our results will show, shared data accesses constitute a substantial number of L1 misses, suggesting we might want to keep the shared portion equally close to all the cores (rather than equally far from all the cores).

Our approach on the other hand comes from studying the characteristics of twenty two multi-threaded applications (including NAS parallel benchmarks [4], SpecOMP [2] and commercial workloads such as Apache [26] and SPECJbb [63]). A detailed characterization of the L2 access patterns of these applications reveals that a substantial number of L2 accesses are to shared blocks, making it important to place them equally close to all the cores. Furthermore, while the number of accesses is substantial, the number of blocks that are widely shared is rather low. These results suggest that we can provide a relatively small L2 space (which we refer to as

the *Center Cell*) – 64KB – at the center, with processing cores around this region. The rest of the L2 space, though shared, is partitioned among the cores and is intended to mainly hold privately-accessed data items. Later in this chapter, we present details on this shared L2 organization for a four-core CMP, together with statistics on the access times to different portions of the L2 from each core. There is a wide design space for exploitation of this organization based on where to place data items, how to search for their presence, how to migrate them based on access patterns (to exploit non-uniformity in access latencies), and what to replace. In this chapter, after pointing out the design choices, we then conduct a detailed evaluation of the proposed architecture using the twenty two applications to show the performance and power benefits of our approach.

The rest of this chapter is organized as follows. Section 5.2 discusses the related work on non-uniform cache architectures. Section 5.3 studies the applications and characterizes their L2 access behavior. In Section 5.4, we present details of our Center-Cell based shared L2 organization, together with the design choices for exploiting this architecture. An evaluation of this architecture is conducted in Section 5.5.

5.2 Related Work

In this section, we discuss the prior work on L2 cache organizations with non-uniform access latency. [9] proposed using a thicker metal for mitigating wire delays, which is a growing problem in circuit design. Kim et al. [36] proposed a non-uniform cache architecture. Their proposal was based on the observation that growing wire delays will force significant changes in the physical layout of large caches. They also evaluated two techniques, called the aggressive broadcast and the energy-friendly multicast search, for locating a block in their architecture. Chishti et al. [15] proposed NuRapid, a non-uniform latency cache organization, which decouples tag placement from data placement. It clusters tags and puts them into places close to the core, and this helps improve both performance and power consumption. Our work is different from these studies since we target a CMP, whereas [36] and [15] focused on single-processor-based architectures.

Several recent papers proposed schemes that extend the non-uniform latency cache idea to CMP environments. For example, Chishti et al. [16] extended NuRAPID [15] to the multi-core processor paradigm. Their approach was based on replication to keep the frequently accessed shared blocks close to processor cores that use them. Zhang and Asanovic [73] discuss an approach called victim replication. This approach kept copies of local primary cache victims within the local L2 cache slices. While both these schemes ([16] and [73]) are effective in reducing access latency to shared data, they require coherence maintenance. Specifically, replicating data within L2 requires a complex coherency control protocol which itself can be costly from both performance and power perspectives. In comparison, the approach proposed in this chapter does not replicate data blocks; instead, we reduce latency and power consumption in accessing

shared blocks by placing them into a small center cell, which is very close to all the processors. Therefore, from an implementation point-of-view, our approach is simpler.

Beckmann and Wood [10] proposed a block-migration scheme to extend the NUCA for the CMPs, but their protocol is complex and their results rely on an oracle-based search. Practical implementation of this search is not elaborated on in the paper. One of the problems associated with this approach is that it puts the shared data into a place within L2, which is equally far from the processors that share it. Each access to such a block tries to bring the block closer to the processor that requests it. However, as a result, heavily shared blocks tend to cluster in the middle regions which are far from the processors. This in turn increases the number of migrations and causes extra latency and power consumption. In fact, one can view this scheme as a variant of ours with a very large center cell *CC.Large*. In contrast, we keep the center cell very small and close to all the processors, and this cuts the number of block migrations significantly, as demonstrated by our experimental results. The reduction in block migrations also helps reduce power consumption.

Huh et al. [31] studied how to partition the NUCA L2 to reduce the interconnect traffic, and, thus, improve the performance and power consumption. Similar to [10], their approach kept the shared data far from the requesting cores, if both cores were on the opposite sides of the die. Iyer [33], Kim et al. [37] and Suh et al. [65] studied the sharing fairness of L2 cache for CMPs to prevent one thread from polluting the cache so that the overall throughput can be improved.

5.3 Motivation

The first consideration in our search for a suitable L2 organization is finding a layout, based on application characteristics. Specifically, we would like to understand the sharing behavior of multithreaded applications at the L2 level, to figure out how much data is really (and actively) shared.

5.3.1 Workloads

We use a large number of diverse multithreaded benchmarks for this study:

- *Scientific Applications*: We use both NAS Parallel Benchmarks (NPB 3.2) [4] and SpecOMP [22]. The NAS Parallel Benchmarks, derived from computational fluid dynamics (CFD) applications, are designed to evaluate the performance of large parallel computers. The benchmarks include five kernels and three pseudo-applications, and we use the Class A benchmarks in our study. SpecOMP has been designed by SPEC to evaluate shared memory multiprocessor (SMP) system performance for OpenMP, covering eleven applications, and we use the Class M benchmarks for our experiments.

- *Commercial Applications*: We use two server workloads including SPECJbb [63] and Apache [26]. SPECJbb evaluates the performance of server side Java by emulating a three-tier client/server system (with an emphasis on the middle tier). It measures the performance of the CPU, memory system, and the system scalability by exercising the Java Virtual Machine (JVM), Just-In-Time (JIT) compiler and some aspects of the operating system. Apache is a widely used open-source multithreaded HTTP web server. In this

work, we use Apache 4.0 for Sparcs and the SURGE web server benchmark [6] as the client to initiate the requests.

All these benchmarks were run on the Simics [43] complete system simulator with the SPARC target. Since our primary interest is in the L2 accesses, we collected the L1 miss traces and used them in all our experiments in the interest of simulation time. We collected the L2 cache access traces for all the benchmarks using a simulated system with 4 cores, each with a private 16KB L1 cache. The L1 caches are kept coherent using the MESI protocol.

For all the NAS Parallel benchmarks and SpecOMP benchmarks, we marked the initialization phase of the benchmarks at the source code level. Trace collection was started 100 million instructions after the end of the initialization phase to warm up the caches. We collected 400 million subsequent L1 misses for our studies, which was roughly 2.2 GB in the compressed form for each application. For the SPECjbb benchmark, since we have no precise control to demarcate the initialization phase, we warmed up the cache for the first 15 billion instructions, which covers the test, initialization, and terminal ramp-up phases. We then collected 400 million L1 misses (during timing measurement) for our experiments. In the case of Apache, we started the trace collection after the SURGE [6] client has successfully retrieved 600 web pages, beyond which point the utilization of our CPUs reached almost 100 percent.

The collected trace for each application contains the initiator of the memory request, the physical and virtual addresses, the time elapsed (in cycles) since the last access from that core, the size of the memory request, whether it is a kernel or user access, instruction or data, read or write, etc. The number of L2 accesses per thousand instructions for each benchmark along with the number of simulated cycles are given in Table 5.1.

Application	L2 accesses per thousand instructions	Simulated cycles per core with perfect L2 (in millions)
bt.a	56.22	4,806
cg.a	127.21	2,167
ep.a	13.30	17,300
ft.a	134.88	1,865
is.a	177.45	1,168
lu.a	54.08	4,939
luhp.a	58.98	4,600
mg.a	89.44	2,976
sp.A	51.19	5,060
ua.a	27.25	9,303
310.wupwise	8.85	20,030
312.swim	103.61	2,084
314.mgrid	64.18	3,617
316.applu	15.33	28,075
318.galgel	11.87	25,335
320.equake	5.41	65,537
324.apsi	25.10	4,672
328.fma3d	63.08	4,205
330.art	4.15	6,322
332.ammp	3.23	34,515
SPECJbb	45.23	5,241
Apache	41.3	1,296
Average:	54.73	11,596

Table 5.1. Our applications and important statistics.

5.3.2 Characterization

We examine the spatial and temporal L2 sharing behavior of these applications using the above traces. In the following results, we assume the L2 cache has infinite capacity since we are interested in the application characteristics.

Figure 5.1 (a) illustrates the spatial sharing characteristics of these benchmarks. The Y-axis plots the cumulative fraction of L2 blocks, and the points ”s2”, ”s3”, and ”s4” on the X-axis correspond to the fraction of total blocks in L2 that are shared by 2, 3, and 4 cores respectively. Note that since L2 is of infinite capacity, this sharing is across the entire run of the experiment. The point marked as ”private” in Figure 5.1 (a) corresponds to the fraction of L2 blocks that are ever accessed by only one core.

Although there are some individual differences across these benchmarks, the general trend (the exceptions are 314.mgrid, lu.a, and luhp.a) we observe is that the percentage of blocks that are shared is a much smaller fraction of the blocks that are privately accessed by a core. In fact, on the average, the percentage of blocks ever shared by two or more cores constitute only 36% of the total L2 blocks across these twenty two benchmarks.

