The Dissertation Committee for Muhammad Aater Suleman
certifies that this is the approved version of the following dissertation:

An Asymmetric Multi-core Architecture for Efficiently Accelerating Critical Paths in Multithreaded Programs

Committee:

Yale Patt, Supervisor

Derek Chiou

Mattan Erez

Keshav Pingali

Eric Sprangle

An Asymmetric Multi-core Architecture for Efficiently Accelerating Critical Paths in Multithreaded Programs

by

Muhammad Aater Suleman, B.S.E.E.; M.S.E.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2010

Dedicated to my parents.

Acknowledgments

First and foremost, I thank my advisor Prof. Yale Patt. Most of what I know today about computer architecture, technical writing, and public speaking is what I have learned from him. I also thank him for his advice and concern in matters that were not directly related to my education. What I have learned from him will stay with me for the rest of my life.

I am grateful to Eric Sprangle for giving me the opportunity to work at Intel for four summer internships; and for teaching me processor simulators, the importance of first order insights, and the industrial perspective on computer architecture. Technical discussions with Eric have always enhanced my ideas.

Special thanks to Moinuddin Qureshi and Onur Mutlu for being my mentors in graduate research. They have both been a source of inspiration and excellent co-authors. I especially thank Onur for listening to my never-ending rants and Moin for his brotherly attitude.

Many people and organizations have contributed to this dissertation and my life as a student. I thank them all for their contributions. I especially thank:

- Lori Magruder for teaching me how to conduct research.
- Kevin Lepak for supervising my first computer architecture research project.
- Hyesoon Kim for working with me on 2D-Profiling.
- Derek Chiou and Mattan Erez for their helpful feedback on this thesis.
- Jose Joao and Khubaib for *always* being there and for the technical discussions.
- Veynu Narasiman for answering random questions on English grammar and for letting me borrow his stuff.
- Eiman Ebrahimi and Chang Joo Lee for listening to my rants.
- Rustam Miftakhutdinov for his acute feedback.

- Anwar Rohillah, Danny Lynch, and Francis Tseng for providing feedback on my ideas.
- Siddharth, Faisal, Owais, Amber, and Tauseef for being helpful in work and otherwise.
- Intel Corporation for the Intel PhD Fellowship and UT for the Prestigious Graduate Dean Fellowship.

I cannot express with words my indebtness to my parents, Sheikh Muhammad Suleman and Nusrat Suleman. They have always supported my decisions even if it required them to change the course of their life. They were my first teachers. It is from them I learned the value of education and hard work. I will just thank them by saying that any value you will find in this thesis is due to them.

I also want to thank my brother Amer Suleman for being an inspiration, for his advice, and for buying me my first electronic science lab and my first personal computer; both these gifts helped me identify my career path. I also want to thank my sister Sarwat Ajmal for being a caring sister, and for tutoring me elementary science. This acknowledgement is incomplete without the mention of my relatives Muhammad Ajmal, Mahvash Amer, Zaynah, Namrah, Areeba, Azal, and Abeera.

At the end, I deeply thank God as it is He who created the opportunities for me and brought me in contact with the right people. I now start this thesis in the name of God, the most Magnificent, the most Merciful.

An Asymmetric Multi-core Architecture for Efficiently Accelerating Critical Paths in Multithreaded Programs

Muhammad Aater Suleman, Ph.D.
The University of Texas at Austin, 2010

Advisor: Yale Patt

Extracting high-performance from Chip Multiprocessors (CMPs) requires that the application be parallelized i.e., divided into *threads* which execute concurrently on multiple cores. To save programmer effort, difficult to parallelize program portions are often left as serial. We show that common serial portions, i.e., non-parallel kernels, critical sections, and limiter stages in a pipeline, become the critical path of the program when the number of cores increases, thereby limiting performance and scalability. We propose that instead of burdening the software programmers with the task of shortening the serial portions, we can accelerate the serial portions using hardware support. To this end, we propose the *Asymmetric Chip-Multiprocessor (ACMP)* paradigm which provides one (or few) fast core(s) for accelerated execution of the serial portions and multiple slow, small cores for high throughput on the parallel portions. We show a concrete example implementation of the ACMP which consists of one large, high-performance core and many small, power-efficient cores. We develop hardware/software mechanisms to accelerate the execution of serial portions using the ACMP, and further improve the ACMP by proposing mechanisms to tackle common overheads incurred by the ACMP.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Problem

It has become difficult to build large monolithic processors because of excessive design complexity and high power requirements. Consequently, industry has shifted to Chip-Multiprocessor (CMP) architectures that tile multiple simpler processor cores on a single chip [8, 52, 54, 79]. Industry trends show that the number of cores will increase every process generation [8, 79]. However, because of power constraints, each core on a CMP is expected to become simpler and power-efficient, and will have lower performance. Therefore, the performance of single-threaded applications may not increase with every process generation. To extract high performance from such architectures, the application must be divided into multiple entities called *threads*. Threads execute concurrently on multiple cores, thereby increasing performance.

CMPs have the potential to provide speedups proportional to the number of cores on the chip if the program can be parallelized completely. However, not all portions of a program are amenable to parallel execution. Programmers either invest the enormous effort required to parallelize these portions or save the effort by leaving such portions as serial (single thread).

These serial portions become a major performance limiter at high core counts. As the number of cores increases, the time inside the parallel portions reduces while the time spent inside the serial portions remains constant (or increases). Consequently, with a large number of cores, a serial portion –no matter how small– can form the critical path through the program. We identify three major sources of serialization in multi-threaded applications: *non-parallel kernels*, *critical sections*, and

limiter pipeline stages.

Non-Parallel Kernels: Kernels that are prohibitively difficult or impossible to parallelize are left completely serial. We call such kernels *Non-Parallel Kernels.* Non-Parallel Kernels execute on a single core of the CMP while the other cores remain idle. They limit the achievable speedup. Reducing the execution time spent in such kernels not only reduces overall execution time but also increases the achievable speedup.

Critical Sections: Serialization can even occur in the parallelized portions when threads contend for shared data. In shared memory systems, multiple threads are not allowed to update shared data concurrently, known as the *mutual exclusion* principle [57]. Instead, accesses to shared data are encapsulated in regions of code guarded by synchronization primitives (e.g. locks). Such guarded regions of code are called *critical sections*. The semantics of a critical section dictate that only one thread can execute it at a given time. Any other thread that requires access to shared data must wait for the current thread to complete the critical section. Thus, when there is contention for shared data, execution of threads gets serialized, which reduces performance. As the number of threads increases, the contention for critical sections also increases. Therefore, in applications that have significant data synchronization (e.g. Mozilla Firefox, MySQL [2], and operating system kernels [83]), critical sections limit both performance (at a given number of threads) and scalability. Techniques to accelerate the execution of critical sections can reduce serialization, thereby improving performance and scalability.

Limiter-stages: *Pipeline parallelism* is a popular software approach to split the work in a loop among threads. In pipeline parallelism, the programmer/compiler splits each iteration of a loop into multiple work-quanta where each work-quantum executes in a different pipeline stage. Each pipeline stage is executed on one or more cores. The performance of a pipeline is limited by the execution rate of the slowest stage. When the slowest stage does not scale, the overall performance saturates and more cores cannot increase performance. Thus, highest performance

can be achieved only when the maximum possible resources are allocated for the acceleration of the slowest stage.

1.2 Thesis Statement

Execution of serial bottlenecks can be accelerated using faster cores in an asymmetric chip multiprocessor.

1.3 Contributions

This dissertation makes the following contributions:

1. It proposes an asymmetric multi-core paradigm which we call the *Asymmetric Chip Multiprocessor (ACMP)*. ACMP can run parallel program portions at high throughput and serial bottlenecks at an accelerated execution rate. We provide an in-depth analysis of industry trends which motivate the ACMP paradigm and prove its feasibility using simple first-order analytic models. As a concrete example of the ACMP paradigm, this thesis designs a CMP with one fast, large core for accelerating the serial program portions; and many slow, small cores for speedily executing the parallel program portions.

2. It proposes simple analytic models to show the importance of the three major sources of serialization in multi-threaded programs. This thesis describes in detail the microarchitecture, OS, ISA, compiler, and library support required for the ACMP to accelerate these serial portions.

3. It proposes *Accelerated Non-Parallel Kernels (ANP)*, a thread scheduling mechanism to accelerate the non-parallel kernels using the ACMP.

4. It proposes the *Accelerated Critical Sections (ACS)* mechanism to accelerate critical sections, thereby reducing thread serialization. We comprehensively describe the instruction set architecture (ISA), compiler/library, hardware, and the operating system support needed to implement ACS.

5. It proposes *Accelerated Limiter Stage (ALS)*, a mechanism to choose and accelerate the limiter stages in pipeline (streaming) workloads. We describe the algorithm to identify the limiter stage and the operating system/library support for ALS in detail.

6. We further show that ACMP's performance is limited due to the cache misses incurred in transferring a task from a small core to the large core. We propose *Data Marshaling (DM)* to overcome the overhead of transferring data among the cores of the ACMP.

1.4 Dissertation Organization

This dissertation is divided into nine chapters. Chapter 2 provides the background and motivation for the work. Chapter 3 presents the proposed ACMP architecture. Chapters 4, 5, and 6 describe how the ACMP can accelerate non-parallel kernels, critical sections, and limiter pipeline stages. Chapter 7 proposes the Data Marshaling mechanism. Chapter 8 describes the related work. Chapter 9 concludes the dissertation.

Chapter 2

Serial Bottlenecks

It has become difficult to build large monolithic processors because of their excessive design complexity and high power consumption. Consequently, industry has shifted to Chip-Multiprocessors (CMPs) [54, 99, 103] that provide multiple processing cores on a single chip. While CMPs are power-efficient, they do not improve performance of legacy applications written to run on a single core. To extract performance from CMPs, programmers must split their programs into multiple entities called *threads*. Threads operate on different portions of the same problem concurrently, thereby improving performance. However, it is difficult or impossible to parallelize some program portions. Such portions are often left as *serial*. The serial portions can form the critical path through the program as the number of cores increases. The three most common sources of serialization are non-parallel kernels, critical sections, and limiter pipeline stages.

2.1 Non-Parallel Kernels

Program portions which are difficult or impossible to parallelize without changing the algorithm are often left as serial to save programmer effort. Examples of difficult to parallelize kernels include loops with dependent iterations or loops with early exit conditions. These are classic Amdahl bottlenecks [10].

Figure 2.1(a) shows a simple program with two kernels K1 and K2. Kernel K1 computes the minimum of two arrays A and B by comparing each element in A with the corresponding element in B and choosing their minimum. K1's iterations are independent, i.e., they operate on different data. Kernel K2 is the code for a *2-tap averaging IIR filter*. K2 sets each element in array A (A[i]) equal to the average

of the element itself (A[i]) and the previous element in A (A[i-1]). Note that the kernel requires the new value of A[i] to compute A[i+1] which makes each iteration dependent on the previous iteration.

Since its iterations are independent, K1 can be parallelized easily such that each core[1] executes different iterations of the loop concurrently. In contrast, K2 cannot be parallelized because each iteration requires the result of the previous iteration. This makes K2 a non-parallel kernel and only one core can execute it at a given time. Note that it may be possible to parallelize K2 if the algorithm can tolerate inaccuracy. However, such algorithmic assumptions are domain-specific and not always applicable.

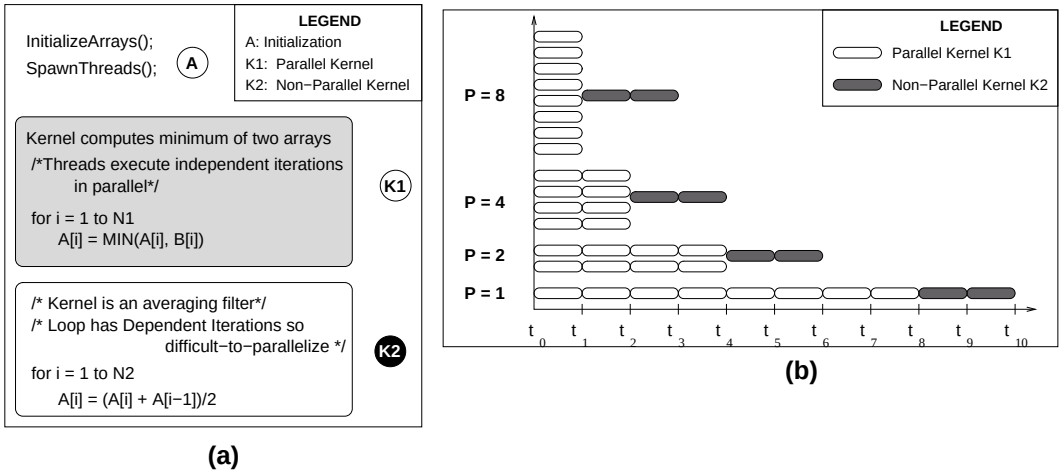


Figure 2.1: Parallel and non-parallel part in a program (a) Code example (b) Execution timeline on the baseline CMP

Figure 2.1(b) shows a simplified execution timeline of this multi-threaded program as the number of cores (P) increases, assuming K1 requires eight units of time for execution and K2 requires two units of time for execution. Since K1 can be parallelized, its work can be distributed evenly across multiple cores. Thus, when P is equal to 2, K1 executes on both cores and finishes in four units of time while K2 still requires two time units. Total execution time is 6 units. Similarly at 4 cores, K1 executes in only 2 time units and the execution time of K1 and K2 become equal.

[1]We use cores and threads interchangeably.

Overall execution time reduces by only 33% to 4 units. If we double the number of cores to 8, K2 begins to dominate overall execution time and the overall time reduces by only 25%. With an infinite number of cores, K1 takes zero time but K2 still takes 2 times units. Therefore, as the number of cores increases, increasing the number of cores becomes less beneficial for performance.

Suppose the same program executes on a hypothetical architecture which accelerates K2 by a factor of 2. The program's execution time will reduce for 1, 2, 4, and 8 cores. At 8 cores, the program will finish in two time units compared to the three time units in the baseline CMP. This shows that accelerating the serial bottleneck (which is only 20% of the program) by 2x reduces overall execution time by 33%. Furthermore, the execution time with an infinite number of cores is only one time unit instead of the two time units when K2 was not accelerated. This improvement from accelerating non-parallel kernels further increases with the number of cores.

2.1.1 Analysis

Amdahl's law [10] provides a simplified model to predict the performance impact of non-parallel kernels. It assumes that as the number of processors increase, the time taken to execute the parallel portion reduces linearly but the time taken to execute the non-parallel portion remains unchanged. Let us assume that $Speedup_N$ is the speedup achieved by N processors over a single processor and (α) is the fraction of the application which is parallelized (and $1 - \alpha$ is the fraction of the application which is not parallelized). We can define $Speedup_N$ using Equation 2.4.

$$Speedup_N = \frac{1}{\frac{\alpha}{N} + (1 - \alpha)} \tag{2.1}$$

Equation 2.4 shows the speedup as N approaches infinity. This is the maximum possible speedup achievable by a program.

$$\lim_{N \to \infty} Speedup_N = \frac{1}{1 - \alpha} \tag{2.2}$$

Note that $Speedup_N$ becomes a function of *only* the length of the non-parallel portion $(1-\alpha)$. Thus, non-parallel kernels limit the achievable speedup.

If we accelerate the execution of the non-parallel part by a factor S, the speedup will be:

$$Speedup_N = \frac{1}{\frac{\alpha}{N} + \frac{1-\alpha}{S}} \tag{2.3}$$

When N approaches infinity, the speedup becomes:

$$\lim_{N\to\infty} Speedup_N = \frac{S}{(1-\alpha)} \tag{2.4}$$

Thus, accelerating the non-parallel kernel can increase performance and the achievable speedup.

2.2 Critical Sections

Accesses to shared data are encapsulated inside critical sections. Only one thread can execute a particular critical section at any given time. Critical sections are different from Amdahl's serial bottleneck: during the execution of a critical section, other threads that do not need to execute the same critical section can make progress. In contrast, no other thread exists in Amdahl's serial bottleneck.

Figure 2.2(a) shows the code for a multi-threaded kernel which solves the 15-puzzle problem [109]. In this kernel, each thread dequeues a work quantum from the priority queue (PQ) and attempts to solve it. If the thread cannot solve the problem, it divides the problem into sub-problems and inserts them into the priority queue. While this example is from the workload `puzzle`, this is a very common parallel implementation of many branch-and-bound algorithms [60]. The kernel consists of three parts. The initial part A and the final part E are the non-parallel Amdahl's serial bottleneck since only one thread exists in those sections. Part B is the parallel part, executed by multiple threads. It consists of code that is both inside

the critical section (C1 and C2, both protected by lock X) and outside the critical section (D1 and D2).

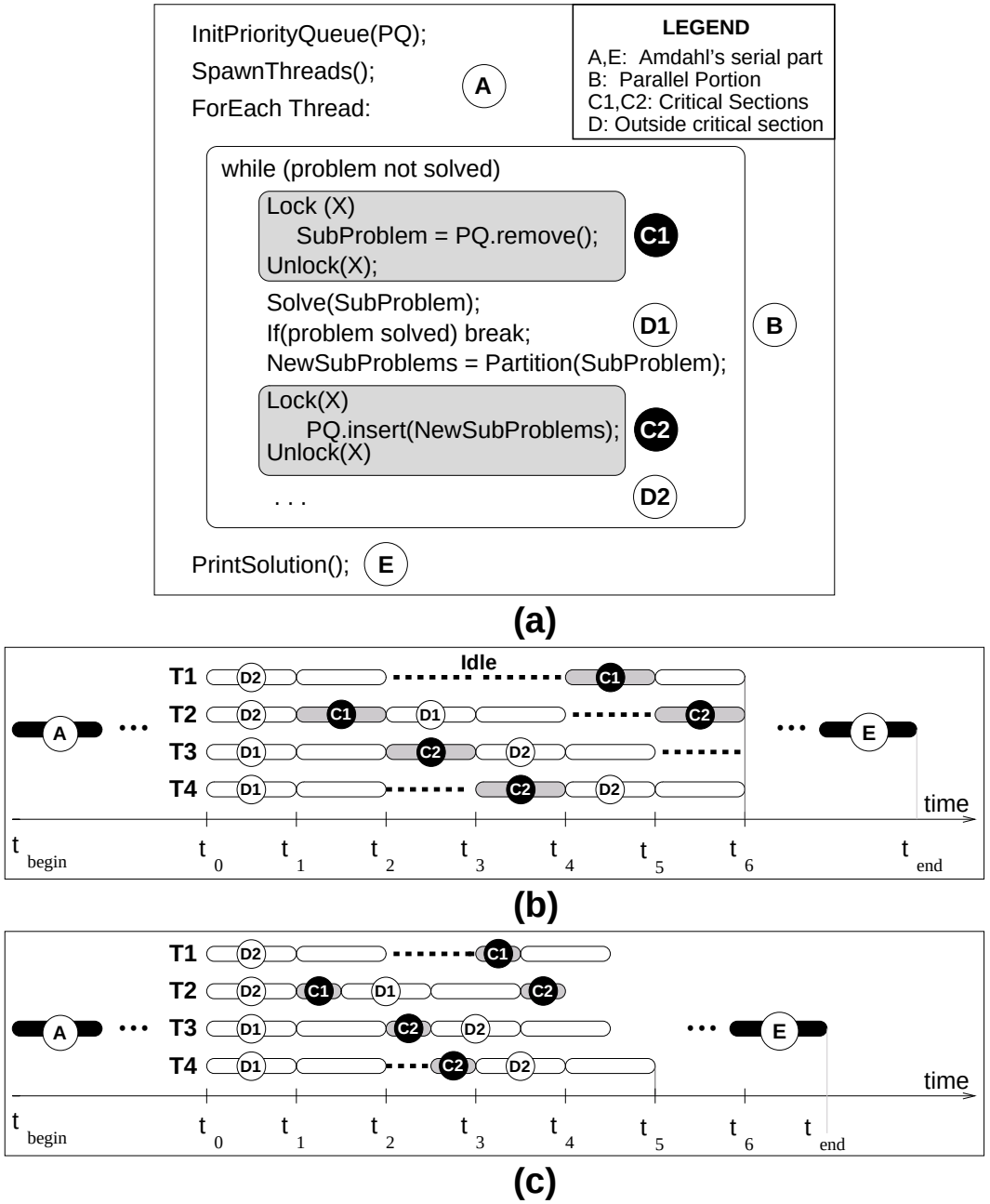


Figure 2.2: Serial part, parallel part, and critical section in 15-puzzle (a) Code example (b) Execution timeline on the baseline CMP (c) Execution timeline with accelerated critical sections.

Figure 2.2(b) shows the execution timeline of the kernel shown in Fig-

ure 2.2(a) on a 4-core CMP. After the serial part A, four threads (T1, T2, T3, and T4) are spawned, one on each core. Once part B is complete, the serial part E is executed on a single core. We analyze the serialization caused by the critical section in steady state of part B. Between time t_0 and t_1, all threads execute in parallel. At time t_1, T2 starts executing the critical section while T1, T3, and T4 continue to execute code independent of the critical section. At time t_2, T2 finishes the critical section and three threads (T1, T3, and T4) contend for the critical section – suppose T3 wins and enters the critical section. Between time t_2 and t_3, T3 executes the critical section while T1 and T4 remain idle, waiting for T3 to exit the critical section. Between time t_3 and t_4, T4 executes the critical section while T1 continues to wait. T1 finally gets to execute the critical section between time t_4 and t_5.

This example shows that the time taken to execute a critical section significantly affects not only the thread that executes it but also the threads that are waiting to enter the critical section. For example, between t_2 and t_3 there are two threads (T1 and T4) waiting for T3 to exit the critical section, without performing any useful work. Therefore, accelerating the execution of the critical section not only improves the performance of T3 but also reduces the useless waiting time of T1 and T4. Figure 2.2(c) shows the execution of the same kernel assuming that critical sections take half as long to execute. Halving the time taken to execute critical sections reduces thread serialization which significantly reduces the time spent in the parallel portion. Thus, accelerating critical sections can provide significant performance improvement.

On average, the critical section shown in Figure 2.2(a) executes 1.5K instructions. When inserting a node into the priority queue, the critical section accesses multiple nodes of the priority queue (implemented as a heap) to find a suitable place for insertion. Due to its lengthy execution, this critical section incurs high contention. Our evaluation shows that when the workload is executed with 8 threads, on average 4 threads wait for this critical section. The average number of waiting threads increases to 16 when the workload is executed with 32 threads. In

contrast, when this critical section is accelerated, the average number of waiting threads reduces to 2 and 3, for 8 and 32-threaded execution respectively.