While the above observation may suggest that optimizations for sharing are not important, we note that the percentage of accesses to shared blocks paints a completely different picture, as illustrated in Figure 5.1 (b). In fact, shared accesses dominate the L2 accesses in many of the benchmarks, averaging 78% of the total L2 accesses across our twenty two benchmarks. This makes it essential to optimize the latency for shared data accesses in L2.

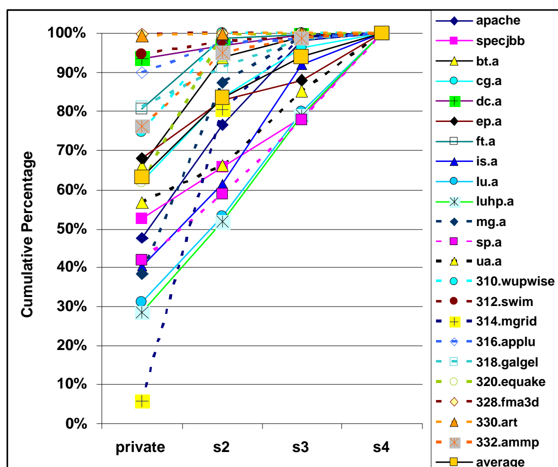
It is possible that, despite this magnitude of shared block accesses, the accesses from the different cores are temporally separated such that they could still be treated as virtually private

accesses to a core. In Figure 5.2, we plot the temporal characteristics of accesses to shared blocks. In this graph, we use the notion of a *running distance*. We define the running distance as the number of accesses by a core to a L2 block before it is accessed by another core. For instance, D0 denotes that after a core accesses a block, the next access to the same block is from another core. D1 denotes that there is one more access from the same core, before another core accesses it. Larger the running distance, the higher the temporal locality from the same core, making it appear more private than shared. As can be gleaned from Figure 5.2, fewer than 15% of the running distances ever cross 4. Most running distances are lower than 3, with single touch accesses (before another core touches the block) constituting a significant fraction (over 30%) in many benchmarks. That is, there is a high degree of active sharing in these benchmarks.

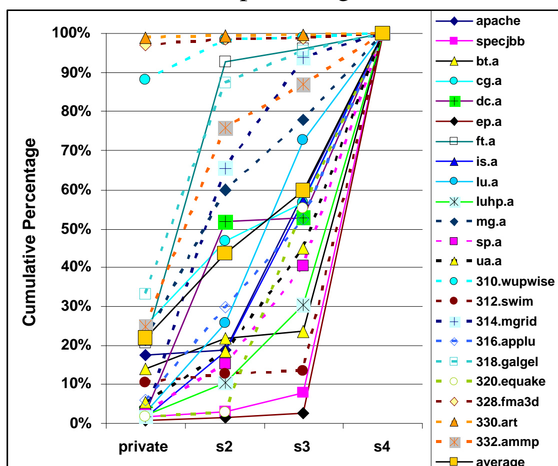
All these characteristics point out that latency of accesses to shared blocks in L2 is very critical. It is not just there are high number of accesses to such data, but the accesses from different cores are temporally interleaved (i.e., the blocks are actively shared). However, the leverage that we may have towards optimizing accesses for such data is the observation that the number of blocks that fall in this category is relatively low. These observations motivate our L2 organization described in the next section.

5.4 Center-Cell based L2 Organization and Data Lookup

The results from the previous section show that the number of L2 blocks shared among processors is not very high, whereas the number of accesses to these shared blocks is high. Therefore, while the L2 space allocated to shared blocks can be small, its position within L2



(a) Cumulative percentage of L2 blocks



(b) Cumulative percentage of accesses to
L2 blocks (private or shared by 2,3,4)

Fig. 5.1. Percentage of L2 blocks and accesses. ”s2”, ”s3” and ”s4” on X-axis indicate that the number of cores sharing/accessing the blocks is 2, 3 and 4 respectively. Each graph is drawn as a cumulative percentage starting from the ”Private” case.

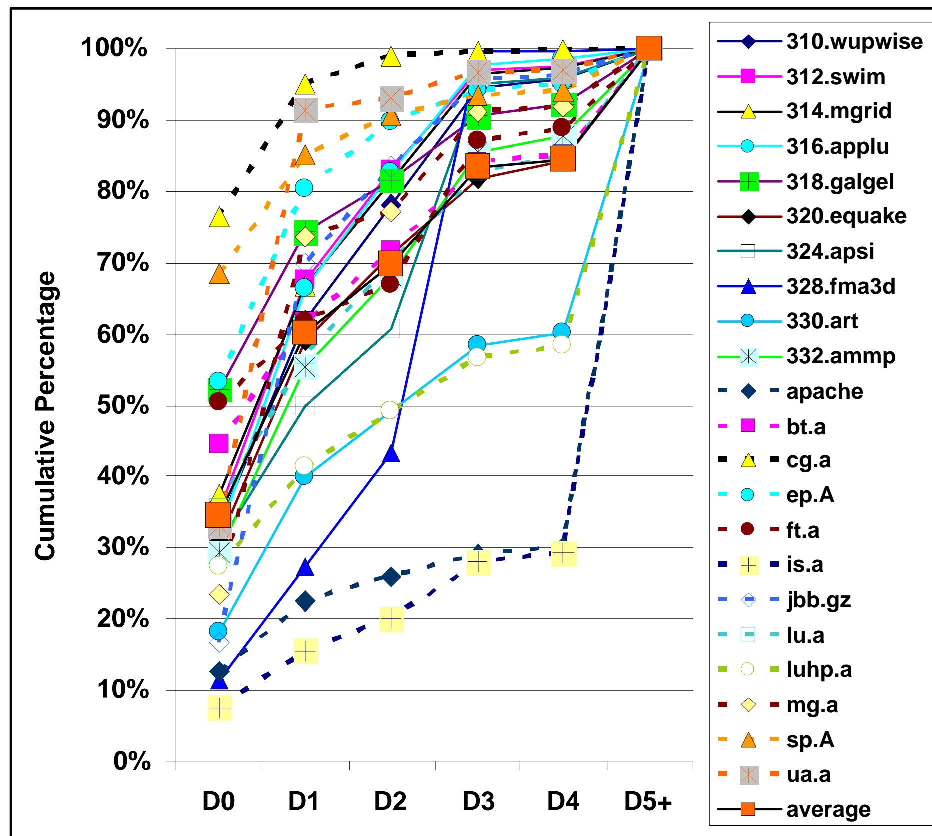


Fig. 5.2. Running distance distribution of the shared blocks in L2. X axis shows the running distance, and the Y axis shows the cumulative percentage of occurrences of that running distance.

is critical[1]. In particular, since some of these blocks are widely shared across processors (see Figure 5.1), it may not be a good option to place them close to only a single processor or into a location which is far from all processors (as in the case of [10]).

Based on these observations, we propose a new L2 organization depicted in Figure 5.3. In this organization, the L2 space is divided into multiple cells of equal size, and one cell (of 64KB), referred to as the *center cell*, is reserved for the shared blocks. The statistics presented in the previous section suggest that this small capacity for the center cell should be sufficient to capture most of the actively shared blocks. Note that this shared cell is placed in the middle to enable fast access from every processor core in the architecture. The remaining cells are distributed in the L2 space as illustrated in Figure 5.4 from the perspective of a single processor. As far as the access latencies are concerned, the cells form three non-overlapping *rings* around a processor core. The first ring, Ring-0, is the fastest one. It contains 18 cells, with access latencies ranging from 4 cycles to 12 cycles. The second ring, Ring-1, accommodates 25 cells, with a maximum latency of 20 cycles. Finally, the outermost ring, Ring-2, includes the remaining 21 cells, with a maximum access latency of 32 cycles. A cell residing in any of these rings is referred to as the *preferred cell* for that core; i.e., all these 64 cells are the preferred cells for that core, whereas the remaining cells in the cache are termed as the *non-preferred* cells. The goal in this design is to cluster the privately-accessed data of each core into its preferred cells to reduce access latency. Clearly, we want the most frequently used blocks to reside in Ring-0 cells. Note that such a design also enables the use of clustered tag [15], if desired, to obtain a summary of the L2 tags

[1] Note that our L2 is shared, and therefore, a shared block can be anywhere in the L2. However, we want to place actively shared blocks into a certain cell (center cell) for improving performance.

to quickly determine if a cache line exists in the preferred cells or not. Consequently, both access latency and dynamic power consumption can be cut down.

An important question at this point is how fast an access to the center cell can be. There are two primary factors that influence the access latency of a cell in this organization: its capacity and its distance from the processor that accesses it. In this case, the capacity does not present a problem since it is really small (64KB). As far as the distance is concerned, we assume the availability of a low latency long wire [9], which is typically used for fast interconnects. We use additional set of low latency wires, shown as the big “I” in Figure 5.3, to connect the center cell to the point of entry of the preferred cells. The results we obtained from using the Berkeley Predictive Technology Model (BPTM) [12] to calculate the wire delay for 3200um global wire at 25nm technology are shown in Table 5.2. Our calculations indicate that the access latency for the center cell is 5 cycles at 10 GHZ. Overall, the center-cell based L2 organization shown in Figure 5.3 provides very fast access to shared blocks (considering the fact that the fastest preferred cell has a latency of 4 cycles as shown in Figure 5.4). As a result, we do not need to migrate back and forth a shared block frequently between different regions of L2, as in the case of prior work [10].