Note that critical sections can be shortened if programmers lock the shared data at a finer-granularity, e.g., using a different lock for every node in the data structure. We show in Section 5.4.1.2 on page 52 that as the number of cores increases, even fine-grain critical sections begin to incur contention, thereby reducing performance.

2.2.1 Analysis

The effect of critical sections on overall performance can also be demonstrated using simple analytic models. We broadly classify critical sections in two categories: *update* and *reduction* critical sections. Update critical sections protect shared data which is continuously read and modified by multiple threads during the execution of a kernel. In contrast, reduction critical sections protect data which is modified by the threads only at the end of the execution of a kernel.

Update Critical Sections:

Update critical sections occur in the midst of the parallel kernels. They protect shared data which multiple threads try to read-modify-write *during* the kernel's execution, instead of waiting till the end of the kernel's execution. Their execution can be overlapped with the execution of non-critical-section code. For example, critical sections C1 and C2 from the workload `puzzle` (shown in Figure 2.2(a)) are update critical sections because they are executed every iteration of the loop and their execution can be overlapped with the execution of non-critical section code D1.

For simplicity, lets assume a kernel which has only one critical section. Each iteration of the loop spends one unit of time inside the critical section and three units of time outside the critical section. Figure 2.3 demonstrates the execution timeline of this critical-section-intensive application. When a single thread executes, only 25% of execution time is spent executing the critical section. If the same loop is

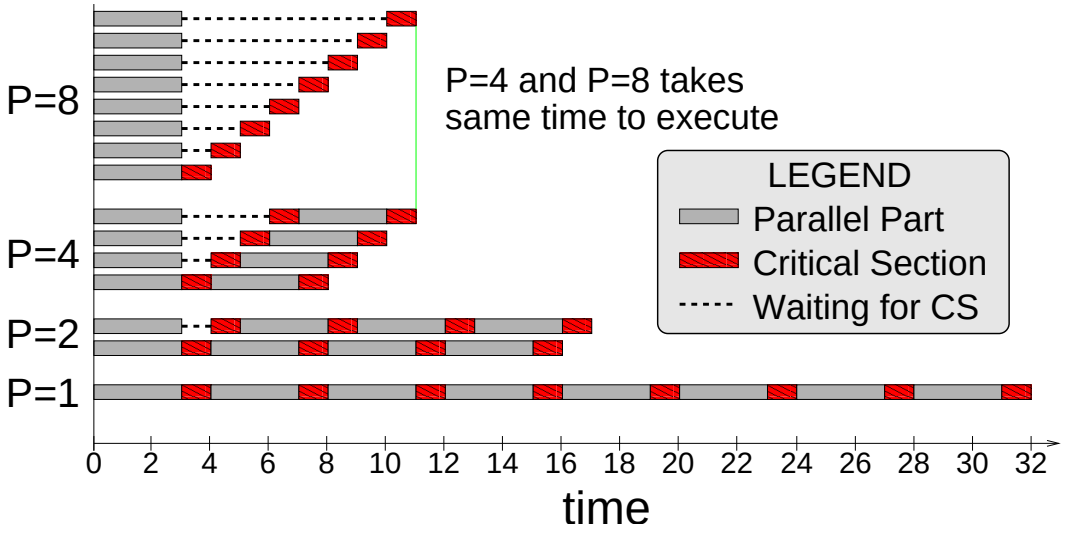


Figure 2.3: Example for analyzing update critical section limited systems

split across two threads, the execution time reduces by 2x. Similarly, increasing the number of threads to four further reduces execution time. As the critical section is always busy, the system becomes critical-section-limited and further increasing the number of threads from four to eight does not reduce the execution time.

Note that the time decreases linearly with the number of threads until the loop becomes critical-section-limited, after which the execution time does not decrease. Let T_{CS} be the time spent in the critical section and T_{NoCS} be the time to execute the non-critical-section part of the program. Let, (T_P) be the time to execute the critical sections and the parallel part of the program when there are P threads. Assuming that all loop iterations run both the critical section and the non-critical-section code, Equation 2.5 shows the execution time, T_p, of a loop with N iterations using P processors.

$$T_p = N \times MAX\left(\frac{T_{NoCS} + T_{CS}}{P}, T_{CS}\right) \tag{2.5}$$

From the above equation, we conclude that a workload becomes critical-section-limited when:

$T_{CS} \geq \frac{T_{NoCS}+T_{CS}}{P}$

We compute (P_{CS}), i.e., the number of threads required to saturate the execution time, by solving the above inequality for P:

$$P_{CS} \geq \frac{T_{NoCS} + T_{CS}}{T_{CS}} \tag{2.6}$$

Thus, when the number of threads is greater than or equal to P_{CS}, critical sections form a critical path through the program, dominating the overall execution time, and limiting scalability. Accelerating the execution of critical sections can reclaim the performance lost due to critical sections.

Generalizing to multiple independent critical sections: To reduce the contention for critical sections, many applications use different locks to protect disjoint data. Since these critical sections are protecting disjoint data, they can execute concurrently, thereby increasing throughput. For example, consider a loop with two independent critical sections CS1 and CS2. Now assume that CS1 executes 25% of the execution time and CS2 executes 12.5% of the execution time. The remaining 62.5% of time is spent executing the parallel portion.

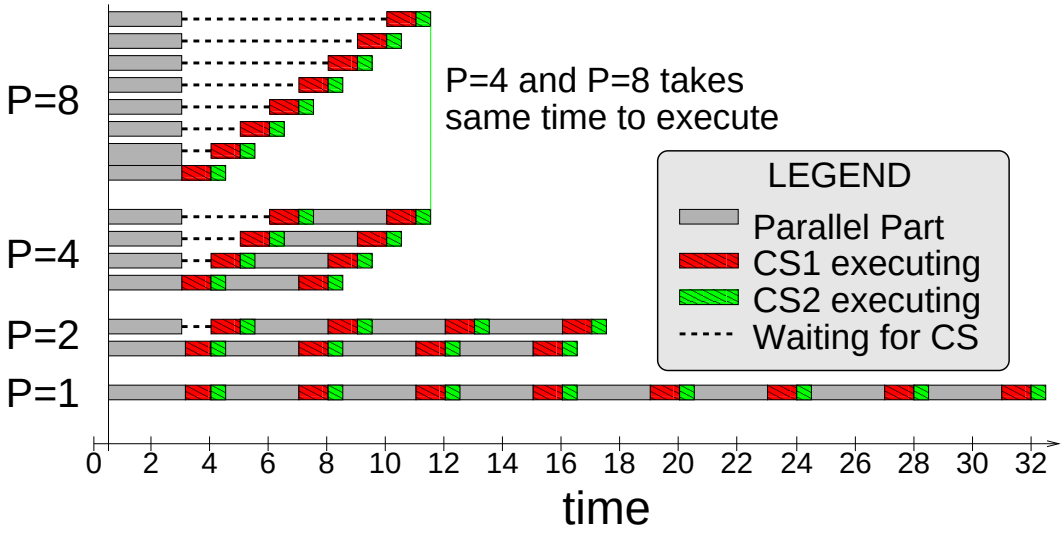


Figure 2.4: Example for analyzing multiple update critical sections

Figure 2.4 demonstrates the execution timeline of this kernel at 1, 2, 4, and 8 threads. Note that the execution time decreases linearly with the number of threads

until the loop becomes limited by CS1, after which the execution time ceases to reduce. Further note that there is no contention for CS2 even with eight cores because once the execution time stops reducing (due to the limitations caused by CS1), the rate at which CS2 is called does not increase with more threads. The reason CS1, not CS2, is the critical path is because CS1 is longer than CS2. Thus, in workloads with many critical sections, the longest critical section, which always has the highest contention, is the performance limiter.

We analyze the impact of multiple critical sections on overall execution time using an analytic model. Let $T_{CS_{all}}$ be the sum of time spent in all critical sections, $T_{CS_{longest}}$ be the time spent in the longest critical section, and T_{NoCS} be the time to execute the parallel part of the program. Let, (T_P) be the time to execute the critical sections and the parallel part of the program when there are P threads. Equation 2.7 shows the execution time, T_p, of a loop with N iterations using P processors.

$$T_p = N \times MAX\left(\frac{T_{NoCS} + T_{CS_{all}}}{p}, T_{CS_{longest}}\right) \tag{2.7}$$

Thus, once the program becomes critical-section-limited, a mechanism to speedup the longest critical section by a factor S will provide an overall speedup of S. Note that sometimes accelerating the longest critical sections makes it faster than the second longest critical section. This makes the previously-second-longest critical section the longest critical section, making it the critical path. Therefore, acceleration is most effective if it done carefully to balance the execution rates of critical sections.

Reduction Critical Sections:

Reduction critical sections occur at the end of kernels and are used to combine the intermediate results computed by individual threads. The key difference between update and reduction critical sections is that, unlike update critical sections, reduction critical sections occur at the *end* of a kernel and their execution *cannot* be overlapped with the execution of the non-critical-section code. Since

every thread executes the critical section, the total time spent in executing the critical sections increases with the number of threads. Furthermore, as the number of threads increase, the fraction of execution time spent in the parallelized portion of the code reduces. Thus, as the number of threads increase, the total time spent in the critical sections increases and the total time spent outside critical sections decreases. Consequently, critical sections begin to dominate the execution time and the overall execution time starts to increase.

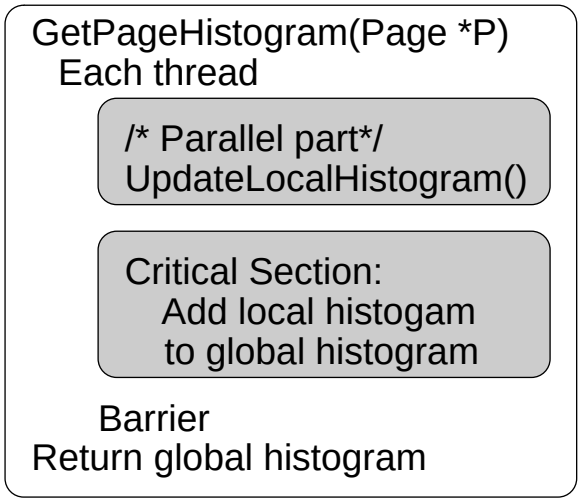


Figure 2.5: A function from `PageMine` that counts the occurrence of each ASCII character on a page of text

We first show a simple example where the time inside the critical sections increases linearly with the number of threads. Let us consider an example kernel. Figure 2.5 shows a function from `PageMine`[2] that counts the number of times each ASCII character occurs on a page of text. This function divides the work across T threads, each of which gathers the histograms for its portion of the page ($PageSize/T$) and adds it to the global histogram. Updates to the local histogram

[2]The code for `PageMine` is derived from the data mining benchmark `rsearchk` [70]. This kernel generates a histogram, which is used as a signature to find a page similar to a query page. This kernel is called iteratively until the distance between the signatures of the query page and a page in the document is less than the threshold. In our experiments, we assume a page-size of 5280 characters (66 lines of 80 characters each) and the histograms consists of 128 integers, one for each ASCII character.

can execute in parallel without requiring data-synchronization. On the other hand, updates to the global histogram, which is a shared data-structure, are guarded by a critical section. Therefore, one and only one thread can update the global histogram at a given time. As the number of threads increase, the fraction of execution time spent in gathering local histograms decreases because each thread has to process a smaller fraction of the page. Whereas, the number of updates to the global histogram increases, which increases the total time spent in updating the global histogram.

Figure 2.6 shows the execution of a program which spends 20% of its execution time in the critical section and the remaining 80% in the parallel part. The overall execution time with one thread is 10 units.

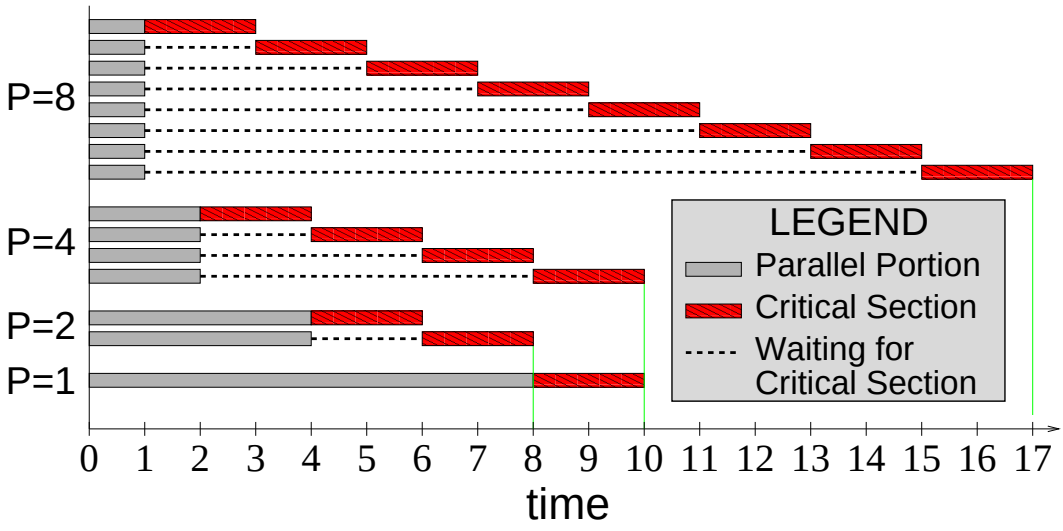


Figure 2.6: Example for analyzing impact of critical sections

When the program is executed with two threads, the time taken to execute the parallel part is reduced to four units while the total time to execute the critical section increases from two to four units. Therefore, the total execution time reduces from 10 units to 8 units. However, overall execution time reduces with additional threads only when the benefit from reduction in the parallel part is more than the increase in the critical section. For example, increasing the number of threads to

four reduces the time for the parallel part from four to two units but increases the time for the critical section from four to eight units. Therefore, increasing the number of threads from two to four increases the overall execution time from 8 units to 10 units. Similarly, increasing the number of threads to eight further increases the overall execution time to 17 units.

We can analyze the impact of critical sections on overall execution time using an analytical model. Let T_{CS} be the time spent in the critical section and T_{NoCS} be the time to execute the *parallel part* of the program. Let, (T_P) be the time to execute the critical sections and the parallel part of the program when there are P threads. Then, T_P can be computed as:

$$T_P = \frac{T_{NoCS}}{P} + P \cdot T_{CS} \tag{2.8}$$

The number of threads (P_{CS}) required to minimize the execution time can be obtained by differentiating Equation 2.8 with respect to P and equating it to zero.

$$\frac{d}{dP} T_P = -\frac{T_{NoCS}}{P^2} + T_{CS} \tag{2.9}$$

$$P_{CS} = \sqrt{\frac{T_{NoCS}}{T_{CS}}} \tag{2.10}$$

Equation 2.10 shows that (P_{CS}) increases only as the *square-root* of the ratio of time outside the critical section to the time inside the critical section. Therefore, even if the critical section is small, the system can become critical section limited with just a few threads. For example, if the critical section accounts for only 1% of the overall execution time, the system becomes critical section limited with just 10 threads. Therefore, reducing T_{CS} by accelerating critical sections will significantly reduce overall execution time and increase scalability.

Note that the time inside the reduction critical sections does not always increase *linearly* with the number of threads, which was the case in our previous example. Sometimes, the programmers are able to partially parallelize reduction by

splitting it into steps. For example, when the kernel from `PageMine` is running on four cores, the global histogram can be computed in two steps (steps I and II). In step I, thread 0 adds the local histograms computed by threads 0 and 1 in the parallel program portion, and thread 2 adds the local histograms computed by threads 2 and 3 in the parallel program portion. In step II, thread 0 adds the temporary histograms computed by threads 0 and 2 in step I, thus computing the final global histogram. Notice that this does not eliminate the critical section but only shortens it: step II is still a critical section that only one thread can execute at a given time. As shown previously, when the number of cores increases, even a short critical section can begin to limit overall performance.

2.3 Limiter stage in a pipeline

Pipeline parallelism is a popular software approach to split the work in a loop among threads. In pipeline parallelism, the programmer/compiler splits each iteration of a loop into multiple work-quanta where each work-quantum executes in a different pipeline stage. Recent research has shown that pipeline parallelism is applicable to many different types of workloads, e.g, streaming [100], recognition-mining-synthesis workloads [15], compression/decompression [47], etc. In pipeline parallel workloads, each stage is allocated one or more *worker* threads and an *in-queue* which stores the work quanta to be processed by the stage. A worker thread pops a work quanta from the in-queue of the stage it is allocated to, processes the work, and pushes the work on the in-queue of the next stage.

Figure 2.7(a) shows a loop which has N iterations. Each iteration is split into 3 stages: A, B, and C. Figure 2.7(b) shows a flow chart of the loop. The three stages of the ith iteration are labeled Ai, Bi, and Ci. Figure 2.7(c) shows how this loop gets executed sequentially on a single processor. The time t_0 is the start of iteration 0 of the loop. The time t_3 is the end of iteration 0, and the start of iteration 1, and so on. Figure 2.7(d) shows how this program gets executed using pipeline parallelism on three processors. Each core works on a separate part of the

iteration (P0 executes stage A, P1 executes stage B, and P2 executes stage C), and the iteration gets completed as it traverses from left to right, and top to bottom. Note that we show for simplicity that each stage has one core but it is possible to allocate multiple cores per stage or share a core among stages.

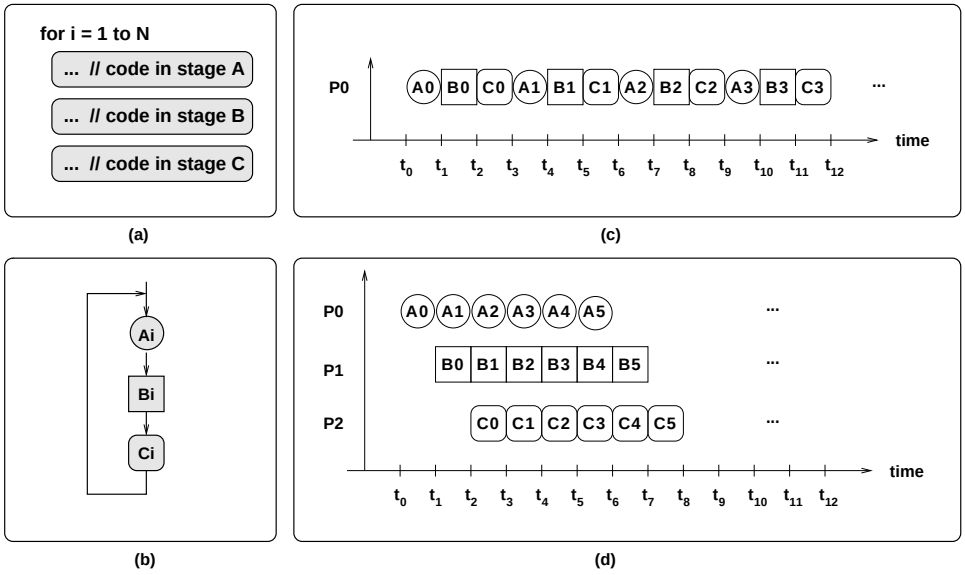


Figure 2.7: (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.

Consider a kernel from the workload `compress`. This kernel compresses the data in an input file and writes it to an output file. Each iteration of this kernel reads a block from the input file, compresses the block, and writes the compressed block to the output file. Figure 2.8 shows the pipeline of this kernel. Stage S1 allocates the space to save the uncompressed and the compressed block. S2 reads the input and S3 compresses the block. When multiple threads/cores are allocated to each stage, iterations in a pipeline can get out of order. Since blocks must be written to the file in-order, S4 re-orders the quanta and writes them to the output file. S5 deallocates the buffers allocated by S1. This kernel can execute on a 5-core CMP such that each stage executes on one core. At any point in time, cores will be busy executing different portions of five different iterations, thereby increasing

performance. In reality, when the pipeline executes, cores executing different stages of a pipeline often wait on other cores and remain idle. This limits concurrency and reduces performance.

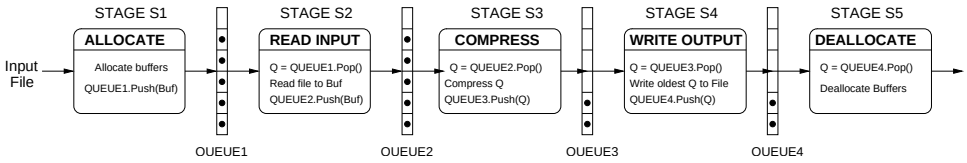


Figure 2.8: File compression algorithm executed using pipeline parallelism

2.3.1 Analysis

We define *throughput* of a pipeline stage as the number of iterations processed in a given amount of time. Thus, the throughput τ_i of a pipeline stage i can be defined as:

$$\tau_i = \frac{Num\ \ Iterations\ \ Processed}{Time} \tag{2.11}$$

The overall throughput, τ, of the whole pipeline is limited by the throughput of the slowest stage of the pipeline. Therefore:

$$\tau = MIN(\tau_0, \tau_1, \tau_2, ...) = \tau_{min} \tag{2.12}$$

Thus, for example, if the slowest stage of the pipeline for compression shown in Figure 2.8 is S3 (compress), then performance will be solely determined by the throughput of S3. Let $LIMITER$ be the stage with the lowest throughput. Then stages other than the LIMITER will wait on the LIMITER stage and their cores will be under-utilized.

A common method used to increase the throughput of the LIMITER stage is to increase the number of cores allocated to it. However, more cores help if and only if the LIMITER stage scales with the number of cores (increasing the

number of allocated cores increases its throughput). Unfortunately, the throughput of a stage does not always increase with the number of cores due to contention for shared data and resources (i.e. data-synchronization, cache-coherence). When a stage does not scale, allocating more cores to the stage either does not improve its throughput or can in some scenarios reduce its throughput [95]. Thus, once the pipeline becomes limited by a non-scalable LIMITER, performance saturates. The only way to further improve its performance is by accelerating the LIMITER stage.

Accelerating the LIMITER stage can increase the throughput of the LIMITER. This can either change the LIMITER or increase the overall throughput of the pipeline as much as the speedup from acceleration.

Conclusions: Non-parallel kernels, critical sections, and limiter pipeline stages can form the critical path through the program as the number of cores increases. When performance is limited by a serial portion, adding more cores does not improve overall performance. This creates the need to reduce the execution time inside the serial portion.

Chapter 3

Asymmetric Chip Multiprocessor

3.1 Current CMP Architectures

Industry is using two common approaches when designing chip multiprocessors:

Tile-Large Approach: The most popular approach is to tile a few large cores(Figure 5.1(a)). This provides high single-thread performance but low parallel throughput. Examples of this approach are AMD Opteron [8], Intel Core2Duo [79], and IBM Power5 [52]. They primarily target multiple single-threaded programs.