Material	**Technology**	**W**	**S**	**Length**	**T**	**H**	**Dielectric**	**Delay**
Cu	25nm	0.35um	0.35um	3200um	1.2um	0.15um	2.0	0.06ns (1 cycle)

Table 5.2. Important parameters for a 3200um low-latency wire at 25nm.

There are three critical issues that need to be addressed in the context of this center cell based L2 organization: (1) Placement: Where should a new (incoming) L2 block be placed in the L2 space?; (2) Migration/Eviction: When/how should a block be migrated within L2 based on variations in interprocessor data access and sharing patterns? Also, what happens to a block that needs to be replaced (evicted) from a cell?; and (3) Search: How should we search for a block in L2? The rest of this section discusses these issues in detail. Our goal in studying these questions is to understand the ways in which the access non-uniformity in this L2 organization can be exploited.

5.4.1 Placement

A placement scheme decides where to store a new block coming from off-chip memory. While one can place it into any randomly-selected cell in L2, a better option would be something like a *first-touch* policy, where the block is placed into one of the fastest cells (4 cycle latency) in the preferred region of the core that brings it into L2. Since our early evaluation showed that the first touch policy is clearly superior to the random placement policy, we use the former in all our experiments.

5.4.2 Migration and Eviction

In this center cell based L2 architecture, a block can migrate from one cell to another in two ways: *access triggered* and *eviction triggered.* When a block is requested by a processor, we may want to migrate it to a different cell than its current one to better exploit non-uniform access latencies in the future accesses. For example, while the block is in a preferred cell of a

core, an access to it by another core moves it to the center cell[2]. It is possible at this point that the center cell is full, and consequently, a block needs to be evicted from it to create space for the new block. Our approach evicts the victim block to one of the lowest latency preferred cells (i.e., 4 cycle cells) of the core that used the block in question last time. This is an example of eviction triggered migration. In our architecture, an eviction triggered migration can cause subsequent evictions. For example, a block evicted from the center cell into a preferred cell of 4 cycles can cause another block from that cell to be evicted to a cell of 8 cycles, and this in turn, can cause yet another block from the 8 cycle cell to be evicted to a cell of 12 cycles, and so on. Figure 5.5 illustrates this cascade evictions. Note that, while, as explained above, these evictions can be triggered by a block evicted from the center cell, they can also occur when a new block coming to L2 displaces a block that resides in the 4 cycle cell (i.e., as a result of placement), or when a block coming from a high latency cell displaces a block from a low latency cell. Figure 5.6 illustrates an example migration and eviction. Note that, in our L2 organization, when a blocks is brought into the center cell, it remains there until it is evicted.

5.4.3 Search

In our L2 organization, data lookup can be performed in different ways, depending on how tags and data are organized, each exhibiting a different trade-off between latency and power consumption. In this work, we study two different schemes which we call *CT (Clustered Tag)* and *DT (Distributed Tag)*. Each scheme has its own tag/data organization. In the *DT* scheme, tag and data are stored close to each other in the same cell. While it is possible to lookup the

[2] Another type of access triggered migration is to move a block from a high latency cell to a low latency cell upon an request.

requested data by probing the entire cache, this would normally be very costly. A better option would be to first search the data in nearby cells (i.e., the ones with low access latency) and then expand the search to other cells upon failure. This should work fine in principle due to locality of data references. Based on this observation, in the *DT* scheme, the requesting core probes its own preferred cells (as will be discussed shortly in detail) and the center cell first. In case of failure, the request is forwarded, by the center cell, to the preferred cells of the other cores. The reason that we employ the center cell for forwarding the request is to avoid a potential race condition. It could so happen that two cores start requesting the same block which sits in a third core's preferred cell. In such a case, if the center cell does not keep track of all the outstanding requests to other preferred cells, the first request will bring the block to the center cell, while the second request will miss in the third core's preferred cell and will subsequently issue a request to the off-chip memory. To prevent this from happening, the center cell in our implementation maintains an *outstanding request queue* (only for the requests directed to other preferred cells) so that the second request in question could be suppressed if it hits in the outstanding request queue of the center cell. In the rest of this chapter, we refer to this search strategy as *CC.DT*, which means center cell based organization which uses distributed tags.

The behavior of this search scheme can be affected by tuning the order in which different cells are looked up. Specifically, we implemented three different variants of *CC.DT*:

- Aggressive Search will start probing the non-preferred cells immediately if the requested block is missing in its Ring-0 (and in the center cell). This illustrated in Figure 5.7(a).

- Moderate Search will start probing the non-preferred cells only when the requested block is missing in its Ring-1. This is shown in Figure 5.7(b).

- Conservative Search, shown in Figure 5.7(c), will start probing the non-preferred cells only when the requested block is missing in its Ring-3.

There is a tradeoff between performance (lookup delay) and power consumption among these three probing (search) schemes. The aggressive search scheme is clearly preferred from the performance angle. On the other hand, one can also expect its power consumption to be very high. The conservative search scheme represents the other extreme where latency can be very high (as accesses to preferred cells and non-preferred cells are serialized). However, its power consumption will be low in cases the requested block is caught in Ring-1 of the requester. The moderate search strikes a balance between these two extremes.

In the *Clustered Tag* scheme (denoted as *CC.CT* in our discussion), tags are clustered together and stored in places close to the core, similar to [15]. When a request comes to a preferred/non-preferred region, we first search the tags to determine the data cell in which the requested block resides, which can be accomplished in 4 cycles, based on our layout. Once this cell is located, we directly access it for retrieving the block. In our implementation of *CC.CT*, however, we still use the concurrent tag-data search for the center cell. This is because (1) this cell is small and (2) its latency affects the performance of all four cores. Consequently, in *CC.CT*, when a core misses in L1, it first probes the center cell and the tags of its preferred cells. If the request hits in the center cell, only the center cell will be accessed. If the tags of the preferred cells report a hit, the core needs to send its request only to the cell that has the block. This approach, depicted in Figure 5.7(d), tends to reduce unnecessary search in other cells and can be beneficial from the dynamic energy viewpoint.

Figure 5.8 shows the reachable cache percentage (i.e., the fraction of L2 cache that can be accessed) with a given number of cycles. As we can see from the figure, among the *CC.DT* schemes, more aggressive search options can cover larger portion of the cache under the same latency value, compared to the less aggressive search options. For example, given 30 cycles, the aggressive, moderate, and conservative search schemes can cover 60%, 40% and 25% of L2 cache, respectively. On the other hand, *CC.CT* performs better than all the *CC.DT* variants beyond 12 cycles. But, up to 12 cycles, *CC.CT* covers much less cache capacity than all the *CC.DT* schemes due to its serial tag-data access.

5.5 Results

5.5.1 Experimental Parameters

The simulated system has 16MB+64KB shared L2 cache with each core having 4MB *preferred* cells, and the *center cell* is 64KB. As discussed before, the access latency to the center cell from any of the cores is 5 cycles. The access times to the preferred cells have already been explained (see Figure 5.4). The rest of the simulation parameters are given in Table 5.3.

The schemes that we are experimentally comparing are summarized in Table 5.4. We have already described our *CC.CT* and *CC.DT* schemes where we use a 64KB center cell, with the difference being whether the tags are distributed or clustered. In the *M.CT* architecture, we do not have a center cell, i.e., L2 has only preferred regions; each is 4MB with their access latencies given in Figure 5.4. The migration scheme automatically migrates the block to the core accessing the block upon each request. Consequently, under *M.CT*, a shared block can keep shuttling between the preferred regions of the different cores. While the search strategy

Parameter	Value
Instruction Set	Sparc V9
CPU Cores	4 In-Order Single Issue cores
Core Speed	10GHZ
L1 I & D Caches	16KB, 4-way, 3 cycles
Center Cell	64KB, 4-way, 3 cycles
	4 outstanding requests
L2 Cell	64KB, 4-way, 3 cycles
Total L2 capacity	16MB
L1,L2 cache block size	64 Bytes
Memory latency	200 cycles

Table 5.3. Simulation parameters.

is to serially probe tag and data in the case of *M.CT*, the search is done in parallel for *M.DT*. The replacement mechanism for *M.CT/M.DT* works the same way as *CC.CT/CC.DT*, except that there is no center cell to be considered. Note that, like *CC.DT*, *M.DT* can also have different search options such as Aggressive, Moderate, and Conservative.

Finally, in order to demonstrate that it is important for the center cell to be equally close to all the cores, rather than equally distant from them (as in [10]), we also introduce one more scheme called *CC.Large* where the center cell is much larger (4MB versus our default 64KB) with a longer access latency (15 cycles from each core). In this scheme, the preferred L2 regions have 3 MB for each core, with their maximum latency now reduced from 32 to 24 cycles.

5.5.2 Reduction of Cache Block Migration

Our first set of results is intended to show that our approach can considerably reduce the number of block migrations between the different parts of L2, since that can impact both performance and power. Figure 5.9 shows the number of migrations of *CC.DT* normalized with respect to *M.DT*. As can be seen, by adding a center cell, we are able to drastically cut down the

	M.CT	CC.CT	M.DT	CC.DT	CC.Large
Preferred size / core	4MB	4MB	4MB	4MB	3MB
Latencies for preferred cells	tag: 4 cycles + 4, 8, 12, 16, 20, 24, 28, 32	tag: 4 cycles + 4, 8, 12, 16, 20, 24, 28, 32	4, 8, 12, 16, 20, 24, 28, 32	4, 8, 12, 16, 20, 24, 28, 32	4, 8, 12, 16, 20, 24
Center cell size	None	64KB	None	64KB	4MB
Center cell latency	None	5 cycles	None	5 cycles	15 cycles
Minimum access latency	8 cycles	8 cycles	4 cycles	4 cycles	4 cycles
Maximum access latency	40 cycles	41 cycles	64 cycles	65 cycles	49 cycles

Table 5.4. Schemes we compare in our experiments. Table shows the sizes of the center cell and preferred L2 regions of each core. It also shows the latency to these L2 portions from each core.

shuttling of blocks between the different L2 regions. When a block in *CC.DT* is migrated to the center cell, it remains there even when another core accesses it, while in *M.DT*, the blocks would frequently move from one preferred region to another. On the average, the center cell saves about 63% of the block migrations. Note that we achieve these savings with just a 64K center cell, and we do not need a large center cell (as in *CC.Large*) to obtain such drastic reductions (as is also illustrated in the same figure).

5.5.3 Profile of L2 Hits

Having shown that the center cell substantially reduces the number of block migrations within L2, we now illustrate how effective it can be toward satisfying the requests from each core by examining the behavior of CC more closely (note that the hit behavior of *CC.DT* and *CC.CT* are the same). Figure 5.10 (a) shows the breakdown of L2 hits in terms of (i) hits in the preferred cells of the requesting cores, (ii) hits in the center cell, and (iii) hits in the non-preferred cells. We can observe the effectiveness of our scheme from this bar-chart by noting that the number of hits in the non-preferred cells is fairly low (less than 6% on average). The center cell is able

to satisfy a considerable number of requests in many of the applications. Specifically six of the applications - 312.swim, 316.applu, 328.fma3d, apache, jbb and cg.a - have 20% or more hits in the center cell. Note that these were also the applications which showed a low running distance in Section 5.3. The average hit ratio in the center cell is around 15.6% across all applications.

We wish to point out that in addition to the hit ratio (in the center cell), the positional information of the hits in the preferred and non-preferred regions are equally important since the access latencies are quite dependent on the location of the cell that satisfies the requests. Figures 5.10 (b) and (c) give the breakdown of the hits in the preferred and non-preferred L2 cache regions, respectively, in terms of which ring satisfies a request. From the figure for the preferred regions, we see that we are exploiting the NUCA property by meeting more of the requests from Ring-0 which has lower access latency than the other two rings. The figure for the non-preferred accesses gives us additional interesting insights. We note that the hits, here again, are coming from the lower number rings (particularly Ring-0 which satisfies around 78% of the non-preferred hits on the average). This is an indication that the requests from different cores are more temporally close to each other (i.e., more active sharing), which is again another motivation for the architecture proposed in this chapter.

5.5.4 L2 Hit Latency

Having shown the profile of where the data is located on an L2 hit, we now show the actual L2 access latency upon a hit in Figure 5.11. This experiment accounts for the latency of accesses to the different parts of L2 as explained earlier. In nineteen out of the twenty two applications, *CC.DT* (with the *aggressive* search option) has the lowest hit latency. In general, CC does better than M, because of the presence of the center cell which is equally accessible

by all the cores. In the M schemes, the high sharing behavior is causing the blocks to migrate frequently between the different regions, as was observed earlier, leading to a higher access latency overall (i.e., a core may frequently need to go to a non-preferred region to get its blocks). On the other hand, CC can locate the data in the center cell rather than having to go to a non-preferred region each time a core misses in its preferred region.

CC is also a better option than *CC.Large* since (i) in the case of a hit in the center cell, the latter incurs a much higher access latency than the former (20 cycles versus 5 cycles) because of its much larger size, (ii) as noted earlier in our characterization studies, we only need a small center cell to hold much of the shared working set, making the hit rates in the center cell of CC and *CC.Large* not very different, and (iii) in *CC.Large* we are cutting down the size of the preferred regions in the expectation of increasing the hit rates of the center cell.

Between the *CC.DT* (which uses aggressive searches) and *CC.CT* search techniques, the latter always incurs an additional tag look up latency (4 cycles) before the data is accessed, while the former does both in parallel for a given L2 cell. Since our statistics provided earlier show that most of the accesses hit in Ring-0, the tag accesses in *CC.CT* get in the critical path making it slower. The exceptions are cg.a, and to a lesser extent, SPECJbb. As observed earlier in Figure 5.10 (a), cg.a has the most non-preferred hits. In *CC.CT*, having all the tags clustered together helps us detect preferred region misses faster, to initiate the search in the non-preferred regions earlier (where it is found in Ring-0). Overall, the average L2 hit latencies for *M.CT*, *M.DT*, *CC.CT* and *CC.DT* under the aggressive search option are 15.3, 12.9, 14.2 and 11.2 cycles, respectively. That is *CC.DT* achieves 9% saving over *M.DT* on average.

Having shown the aggressive search results for *CC.DT*, we now examine the L2 hit latencies for the *moderate* and *conservative* search schemes, and we present the results in Figure

5.12. The results for *CC.DT* under the aggressive search option are reproduced here for the ease of comparison. As expected, the aggressive scheme does better since the search ends quicker for the hits in the non-preferred regions. This benefit would be more significant for applications which have higher hits in non-preferred regions as is the case of cg.a and SPECJbb. On the average, the results with the different variants of *CC.DT* are within 8% of each other, and thus one may simply want to employ the moderate or conservative search schemes in the interest of lowering interconnect traffic and reducing power.

5.5.5 Unnecessary Non-Preferred Lookups

In the aggressive and moderate (*CC.DT*) search schemes, it is possible for a search to be triggered in the non-preferred region even if the data block is present in a higher numbered ring of the preferred region. The distribution of such redundant lookups is illustrated in Figure 5.13, where we find that around 28.7% (on the average) of the L2 accesses incur these unnecessary non-preferred lookups, compared to 11.5% for the moderate search scheme. There results reiterate the earlier observation that it would be better to opt for either a moderate (conservative) search technique, which would incur much fewer (no) unnecessary non-preferred lookups (for network traffic and power considerations), while providing as good performance as the aggressive technique.

5.5.6 Number of L2 Cells Accessed

Even though the sequential tag and data access for *CC.CT* incurs higher latency before getting the data even from Ring-0, it will not unnecessarily probe too many L2 cells. Thus, it can possibly be more efficient from the dynamic power considerations. In Figure 5.14, we

show the average number of L2 cells accessed per L2 hit in *CC.CT* (which can be viewed as the lower bound), and *CC.DT* with aggressive, moderate and conservative probing. In general, we find that the aggressive search option probes a lot more cells than the other schemes making this approach less preferable than the others. The moderate approach provides a considerable reduction in the probed cells, making it a more attractive alternative. The only exception is cg.a, where the aggressive search can find the block faster in a non-preferred region, thus avoiding the probe of other cells (in larger numbered rings) of the preferred region.

5.5.7 Average Access Energy

Figure 5.15 shows the average energy consumption number per L2 access. To calculate the data and tag access power numbers, we used CACTI [69]. This is shown for our proposed CC schemes (*CC.CT*, and *CC.DT* with the moderate search option), and compared with the corresponding schemes without a center cell (*M.CT* and *M.DT* with moderate), and the scheme which uses a large center cell (*CC.Large* with moderate). We present the results in two graphs for ease of comparison. In Figure 5.15 (a), we compare *M.CT* and *CC.CT*, and in (b), we compare *M.DT*, *CC.DT*, and *CC.Large*.

Similar to the latency results, we find that the large center cell (*CC.Large*) does poorly in terms of the energy consumption as well. As noted earlier, shared data access constitute a substantial portion of the L2 accesses, and by making the center cell large, we are incurring much more energy per access. This is another reason in favor of opting for a much smaller center cell as in our proposal. When we look at Figure 5.15 (a), we see that *CC.CT* saves 6% energy on average than *M.CT*. Similarly, from Figure 5.15 (b), one can observe that *CC.DT* saves

15% energy than *M.DT*. The results show that having a small center cell in the organization is important from power perspective as well.

5.5.8 Execution time

We finally present the overall execution cycles for the different schemes across our benchmarks in Figure 5.16. Since our point here is to illustrate the benefit of our center cell architecture, we present the reduction in execution time for the different schemes with respect to that for the *M.CT* scheme (which frequently shuttles the shared blocks among the preferred regions of each core).

As we can see from these results, in most of the applications the *CC.DT* scheme does the best, providing 2.1% execution time improvement over *M.CT* on the average. In some applications, such as ft.a, it provides as much as 6.3% performance improvement. In the few cases where *CC.DT* does not fare as well, as in cg.a and SPECJbb where the hits in the non-preferred regions are high (see Figure 5.10), we note that *CC.CT* does the best. As noted earlier, in addition to the non-preferred region hits, *CC.CT* benefits from the hits in the lower latency center cell as well compared to *M.CT*. Finally, the choice of a small center cell is again reiterated by the poor performance of *CC.Large*, as in Apache, where the hits in the higher latency center cell lead to worse execution time.