Tile-Small Approach: Sun Microsystem's Niagara [54] and Intel's Larrabee [86] processor has taken a different approach(Figure 5.1(b)). Since each small core is more area-efficient than the large core, it provides high parallel throughput but low single thread performance. These chips are designed for workloads with massive amount of parallelism, e.g., server and graphics workloads.

Neither of the two approaches will be best suited for future applications. Since the motivation exists, programmers are likely to parallelize some portions of their programs which makes the Tile-Large approach low-performance. Taking advantage of Tile-Small requires the application to be completely parallel. It is unreasonable to expect that common programmers will be able write massively

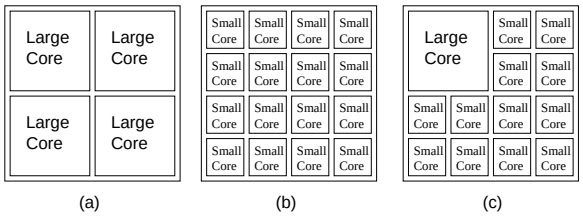


Figure 3.1: CMP Architecture approaches. (a) Tile-Large (b) Tile-Small (c) Asymmetric

parallel programs like the sever and graphics programmers. There are three reasons: lack of domain specific knowledge, variations in target platforms, and time/financial constratins. We demonstrate this with some examples and facts.

- We describe a real-world example of the MySQL database. The standard MySQL database engine is known to have poor scalability (its performance saturates between 10-20 threads) [5, 6]. Yet, it is being used in large scale systems such as the social networking website, Facebook (wwww.facebook.con) [87]. The reason is that Facebook has specialized MySQL to their specific domain. They call is Facebook SQL (FQL) [1]. FQL is different from MySQL in two ways. First, unlike MySQL, FQL no longer guarantees that updates to the shared data (about users) are visible to all users immediately. For example, when a user changes his/her profile on Facebook, it is not guaranteed that the change will be visible to all Facebook users instantaneously. Second, FQL does not assign a unique ID to every entry in their database, a feature supported by the baseline MySQL. Both these changes eliminate major critical sections. Facebook further replicates their databases in data centers all over the world and use caching extensively, which incurs a large cost. Thus, MySQL's scaling was only made possible by programmer effort and substantial financial support.

- Recall the kernel K2 in Figure 2.1(b). This kernel does not exhibit any parallelism as-is. Now suppose that we know that the kernel will be used to smooth the pixels in a video frame. With the domain-specific knowledge that inaccuracies in pixel values are tolerable, we can re-factor the kernel where each pixel can be computed as the weighted mean of the old values of the last k pixels. This will break the inter-iteration dependency, thereby making the kernel parallelizable. Such optimizations are only made possible by domain-specific knowledge.

- Experts in graphics programming [7] shows that it is often necessary to employ hardware-specific and input-set-specific optimizations to extract paral-

lelism out of programs. This is because factors like cache sizes, communication latencies, lock handling operations, all have high impact on parallel program performance. General purpose programmers are often unfamiliar with their target hardware platforms and input sets. For example, a video player can be run on many different systems with a large variety of videos as the input. Thus, general purpose programmers are forced to take conservative approaches and it is difficult for them to optimize the programs in the same way other the game programmers can.

- General purpose programmers are often under time and resource constraints. Discussions with IBM Blue Gene software team reveals that just optimizing the performance of a functionally correct Blue Gene program can take several man years. Unavailability of time and resources also reduces the ability of programmers to fully tune their code.

- Despite the challenges, general-purpose programmers are being burdened with the task of identifying parallelism. Further expecting programmers to fully optimize their code often leads to infeasible results. For example, Mirano et al. [66] show how attempts to shorten critical sections have led to data-race bugs in common programs such as Mozilla Firefox and several Microsoft products.

We conclude that future workloads will require CMP architectures which can shorten serial portions, without requiring programmer effort. Such mechanisms can improve performance of existing programs, provide higher performance for sub-optimized code, and make parallel programming more available to the average programmer.

3.2 Our Solution

For high-performance execution of multithreaded programs, we propose the *Asymmetric Chip Multiprocessor (ACMP)* paradigm. ACMP paradigm offers two

types of cores: fast cores for speedy execution of the serial portions and an array of power-efficient, small cores for high-throughput on the parallel portions. Note that the central idea behind the ACMP paradigm is that the future CMPs shall provide asymmetric performance characteristics. This performance asymmetry can be created in multiple ways, e.g., by making a core more agressive or by increasing a core's frequency [24]. This thesis develops an example implementation where the faster cores are implemented by making them larger and more-powerful compared to the other small cores. To make the design pragmatic, we chose cores similar to current and past Intel cores.

3.3 ACMP Architecture

The Asymmetric Chip Multi-processor (ACMP) is a shared memory and homogeneous ISA CMP. It provides one or a few large cores and many small cores.

3.3.1 ISA

All cores support all instructions and software is unable to distinguished between a large core and a small core functionality-wise. The ISA supports one new instruction which can be used by software to query the type (whether small or large) of the core it is running on. Similar instructions, which provide information about the underlying microarchitecture, already exist in modern ISAs, e.g., CPUID instruction in x86 [46]. We envision that core-type can be an additional field in the output of the CPUID instruction.

3.3.2 Interconnect

All cores, both small and large, share the same interconnect for high bandwidth, low-latency communication. Our example implementation uses a bi-directional ring interconnect with seperate control and data lines. The width of the interconnect is 512 bits, which is exactly one cache line.

3.3.3 Caches

The cache hierarchy is the same as any symmetric CMP and can include multiple levels of private and/or shared caches. In our example, each core has its private L1 and L2 caches. To further increase the large core's performance, the large core is given a larger cache than the L2 cache of the smaller cores (1MB for the large core vs. 256KB each for the small cores). All cores share an L3 cache.

3.3.4 Cache Coherence

ACMP supports shared memory and any private caches must be kept coherent using a hardware cache coherence protocol. In our implementation, the private L1s are write-through and L2s are kept coherent using a coherence directory-based MOESI protocol.

3.3.5 Large core

The large core's purpose is to provide a high single-thread IPC. It has the characteristics of an aggressive state-of-the-art core. It has a wide issue width, a deep pipeline, out of order execution, several ALUs, a powerful branch predictor, an aggressive prefetcher, a powerful indirect-branch predictor, etc. In our example implementation, we use a 4-wide out-of-order machine similar to each core in the Intel Core2 Quad.

3.3.6 Small core

The small core's purpose is to run code power-efficiently. It has a shallow in-order pipeline, a small branch predictor, a simpler prefetcher, and it does not have a dedicated indirect branch predictor. In our implementation, each small core is similar to a Pentium core [44]. It is 2-wide with a 5-stage pipeline with a 4KB GSHARE branch predictor.

3.4 Design Trade-offs in ACMP

There are two key trade-offs which impact the design and use of the ACMP.

1. Area vs. Performance: The large core is less area-efficient: it takes the same area as four small cores but provides a lower throughput than four small cores. Thus, in serial regions of code, while the large core of the ACMP executes faster, it consumes more power compared to a small core. This increased energy is tolerable if the increase in performance is substantial.

2. Hardware Cost vs. Software Cost: The ACMP requires two different types of cores to be integrated on the same chip. This may increase design costs. However, industry is already building chips with different types of execution units e.g. the IBM Cell Processor [40] and numerous system-on-chip designs. By integrating the large core, the ACMP provides higher performance in the serial bottlenecks which makes overall performance more tolerant to the length of the serial portion. Consequently, programmer effort can be saved by parallelizing only the easier to parallelize kernels. Thus, the hardware cost can be amorized.

The next four chapters describes mechanisms to identify and accelerate non-parallel kernels, critical sections, and limiter stages using the ACMP..

Chapter 4

ACMP for Accelerating Non-Parallel Kernels

Non-parallel kernels execute as single threads. Such regions of code often occur at the beginning of the program when no threads have been spawned or at the end when all threads have finished their work. In some cases, such regions are also interleaved with parallel kernels. We propose a mechanism to accelerate the execution of non-parallel kernels using the ACMP. We call it *Accelerated Non-Parallel Kernels (ANP).*

4.1 Architecture

Accelerating Non-Parallel kernels (ANP) requires a simple change in the thread scheduler: any time there is only one active thread, it must execute on the large core. The operating system uses the newly added CPUID instruction to identify the large core and stores this information. Figure 4.1 shows the thread scheduling algorithm. The list of active threads is initially empty. The master thread, which is the first entry in the list, is spawned on the large core. The master thread spawns additional worker threads and either waits for the worker threads or executes a portion of the work itself. The worker threads execute the parallel region. As the worker threads finish their work and exit, they remove themselves from the list of active threads. When the second last thread exits and the parallel region finishes, the single remaining thread, if not already on the large core, is migrated to the large core.

```
On creation of thread t:
  Add thread t to thread list
  If thread t is the only thread
      Spawn it on the large core
  Else
      Spawn it on a free small core
```

```
On exit of thread t:
  Delete thread t from thread list
  If thread list has only only one thread
        and that thread is not on the large core
      Move it to the large core
```

Figure 4.1: Scheduling algorithm for accelerating non-parallel kernels.

4.2 Performance Trade-offs in ANP

Accelerating the non-parallel kernels using the ACMP involves three performance trade-offs:

1. Peak Parallel Throughput vs. Serial Thread Performance: ACMP replaces a few small cores with one large core. This reduces peak parallel throughput. However, this reduction in throughput is compensated by the accelerated execution of the non-parallel region. Figure 4.2 shows how the ACMP, Tile-Small, and Tile-Large compare as degree of parallelism increases. We assume a large core takes the area of four small cores. Note that Tile-Small outperforms the ACMP at higher parallelism while the ACMP outperforms both competing approaches for a wide range of parallelism. Moreover, the fraction of parallel throughput lost due to the large core reduces as the total number of cores increase. For example, replacing four small cores with one large core reduces throughput by 50% in an 8-core CMP but by only 6.25% in a 64-core CMP. Therefore, the ACMP approach becomes more and more feasible as we are able to increase the number of cores on the chip.

Figure 4.2 plots the parallelism required for Tile-Small to outperform ACMP as a function of chip area. The cut-off point moves higher and higher as chip area increases. We conclude that ACMP will become applicable to more and more workloads in the future.

2. Thread migration overhead vs. Accelerated Execution: When a thread

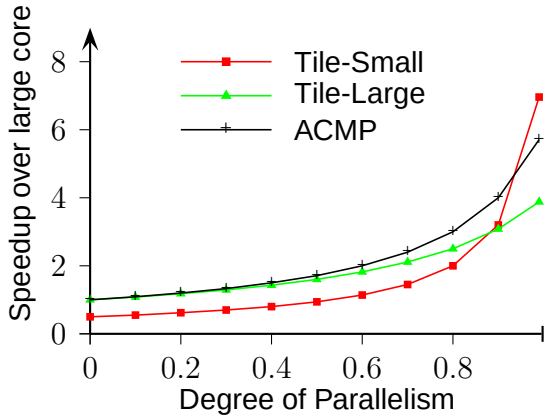


Figure 4.2: Performance vs. Degree of Parallelism at a fixed peak power for baseline CMP architectures

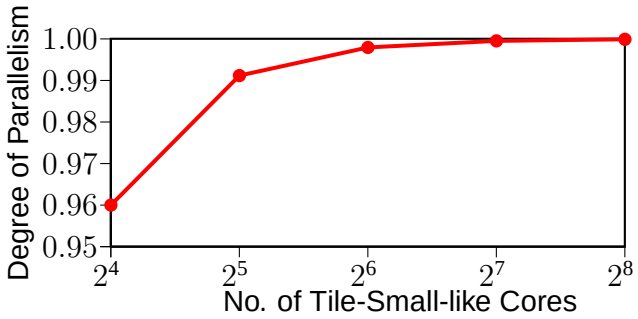


Figure 4.3: Degree of parallelism needed in application for Tile-Small approach to outperform ACMP approach vs. die area.

is migrated from a small core to a large core or vice versa, the ACMP incurs the overhead of sending the register values, the program counter, and the stack pointer. However, such data migrations are rare and only occur when a parallel region begins and ends. This overhead can get amortized due to the accelerated execution of the serial bottleneck. ACMP can reduce performance if either the non-parallel regions are very small or the large core does not accelerate the execution. We find that not to be case across our benchmarks.

3. Cache locality in the ACMP is similar to that of the baseline CMP. The reason is that the data generated in the parallel region is spread evenly across cores. In the baseline CMPs, when one of the regular cores execute the serial portion, it gathers the required data from other cores. In the ACMP, the scenario does not

change. The large core incurs the same number of cache misses to gather the data from the small cores. If a serial portion requires data from the previous serial portion, the ACMP performs exactly the same as the baseline.

4.3 Evaluation Methodology

Table 4.1 shows the configuration of the simulated CMPs, using our in-house cycle-level x86 simulator. The large core occupies the same area as four smaller cores: the smaller cores are modeled after the Intel Pentium processor [44], which requires 3.3 million transistors, and the large core is modeled after the Intel Pentium-M core, which requires 14 million transistors [30]. We evaluate two different CMP architectures: a symmetric CMP (SCMP) consisting of all small cores; and an asymmetric CMP (ACMP) with one large core with 2-way SMT and remaining small cores which accelerates the non-parallel kernels. Unless specified otherwise, all comparisons are done at equal area budget. We specify the area budget in terms of number of small cores.

4.3.1 Workloads

Table 4.2 shows our benchmarks. We divide these workloads into two categories: workloads with coarse-grained locking and workloads with fine-grained locking. All workloads were simulated to completion.

The database workloads `oltp-1`, `oltp-2`, and `sqlite` were compiled with gcc 4.1 using -O3 flag. The number of threads were set by changing the number of clients in the input set. specjbb was compiled using gcj 4.1. The number of threads were set by changing the number of warehouses. All other workloads were compiled with the Intel C Compiler [43] using the `-O3` flag and were simulated to completion. For `is` and `ep`, we use the reference OpenMP implementation with the input sets as shown. We acknowledge that these workloads can be optimized by investing more programmer effort. However, we use them as-is to make a point that ACMP is very effective at improving performance of sub-optimal code.

Table 4.1: Configuration of the simulated machines

Small core	2-wide In-order, 2GHz, 5-stage. L1: 32KB write-through. L2: 256KB write-back, 8-way, 6-cycle access
Large core	4-wide Out-of-order, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, L1: 32KB write-through. L2: 1-MB write-back, 16-way, 8-cycle
Interconnect	64-bit wide bi-directional ring, all queuing delays modeled, ring hop latency of 2 cycles (latency between one cache to the next)
Coherence	MESI, On-chip distributed directory similar to SGI Origin [59], cache-to-cache transfers. # of banks = # of cores, 8K entries/bank. We the delay and contention of all transactions to and from the directory.
L3 Cache	8MB, shared, write-back, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Row buffer hit: 25ns, Row buffer miss: 50ns, Row buffer conflict: 75ns
Memory bus	4:1 cpu/bus ratio, 64-bit wide, split-transaction, pipelined bus
Area-equivalent CMPs where area is equal to N small cores. We vary N from 1 to 32	
SCMP	N small cores, One small core runs serial part, all N cores run parallel part, conventional locking (Max. concurrent threads = N)
ACMP	1 large core and N-4 small cores; large core runs serial part, 2-way SMT on large core and small cores run parallel part, conventional locking (Maximum number of concurrent threads = N-2)

We briefly describe the benchmarks whose source code is not publicly available. `iplookup` is an Internet Protocol (IP) packet routing algorithm [105]. Each thread maintains a private copy of the routing table, each with a separate lock. On a lookup, a thread locks and searches its own routing table. On an update, a thread locks and updates all routing tables. Thus, the updates, although infrequent, cause substantial serialization and disruption of data locality.

`puzzle` solves a 15-Puzzle problem [109] using a branch-and-bound algorithm. There are two shared data structures: a work-list implemented as a priority heap and a memoization table to prevent threads from duplicating computation. Priority in the work-list is based on the Manhattan distance from the final solution. The work-list (heap) is traversed every iteration, which makes the critical sections long and highly contended for.

`webcache` implements a shared software cache used for caching “pages”

Table 4.2: Simulated workloads

Locks	Workload	Description	Source	Input set	# of disjoint critical sections	What is Protected by CS?
Coarse	ep	Random number generator	NAS suite [13]	262144 nums.	3	reduction into global data
	is	Integer sort	NAS suite [13]	n = 64K	1	buffer of keys to sort
	pagemine	Data mining kernel	MineBench [70]	10Kpages	1	global histogram
	puzzle	15-Puzzle game	[109]	3x3	2	work-heap, memoization table
	qsort	Quicksort	OpenMP SCR [27]	20K elem.	1	global work stack
	sqlite	sqlite3 [3] database engine	SysBench [4]	OLTP-simple	5	database tables
	tsp	Traveling salesman prob.	[55]	11 cities	2	termination cond., solution
Fine	iplookup	IP packet routing	[105]	2.5K queries	# of threads	routing tables
	oltp-1	MySQL server [2]	SysBench [4]	OLTP-simple	20	meta data, tables
	oltp-2	MySQL server [2]	SysBench [4]	OLTP-complex	29	meta data, tables
	specjbb	JAVA business benchmark	[90]	5 seconds	39	counters, warehouse data
	webcache	Cooperative web cache	[101]	100K queries	33	replacement policy

of files in a multi-threaded web server. Since, a cache access can modify the contents of the cache and the replacement policy, it is encapsulated in a critical section. One lock is used for every file with at least one page in the cache. Accesses to different files can occur concurrently.

`pagemine` is derived from the data mining benchmark `rsearchk` [70]. Each thread gathers a local histogram for its data set and adds it to the global histogram inside a critical section.

4.4 Evaluation

Recall that accelerating non-parallel kernels using the ACMP makes a trade-off: it provides increased serial performance at the expense of peak parallel throughput. Thus, ACMP improves performance if the benefit obtained by accelerating the non-parallel portion is more than the loss in throughput due to replacing four small cores with a single large core. ACMP's benefit will be higher for the workloads

with longer non-parallel portions. ACMP's cost, the loss of peak parallel throughput, will be most noticeable for workloads which scale well but it will matter less for workloads with poor scalability. Furthermore, the loss in peak parallel throughput will be lower as the area budget of the chip increases. Thus, at small area budgets, ACMP is expected to perform best for non-scalable workloads. At high area budgets, ACMP should be effective for both scalable and non-scalable workloads.

4.4.1 Performance with Number of Threads Set Equal to the Number of Available Thread Contexts

Figure 4.4 shows the execution time of the ACMP normalized to the SCMP, both at an area-budget of 8 cores. The ACMP significantly reduces execution time in workloads which have significant non-parallel kernels. For example, ACMP improves performance of `is` by 2x because `is` spends 84% of its instructions in the non-parallel portion of the program. Recall that the implementation of `is` that we use is the reference implementation, where minimal programmer effort has been invested in optimizing the code. Thus, ACMP is providing a much higher performance at lower programmer effort compared to the SCMP. Similarly, in programs `ep` and `qsort`, ACMP outperforms the SCMP by accelerating the long serial bottleneck. In contrast, ACMP reduces performance for `sqlite`, `tsp`, `iplookup`, `mysql-1`, `mysql-2`, `webcache`, and `specjbb`. These workloads have scalable parallel portions and small serial portions. Most noticeably, ACMP increases the execution time of `iplookup` by almost 2x compared to SCMP. This is because in `iplookup` the serial part is practically non-existent (only 0.1% of all dynamic instructions) and, as we show in Section 5.4.1.2, the parallel code is very scalable as it uses fine-grain locking. On average, ACMP and SCMP perform similar when area budget is 8 cores.

Systems area-equivalent to 16 and 32 small cores: As the chip area increases, ACMP's cost reduces since the fractional reduction in peak parallel throughput due to the big core reduces. Furthermore, ACMP's benefit increases since the parallel portion gets faster as it is split across a larger number number of

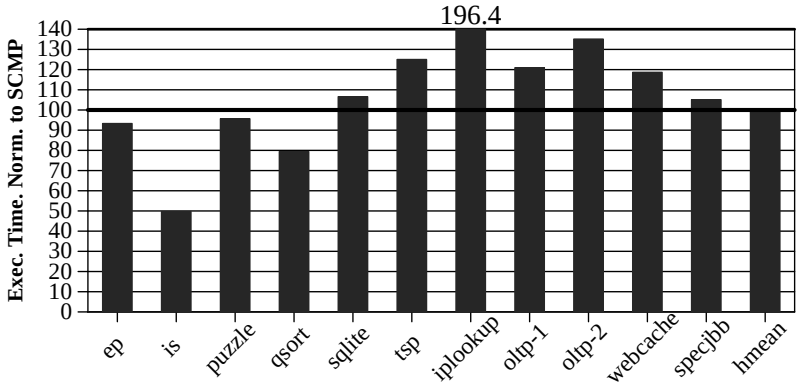


Figure 4.4: Normalized execution time of ACMP at an area budget of 8 cores.

cores, thereby making the serial portion a larger fraction of the execution time.

Figure 4.5 shows the execution time of ACMP normalized to an equal-area SCMP, both with an area budget of 16. ACMP begins to outperform SCMP in some of the workloads where it performed worse than the SCMP when the area budget was 8. For example, ACMP increased the execution time of `sqlite` at an area budget 8 but reduces its execution time at an area budget of 16. `tsp` is the only workload where ACMP's benefit reduces when the area budget increases. This is not because of a property of the ACMP or SCMP but because `tsp` takes a different path through the algorithm when more threads are available. In all other workloads, having a larger area budget helps the ACMP.

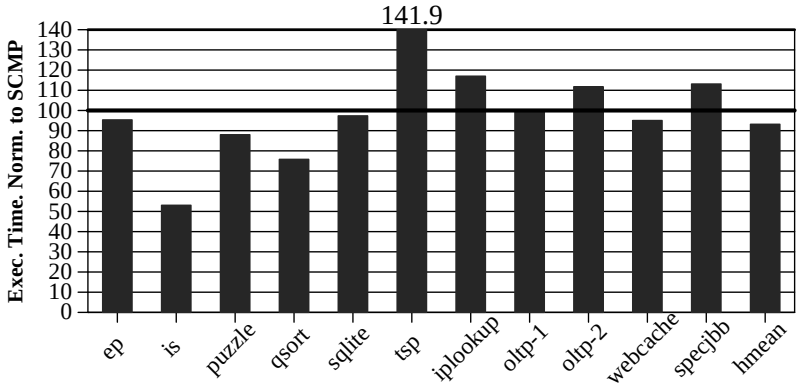


Figure 4.5: Normalized execution time of ACMP at an area budget of 16 cores.