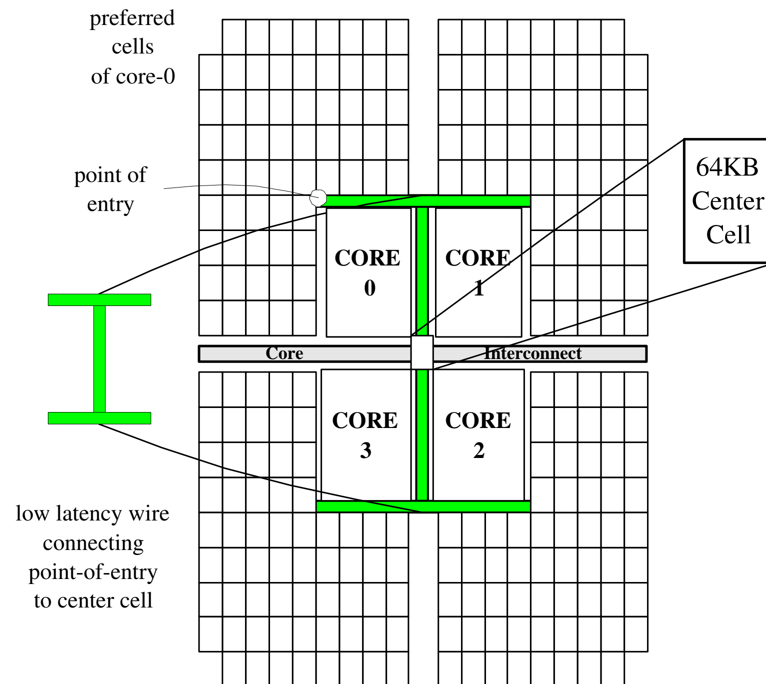


Fig. 5.3. Four-core 16MB L2 chip layout.

	32	28	24	20	20	24	28	32
32	28	24	20	16	16	20	24	28
28	24	20	16	12	12	16	20	24
24	20	16	12	8	8	12	16	20
20	16	12	8	4	4	8	12	16
20	16	12	8	4				
24	20	16	12	8				
28	24	20	16	12				
32	28	24	20	16				

Ring 2
Ring 1
Ring 0
Point of Entry

Fig. 5.4. Layout of the preferred cells and access latencies as seen from a core's perspective.

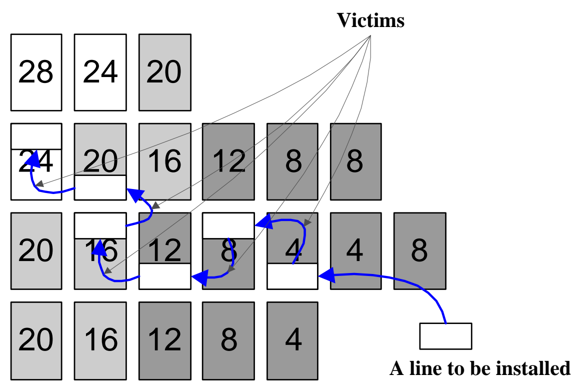


Fig. 5.5. Cascade evictions of L2 blocks.

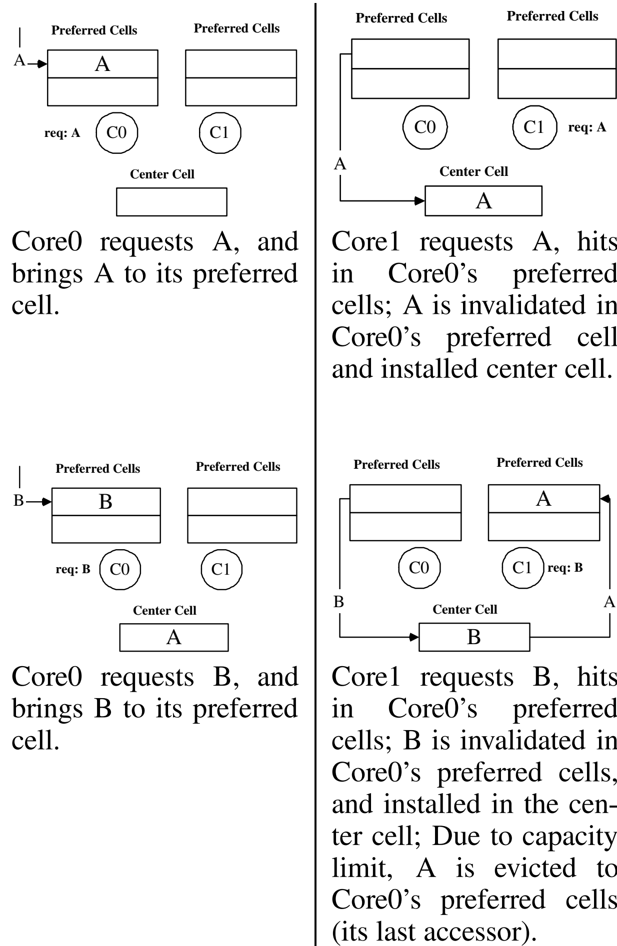
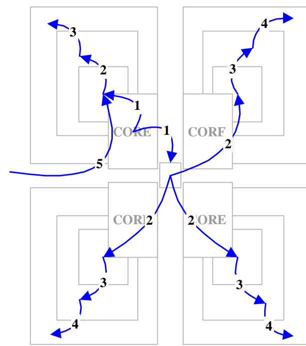
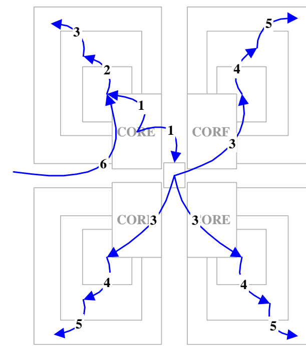


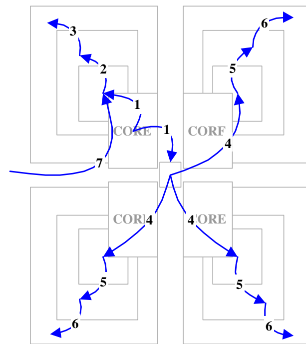
Fig. 5.6. Example migration and eviction.



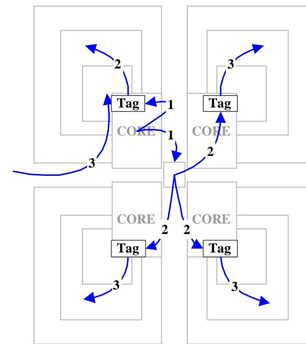
(a) *CC.DT* with Aggressive search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1 and all three non-preferred Ring-0
3. If still missing, probe preferred Ring-2 and all three non-preferred Ring-1
4. Probe all three non-preferred Ring-2 if needed
5. Bring data in through memory interface



(b) *CC.DT* with Moderate search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1
3. If still missing, probe preferred Ring-2 and all three non-preferred Ring-0
4. Probe all three non-preferred Ring-1 if needed
5. Probe all three non-preferred Ring-2 if needed
6. Bring data in through memory interface



(c) *CC.CT* with Conservative search
1. Probe both Ring-0 and center cell
2. If miss in both, probe preferred Ring-1
3. If still missing, probe preferred Ring-2
4. Probe all three non-preferred Ring-0
5. Probe all three non-preferred Ring-1
6. Probe all three non-preferred Ring-2
7. Bring data in through memory interface



(d) *CC.CT*
1. Probe the tags of preferred cells and center cell
2. If miss, probe non-preferred tags
3. If miss in non-preferred cells, bring data through memory interface (note: access could happen in any of the rings, thus the corresponding latency value varies)

Fig. 5.7. The search (probing) strategies employed by *CC.CT* and by different variants of *CC.DT*

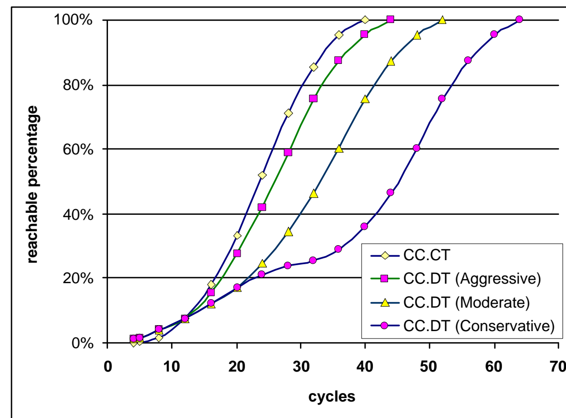


Fig. 5.8. Fraction of L2 cache that can be accessed under a given number of cycles.