Figure 4.6 shows the execution time of ACMP, normalized to SCMP, when the area budget is 32. Increasing the area budget to 32 further increases the ACMP's

benefit. For all workloads, the ACMP either reduces the execution time or does not impact it by more than 1%. Overall, when area is 32, ACMP reduces execution time by 19% compared to the equal area SCMP.

We conclude that ACMP, when accelerating only the serial portions, can significantly improve performance of multi-threaded workload for today's CMP. Furthermore, its benefit increases with the number of cores.

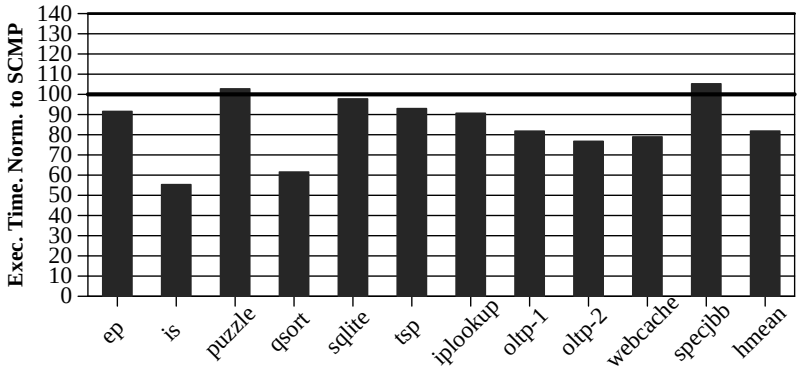


Figure 4.6: Normalized execution time of ACMP at an area budget of 32 cores.

4.4.2 Scalability

Figure 4.7 shows the speedup of SCMP and ACMP over a single small core as the chip area increases. The performance benefit of ACMP increases as chip area increases. Note that the peak performance with the ACMP (marked with a filled dot) is always higher or very similar to that of the SCMP. To explain the results, we split our workloads into three categories:

(1) Workloads which are non-scalable and have long non-parallel kernels, e.g., `is`, `qsort`. ACMP consistently improves their performance as they do not suffer from the loss in throughput due the ACMP but they benefit from the higher serial thread performance provided by the ACMP.

(2) Workloads which are non-scalable but do not have long non-parallel kernels, e.g., `ep`, `pagemine`, `sqlite`, `tsp`, `oltp-1`, `oltp-2`. Such workloads should remain unaffected by the ACMP as they do not suffer due to the reduced

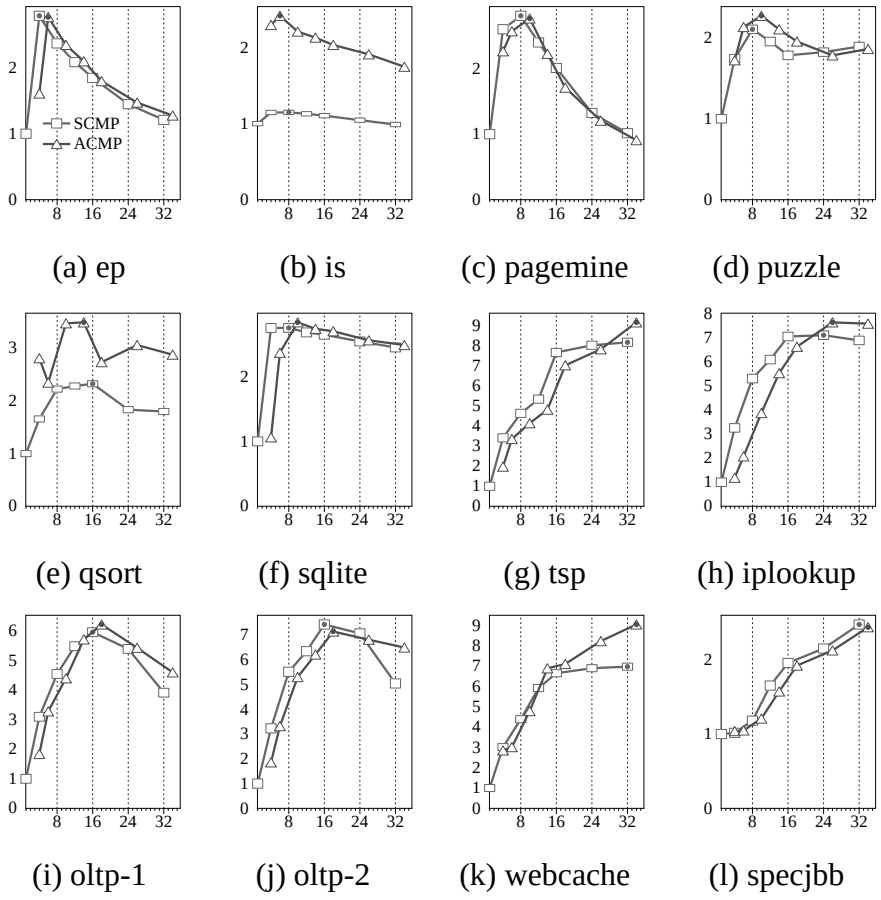


Figure 4.7: Speedup over a single small core. Y-axis is the speedup over a single small core and X-axis is the chip area in terms of small cores.

throughput (since they are non-scalable) and also do not benefit from the large cores due to absence of non-parallel kernels.

(3) Workloads which are scalable with but do not have non-parallel kernels (e.g., `iplookup`, `specjbb`, `webcache`). ACMP reduces their performance as ACMP's benefit of having accelerated non-parallel kernels is unable to outweigh ACMP's cost of having fewer threads. In such workloads, ACMP's benefit will increase as the chip area increases.

In summary, ACMP is able to improve (or not impact) performance of most workloads; and ACMP is expected to improve performance of other workloads in the future when chip area increases.

4.4.3 ACMP with Best Number of Threads

Unless otherwise specified, most systems set the number of threads equal to the number of available thread contexts. This is not always optimal for performance as having more threads than required can degrade performance [95]. Instead, the number of threads must be chosen such that the execution time is minimized. Figure 4.8 shows the execution time of ACMP normalized to the SCMP where for each configuration-workload pair, the number of threads is set to the number of threads required to minimize the execution time for that configuration-workload pair. We choose the best threads for each workload-configuration pair by simulating all possible number of threads and choosing the number of threads at which execution time is minimized. ACMP effectively reduces the execution time on all workloads and on average the ACMP reduces execution time by 17%.

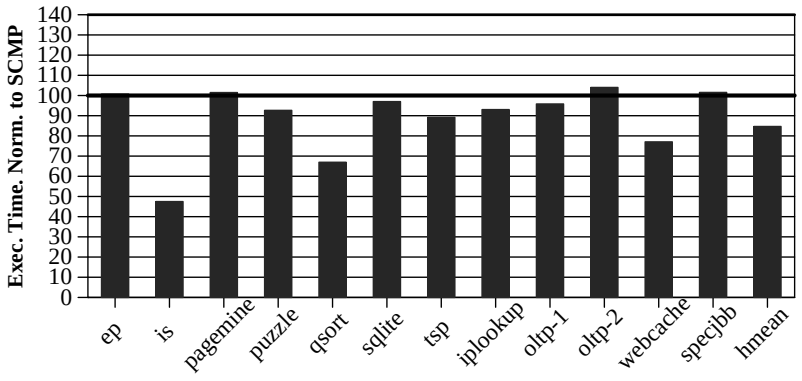


Figure 4.8: Normalized execution time of ACMP at Best Threads.

Chapter 5

ACMP for Accelerating Critical Sections

To overcome the performance bottleneck of critical sections, we propose *Accelerated Critical Sections (ACS)* [97]. In ACS, both the critical sections and the serial part of the program execute on a large core, whereas the remaining parallel parts execute on the small cores. Executing the critical sections on a large core reduces the execution latency of the critical section, thereby improving performance and scalability.

5.1 Architecture

Figure 5.1 shows an example ACS architecture implemented on an ACMP consisting of one large core (P0) and 12 small cores (P1-P12). ACS executes the serial part of the program on the large core P0 and parallel part of the program on the small cores P1-P12. When a small core encounters a critical section, it sends a "critical section execution" request to P0. P0 buffers this request in a hardware structure called the *Critical Section Request Buffer (CSRB)*. When P0 completes the execution of the requested critical section, it sends a "done" signal to the requesting core. To support such accelerated execution of critical sections, ACS requires support from the ISA (i.e., new instructions), from the compiler, and from the on-chip interconnect. We describe these extensions in detail next.

5.1.1 ISA Support

ACS requires two new instructions: *CSCALL* and *CSRET*. CSCALL is similar to a traditional CALL instruction, except it is used to execute critical section code on a remote, large processor. When a small core executes a CSCALL in-

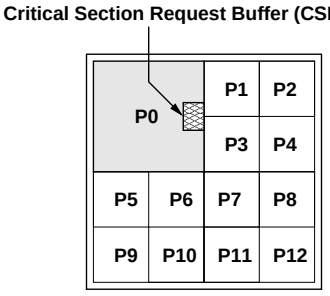


Figure 5.1: ACS on ACMP with 1 large core and 12 small cores

CSCALL LOCK_ADDR, TARGET_PC	CSRET LOCK_ADDR
On small core: STACK_PTR <– SP Send CSCALL Request to large core with Arguments: LOCK_ADDR TARGET_PC, STACK_PTR, CORE_ID Stall until CSDONE signal received *On large core:* Enqueue in CSRB Wait until HEAD ENTRY in CSRB Acquire lock at LOCK_ADDR SP <– STACK_PTR PC <– TARGET_PC	*On large core:* Release lock at LOCK_ADDR Send CSDONE to REQ_CORE *On small core:* Retire CSCALL

Figure 5.2: Format and operation semantics of new ACS instructions

struction, it sends a request for the execution of critical section to P0 and waits until it receives a response. CSRET is similar to a traditional RET instruction, except that it is used to return from a critical section executed on a remote processor. When P0 executes CSRET, it sends a CSDONE signal to the small core so that it can resume execution. Figure 5.2 shows the semantics of CSCALL and CSRET. CSCALL takes two arguments: LOCK_ADDR and TARGET_PC. LOCK_ADDR is the memory address of the lock protecting the critical section and TARGET_PC is the address of the first instruction in the critical section. CSRET takes one argument, LOCK_ADDR corresponding to the CSCALL.

5.1.2 Compiler/Library Support

The CSCALL and CSRET instructions encapsulate a critical section. CSCALL is inserted before the "lock acquire" and CSRET is inserted after the "lock release." The compiler/library inserts these instructions automatically without requiring any modification to the source code. The compiler must also remove any register dependencies between the code inside and outside the critical section. This avoids transferring register values from the small core to the large core and vice versa before and after the execution of the critical section. To do so, the compiler performs *function outlining* [111] for every critical section by encapsulating the critical section in a separate function and ensuring that all input and output parameters of the function are communicated via the stack. Several OpenMP compilers already

do function outlining for critical sections [20, 62, 85]. Therefore, compiler modifications are limited to the insertion of CSCALL and CSRET instructions. Figure 5.3 shows the code of a critical section executed on the baseline (a) and the modified code executed on ACS (b).

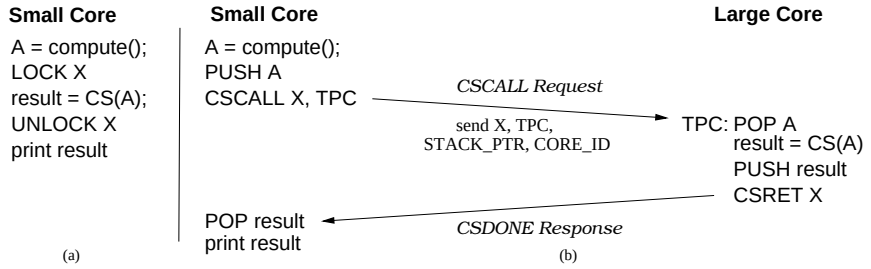


Figure 5.3: Source code and its execution: (a) baseline (b) with ACS

5.1.3 Hardware Support

5.1.3.1 Modifications to the small cores

When a CSCALL is executed, the small core sends a CSCALL request along with the stack pointer (STACK_PTR) and its core ID (CORE_ID) to the large core and stalls, waiting for the CSDONE response. The CSCALL instruction is retired when a CSDONE response is received. Such support for executing certain instructions remotely already exists in current architectures: for example, all 8 cores in Sun Niagara-1 [54] execute floating point (FP) operations on a common remote FP unit.

5.1.3.2 Critical Section Request Buffer

The Critical Section Request Buffer (CSRB), located at the large core, buffers the pending CSCALL requests sent by the small cores. Figure 5.4 shows the structure of the CSRB. Each entry in the CSRB contains a valid bit, the ID of the requesting core (REQ_CORE), the parameters of the CSCALL instruction, LOCK_ADDR and TARGET_PC, and the stack pointer (STACK_PTR) of the requesting core. The number of entries in the CSRB is equal to the maximum possible number of concurrent CSCALL instructions. Because each small core can execute

at most one CSCALL instruction at any time, the number of entries required is equal to the number of small cores in the system (Note that the large core does not send CSCALL requests to itself). For a system with 12 small cores, the CSRB has 12 entries, 25-bytes[1] each. Thus, the storage overhead of the CSRB is 300 bytes.

The circuit for CSRB includes the logic to insert/remove entries from the CSRB in FIFO order and a state machine which works as follows. When a CSCALL request is received, the CSRB enqueues the incoming request. When the large core is idle, the CSRB supplies the oldest CSCALL request in the buffer to the core. When the large core completes the critical section, the CSRB dequeues the corresponding entry and sends a CSDONE signal to the requesting core. Due to the simplicity of the logic, reading/writing from the buffer in FIFO takes a single cycle.

The CSRB has a single read/write port. Two entities can contend for this port: the large core (to dequeue a request) or the interconnect (to insert a CSCALL in the CSRB). When there is contention, we always give priority to the large core because delaying the large core extends critical section execution but delaying the CSCALL insertion has no performance impact as the CSCALL request would have waited in the CSRB anyways. Note that this wastes a cycle if the CSRB is empty and the large core is idle. However, wasting a cycle in this case does not degrade performance because if the CSRB is empty, there is low contention for the critical sections and hence critical section performance is less critical. Further note that the likelihood that both an incoming CSCALL request and the large core attempt to access the CSRB concurrently is extremely low because accesses to CSRB only happen at the start and finish of critical sections, which are infrequent events (usually hundreds of cycles apart).

[1]Each CSRB entry has one valid bit, 4-bit REQ_CORE, 8 bytes each for LOCK_ADDR, TARGET_PC, and STACK_PTR.

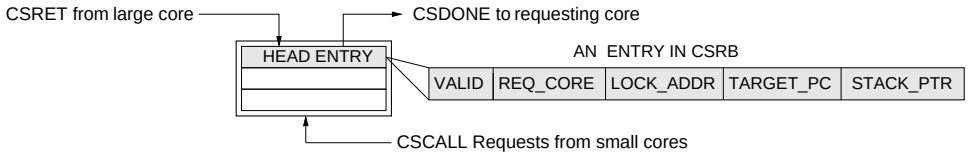


Figure 5.4: Critical Section Request Buffer (CSRB)

5.1.3.3 Modifications to the large core

When the large core receives an entry from the CSRB, it loads its stack pointer register with STACK_PTR and acquires the lock corresponding to LOCK_ADDR (as specified by program code). It then redirects the program counter to TARGET_PC and starts executing the critical section. When the core retires the CSRET instruction, it releases the lock corresponding to LOCK_ADDR and removes the HEAD ENTRY from the CSRB. Thus, ACS executes a critical section similar to a conventional processor by acquiring the lock, executing the instructions, and releasing the lock. However, it does so at a higher performance because of the aggressive configuration of the large core.

5.1.3.4 Interconnect Extensions

ACS introduces two new transactions on the on-chip interconnect: CSCALL and CSDONE. The interconnect transfers the CSCALL request (along with its arguments) from the smaller core to the CSRB and the CSDONE signal from the CSRB to the smaller core. Similar transactions already exist in the on-chip interconnects of current processors. For example, Sun Niagara-1 [54] uses such transactions to interface cores with the shared floating point unit.

5.1.4 Operating System Support

ACS requires modest support from the operating system (OS). When executing on an ACS architecture, the OS allocates the large core to a single application and does not schedule any threads onto it. Additionally, the OS sets the control registers of the large core to the same values as the small cores executing the application. As a result, the program context (e.g. processor status registers,

and TLB entries) of the application remains the coherent in all cores, including the large core. Note that ACS does not require any special modifications because such support already exists in current CMPs to execute parallel applications [46].

5.1.5 Reducing False Serialization in ACS

Critical sections that are protected by different locks can be executed concurrently in a conventional CMP. However, in ACS, their execution gets serialized because they are all executed sequentially on the single large core. This "false serialization" reduces concurrency and degrades performance. We reduce false serialization using two techniques. First, we make the large core capable of executing multiple critical sections concurrently[2], using simultaneous multithreading (SMT) [104]. Each SMT context can execute CSRB entries with different LOCK_ADDR. Second, to reduce false serialization in workloads where a large number of critical sections execute concurrently, we propose *Selective Acceleration of Critical Sections (SEL)*. The key idea of SEL is to estimate the occurrence of false serialization and adaptively decide whether or not to execute a critical section on the large core. If SEL estimates false serialization to be high, the critical section is executed locally on the small core, which reduces contention on the large core.

Implementing SEL requires two modifications: 1) a bit vector at each small core that contains the ACS_DISABLE bits and 2) logic to estimate false serialization. For the purpose of making our explanation simple, we assume that the ACS_DISABLE bit vector contains one bit per critical section and is indexed using the LOCK_ADDR (we later show how a practical design can use a very small 16-bit vector). When the smaller core encounters a CSCALL, it first checks the corresponding ACS_DISABLE bit. If the bit is 0 (i.e., false serialization is low), a CSCALL request is sent to the large core. Otherwise, the CSCALL and the critical section is executed locally.

[2] Another possible solution to reduce false serialization is to add additional large cores and distribute the critical sections across these cores. However, further investigation of this solution is an interesting research direction, but is beyond the scope of this thesis.

False serialization is estimated at the large core by augmenting the CSRB with a table of saturating counters, which track the false serialization incurred by each critical section. We quantify false serialization by counting the number of critical sections present in the CSRB for which the LOCK_ADDR is different from the LOCK_ADDR of the incoming request. If this count is greater than 1 (i.e. if there are at least two independent critical sections in the CSRB), the estimation logic adds the count to the saturating counter corresponding to the LOCK_ADDR of the incoming request. If the count is 1 (i.e. if there is exactly one critical section in the CSRB), the corresponding saturating counter is decremented. If the counter reaches its maximum value, the ACS_DISABLE bit corresponding to that lock is set by sending a message to all small cores. Since ACS is disabled infrequently, the overhead of this communication is negligible. To adapt to phase changes, we reset the ACS_DISABLE bits for all locks and halve the value of the saturating counters periodically (every 10 million cycles). We reduce the hardware overhead of SEL by hashing lock address into a small number of sets. Our implementation of SEL hashes lock addresses into 16 sets and uses 6-bit counters. The total storage overhead of SEL is 36 bytes: 16 counters of 6-bits each and 16 ACS_DISABLE bits for each of the 12 small cores.

5.2 Performance Trade-offs in ACS

There are three key performance trade-offs in ACS that determine overall system performance:

1. Faster critical sections vs. Fewer threads: ACS executes selected critical sections on a large core, the area dedicated to which could otherwise be used for executing additional threads. ACS could improve performance if the performance gained by accelerating critical sections (and serial program portions) outweighs the loss of throughput due to the unavailability of additional threads.

ACS's performance improvement becomes more likely when the number of cores on the chip increases. There are two reasons. First, the marginal loss in

parallel throughput due to the large core becomes relatively small (for example, if the large core replaces four small cores, then it reduces 50% of the smaller cores in a 8-core system but only 12.5% of cores in a 32-core system) Second, more cores allow concurrent execution of more threads, which increases contention by increasing the probability of each thread waiting to enter the critical section [83]. When contention is high, faster execution of a critical section reduces not only critical section execution time but also the contending threads' waiting time.

2. CSCALL/CSDONE signals vs. Lock acquire/release: To execute a critical section, ACS requires the communication of CSCALL and CSDONE transactions between a small core and a large core. This communication over the on-chip interconnect is an overhead of ACS, which the conventional lock acquire/release operations do not incur. On the other hand, a lock acquire operation often incurs cache misses [76] because the lock needs to be transferred from one cache to another. Each cache-to-cache transfer requires two transactions on the on-chip interconnect: a request for the cache line and the response, which has similar latency to the CSCALL and CSDONE transactions. ACS can reduce such cache-to-cache transfers by keeping the lock at the large core, which can compensate for the overhead of CSCALL and CSDONE. ACS actually has an advantage in that the latency of CSCALL and CSDONE can be overlapped with the execution of another instance of the same critical section. On the other hand, in conventional locking, a lock can only be acquired after the critical section has been completed, which *always* adds a delay before critical section execution.

3. Cache misses due to private data vs. cache misses due to shared data: In ACS, private data that is referenced in the critical section needs to be transferred from the cache of the small core to the cache of the large core. Conventional locking does not incur this cache-to-cache transfer overhead because critical sections are executed at the local core and private data is often present in the local cache. On the other hand, conventional systems incur overheads in transferring shared data: in such systems, shared data "ping-pongs" between caches as different threads execute the critical section and reference the shared data. ACS eliminates the transfers of

shared data by keeping it at the large core,[3] which can offset the misses it causes to transfer private data into the large core. In fact, ACS can decrease cache misses if the critical section accesses more shared data than private data. Note that ACS can improve performance even if there are equal or more accesses to private data than shared data because the large core can still 1) improve performance of other instructions and 2) hide the latency of some cache misses using latency tolerance techniques like out-of-order execution.

In summary, ACS can improve overall performance if its performance benefits (faster critical section execution, improved lock locality, and improved shared data locality) outweigh its overheads (reduced parallel throughput, CSCALL and CSDONE overhead, and reduced private data locality).

5.3 Evaluation Methodology

Table 5.1 shows the configuration of the simulated CMPs, using our in-house cycle-accurate x86 simulator. We evaluate three different CMP architectures: a symmetric CMP (SCMP) consisting of all small cores; an asymmetric CMP (ACMP) with one large core with 2-way SMT and remaining small cores; and an ACMP augmented with support for the ACS mechanism (ACS). Unless specified otherwise, all comparisons are done at equal area budget. We specify the area budget in terms of the number of small cores. Unless otherwise stated, the number of threads for each application is set equal to the number of threads that minimizes the execution time for the particular configuration; e.g. if the best performance of an application is obtained on an 8-core SCMP when it runs with 3 threads, then we report the performance with 3 threads. In both ACMP and SCMP, conventional lock acquire/release operations are implemented using the Monitor/Mwait instructions, part of the SSE3 extensions to the x86 ISA [45]. In ACS, lock acquire/release

[3] By keeping all shared data in the large core's cache, ACS reduces the cache space available to shared data compared to conventional locking (where shared data can reside in any on-chip cache). This can increase cache misses. However, we find that such cache misses are rare and do not degrade performance because the private cache of the large core is large enough.

instructions are replaced with CSCALL/CSRET instructions.

Small core	2-wide In-order, 2GHz, 5-stage. L1: 32KB write-through. L2: 256KB write-back, 8-way, 6-cycle access
Large core	4-wide Out-of-order, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, L1: 32KB write-through. L2: 1-MB write-back, 16-way, 8-cycle
Interconnect	64-bit wide bi-directional ring, all queuing delays modeled, ring hop latency of 2 cycles (latency between one cache to the next)
Coherence	MESI, On-chip distributed directory similar to SGI Origin [59], cache-to-cache transfers. # of banks = # of cores, 8K entries/bank
L3 Cache	8MB, shared, write-back, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Row buffer hit: 25ns, Row buffer miss: 50ns, Row buffer conflict: 75ns
Memory bus	4:1 cpu/bus ratio, 64-bit wide, split-transaction, pipelined bus.
Area-equivalent CMPs. Area = N small cores. N varies from 1 to 32	
SCMP	N small cores, One small core runs serial part, all N cores run parallel part, conventional locking (Max. concurrent threads = N)
ACMP	1 large core and N-4 small cores; large core runs serial part, 2-way SMT on large core and small cores run parallel part, conventional locking (Maximum number of concurrent threads = N-2)
ACS	1 large core and N-4 small cores; (N-4)-entry CSRB on the large core, large core runs the serial part, small cores run the parallel part, 2-way SMT on large core runs critical sections using ACS (Max. concurrent threads = N-4)

Table 5.1: Configuration of the simulated machines

5.3.1 Workloads

Our main evaluation focuses on 12 critical-section-intensive workloads shown in Table 5.2. We define a workload to be critical-section-intensive if at least 1% of the instructions in the parallel portion are executed within critical sections. We divide these workloads into two categories: workloads with coarse-grained locking and workloads with fine-grained locking. We classify a workload as using coarse-grained locking if it has at most 10 critical sections. Based on this classification, 7 out of 12 workloads use coarse-grain locking and the remaining 5 use fine-grain locking. All workloads were simulated to completion. Refer to Section 4.3 for a detailed description of simulated workloads.

In the ensuing discussion, we refer to Table 5.3, which shows the character-

Locks	Workload	Description	Source	Input set
Coarse	ep	Random number generator	[13]	262144 nums.
	is	Integer sort	[13]	n = 64K
	pagemine	Data mining kernel	[70]	10Kpages
	puzzle	15-Puzzle game	[109]	3x3
	qsort	Quicksort	[27]	20K elem.
	sqlite	sqlite3 [3] database engine	[4]	OLTP-simple
	tsp	Traveling salesman prob.	[55]	11 cities
Fine	iplookup	IP packet routing	[105]	2.5K queries
	oltp-1	MySQL server [2]	[4]	OLTP-simple
	oltp-2	MySQL server [2]	[4]	OLTP-complex
	specjbb	JAVA business benchmark	[90]	5 seconds
	webcache	Cooperative web cache	[101]	100K queries

Table 5.2: Simulated workloads

istics of each application, to provide insight into the performance results.

Table 5.3: Benchmark Characteristics. Shared/Private is the ratio of *shared* data (cache lines that are transferred from other cores) to *private* data (cache lines that hit in the private cache) accessed inside a critical section. Contention is the average number of threads waiting for critical sections when the workload is executed with 4, 8, 16, and 32 threads on the SCMP.

Workload	% of Non-parallel instr.	% of parallel instr. in critical sections	# of disjoint critical sections	Avg. instr. in critical section	Shared/ Private	Contention 4	8	16	32
ep	13.3	14.6	3	620618.1	**1.0**	1.4	1.8	4.0	8.2
is	84.6	8.3	1	9975.0	**1.1**	2.3	4.3	8.1	16.4
pagemine	0.4	5.7	1	531.0	**1.7**	2.3	4.3	8.2	15.9
puzzle	2.4	69.2	2	926.9	**1.1**	2.2	4.3	8.3	16.1
qsort	28.5	16.0	1	127.3	0.7	1.1	3.0	9.6	25.6
sqlite	0.2	17.0	5	933.1	**2.4**	1.4	2.2	3.7	6.4
tsp	0.9	4.3	2	29.5	0.4	1.2	1.6	2.0	3.6
iplookup	0.1	8.0	4	683.1	0.6	1.2	1.3	1.5	1.9
oltp-1	2.3	13.3	20	277.6	0.8	1.2	1.2	1.5	2.2
oltp-2	1.1	12.1	29	309.6	0.9	1.1	1.2	1.4	1.6
specjbb	1.2	0.3	39	1002.8	0.5	1.0	1.0	1.0	1.2
webcache	3.5	94.7	33	2257.0	**1.1**	1.1	1.1	1.1	1.4

5.4 Evaluation

We make three comparisons between ACMP, SCMP, and ACS. First, we compare their performance on systems where the number of threads is set equal to the optimal number of threads for each application under a given area constraint. Second, we compare their performance assuming the number of threads is set equal to the number of cores in the system, a common practice employed in many existing systems. Third, we analyze the impact of ACS on application scalability i.e., the number of threads over which performance does not increase.

5.4.1 Performance with the Optimal Number of Threads

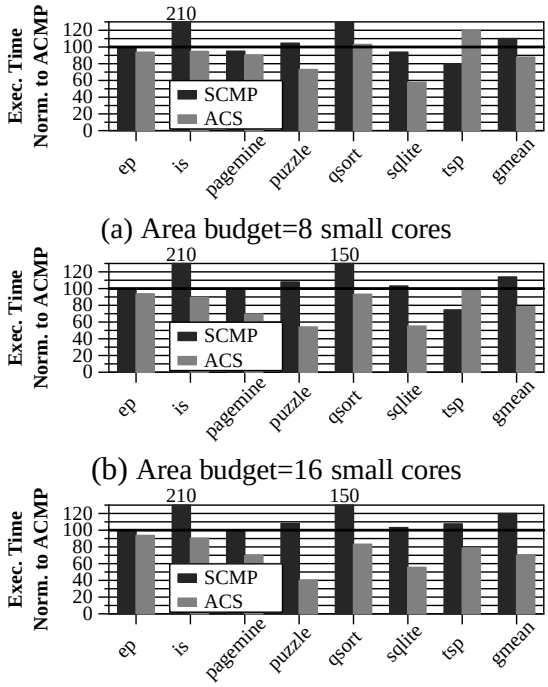
Systems sometimes use profile or run-time information to choose the number of threads that minimizes execution time [95]. We first analyze ACS with respect to ACMP and SCMP when the optimal number of threads are used for each application on each CMP configuration.[4] We found that doing so provides the best baseline performance for ACMP and SCMP, and a performance comparison results in the lowest performance improvement of ACS. Hence, this performance comparison penalizes ACS (as our evaluations in Section 5.4.2 with the same number of threads as the number of thread contexts will show). We show this performance comparison separately on workloads with coarse-grained locks and those with fine-grained locks.

5.4.1.1 Workloads with Coarse-Grained Locks

Figure 5.5 shows the execution time of each application on SCMP and ACS normalized to ACMP for three different area budgets: 8, 16, and 32. Recall that when area budget is equal to N, SCMP, ACMP, and ACS can execute up to N, N-2, and N-4 parallel threads respectively.

Systems area-equivalent to 8 small cores: When area budget equals 8, ACMP significantly outperforms SCMP for workloads with high percentage of instructions in the serial part (85% in `is` and 29% in `qsort` as Table 5.3 shows). In `puzzle`, even though the serial part is small, ACMP improves performance because it improves cache locality of shared data by executing two of the six threads on the large core, thereby reducing cache-to-cache transfers of shared data. SCMP outperforms ACMP for `sqlite` and `tsp` because these applications spend a very small fraction of their instructions in the serial part and sacrificing two threads for improved serial performance is not a good trade-off. Since ACS devotes the two

[4] We determine the optimal number of threads for an application by simulating all possible number of threads and using the one that minimizes execution time. The interested reader can obtain the optimal number of threads for each benchmark and each configuration by examining the data in Figure 6.11. Due to space constraints, we do not explicitly quote these thread counts.



(a) Area budget=8 small cores

(b) Area budget=16 small cores

(c) Area budget=32 small cores

Figure 5.5: Execution time of workloads with coarse-grained locking on ACS and SCMP normalized to ACMP

SMT contexts on the large core to accelerate critical sections, it can execute only four parallel threads (compared to 6 threads of ACMP and 8 threads of SCMP). Despite this disadvantage, ACS reduces the average execution time by 22% compared to SCMP and by 11% compared to ACMP. ACS improves performance of five out of seven workloads compared to ACMP. These five workloads have two common characteristics: 1) they have high contention for the critical sections, 2) they access more shared data than private data in critical sections. Due to these characteristics, ACS reduces the serialization caused by critical sections and improves locality of shared data.

Why does ACS reduce performance in `qsort` and `tsp`? The critical section in `qsort` protects a stack that contains indices of the array to be sorted. The insert operation pushes two indices (private data) onto the stack by changing the stack pointer (shared data). Since indices are larger than the stack pointer, there are

more accesses to private data than shared data. Furthermore, contention for critical sections is low. Therefore, `qsort` can take advantage of additional threads in its parallel portion and trading off several threads for faster execution of critical sections lowers performance. The dominant critical section in `tsp` protects a FIFO queue where an insert operation reads the node to be inserted (private data) and adds it to the queue by changing only the head pointer (shared data). Since private data is larger than shared data, ACS reduces cache locality. In addition, contention is low and the workload can effectively use additional threads.

Systems area-equivalent to 16 and 32 small cores: Recall that as the area budget increases, the overhead of ACS decreases. This is due to two reasons. First, the parallel throughput reduction caused by devoting a large core to execute critical sections becomes smaller, as explained in Section 5.2. Second, more threads increases contention for critical sections because it increases the probability that each thread is waiting to enter the critical section. When the area budget is 16, ACS improves performance by 32% compared to SCMP and by 22% compared to ACMP. When the area budget is 32, ACS improves performance by 42% compared to SCMP and by 31% compared to ACMP. In fact, the two benchmarks (`qsort` and `tsp`) that lose performance with ACS when the area budget is 8 experience significant performance gains with ACS over both ACMP and SCMP for an area budget of 32. For example, ACS with an area budget of 32 provides 17% and 22% performance improvement for `qsort` and `tsp` respectively over an equal-area ACMP. With an area budget of at least 16, ACS improves the performance of *all* applications with coarse-grained locks. We conclude that ACS is an effective approach for workloads with coarse-grained locking even at small area budgets. However, ACS becomes even more attractive as the area budget in terms of number of cores increases.

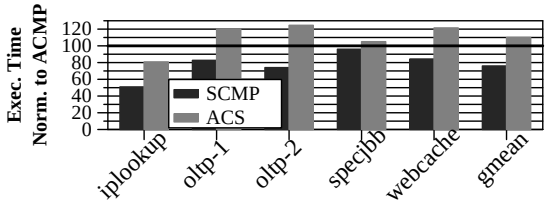
5.4.1.2 Workloads with Fine-Grained Locks

Figure 5.6 shows the execution time of workloads with fine-grained locking for three different area budgets: 8, 16, and 32. Compared to coarse-grained

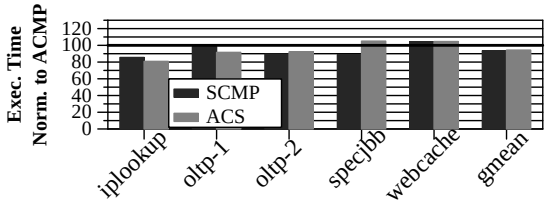
locking, fine-grained locking reduces contention for critical sections and hence the serialization caused by them. As a result, critical section contention is negligible at low thread counts, and the workloads can take significant advantage of additional threads executed in the parallel section. When the area budget is 8, SCMP provides the highest performance (as shown in Figure 5.6(a)) for all workloads because it can execute the most number of threads in parallel. Since critical section contention is very low, ACS essentially wastes half of the area budget by dedicating it to a large core because it is unable to use the large core efficiently. Therefore, ACS increases execution time compared to ACMP for all workloads except `iplookup`. In `iplookup`, ACS reduces execution time by 20% compared to ACMP but increases it by 37% compared to SCMP. The critical sections in `iplookup` access more private data than shared data, which reduces the benefit of ACS. Hence, the faster critical section execution benefit of ACS is able to overcome the loss of 2 threads (ACMP) but is unable to provide enough improvement to overcome the loss of 4 threads (SCMP).

As the area budget increases, ACS starts providing performance improvement over SCMP and ACMP because the loss of parallel throughput due to the large core reduces. With an area budget of 16, ACS performs similarly to SCMP (within 2%) and outperforms ACMP (by 6%) on average. With an area budget of 32, ACS's performance improvement is the highest: 17% over SCMP and 13% over ACMP; in fact, ACS outperforms both SCMP and ACMP on all workloads. Hence, we conclude that ACS provides the best performance compared to the alternative chip organizations, even for critical-section-intensive workloads that use fine-grained locking.

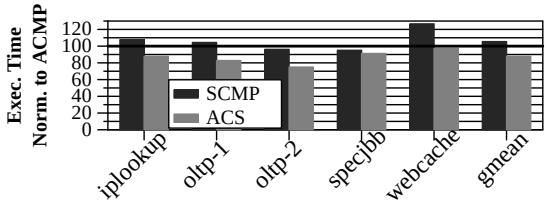
Depending on the scalability of the workload and the amount of contention for critical sections, the area budget required for ACS to provide performance improvement is different. Table 5.4 shows the area budget required for ACS to outperform an equivalent-area ACMP and SCMP. In general, the area budget ACS requires to outperform SCMP is higher than the area budget it requires to outperform ACMP. However, `webcache` and `qsort` have a high percentage of serial instruc-



(a) Area budget=8 small cores



(b) Area budget=16 small cores



(c) Area budget=32 small cores

Figure 5.6: Execution time of workloads with fine-grained locking on ACS and SCMP normalized to ACMP

tions; therefore ACMP becomes significantly more effective than SCMP for large area budgets. For all workloads with fine-grained locking, the area budget ACS requires to outperform an area-equivalent SCMP or ACMP is less than or equal to 24 small cores. Since chips with 8 and 16 small cores are already in the market [54], and chips with 32 small cores are being built [86, 103], we believe ACS can be a feasible and effective option to improve the performance of workloads that use fine-grained locking in near-future multi-core processors.

	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
ACMP	6	6	6	4	12	6	10	6	14	10	18	24
SCMP	6	4	6	4	8	6	18	14	14	16	18	14

Table 5.4: Area budget (in terms of small cores) required for ACS to outperform an equivalent-area ACMP and SCMP

Summary: Based on the observations and analyses we made above for workloads with coarse-grained and fine-grained locks, we conclude that ACS provides significantly higher performance than both SCMP and ACMP for both types of workloads, except for workloads with fine-grained locks when the area budget is low. ACS's performance benefit increases as the area budget increases. In future systems with a large number of cores, ACS is likely to provide the best system organization among the three choices we examined. For example, with an area budget of 32 small cores, ACS outperforms SCMP by 34% and ACMP by 23% averaged across all workloads, including both fine-grained and coarse-grained locks.

5.4.2 Performance with Number of Threads Set Equal to the Number of Available Thread Contexts

In the previous section, we used the optimal number of threads for each application-configuration pair. When an estimate of the optimal number of threads is not available, many current systems use as many threads as there are available thread contexts [47, 73]. We now evaluate ACS assuming the number of threads is set equal to the number of available contexts. Figure 6.11 shows the speedup curves of ACMP, SCMP, and ACS over one small core as the area budget is varied from 1 to 32. The curves for ACS and ACMP start at 4 because they require at least one large core which is area-equivalent to 4 small cores.

Number of threads	No. of max. thread contexts			Optimal		
Area Budget	8	16	32	8	16	32
SCMP	0.93	1.04	1.18	0.94	1.05	1.15
ACS	0.97	0.77	0.64	0.96	0.83	0.77

Table 5.5: Average execution time normalized to area-equivalent ACMP

Table 5.5 summarizes the data in Figure 6.11 by showing the average execution time of ACS and SCMP normalized to ACMP for area budgets of 8, 16, and 32. For comparison, we also show the data with optimal number of threads. With an area budget of 8, ACS outperforms both SCMP and ACMP on 5 out of 12 benchmarks. ACS degrades average execution time compared to SCMP by 3% and outperforms ACMP by 3%. When the area budget is doubled to 16, ACS out-

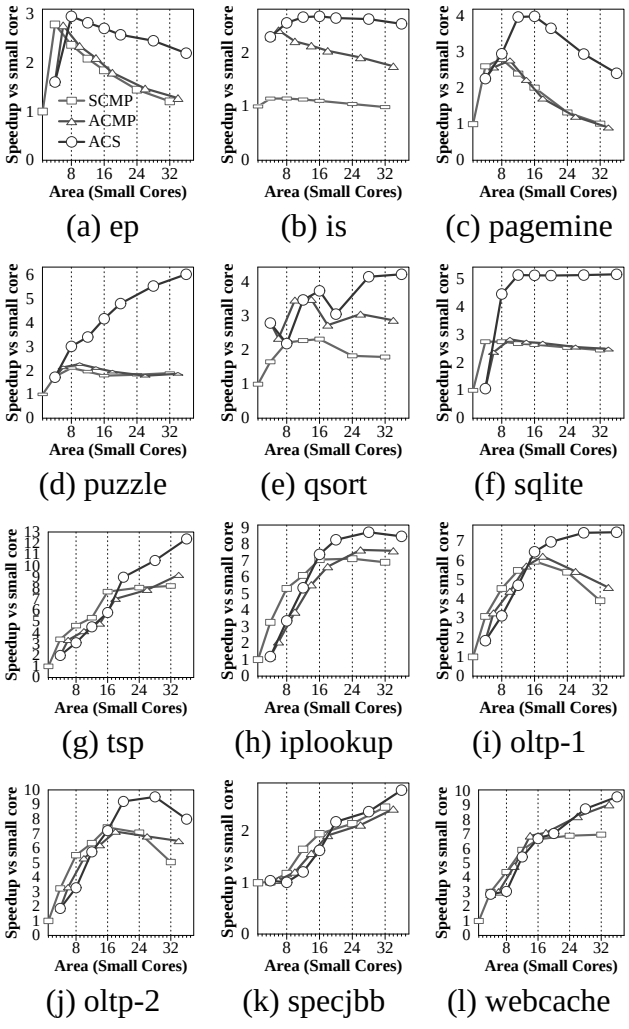


Figure 5.7: Speedup over a single small core

performs both SCMP and ACMP on 7 out of 12 benchmarks, reducing average execution time by 26% and 23%, respectively. With an area budget of 32, ACS outperforms both SCMP and ACMP on all benchmarks, reducing average execution time by 46% and 36%, respectively. Note that this performance improvement is significantly higher than the performance improvement ACS provides when the optimal number of threads is chosen for each configuration (34% over SCMP and 23% over ACMP). Also note that when the area budget increases, ACS starts to consistently outperform both SCMP and ACMP. This is because ACS tolerates contention among threads better than SCMP and ACMP. Table 5.6 compares the contention of SCMP, ACMP, and ACS at an area budget of 32. For `ep`, on average more than 8 threads wait for each critical section in both SCMP and ACMP. ACS reduces the waiting threads to less than 2, which improves performance by 44% (at an area budget of 32).

Workload	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
SCMP	8.2	16.4	15.9	16.1	25.6	6.4	3.6	1.9	2.2	1.6	1.2	1.4
ACMP	8.1	14.9	15.5	16.1	24.0	6.2	3.7	1.9	1.9	1.5	1.2	1.4
ACS	1.5	2.0	2.0	2.5	1.9	1.4	3.5	1.8	1.4	1.3	1.0	1.2

Table 5.6: Contention (see Table 3 for definition) at an area budget of 32 (Number of threads set equal to the number of thread contexts)

We conclude that, even if a developer is unable to determine the optimal number of threads for a given application-configuration pair and chooses to set the number of threads at a point beyond the saturation point, ACS provides significantly higher performance than both ACMP and SCMP. In fact, ACS's performance benefit is even higher in systems where the number of threads is set equal to number of thread contexts because ACS is able to tolerate contention for critical sections significantly better than ACMP or SCMP.

5.4.3 Application Scalability

We examine the effect of ACS on the number of threads required to minimize the execution time. Table 5.7 shows number of threads that provides the best

performance for each application using ACMP, SCMP, and ACS. The best number of threads were chosen by executing each application with all possible threads from 1 to 32. For 7 of the 12 applications (`is`, `pagemine`, `puzzle`, `qsort`, `sqlite`, `oltp-1`, and `oltp-2`) ACS improves scalability: it increases the number of threads at which the execution time of the application is minimized. This is because ACS reduces contention due to critical sections as explained in Section 5.4.2 and Table 5.6. For the remaining applications, ACS does not change scalability.[5] We conclude that if thread contexts are available on the chip, ACS uses them more effectively compared to ACMP and SCMP.

Workload	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
SCMP	4	8	8	8	16	8	32	24	16	16	32	32
ACMP	4	8	8	8	16	8	32	24	16	16	32	32
ACS	4	12	12	32	32	32	32	24	32	24	32	32

Table 5.7: Best number of threads for each configuration

5.4.4 Performance of ACS on Critical Section Non-Intensive Benchmarks

We also evaluated all 16 benchmarks from the NAS [13] and SPLASH [110] suites that are not critical-section-intensive. These benchmarks contain regular data-parallel loops and execute critical sections infrequently (less than 1% of the executed instructions). Detailed results of this analysis are presented in [94]. We find that ACS does not significantly improve or degrade the performance of these applications. When area budget is 32, ACS provides a modest 1% performance improvement over ACMP and 2% performance reduction compared to SCMP. As area budget increases, ACS performs similar to (within 1% of) SCMP. We conclude that ACS will not significantly affect the performance of critical section non-intensive workloads in future systems with large number of cores.