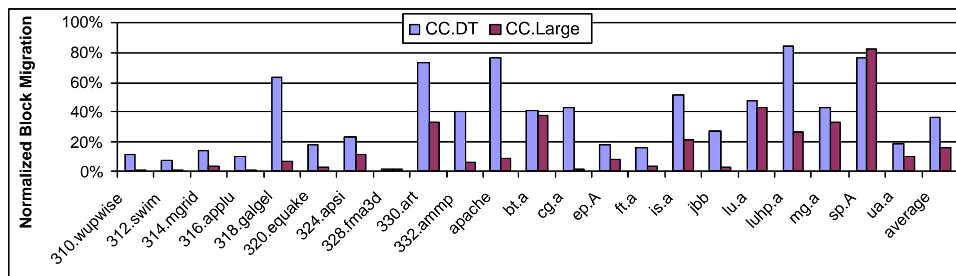
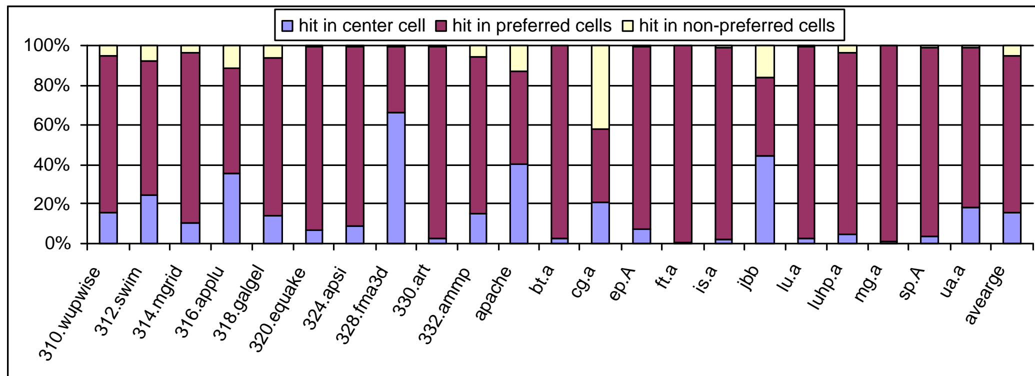
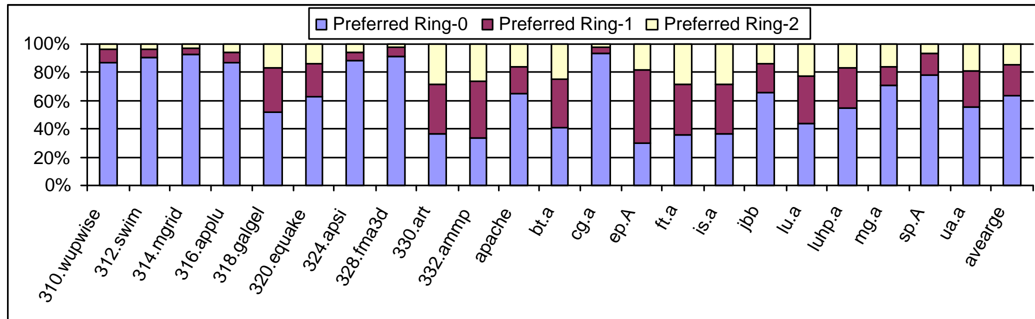


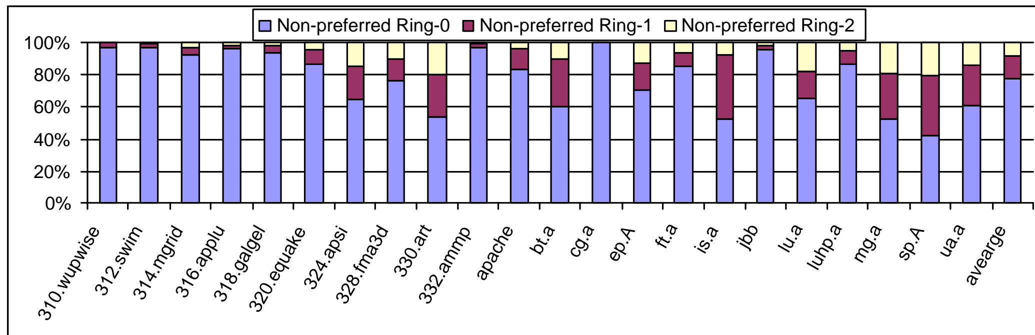
Fig. 5.9. Normalized cache block migration with *CC.DT* and *CC.Large* with respect to *M.DT*.



(a) Breakdown of L2 hits.



(b) Breakdown of L2 hits in preferred regions.



(c) Breakdown of L2 hits in non-preferred regions.

Fig. 5.10. L2 hit distribution.

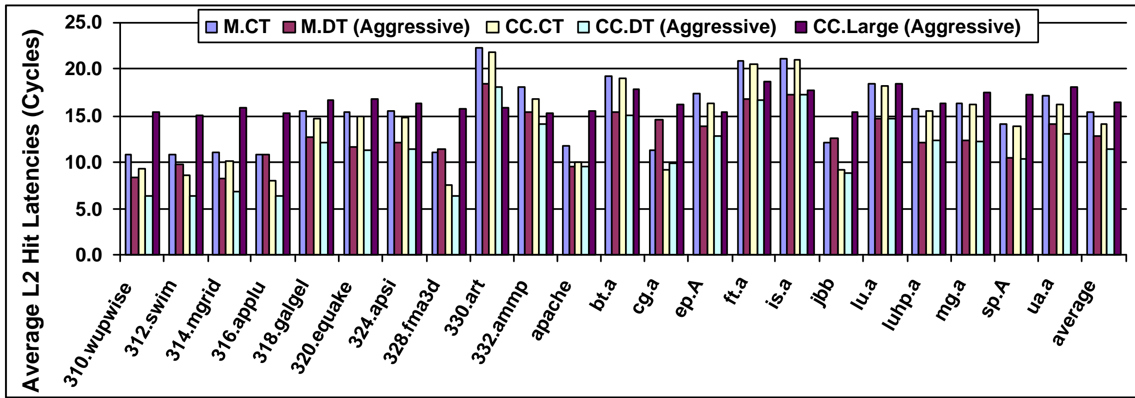


Fig. 5.11. Average L2 hit latencies with the different schemes.

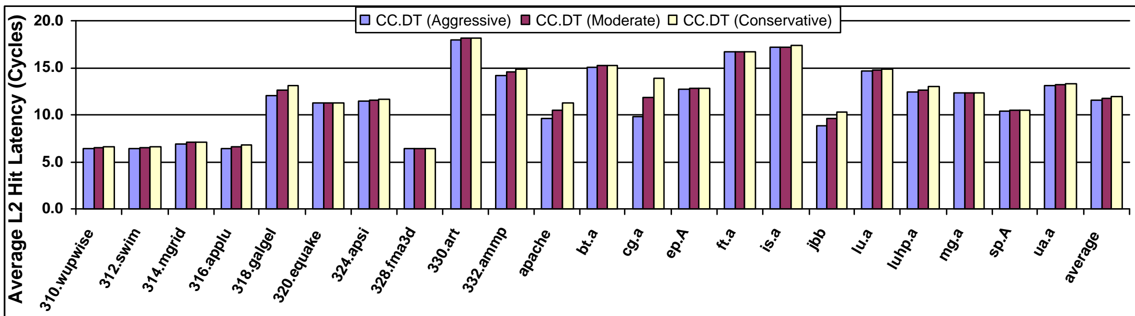


Fig. 5.12. Average L2 hit latencies for *CC.DT* under the different search options.

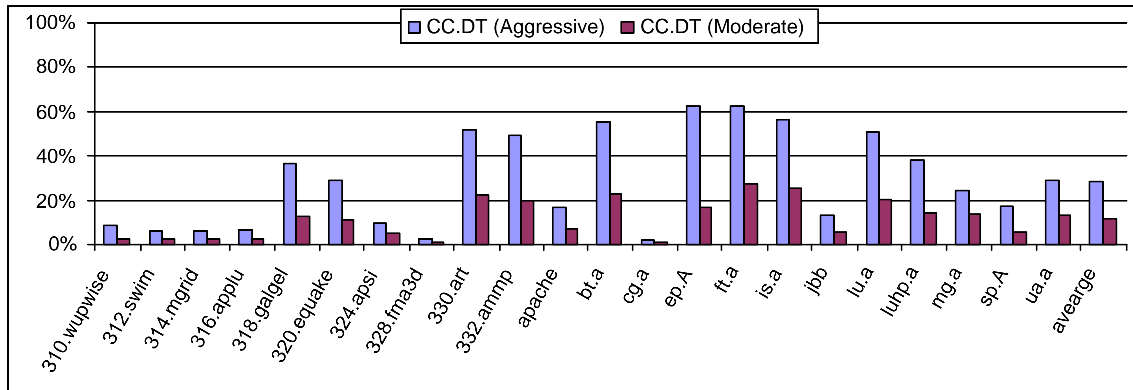


Fig. 5.13. Unnecessary non-preferred lookups under the aggressive and moderate schemes as a percentage of L2 accesses.