[5] Note that Figure 6.11 provides more detailed information on ACS's effect on the scalability of each application. However, unlike Table 7, the data shown on the x-axis is area budget and not number of threads.

5.5 Sensitivity of ACS to System Configuration

5.5.1 Effect of SEL

ACS uses the SEL mechanism (Section 5.1.5) to selectively accelerate critical sections to reduce false serialization of critical sections. We evaluate the performance impact of SEL. Since SEL does not affect the performance of workloads that have negligible false serialization, we focus our evaluation on the three workloads that experience false serialization: `puzzle`, `iplookup`, and `webcache`. Figure 5.8 shows the normalized execution time of ACS with and without SEL for the three workloads when the area budget is 32. For `iplookup` and `webcache`, which has the highest amount of false serialization, using SEL improves performance by 11% and 5% respectively over the baseline. The performance improvement is due to acceleration of *some* critical sections which SEL allows to be sent to the large core because they do not experience false serialization. In `webcache`, multiple threads access pages of different files stored in a shared cache. Pages from each file are protected by a different lock. In a conventional system, these critical sections can execute in parallel, but ACS without SEL serializes the execution of these critical sections by forcing them to execute on a single large core. SEL disables the acceleration of 17 out of the 33 locks, which eliminates false serialization and reduces pressure on the large core. In `iplookup`, multiple copies of the routing table (one for each thread) are protected by disjoint critical sections that get serialized without SEL. `puzzle` contains two critical sections protecting a heap object (PQ) and a memoization table. Accesses to PQ are more frequent than to the memoization table, which results in false serialization for the memoization table. SEL detects this serialization and disables the acceleration of the critical section for the memoization table. On average, across all 12 workloads, ACS with SEL outperforms ACS without SEL by 15%. We conclude that SEL can successfully improve the performance benefit of ACS by eliminating false serialization without affecting the performance of workloads that do not experience false serialization.

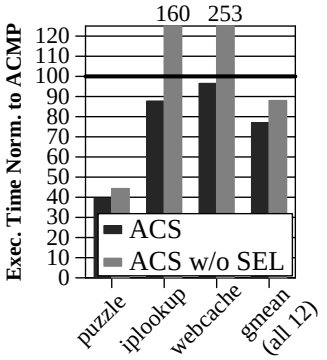


Figure 5.8: Impact of SEL.

5.5.2 Effect of using SMT on the Large Core

We have shown that ACS significantly improves performance over SCMP and ACMP when the large core provides support for SMT. The added SMT context provides ACS with the opportunity to concurrently execute critical sections that are protected by different locks on the high performance core. When the large core does not support SMT, contention for the large core can increase and lead to false serialization. Since SMT is not a requirement for ACS, we evaluate ACS on an ACMP where the large core does not support SMT and executes only one thread. Figure 5.9 shows the execution time of ACS without SMT normalized to ACMP without SMT when the area budget is 32. On average, ACS without SMT reduces execution time by 22% whereas ACS with SMT by 26%. Thus, SMT provides 4% performance benefit by reducing false serialization of critical sections.

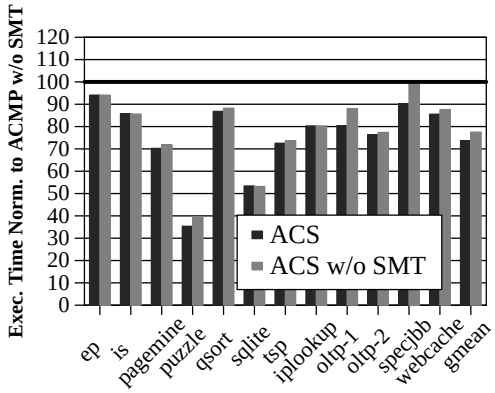


Figure 5.9: Impact of SMT.

5.5.3 ACS on Symmetric CMPs: Effect of Only Data Locality

Part of the performance benefit of ACS is due to improved locality of shared data and locks. This benefit can be realized even in the absence of a large core. A variant of ACS can be implemented on a symmetric CMP, which we call *symmACS*. In symmACS, one of the small cores is dedicated to executing critical sections. This core is augmented with a CSRB and can execute the CSCALL requests and CSRET instructions. Figure 5.10 shows the execution time of symmACS and ACS normalized to SCMP when area budget is 32. SymmACS reduces execution time by more than 5% compared to SCMP in `is`, `puzzle`, `sqlite`, and `iplookup` because more shared data is accessed than private data in critical sections.[6] In `ep`, `pagemine`, `qsort`, and `tsp`, the overhead of CSCALL/CSRET messages and transferring private data offsets the shared data/lock locality advantage of ACS. Thus, overall execution time increases. On average, symmACS reduces execution time by only 4% which is much lower than the 34% performance benefit of ACS. Since the performance gain due to improved locality alone is relatively small, we conclude that most of the performance improvement of ACS comes from accelerating critical section execution using the large core.

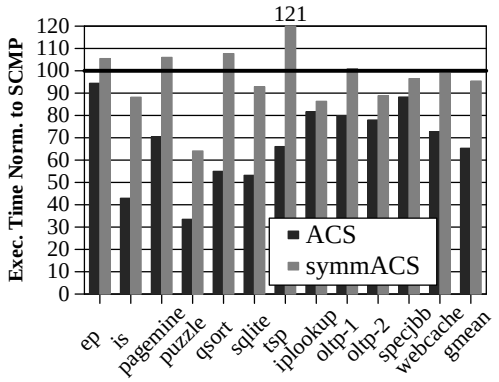


Figure 5.10: ACS on symmetric CMP.

[6]Note that these numbers do not correspond to those shown in Table 5.3 on page 49. The Shared/Private ratio reported in Table 5.3 is collected by executing the workloads with 4 threads. On the other hand, in this experiment, the workloads were run with the optimal number of threads for each configuration.

5.5.4 Interaction of ACS with Hardware Prefetching

Part of the performance benefit of ACS comes from improving shared data/lock locality, which can also be partially improved by data prefetching [81, 102]. To study the effect of prefetching, we augment each core with a L2 stream prefetcher [99] (32 streams, up to 16 lines ahead).

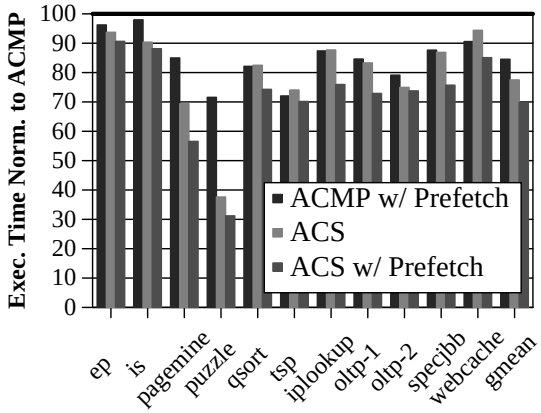


Figure 5.11: Impact of prefetching

Figure 5.11 shows the execution time of ACMP with a prefetcher, ACS (with and without a prefetcher), all normalized to an ACMP without a prefetcher (area budget is 32). On all benchmarks, prefetching improves the performance of both ACMP and ACS, and ACS with a prefetcher outperforms ACMP with a prefetcher. However, in `puzzle`, `qsort`, `tsp`, and `oltp-2`, ACMP benefits more from prefetching than ACS because these workloads contain shared data structures that lend themselves to prefetching. For example, in `tsp`, one of the critical sections protects an array. All elements of the array are read, and often updated, inside the critical section which leads to cache misses in ACMP. The stream prefetcher successfully prefetches this array, which reduces the execution time of the critical sections. As a result, ACMP with a prefetcher is 28% faster. Because ACS already reduces the misses for the array by keeping it at the large core, the improvement from prefetching is modest compared to ACMP (4%). On average, ACS with prefetching reduces execution time by 18% compared to ACMP with prefetching and 10% compared to ACS without prefetching. Thus, ACS interacts positively with a stream prefetcher and both schemes can be employed together.

Chapter 6

ACMP for Accelerating the Limiter Stage

The overall throughput of a kernel with pipeline parallelism is dictated by the pipeline stage which has the lowest throughput. We call this the LIMITER stage. It is desirable to run the LIMITER as fast as possible. The large core in the ACMP can be used to accelerate the execution of the LIMITER, thereby increasing its throughput. To this end, we propose *Accelerated Limiter Stage (ALS)*. Similar to the Accelerating Critical Sections (ACS) mechanism, ALS also runs the non-parallel part of the program on the large core.

6.1 Key Insights

We develop two key insights. First, the large core is less power-efficient than the small cores and thus it should never be assigned to the non-LIMITER, non-critical, stages as it will waste power but not improve performance. Second, the LIMITER stage –scalable or non-scalable– can always benefit from running at the large core. For a scalable stage, having the large core increases the throughput fractionally. For example, let us assume an ACMP with a single large core. Suppose that each small core has an IPC of 1 and the large core's IPC is 2. Let us consider a LIMITER which scales up to 5 cores. With 5 small cores, it will be running at an aggregate IPC of 5 (5×1). Whereas, with four small cores and one large core, it will run at an aggregate IPC of 6 ($4 \times 1 + 2$). Thus, a 20% increase in throughput. For non-scalable stages, having the large cores increases the LIMITER's throughput by as much as the speedup of the large core over a small core. For example, ACS can increase the throughput of a non-scalable LIMITER by up to 2x.

6.2 Overview

Implementing ALS on an ACMP does not require any hardware support and only requires modifications to the run-time library. We implement ALS in two steps. First, we maximize the performance of the pipeline workload on the small cores of the ACMP by choosing the best core-to-stage allocation. Once performance has been maximized on the small cores, we assign the large core to the stage with the lowest throughput.

To choose the best core-to-stage allocation, we propose *Feedback-Driven Pipelining (FDP)*. FDP identifies the limiter by choosing a core-to-stage allocation at run-time such that execution time is minimized. Section 6.4 describes the support required to accelerate the limiter stage using ACMP.

6.3 Feedback-Driven Pipelining: Optimizing the pipeline

Choosing the core-to-stage allocation which maximizes performance is a well-known challenge. The programmer is often assigned the task of choosing the best core-to-stage allocation. This is infeasible as it burdens the programmers and also leads to sub-optimal performance in case the LIMITER stage changes at run-time due a variation in input set or machine configuration. To minimize the effect of this problem, we propose an automated mechanism to choose the LIMITER stage. We call it *Feedback-Driven Pipelining (FDP)*. By choosing the best stage-to-core allocation at run-time, FDP saves programmer effort, is more robust to changes in input set and machine configuration, and saves power by combining the stages which are much faster than the LIMITER on a single core.

6.3.1 Overview

FDP uses runtime information to choose core-to-stage allocation for best overall performance and power-efficiency. Figure 6.1 shows an overview of the FDP framework.

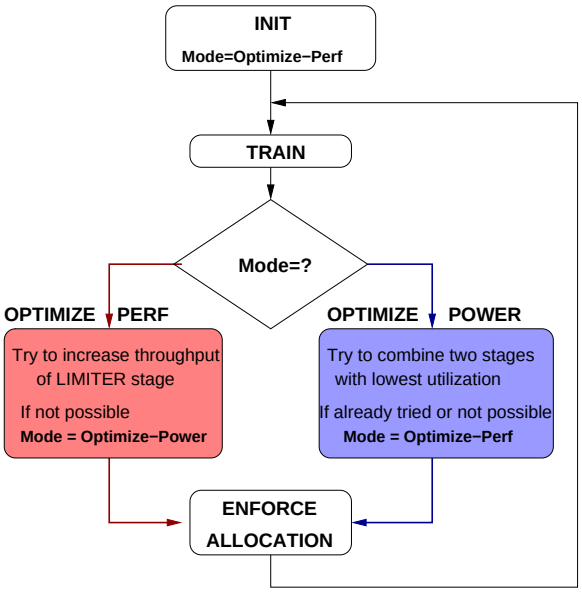


Figure 6.1: Overview of FDP.

FDP operates in two modes: one that optimizes performance (`Optimize-Perf`) and other that optimizes power (`Optimize-Power`). Initially, each stage in the pipeline is allocated one small core. FDP first tries to achieve the highest performance, and then it tries to optimize power. FDP is an iterative technique that contains three phases: training, re-allocation of cores to stages, and enforcement of the new allocation. The training phase gathers runtime information for each stage of the pipeline, and is helpful in determining the throughput and core utilization of each stage. Based on this information, the performance-optimization mode identifies the LIMITER stage and tries to increase its throughput by allocating more cores to it. When it can no longer improve performance (as there may be no spare cores or adding cores does not help improve performance) FDP switches to power-optimization mode. In this mode, FDP tries to assign the stages with lowest utilization to one core, as long as the combined stage does not become the LIMITER stage. The core thus saved can be used to improve performance or turned off to save power. Every time FDP chooses a new core-to-stage allocation, it enforces the new allocation on the pipeline at the end of the iteration.

Stages:	P0	P1	P2	Avg. Execution Time	Throughput
S0 :	(3K,3)			1K	1/1K
S1 :		(12K, 3)		4K	1/4K
S2 :	(9K, 1)		(21K,2)	10K	**1/5K**

Figure 6.2: Sample output from Train for a pipeline with three stages (S0, S1, S2) on a 3-core machine. Each entry is a 2-tuple: (the sum of time measurements, the number of time measurements) taken for each core-stage pair. Blank entries contain (0,0).

6.3.2 Train

The goal of the training phase is to gather runtime statistics about each stage. To measure execution time of each stage, the processor's cycle count register is read at the beginning and end of each stage. Instructions to read the cycle count register already exist in current ISAs, e.g., the `rdtsc` instruction in the x86 ISA. The difference between the two readings at the start and end of the stage is the execution time of the stage. This timing information is stored in a two-dimensional table similar to the one shown in Figure 6.2. The rows in the table represent stages (S0-S2) and columns represent cores (P0-P2). Each entry in this table is a 2-tuple: the sum and the number of time measurements taken for the corresponding core-stage pair. For each measurement taken, Train adds the measured time to the sum of measured times of the core-stage pair and increments the corresponding number of measurements. For example, if Train measures that executing S0 on P0 took 4K cycles, then it will modify the entry corresponding to S0 and P0 in Table 5 to (7K,4) i.e. (3K+4K, 3+1). Note that if a stage is not assigned to a core, the entry corresponding to the core-stage pair remains (0,0). For example, since S1 is only assigned to P1 and not to P0 and P2, its entries for P0 and P2 are 0. We limit the overhead of measuring the timing information via sampling: we measure it once every 128th work-quanta processed by the stage.

6.3.3 Performance-Optimization

The goal of the performance-optimization mode is to change the core-to-stage allocation in order to improve overall performance. When the mode of operation is performance-optimization, one of the threads invokes this phase once every 2K iterations or 100K processor cycles, whichever is earlier[1]. The phase takes as its input the information collected during training, a table similar to Figure 6.2. The phase first computes the average execution time of all stages. The average execution time of a stage is the sum of all timing measurements recorded in the table for that stage divided by the total number of measurements for that stage. For example, for the table shown in Figure 6.2, the average execution time of stage S2 is 10K cycles computed as (9K+21K)/(1+2). The phase next computes the throughput of each stage as the number of cores assigned to the stage divided by the stage's average execution time (e.g., throughput of S2, which runs on two cores, is 2/10K, i.e., 1/5K). The stage with the lowest throughput is identified as the LIMITER (S2 is the LIMITER stage in our example). If there are free small cores in the system, FDP allocates one of them to the LIMITER. The cores assigned to the LIMITER stage execute in parallel and feed from the in-queue assigned to the LIMITER stage.

To converge to the best decision, it is important that the core-to-stage allocations, that have already been tried, are not re-tried. FDP filters the allocations by maintaining the set of all allocations which have been tried. A new allocation is only enforced if it has not been tried before except when FDP is reverting back to a previous allocation that is known to perform similar to (or better than) the current allocation, while using fewer cores.

FDP increases the number of cores of the LIMITER stage with an implicit assumption that more cores lead to higher throughput. Unfortunately, this assumption is not always true; performance of a stage can saturate at a certain number of cores and further increasing cores wastes power without improving performance. To avoid allocating cores that do not improve performance, FDP always measures

[1] We choose these values empirically.

and stores the performance of the previous allocation. Every time FDP assigns a new core to the LIMITER stage, it compares the new performance with the performance of the previous allocation. If the new performance is higher than the performance with the previous allocation, FDP allocates another core to the LIMITER stage. However, if the new performance is lower than the performance with the previous allocation, FDP reverts to the previous allocation and switches to power-mode.

6.3.4 Power-Optimization

The goal of this mode is to reduce the number of active cores, while maintaining similar performance. When the mode of operation is power-optimization, this phase is invoked once every 2K iterations or 100K processor cycles whichever is earlier. This phase uses the information collected during training to compute the throughput of each stage. To improve power-efficiency, the two stages with the highest throughput that are each allocated to a separate core can be combined to execute on a single core, as long as the resulting throughput is not less than the throughput of the LIMITER stage. This optimization frees up one core which can be used by another stage for performance improvement or turned off for saving power. This process is repeated until no more cores can be set free. At this point, FDP reverts to performance mode.

6.3.5 Enforcement of Allocation

FDP changes the allocation of cores to stages dynamically. To facilitate dynamic allocation we add a data structure which stores for each core the list of stages allocated to it. The core processes the stages allocated to it in a round-robin fashion. FDP can modify the allocation in three ways. First, when a free core is allocated to the LIMITER stage, the LIMITER stage is added to the list of the free core. Second, when a stage is removed from a core, it is deleted from the core's list. Third, when stages on two different cores are combined on to a single core, the list of one of the cores is merged with the list of other core and emptied.

6.3.6 Programming Interface for FDP

The FDP library contains the code for measuring and recording the execution time of each stage. It also maintains sampling counters for each allocation to limit instrumentation overhead. It automatically invokes performance-optimization or power-optimization phases at appropriate times without programmer intervention. To interface with this library, the programmer must insert in the code the four library calls shown in Figure 6.3.

```
void FDP_Init (num_stages)
void FDP_BeginStage (stage_id)
void FDP_EndStage (stage_id)
int  FDP_GetNextStage ()
```

Figure 6.3: FDP library interface.

The `FDP_Init` routine initializes storage for FDP and sets the mode to optimize performance. The training phase of FDP reads the processor's cycle count register at the start and end of every stage. To facilitate this, a call to `FDP_BeginStage` is inserted after the work-quanta is read from the respective queue and before it is processed. Also, a call to `FDP_EndStage` is inserted after the processing of the quanta is complete but before it is pushed to the next stage. The arguments of both function calls is the stage id. Once a core completes a work-quanta, it needs to know which stage it should process next. This is done by calling the `FDP_GetNextStage` function. FDP obtains the id of the core executing an FDP function by invoking a system call.

FDP only requires modifications to the code of the worker thread in a pipeline program, not the code which does the actual computation for the stage. Thus, FDP can be implemented in the infrastructures commonly used as foundation for implementing pipeline programs, e.g., Intel Threading Building Blocks [47].

Figure 6.4 shows how the code of the worker loop is modified to interface with the FDP library. The four function calls are inserted as follows.

```
1:  FDP_Init ()
2:      while (!DONE)
3:          stage_id = FDP_GetNextStage ()
4:          Pop an iteration i from the stage's in-queue
5:          FDP_BeginStage (stage_id)
6:          Run the the stage of that iteration
7:          FDP_EndStage (stage_id)
8:          Push the iteration to the in-queue of its next stage
```

Figure 6.4: Modified worker loop (additions/modifications are shown in bold)

`FDP_Init` is called before the worker loop begins. Inside the loop the thread calls `FDP_GetNextStage` to get the ID of the next stage to process. The worker thread then pops an entry from the in-queue of the chosen stage. Before executing the computation in stage, it calls the instrumentation routine `FDP_BeginStage`. It then runs the computation and after the computation it calls the instrumentation function `FDP_EndStage`. It then pushes the iteration to the in-queue of the next iteration.

6.3.7 Overheads

FDP is a pure software mechanism and does not require *any* changes to the hardware. FDP only incurs minor latency and software storage overhead. The latency overhead is incurred due to instrumentation and execution of the optimization phases. These overheads are significantly reduced because we only instrument 0.7% (1/128) iterations. The software storage overhead comprises the storage required for the current core-to-stage allocation, the list of previously tried core-to-stage allocations, the table to store execution latencies of each stage, and counters to support sampling. The total storage overhead is less than 4KB in a system with 16 cores and 16 stages. Note that this storage is allocated in the global memory and does not require separate hardware support.

6.4 Accelerating the Limiter Stage

Once FDP has optimized the pipeline using the small cores, ALS can accelerate the stage which is limiting performance by assigning it the large core. Recall that performance with FDP saturates when either the LIMITER stage stops scaling or FDP runs out of cores to assign. When accelerating with the ACMP, these two cases are handled differently.

1. LIMITER stops scaling: Since the LIMITER's throughput has saturated, we know that increasing the number of cores assigned to the LIMITER will only waste power, without providing any performance benefit. Thus, our goal is to increase its throughput without increasing the number of cores assigned to the LIMITER. Thus, when we add the large core to the LIMITER to increase its throughput, we must remove a small core to keep the number of threads the same.

When the LIMITER ceases to scale, ALS assigns the large core to the LIMITER and removes one of the small cores from the LIMITER's core allocation. ALS removes a small core because adding the large core, without removing the small core, will increase the number of parallel threads for the LIMITER; this will be wasteful because we know that the LIMITER cannot leverage any more cores (as its scalability has been saturated). We remove only a single small core (and no additional small cores) because our goal is to improve the LIMITER throughput as much as possible.

We further explain this with an example. Consider a LIMITER which can scale up to three cores. Once three small cores have been assigned to this LIMITER, ALS realizes that more cores cannot be assigned. At this point, ALS adds the large core to LIMITER's allocation. Now, the LIMITER has four cores assigned to it (one large and three small) which is wasteful. To save the power, without reducing performance, ALS removes one of the small cores from the LIMITER's allocation as soon as it assigns the large core to the LIMITER.

2. No more small cores to assign: In case the LIMITER continues to scale and FDP runs out of free small cores to assign, ALS assigns the large core to the

LIMITER but does not remove a small core from the LIMITER's core assignment. Removing a small core is unnecessary in this case because the LIMITER is still scalable and it can leverage the additional core.

How does ALS Identify the large core? The initialization code of each worker threads runs the CPUID instruction upfront and reports the type (small or large) of the core it is running on to the run-time. When FDP finds a stage which deserves the large core, it queries the run-time for the core-id of the large core and assigns the identified stage to the large core.