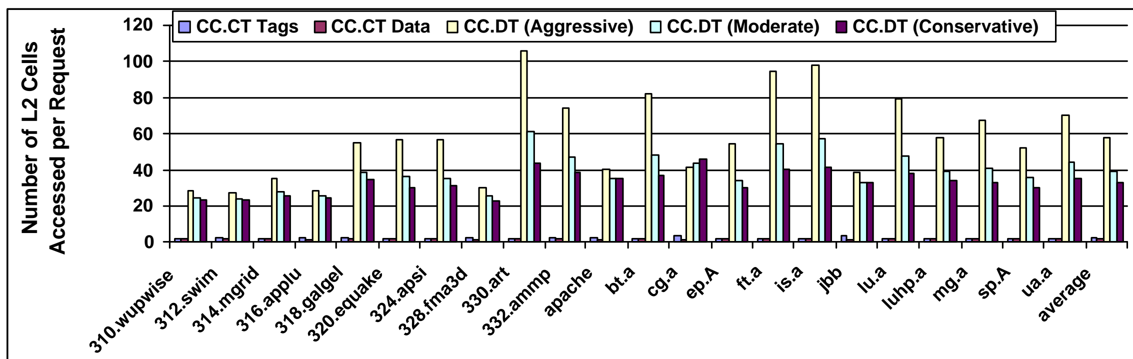
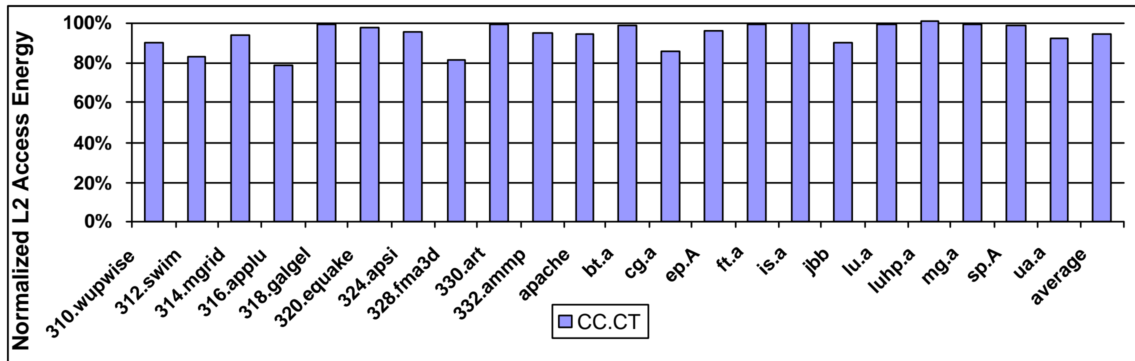
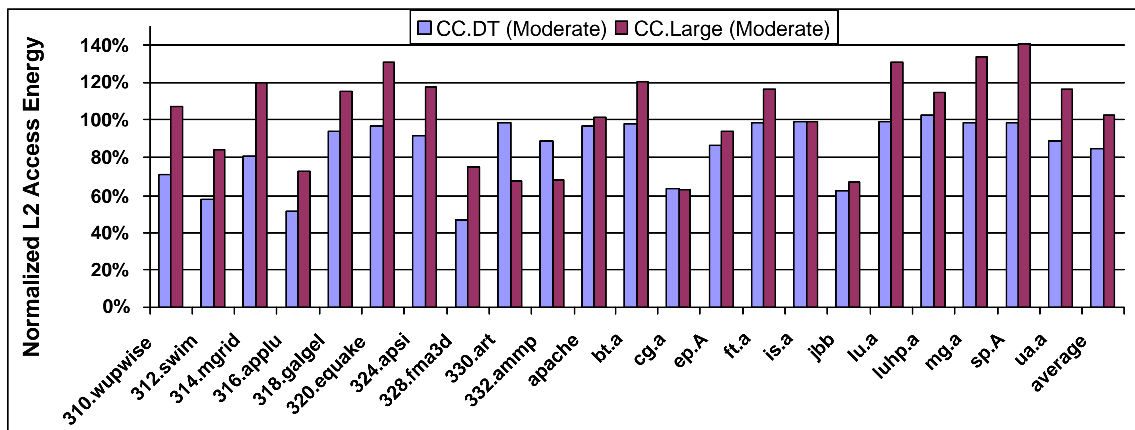


Fig. 5.14. Average number of cells probed for each L2 hit.



(a) *CC.CT* normalized with respect to *M.CT*.



(b) *CC.DT* and *CC.Large* normalized with respect to *M.DT*.

Fig. 5.15. Normalized average energy consumption per L2 access.

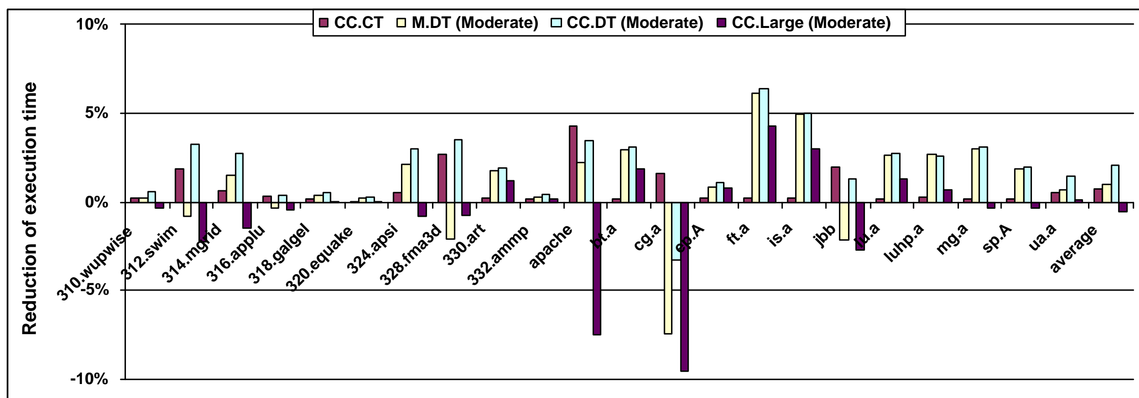


Fig. 5.16. Reduction in execution time with respect to *M.CT*.

Chapter 6

Conclusion and Future Work

Recent developments in silicon technology provide more transistors than ever on a single chip. At the same time, wire delay, design complexities and power consumption prohibit large, unscalable and complex designs. CMPs show promise in tackling these emerging issues. Even though CMPs can be used to accelerate the performance of a single threaded application's execution, or enhance the reliability of the system using redundant threads, this thesis focuses on the traditional use of CMPs, which is primarily to speed up the performance of multi-threaded applications. In this thesis, we first attempt to characterize the multi-threaded application's behavior along two dimensions: execution time and cache demands.

Chapter 3 characterizes the execution time of barrier construct of OpenMP applications, which provides a convenient mechanism for tracking work imbalances among the processors and their resulting idleness. By predicting this idleness, it is possible to scale the frequencies for the duration of the work to be done before getting to the barrier the next time. In this way, we obtain power savings and reduce or eliminate idleness. We show that simple predictors can effectively estimate idleness for different SpecOMP applications. Using detailed complete system (including OS and OpenMP library) simulation of application code, we demonstrate that our approach can provide considerable power savings for these applications: as much as 40% power savings, including 32% on the average across five applications, with little impact on performance. We

also show that this strategy provides higher power savings than a previously proposed strategy that simply switches the executing core to a low power mode during the idleness at the barrier.

To answer why different threads executing the same loop could have different execution times, in Chapter 4 we start our search with an investigation on the processor stalls due to L2 accesses. We present a quantitative measure of load imbalance of cache demands and evaluate the two well-known organizations of L2 level cache: private and shared. In the case of private organization, CPUs are relatively insulated from the load they impose on this level, and usually do not need to traverse a shared interconnect to get to this level in the common case. On the other hand, the advantage of the shared organization is that it can allow one CPU to eat into the cache space of another whenever needed to allow more heterogeneous allocation of this cache space. However, such a capability can also become very detrimental in causing interference between the address streams generated by the different CPUs, thereby incurring additional misses.

Recognizing these trade-offs, we have proposed a new last-line-of-defense organization. In order to balance the load, we use the underlying advantages of the shared structure, and at the same time provide a way to insulate the diverse requirements across the cores whenever needed. The basic idea is to have multiple cache units allocated on a CPU-ID basis (rather than on an address basis as is done in typically banked caches). Each lookup from a CPU looks first at its set of units, but can subsequently lookup other units as well before going off-chip, thus maintaining the shared view. Upon a miss, the request can allocate the block only into one of its units, thereby reducing the interference.

We have proposed a flexible mechanism for implementing this shared processor-based split organization, that allows software to configure the splits spatially (between the CPUs) and temporally (can vary over time). With CMPs capable of targeting a diverse range of workloads,

from commercial high-end workloads to scientific and embedded applications, it becomes important to allow this flexibility for dynamic adaptation. At the same time, this mechanism can integrate easily with existing interconnects and coherence mechanisms. This organization is also fairly power efficient (though not evaluated quantitatively in this thesis), based on our findings that the number of units referenced upon each access is comparable to that of the private and address-based shared mechanisms and, in fact, reduces off-chip accesses.

Our results find that even when we use a single split for each CPU we are doing better than the private (P) or the shared address-based interleaved (SI) organizations since our strategy is able to better balance the load, and reduces the interference. Consequently, in this thesis we have not delved into methods for allocating the splits to the CPUs at runtime, and this is part of our future work. Still, preliminary results with non-uniform allocations between the CPUs show benefits in multi-programmed workloads. In these situations, as we demonstrated, loads can be quite diverse, even in a single program execution if the load imbalance is high.

With more caches integrated onto the die, non-uniformity of cache access is inevitable. We continue our search for the perfect L2 organization adapted for such non-uniformity in Chapter 5. The two main themes of prior work in exploiting non-uniformity of L2 cache latencies for CMPs have primarily tried to use replication (tag and/or data) [16, 73], and explicit migration or staging of the data based on access patterns [10]. When wroking with replication, it is necessary to explicitly maintain consistency. In our work, we have avoided the additional complexity of dealing with replication. Rather, our proposal is more in line with the latter theme of migrating the data to keep access times low for each core.

Chapter 5 presents the idea of using a small, low-latency center cell based L2 organization that is equally close to all the cores. This cell can hold shared data blocks, which constitute

a substantial number of L2 misses though the number of such blocks is itself quite low. Using a large number of diverse multi-threaded benchmarks, we have shown that this architecture is a better alternative, in terms of L2 access latency, access energy and execution time, than a scheme (without center cell) that simply migrates blocks back and forth between the regions closer to each core. It is also a better alternative than having a large center cell (*CC.Large*) and perhaps staging the data between different parts of this large cell (as is done in [10]) since our results show that there is high active sharing for the few blocks. Placing such blocks in a small center cell and not migrating them back and forth can give both performance and power savings.

There are several interesting directions for future work related to discretionarily placing data in the center cell, bringing blocks directly to center cell based on prior history and compiler support for exploiting the center cell based L2 organization. In addition, we are also working on expanding our cache organization to higher number of cores. We would also like to include more benchmarks in our study, especially server workload.

References

[1] David H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 248–259, 1999.

[2] Vishal Aslot, Max J. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[3] Vishal Aslot and Rudolf Eigenmann. Performance characteristics of the spec omp2001 benchmarks. *SIGARCH Comput. Archit. News*, 29(5):31–40, 2001.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarkssummary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 158–165, 1991.

[5] David H. Bailey, T. Harris, Rob Van der Wigngaart, William Saphir, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. technical report nas-95-010, 1995.

[6] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, 1998.

[7] L. Barroso, K. Gharachorloo, A. Nowatzyk, and B. Verghese. Impact of chip-level integration on performance of OLTP workloads. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 4–13, 2000.

[8] Luiz Andre; Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *27th International Symposium on Computer Architecture (ISCA'00)*, pages 282–293, 2000.

[9] Bradford M. Beckmann and David A. Wood. Tlc: Transmission line caches. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 43, 2003.

[10] Bradford M. Beckmann and David A. Wood. Managing wire delay in large chip-multiprocessor caches. In *Proceedings of the 37th annual International Symposium on Microarchitecture*, pages 319–330, 2004.

[11] Shekhar Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999.

[12] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. New paradigm of predictive mosfet and interconnect modeling for early circuit design. In *Proc. of IEEE Custom Integrated Circuit Conference*, pages 201–204, 2000.

[13] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[14] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-Power CMOS Digital Design. *JSSC*, 27(4):473–484, 1992.

[15] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 55, 2003.

[16] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

[17] Marcelo Cintra, Jose; F. Martinez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, 2000.

[18] Joachim Clabes, Joshua Friedrich, Mark Sweet, Jack DiLullo, Sam Chu, Donald Plass, James Dawson, Paul Muench, Larry Powell, Michael Floyd, Balaram Sinharoy, Mike Lee, Michael Goulet, James Wagoner, Nicole Schwartz, Steve Runyon, Gary Gorman, Phillip Restle, Ronald Kalla, Joseph McGill, and Steve Dodson. Design and implementation of the power5 microprocessor. In *Proceedings of the 41st annual conference on Design automation*, pages 670–672, 2004.

[19] *Crusoe Processor Model TM5700/TM5900 Data Book*, 2004. http://www.transmeta.com/crusoe_docs/tm5900_databook_040204.pdf.

[20] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[21] DAC'02: Session: Design methodologies meet network applications and system on chip design, 2002.

[22] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[23] David E. Duarte, N. Vijaykrishnan, and Mary Jane Irwin. A clock power model to evaluate impact of architectural and technology optimizations. *IEEE Transactions on VLSI Systems*, 10(6):676–686, 2002.

[24] Michel Dubois, Jaeheon Jeong, and Ashwini Nanda. Shared cache architectures for decision support systems. *Perform. Eval.*, 49(1-4):283–298, 2002.

[25] D. Marr et al. Hyper-threading technology architecture and microarchitecture, 2002.

[26] Roy T. Fielding and Gail Kaiser. The apache http server project. *IEEE Internet Computing*, 1(4):88–90, 1997.

[27] Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149, 2003.

[28] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, 1998.

[29] Richard Hankins, Trung Diep, Murali Annavaram, Brian Hirano, Harald Eri, Hubert Nueckel, and John P. Shen. Scaling and characterizing database workloads: Bridging the gap between research and practice. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 151, 2003.

[30] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 873–878, 1999.

[31] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A nuca substrate for flexible cmp cache sharing. In *Proceedings of the 19th annual International Conference on Supercomputing*, 2005.

[32] Anoop Iyer and Diana Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 379–386, 2002.

[33] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.

[34] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

[35] I. Kadayif, M. Kandemir, and U. Sezer. An integer linear programming based approach for parallelizing applications in on-chip multiprocessors. In *Proceedings of the 39th conference on Design automation*, pages 703–706, 2002.

[36] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 211–222, 2002.

[37] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122, 2004.

[38] Geraud Krawezik and Franck Cappello. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 118–127, 2003.

[39] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[40] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 81, 2003.

[41] Jian Li, Jose Martinez, and Michael Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, pages 14–23, 2004.

[42] Mikko H. Lipasti and John Paul Shen. Superspeculative microarchitecture for beyond ad 2000. *Computer*, 30(9):59–66, 1997.

[43] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, , and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[44] *MAJC5200*. http://www.sun.com/microelectronics/MAJC/5200wp.html.

[45] Jaydeep Marathe, Anita Nagarajan, and Frank Mueller. Detailed cache coherence characterization for openmp benchmarks. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 287–297, 2004.

[46] Jose F. Martinez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 18–29, 2002.

[47] Doug Matzke. Will physical scalability sabotage performance gains? *Computer*, 30(9):37–39, 1997.

[48] Cameron McNairy and Don Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(02):44–55, 2003.

[49] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Dean Tullsen. Ilp versus tlp on smt. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, 1999.

[50] *MP98:A Mobile Processor*. http://www.labs.nec.co.jp/MP98/.

[51] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 99–110, 2002.

[52] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *23rd International Symposium on Computer Architecture (ISCA'96)*, pages 67–77, 1996.

[53] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, 1996.

[54] Chong-Liang Ooi, Seon Wook Kim, Il Park, Rudolf Eigenmann, Babak Falsafi, and T. N. Vijaykumar. Multiplex: unifying conventional and speculative thread-level parallelism on a chip multiprocessor. In *Proceedings of the 15th international conference on Supercomputing*, pages 368–380, 2001.

[55] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, 1997.

[56] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*, 2004. order #:301170-001.

[57] Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. Reconfigurable caches and their application to media processing. In *Proceedings of the 27th annual international symposium on Computer architecture*, pages 214–224, 2000.

[58] Parthasarathy Ranganathan, Kourosh Gharachorloo, Sarita V. Adve, and Luiz Andre Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Architectural Support for Programming Languages and Operating Systems*, pages 307–318, 1998.

[59] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. *SIGOPS Oper. Syst. Rev.*, 29(5):285–298, 1995.

[60] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Nitya Ranganathan, Doug Burger, Stephen W. Keckler, Robert G. McDonald, and Charles R. Moore. Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp. *ACM Trans. Archit. Code Optim.*, 1(1):62–93, 2004.

[61] Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen, and Eric Debes. The energy efficiency of cmp vs. smt for multimedia workloads. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 196–206, 2004.

[62] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367, 2002.

[63] Specjbb2000 java business benchmark. standard performance evaluation corporation (spec), fairfax, va, 1998. available at http://www.spec.org/osg/jbb2000/.

[64] G. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. In *Thirteenth IASTED International Conference on Parallel and Distributed Computing System, 2001.*, 2001.

[65] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[66] M. Taylor, J. Kim, J. Miller, D. Wentzla, F. Ghodrat, B. Greenwald, H. Ho, m Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[67] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23(2):392–403, 1995.

[68] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, 1991.

[69] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model, 1996.

[70] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, 1999.

[71] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, 1995.

[72] Se-Hyun Yang, Babak Falsafi, Michael D. Powell, Kaushik Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, page 147, 2001.

[73] Michael Zhang and Krste Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005.

Vita

Chun Liu received his Bachelor of Science degree in Computer Science and Engineering in 1999 from Shanghai JiaoTong University in Shanghai, China. He received his Master of Science degree from the same university in 2001. He expects to receive his Ph.D. in August, 2005, from the Department of Computer Science and Engineering at Pennsylvania State University. He spent the summer of 2001 at IBM Watson Research Center in Yorktown Heights, NY. He spent the Fall of 2004 and Spring of 2005 at Advanced Micro Devices in Sunnyvale, CA. His research interests mainly include Computer Architecture, Parallel Computing and Performance Evaluation. He is a student member of IEEE and ACM.