6.5 Performance Trade-offs in ALS

The ACMP replaces four small cores with one large core which makes its peak throughput lower than that of the SCMP. If the LIMITER stage in a workload is scalable then it can benefit more from the four small cores, than the one large core. For such workloads, the ACMP can reduce performance compared to an SCMP. However, this negative effect of the ACMP will reduce in the future as more cores will become available on the chip.

6.6 Evaluation

We partition our evaluation into two parts. First, we show the working of FDP and evaluate its effectiveness at choosing the best core-to-stage allocation. Second, we show how ACMP can improve performance by accelerating the limiter stage identified by FDP.

6.6.1 Effectiveness of FDP

6.6.1.1 Evaluation Methodology

We evaluate FDP using a very different approach than the rest of this thesis. This is because evaluating FDP requires extensive experiments (multiple thousand runs of each workloads). Doing this many experiments on a simulator requires a

long time. Instead of using the simulator, we evaluate FDP by running the workloads on real Intel and AMD machines.

Configurations Our baseline system is a Core2Quad SMP that contains 2 Xeon Chips of four cores each. To show scalability of our technique, we also conduct experiments with an AMD Barcelona SMP machine with four Quad-core chips (results for this machine will be reported in Section 6.6.1.6). Configuration details for both machines are shown in Table 7.1 on page 100. Each system has sufficient memory to accommodate the working set of each of the workloads used in our study.

Table 6.1: System Configuration

Name	**Core2Quad (Baseline)**	**Barcelona**
System	8-cores, 2 Intel Xeon Core2Quad packages	16-cores, 4 AMD Barcelona packages
Frequency	2 GHz	2.2 GHz
L1 cache	32 KB Private	32 KB Private
L2 cache	Shared; 6MB/2-cores	Private; 512KB/core
L3 cache	None	Shared; 8MB/4-cores
DRAM	8 GB	16 GB
OS	Linux CentOS 5	Linux CentOS 5

Workloads: We use 9 workloads from various domains in our evaluation (including 2 from PARSEC benchmark suite [15][2]). Table 7.7 describes each workload and its input set. `MCarlo`, `BScholes`, `mtwister`, and `pagemine` were modified from original code to execute in pipeline fashion.

Measurements: We run all benchmarks to completion and measure the overall execution time of each workload using the GNU time utility. To measure the fine-grained timings, such as, spent inside a particular section of a program, we use the read timestamp-counter instruction (`rdtsc`). We compute the average number of active cores by counting the number of cores that are active at a given time

[2]The remaining PARSEC workloads are data-parallel (not pipelined) and FDP does not increase or decrease their performance

and averaging this value over the entire execution time. We run each experiment multiple times and use the average to reduce the effect of OS interference.

Table 6.2: Workload characteristics.

Workload	Description (No. of pipeline stages)	Input
MCarlo	MonteCarlo simulation of stock options [74] **(6)**	N=400K
compress	File compression using bzip2 algorithm [47] **(5)**	4MB text file
BScholes	BlackScholes Financial Kernel [74] **(6)**	1M opts
pagemine	Derived from rsearchk[70] and computes a histogram **(7)**	1M pages
image	Coverts an RGB image to gray-scale **(5)**	100M pixels
mtwister	Mersenne-Twister PRNG [74] **(5)**	path=200M
rank	Rank strings based on their similarity to an input string **(3)**	800K strings
ferret	Content based similarity search from PARSEC suite[15] **(8)**	simlarge
dedup	Data stream compression using deduplication algorithm from PARSEC suite[15] **(7)**	simlarge

Evaluated Schemes: We evaluate FDP in terms of performance, power consumption, and robustness. We evaluate three core-to-stage allocation schemes. First, the *One Core Per Stage (1CorePS)* scheme which allocates one core to each stage. Second, the *Proportional Core Allocation (Prop)* scheme which allocates cores to stages based on their relative execution rates. Prop runs the application once with 1CorePS and calculates the throughput of each stage. The cores are then allocated in inverse proportion to the throughput of each stage, thus giving more cores to slower stages and vice versa. Third, the *Profile-Based* scheme which allocates cores using static profiling. The Profile-Based scheme runs the program for all possible allocations which assign an *integer* number of cores to each stage and chooses the allocation which minimizes execution time. Note that while the absolute best profile algorithm can try even non-integer allocations by allowing stages to share cores, the number of combinations with such an approach quickly approaches into millions, which makes it impractical for us to quantitatively evaluate such a scheme.

6.6.1.2 Case Studies

FDP optimizes performance as well as power for pipelined workloads at runtime. We now show the working of FDP on both scalable and non-scalable

workloads with the help of in-depth case studies that provide insights on how FDP optimizes execution.

Scalable Workload: Compress

The workload `compress` implements a parallel pipelined bzip2 compression algorithm. It takes a file as input, compresses it, and writes the output to a file. To increase concurrency, it divides the input file into equal size blocks and compresses them independently. It allocates the storage for the compressed and uncompressed data, reads a block from the file, compresses the block, re-order any work quanta which may have become out of order, writes the compressed block to the output file, and deallocates the buffers. Figure 2.8 shows the pipeline of `compress`. Each iteration in `compress` has 5 stages(S1-S5). Each stage can execute concurrently on separate cores, thereby improving performance.

Table 6.3 shows the throughput of each stage when each stage is allocated one core (the allocation 1-1-1-1-1). The throughput of stage S3, which compresses the block, is significantly lower than the other stages. Thus, the overall performance is dominated by S3 (the LIMITER stage). Table 6.3 also shows the throughput when one of the stage receives four cores and all other receive one core. For example, with the 4-1-1-1-1 allocation S1 receives four cores and all other stages get one core. Threads in S1 allocate buffers in the shared heap and contend for the memory allocator, thereby loosing concurrency, hence throughput of S1 improves by only 2.4x with 4x the cores. Whereas, when 4 cores are given to Stage S3, its throughput improves almost linearly by 3.9x because S3 compresses independent blocks without requiring any thread communication.

Table 6.3 also shows the overall execution time with different core allocations. As S3 is the LIMITER stage, increasing the number of cores for other stages does not help reduce the overall execution time. However, when S3 receives more cores, the throughput of S3 increases by 3.9x and overall execution time reduces form 55 seconds to 14 seconds (a speedup of 3.9x). Therefore, to improve performance more execution resources must be invested in the LIMITER stage.

Table 6.3: Throughput of different stages as core allocation is varied. Throughput is measured as iterations/1M cycles.

Core Alloc.	S1	S2	S3	S4	S5	Exec. Time
1-1-1-1-1	284	49	0.4	34	8K	55 sec.
4-1-1-1-1	698	44	0.4	33	6K	55 sec.
1-4-1-1-1	294	172	0.4	35	7K	55 sec.
1-1-4-1-1	304	52	**1.5**	37	7K	**14 sec.**
1-1-1-4-1	279	49	0.4	135	8K	55 sec.
1-1-1-1-4	282	51	0.4	33	31K	55 sec.

We modify the source code of `compress` to include library calls to FDP. FDP measures the throughput of each stage at runtime and regulates the core-to-stage allocation to maximize performance and power-efficiency. Figure 6.5 shows the overall throughput as FDP adjusts the core-to-stage allocation.

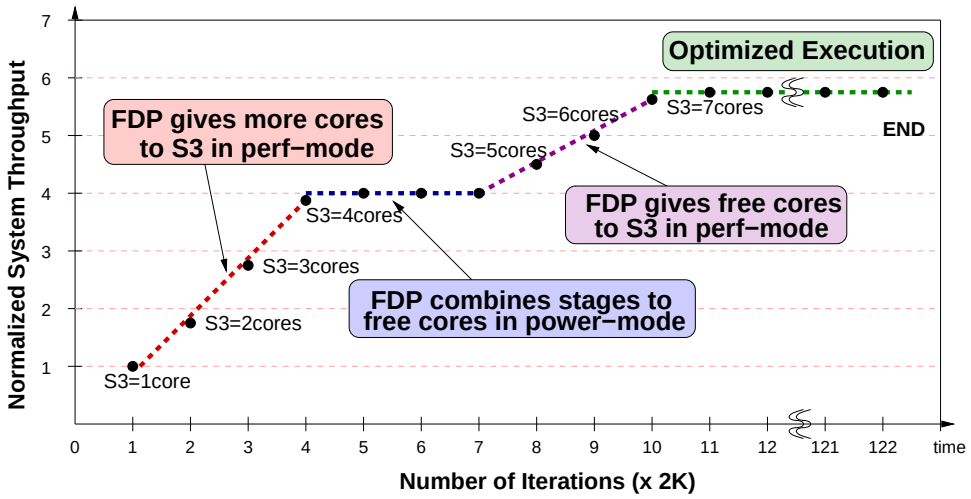


Figure 6.5: Overall throughput of `compress` as FDP adjusts core-to-stage allocation

FDP initially allocates one core to each stage. As execution continues, FDP trains and identifies S3 to be the LIMITER stage. To improve performance FDP increases the number of cores allocated to S3, until it runs out of cores. For our 8-core system, this happens when S3 is allocated 4 cores, and the remaining 4 cores are allocated one each to S1, S2, S4, and S5. After it runs out of cores, FDP begins to operate in power-optimization mode. In the first invocation of this mode, the

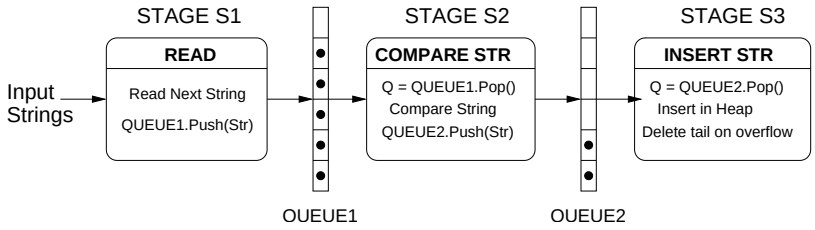


Figure 6.6: Pipeline for matching a stream of strings with a given string

stages with the highest throughput, S1 and S5, are combined to execute on a single core, thereby freeing one core. In the next invocation, FDP combines S1 and S5 with S2 which frees up another core. FDP continues this until all four stages S1, S2, S4, and S5 get combined to execute on a single core. With no opportunity left to reduce power, FDP switches back to performance optimization mode. FDP again identifies S3 as the LIMITER and allocates the 3 free cores to S3. Thus, 7 out of the 8 cores are allocated to S3, and a single core is shared among all other stages. FDP converges in 10 invocations and executes the workload in 9.7 seconds, which is much lower than with the static-best integer allocation (1-1-4-1-1) that requires 14 seconds.

Non-Scalable Workload: Rank

The `rank` program ranks a list of strings based on their similarity to an input string. It returns the top N closest matches (N is 128 in our experiments). Figure 6.6 shows the pipelined implementation for `rank`. Each iteration is divided into 3 stages. The first stage (S1) reads the next string to be processed. The second stage (S2) performs the string comparison, and the final stage (S3) inserts the similarity metric in a sorted heap, and removes the smallest element from the heap (except when heap size is less than N). At the end of the execution, the sorted heap contains the top N closest matches.

Table 6.4 shows the throughput of system when each stage is allocated one core (1-1-1). The throughput of S2, which performs the string comparison, is significantly lower than the other stages in the pipeline. As S2 is the LIMITER, allocating more cores to S2 is likely to improve overall performance. The next three rows in

the table shows the throughput when one of the stage receives 4 cores and the other stages get one core. With the increased core count, S1 and S3 show a speedup of 2.5x and 1.3x, respectively. However, as these stages are not the LIMITER, the overall execution time does not decrease.

Table 6.4: Throughput of different stages as core allocation is varied (measured as iterations/1M cycles).

Core Alloc.	S1	S2	S3	Exec. Time
1-1-1	1116	142	236	17 sec
4-1-1	2523	118	258	19 sec
1-4-1	1005	558	278	13.2 sec
1-1-4	900	117	290	19.2 sec
1-4-2	930	368	285	14.6 sec
1-2-1	1028	274	268	13 sec

When S2 is allocated 4 cores, it shows a speedup of approximately 4x. This is because all cores in S2 work independently without requiring communication. Unfortunately, the overall execution time reduces by only 27%. This is because as S2 scales, its throughput surpasses the throughput of S3. Thus, S3 becomes the LIMITER. Once S3 becomes the LIMITER, the overall execution time is dominated by S3.

As S3 is the LIMITER, we expect to improve overall performance by increasing cores allocated to S3. The table also shows the throughput when additional cores are allocated to S3 (1-4-2). The access to the shared linked data-structure in S3 is protected by a critical section, hence this stage is not scalable and overall performance reduces as the number of cores is increased due to contention for shared data. Thus, increasing core counts for S3 does not help improve performance while consuming increased power.

We modify the source code of `rank` to include library calls to FDP. Figure 6.7 shows the overall throughput and active cores as FDP adjusts the core-to-stage allocation. With the information obtained during training, FDP identifies S2 as the LIMITER stage, and allocates it one extra core (1-2-1). In the next invocation, it

identifies S3 as the LIMITER stage, and increases the core count allocated to S3 (1-2-2). However, as S3 does not scale, FDP withdraws the extra core given to S3, and switches to power-optimization mode. In power-optimization mode, FDP saves power by executing S1 on one of the cores allocated to S2. Thus, the final allocation is S1+S2 on one core, S2 on another core, and S3 on the third core. After this, there are no opportunities left in the pipeline to save power or improve performance, and execution continues on 3 cores completing in 13 seconds (similar to best-static allocation 1-2-1, but with fewer cores).

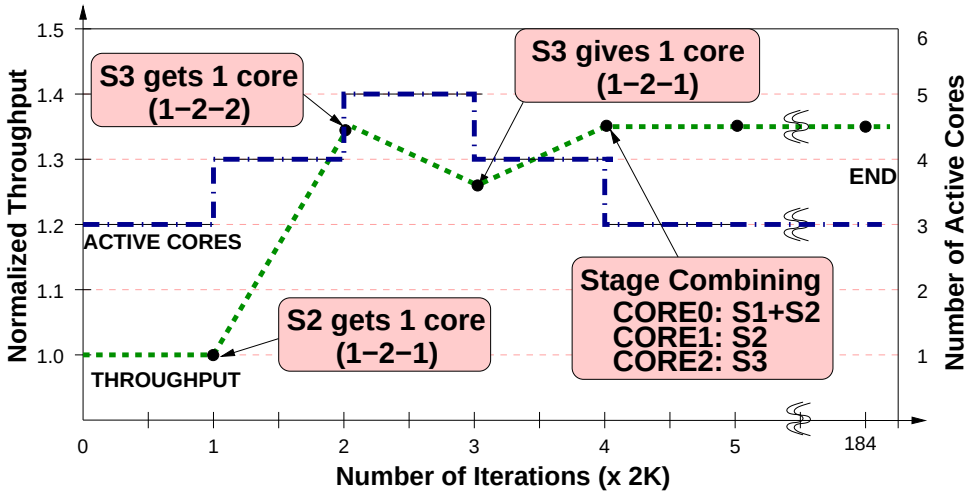


Figure 6.7: Overall throughput and active cores of `rank` as FDP adjusts core-to-stage allocation

6.6.1.3 Performance

Figure 6.8 shows the speed-up when the workloads are executed with the core-to-stage allocation using 1CorePS, Prop, FDP, and Profile-Based. The speedup is relative to execution time with a single core system [3]. The bar labeled *Gmean* is the geometric mean over all workloads. The 1CorePS scheme provides only a marginal improvement, providing minor speedup increase on four out of seven workloads. On the contrary, a Profile-Based allocation significantly improves per-

[3]We run the sequential version without any overheads of multi-threading.

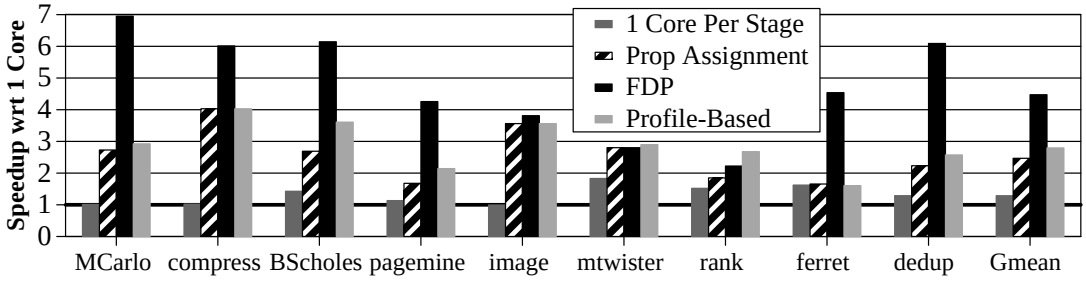


Figure 6.8: Speedup with different core-to-stage allocation schemes.

formance for all workloads, providing an average speedup of 2.86x. However, Profile-Based requires impractical searching through all possible integer allocations. Prop avoids this brute force searching and gets an improvement similar to Profile-Based by providing an average speedup of 2.7x. FDP outperforms or is similar to the comparative schemes on all workloads. `MCarlo` gets near optimal speedup of 7x with FDP because it contains a scalable LIMITER stage and FDP combines all other stages. The workload `rank` has a stage that is not scalable, hence the limited performance improvement with all schemes. FDP provides an average speedup of 4.3x. Note, that this significant improvement in performance comes without any reliance on profile information which is required for both Prop and Profile-Based.

6.6.1.4 Number of Active Cores

FDP tries to increase performance by taking core resources from faster stages and reallocating it to slower stages. When the slowest stage no longer scales with additional cores, the spare cores can be turned off or used for other applications. Figure 6.9 shows the average number of active cores during the execution of the program for 1CorePS, FDP, and Prop/Profile-Based. Both Prop and Profile-Based allocates all the cores in the system, therefore they are shown with the same bar. The bar labeled *Amean* denotes the arithmetic mean over all the workloads.

The number of active cores with the 1CorePS is equal to the number of pipeline stages, which has an average of 5.2 cores. The Prop and Profile-Based

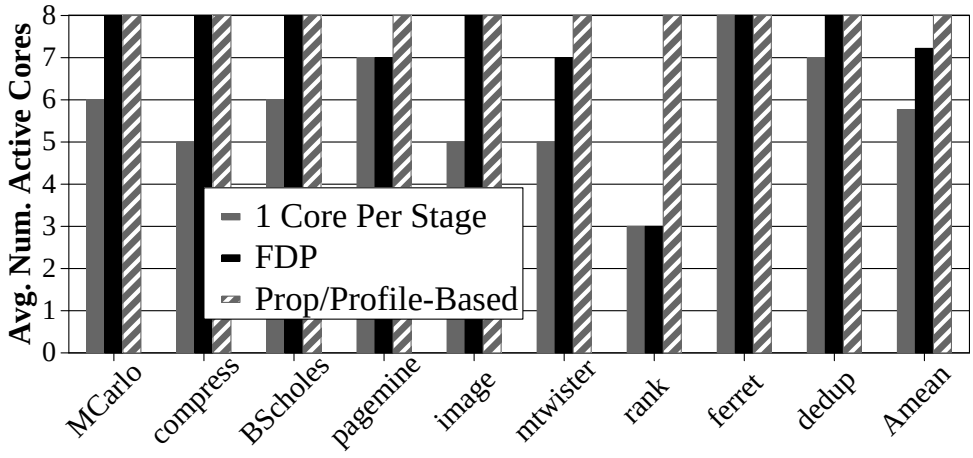


Figure 6.9: Average number of active cores for different core allocation schemes.

schemes use 8 cores. For `Pagemine` and `mtwister`, the performance saturates at 7 cores, so FDP does not use one of the cores in the system. For the workload `rank`, the non-scalable stage means that five out of the eight cores can be turned off. Thus, FDP is not only a performance enhancing technique but also helps with reducing the power consumed by cores when it is not possible to improve performance with more cores. On average, FDP consumes only 7 cores even though it has one and a half times the speedup of the Profile-Based scheme. This means for the same number of active cores, FDP consumes two-thirds the energy as the Profile-Based scheme and has a much reduced energy-delay product.

6.6.1.5 Robustness to Input Set

The best core-to-stage allocation can vary with the input set. Therefore, the decisions based on profile information of one input set may not provide improvements on other input set. To explain this phenomenon, we conduct experiments for the `compress` workload with two additional input sets that are hard to compress. We call these workloads `compress-2` and `compress-3`. The LIMITER stage S3 for `compress-2` (80K cycles) and for `compress-3` (140K cycles) is much smaller than the one used in our studies (2.2M cycles). The non-scalable stage that writes to the output file remains close to 80K cycles in all cases. Thus, the

`compress` workload has limited scalability for the newly added input sets.

Figure 6.10 shows the speedup for the two workloads with 1CorePS, Prop, FDP and Profile-Based. Both Prop and Profile-Based use the decisions made in our original `compress` workload. These decisions in fact result in worse performance than 1CorePS for `compress-2`, because they allocate more cores to the non-scalable stage which results in increased contention. FDP, on the other hand, does not rely on any profile information and allocates only one-core to the non-scalable stage. It allocates two cores to S3 for `compress-2` and 3 cores to S3 for `compress-3`. The runtime adaptation allows FDP to outperform all comparative schemes on all the input sets.

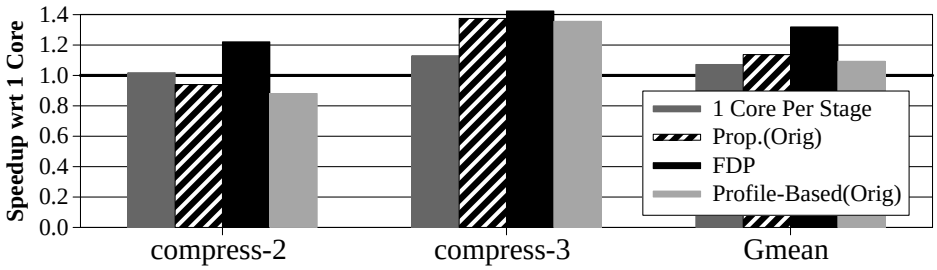


Figure 6.10: Robustness to variations in input set.

6.6.1.6 Scalability to Larger Systems

We use an 8-core machine as our baseline for evaluations. In this section, we analyze the robustness and scalability of FDP to larger systems, using a 16-core AMD Barcelona machine. We do not show results for 1CorePS as they are similar to the 8-core system (all workloads have fewer than 8 stages). Furthermore, a 16-core machine can be allocated to a 6-7 stage pipeline in several thousand ways, which makes evaluating Profile-Based impractical.

Figure 6.11 shows the speedup of Prop and FDP compared to a single core on the Barcelona machine. FDP improves performance of *all* workloads compared to Prop. Most notably, in `image`, FDP obtains almost twice the improvement of Prop. The scalable part of `image`, which transforms blocks of the image from

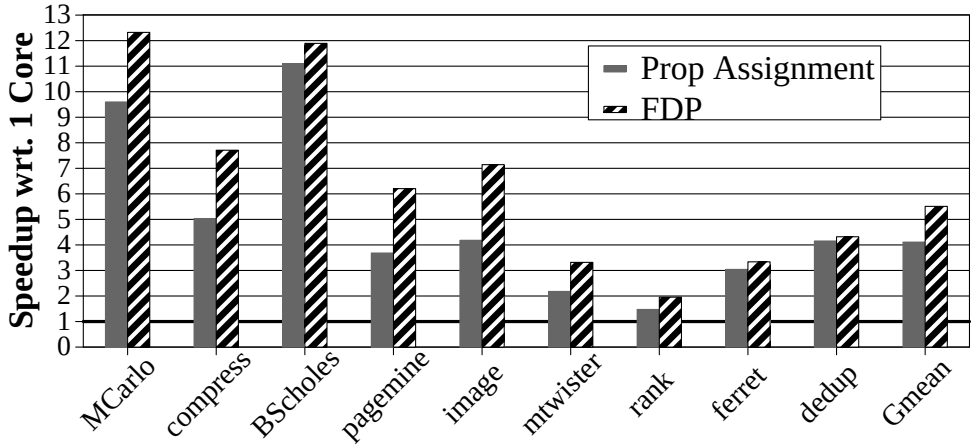


Figure 6.11: FDP's performance on 16-core Barcelona.

colored to gray scale, continues to scale until 6 cores. The other parts, reading and writing from the file, do not scale. Prop allocates cores to each stage proportionally assuming equal scaling. However, the cores allocated to non-scalable parts do not contribute to performance. FDP avoids such futile allocations. On average, FDP provides a speedup of 6.13x compared to 4.3x with Prop.

As the number of cores increases, the performance of some of the workloads starts to saturate. Under such scenarios, there is no room to improve performance but there is a lot of potential to save power. Figure 6.12 shows the average number of active cores during the workload execution with FDP and Prop. Since Prop allocates all cores, the average for Prop is 16. When cores do not contribute to performance FDP can deallocate them, thereby saving power. For example, `pagemine` contains four stages in the pipeline that do not scale because of critical sections. FDP allocates 7 cores to the scalable stage, 1 core each to the non-scalable stages, and 1 more core to the input stage. The remaining four cores remain unallocated. On average, FDP has 11.5 cores active, which means a core power reduction of more than 25%. Thus FDP not only improves overall performance significantly but can also save power.

If all cores were active, then the energy consumed by FDP would be 30% less compared to Prop (measured by relative execution time). Given that FDP uses

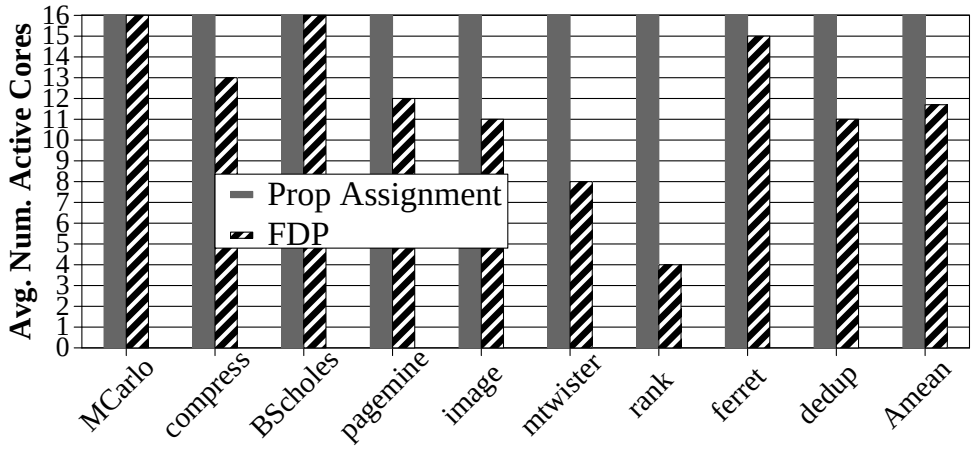


Figure 6.12: FDP's power on 16-core Barcelona.

25% fewer cores than Prop, FDP consumes less than half the energy consumed by Prop. Thus, FDP is an energy-efficient high-performance framework for implementing pipelined programs.

6.6.2 Evaluation of ALS

We now evaluate how well ALS can accelerate the limiter stage.

6.6.2.1 Evaluation Methodology

We simulate two configurations: Baseline (a baseline SCMP), and an ACMP. The parameters of each small core, the interconnect, caches, and the memory sub-system are shown in Table 6.5.

Table 6.6 shows the simulated workloads. We developed a pipelined implementation of the dedup decoder based on the sequential code in PARSEC [15] and call it `dedupD`.

6.6.2.2 Performance at One Core per Stage

We first compare SCMP and ACMP when only one core is assigned to each stage. SCMP assigns a small core to each stage while the ACMP assigns the large core to the limiter stage (the stage with the longest execution time) and a small core

Table 6.5: Configuration of the simulated machines.

Small core	2-wide, 2GHz, 5-stage, in-order
Large core	4-wide, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, out-of-order; 4x the area of small core
Interconnect	64-byte wide bi-directional ring, all queuing delays modeled, minimum cache-to-cache latency of 5 cycles
Coherence	MESI on-chip distributed directory similar to SGI Origin [59], cache-to-cache transfers, # of banks = # of cores, 8K entries/bank
Prefetcher	Aggressive stream prefetcher [99] with 32 stream buffers, can stay 16-lines ahead, prefetches into cores' L2 caches
Caches	Private L1I and L1D: 32KB, write-through, 1-cycle, 4-way. Private L2: 256KB, write-back, 6-cycle, 8-way (1MB, 8-cycle, 16-way for large core). Shared L3: 8MB, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Precharge, activate, column access latencies are 25ns each
Memory bus	4:1 CPU/bus ratio, 64-bit wide, split-transaction
Area-equivalent CMPs. Area is equal to N small cores. We vary N from 1 to 64.	
SCMP	N small cores. Core-to-stage allocation chosen using FDP.
ACMP	1 large core and N-4 small cores; Core-to-stage allocation chosen using FDP; large core runs the limiter stage

Table 6.6: Workload characteristics.

Workload	**Description (No. of pipeline stages)**	**Major steps of computation**	**Input**
black	BlackScholes Financial Kernel [74] **(6)**	Compute each option's call/put value	1M opts
compress	File compression using bzip2 algorithm **(5)**	Read file, compress, re-order, write	4MB text file
dedupE	De-duplication (Encoding) [15] **(7)**	Read, find anchors, chunk, compress, write	simlarge
dedupD	De-duplication (Decoding) [15] **(7)**	Read, decompress, check cache, reassemble/write	simlarge
ferret	Content based search [15] **(8)**	Load, segment, extract, vector, rank, out	simlarge
image	Image conversion from RGB to gray-scale **(5)**	Read file, convert, re-order, write	100M pixels
mtwist	Mersenne-Twister PRNG [74] **(5)**	Read seeds, generate PRNs, box-muller	path=200M
rank	Rank string similarity with an input string **(3)**	Read string, compare, rank	800K strings
sign	Compute the signature of a page of text **(7)**	Read page and compute signature	1M pages

to each of the remaining stages. Figure 6.13 shows the execution time of ACMP (with ALS) normalized to the execution time of SCMP. Note that this is not an equal-area comparison as the ACMP's large core occupies the area of four small cores.

ACMP significantly reduces the execution time of all workloads except

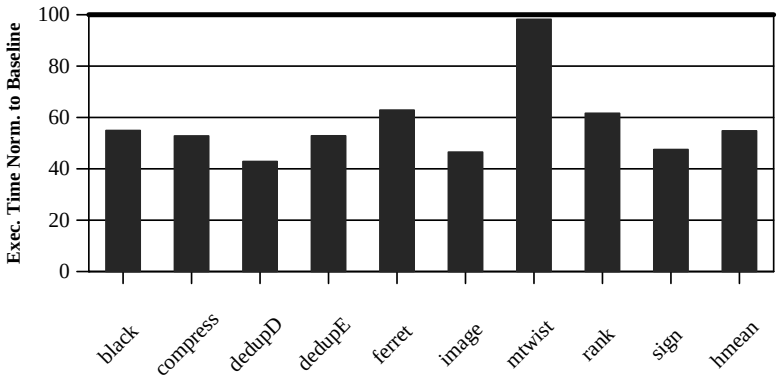


Figure 6.13: Speedup at 1 Core Per Stage (not area-equivalent).

`mtwist` where ACMP's improvement is only marginal. `mtwist` contains two stages, S3 and S4, which have almost the same average execution time (within 3% of each other) . ALS accelerates only S3 which makes S4 the limiter stage and the overall performance stays similar to the baseline. In all other workloads, ACMP reduces execution time significantly. For example, ACMP reduces the execution time of `black` by 44% because the limiter stage in `black` consists of a regular loop with large amounts of memory level parallelism (MLP). When `black's` limiters runs at the large core, the large core is able to exploit this MLP (due to its out of orderness) and accelerate the stage significantly. This translates into very high overall performance for ACMP.

6.6.2.3 Performance at Best Core-to-stage Allocation

We now compare ACMP and SCMP where FDP was used to choose the best core-to-stage allocation for each application-configuration pair. The comparisons are done at an equal-area budget of 32-cores.

Figure 6.14 shows the execution time of ACMP normalized to the execution time of SCMP. ACMP significantly reduces the execution time of seven out of the nine workloads. For example, in `sign`, ACMP reduces execution time by 20%. `sign` consists of four stages of computation: a stage to allocate memory (S1), a stage to compute signatures of input pages of text (S2), a stage to process the

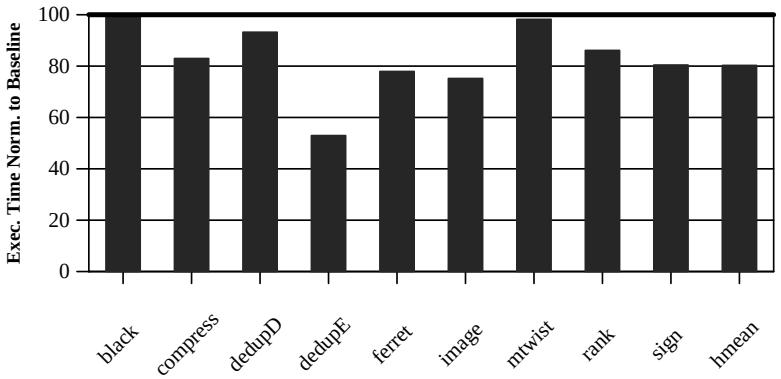


Figure 6.14: Speedup with FDP at an equal area budget of 32 small cores.

signatures (S3), and a stage to deallocate memory resources. S2 is compute-bound while S3 is dominated by a critical section which inhibits its ability to scale with the number of cores. At one core per stage, S2 has the longest execution time and is thus the limiter. As the the number of cores increase, FDP assigns more and more cores to S2. When S2 has been assigned four cores, S3 becomes the limiter. Since S3 does not scale, SCMP's performance saturates. However, for ACMP, FDP then assigns S3 to the large core. Accelerating S3 again makes S2 the limiter which allows the workload to leverage another small core, thereby increasing the overall performance by 20%.

Why does ACMP not benefit `black` and `mtwist`? The dominant stages in `black` and `mtwist` scale with the number of cores. Since the limiter stages scale, they only marginally benefit from the large core in the ACMP. To provide more insights, we further analyzed `black`. `black` has six stages (S0-S6). S0 and S5 perform memory allocation and deallocation respectively. S1-S4 perform different steps of computation and are very scalable. FDP continues to assign more and more cores to the scalable stages. For the 32-core SCMP, FDP assigns S0-S5 one, two, eight, nine, eight, and four cores respectively. We also find that the stages S0, S2, and S3 have approximately the same throughput. Thus, accelerating any one of them using the large cores does not improve performance. We find that as the area budget increases, more cores are assigned to the scalable stages S2 and S3 which makes S0 the limiter stage, thereby providing opportunity for ACMP to

improve performance via accelerating S0.

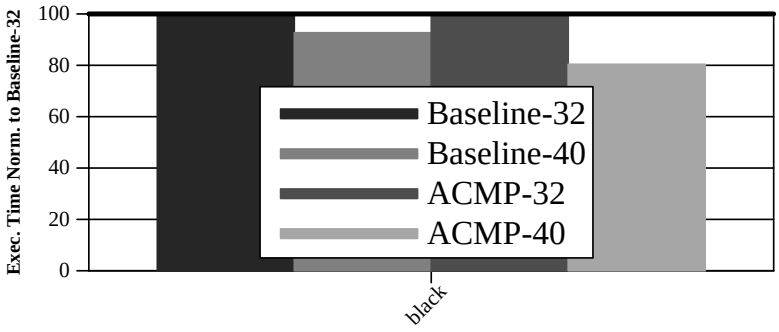


Figure 6.15: Performance of ACMP for `black` at an area budget of 40.

To analyze this effect, we also compare `black's` performance on an ACMP and SCMP at an area budget of 40. Figure 6.15 shows execution time of black on four configurations: the baseline SCMP with an area budget of 32 (baseline-32), the baseline SCMP with an area budget of 40 (baseline-40), ACMP with an area budget of 32 (ACMP-32), and ACMP with an area budget of 40 (ACMP-40). Note that the execution times of Baseline-32 and Baseline-40 are nearly equal. However, ACMP-40 is 20% faster than both SCMP-32 and SCMP-40 and 15% faster than the ACMP-32. We conclude that ACMP increases performance as well as scalability. Furthermore, its benefit will further increase as the chip area increases, and pipeline programs becomes limited by non-scalable stages.

Chapter 7

Data Marshaling

7.1 The Problem

Taking advantage of the ACMP requires shipping work to the large core whenever it is beneficial to run it at the large core. However, when a work-quanta executes at the large core, it incurs cache misses in fetching its working set from the small core. This lowers the IPC of the large core, thereby reducing the benefit of sending the work to the large core. To this end, we propose *Data Marshaling (DM)* which identifies the data needed by the large core and proactively transfers it to the cache of the large core from the cache of the small core. DM is also beneficial for pipeline parallelism where each stage runs on a different core and the input data for the stage has to be transferred from the core executing the previous stage.

We first explain a general implementation of DM and then explain the specifics which make it applicable to the ACMP. To design a general DM, we first generalize the problem by introducing an abstraction layer called *staged execution*.

Staged Execution The central idea of Staged Execution (SE) is to split a program into code *segments* and execute each segment where it runs the best. If more than one cores are equally suited for a segment, different instances of the segment can run on any one of those cores. Thus, the **home core** of an instance of a segment is the core where that particular instance executes. The criteria commonly used to choose a segment's home core include performance criticality, functionality, and data requirements, e.g. critical sections which are on the critical path of the program are best run at the fastest core. Other mechanisms to choose the home core are beyond the scope of this thesis and are discussed in other work on SE [18, 21, 32, 96].

Implementation: Each core is assigned a work-queue, which can be implemented as a hardware or a software structure. The work-queue stores the execution requests to be processed by the corresponding core. Each entry in the queue contains an identifier of the segment to be processed. The identifier can be a segment ID, the address of the first instruction in the segment, or a pointer to a function object. Each core continuously runs in a loop, processing segments from its work-queue. Every time the core completes a segment, it dequeues the next entry from the work-queue and processes the segment referenced by that entry. If the work-queue is empty, the core waits until a request is enqueued or the program finishes.

The program is divided into *segments*, where each segment is best suited for a different core. Each segment, except the first, has a *previous-segment* (the segment that executed before it) and a *next-segment* (the segment that will execute after it).

At the end of each segment is an *initiation routine* for its next-segment. The **initiation routine** performs two tasks. First, it chooses the home core of the next-segment from one of the cores assigned to the nest-segment. Second, it enqueues a request for execution of the next-segment at the chosen home core. We call the core that runs the initiation routine to request a segment's execution the segment's *requesting core*.

We now explain SE using an example. Figure 7.1a shows a code example that computes the values of X, Y, and Z. The code can be divided into three segments: A, B, and C (shown in Figure 7.1b). Segment A is Segment B's previous-segment and Segment C is Segment B's next-segment. At the end of Segment A is Segment B's initiation routine and at the end of Segment B is Segment C's initiation routine. Figure 7.1c shows the execution of this code on a CMP with three cores (P0-P2) with the assumption that the home cores of Segments A, B and C are P0, P1, and P2 respectively. After completing Segment A, P0 runs Segment B's initiation routine, which inserts a request for the execution of Segment B in P1's work-queue. Thus, P0 is Segment B's requesting core. P1 dequeues the entry,

executes segment B, and enqueues a request for Segment C in P2's work-queue. Processing at P2 is similar to the processing at P0 and P1.

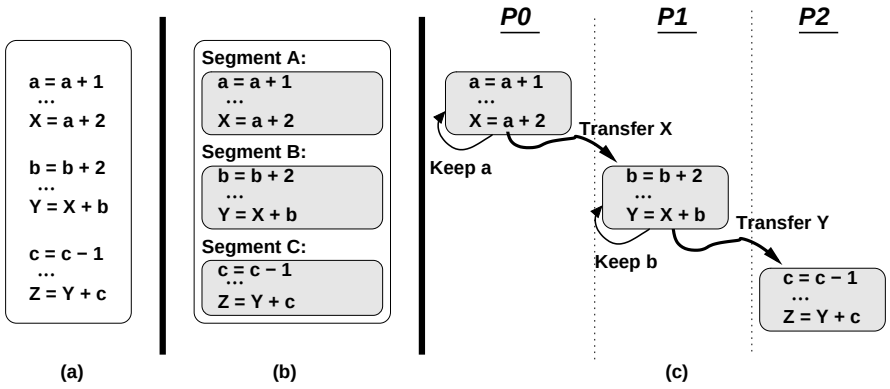


Figure 7.1: (a) Source code, (b) code segments, and (c) their execution in SE.

.

We define the **Inter-Segment Data** of a segment as *the data the segment requires from its previous segment*. For example, Segment B in Figure 7.1 requires the variable X to compute Y. Thus, X is Segment B's inter-segment data. Locality of inter-segment data is very high in models where consecutive segments run on the same core: the first segment generates the data which may remain in the local cache until the next segment can use it. However, in SE, accesses to inter-segment data incur cache misses and the data is transferred from the requesting core to the home core via the cache coherence mechanism. For example, in Figure 7.1c, P1 incurs a cache miss to transfer X from P0's cache. Similarly, P2 incurs a cache miss for Y, which is Segment C's inter-segment data. These cache misses limit SE's performance.

7.2 Mechanism

Data Marshaling (DM) aims to reduce cache misses to inter-segment data. In DM, the requesting core identifies the inter-segment data and marshals it to the home core when it ships a segment to the home core. We first share the key insight that forms the basis for DM.

7.2.1 Key Insight

We first define new terminology. We call the last instruction that modifies the inter-segment data in a segment a *generator*. For example, consider the code in Figure 7.2 with two segments: S1 and S2. In this case, A is S2's inter-segment data: it is written by an instruction in S1, the `STORE` on line 2, and read by an instruction in S2, the LOAD on line 4. Thus, the *STORE* on line 2 is A's generator since there are no other stores to A between this STORE and the initiation of Segment S2. The LOAD on line 1 is *not* a generator because it does not modify A and the STORE on line 0 is *not* a generator since there is a later STORE (the one on line 2) to A. We generalize our definition of generators to cache lines by redefining a generator as *the last instruction that modifies a cache line containing inter-segment data before the next segment begins.*

```
    Segment S1:          ; Previous-segment of S2
0:  STORE A
1:  LOAD A
2:  STORE A              ; A's generator
3:  CALL Initiation(S2)

    Segment S2:          ; Currently executing segment
4:  LOAD A               ; Inter-segment data access
5:  ..
6:  CALL Initiation(S3)
```

Figure 7.2: Concept of "generator of inter-segment data".

We call the set of all generator instructions, i.e. generators of all the cache lines containing inter-segment data, the *generator-set*. We observed that the generator-set of a program is often small and does not vary during the execution of the program and across input sets (See Section 7.3.4.1 for details). This implies that any instruction that has once been identified as a generator stays a generator. Thus, a cache line written by a generator is very likely to be inter-segment data required by the following segment, hence, a good candidate for data marshaling. Based on this observation, we assume that *every cache line written by an instruction in the generator-set is inter-segment data and will be needed for the execution of the next*

segment. We call the set of inter-segment data cache lines generated by a segment its *marshal-set*. DM adds all cache lines written by generators to the marshal-set. When a core finishes the initiation routine of the next segment, all cache lines that belong to the marshal-set are transferred to the next-segment's home core.

We only marshal data between consecutive segments for two reasons: 1) it achieves most of the potential benefit since we measure that 68% of the data required by a segment is written by the immediately preceding segment, and 2) in most execution paradigms, the requesting core only knows where the next segment will run, but not where the subsequent segments will run. Thus, marshaling data to non-consecutive segments requires a substantially complicated mechanism. Efficient mechanisms for marshaling data to non-consecutive segments is a part of our future work.

7.2.2 Overview of the Architecture

The functionality of DM can be partitioned into three distinct parts:

Identifying the generator-set: DM identifies the generator-set at compile-time using profiling. We define the *last-writer* of a cache line to be the last instruction that modified a cache line. Thus, *a line is inter-segment data if it is accessed inside a segment but its last-writer is a previous segment instruction*. Since the generator-set is stable, we assume that last-writers of all inter-segment data are generators. Thus, every time DM detects an inter-segment cache line, it adds the cache line's last-writer to the generator-set (unless it is already in the generator-set). The compiler conveys the identified generator-set to the hardware using new ISA semantics.

Recording of inter-segment data: Every time an instruction in the generator-set is executed, its destination cache line is predicted to be inter-segment data and added to the marshal-set.

Marshaling of inter-segment data: All elements of the marshal-set are transferred, i.e. marshaled, to the home core of the next-segment as soon as the

On every memory access:
If cache-line not present in segment's accessed lines and last-writer is from previous segment
Add last-writer to generator-set

On every store:
Save address of store in current-last-writer

At the end of segment:
Deallocate previous-last-writer
current-last-writer becomes previous-last-writer
Allocate and initialize current-last-writer

Figure 7.3: The profiling algorithm.

execution request is sent to the next-segment's home core.

For example, DM for the code shown in Figure 7.2 works as follows. DM detects A to be inter-segment data, identifies the STORE on line 2 to be A's last-writer, and adds it to the generator-set. When the STORE on line 2 executes again, DM realizes that it is a generator and adds the cache line it modifies to the marshal-set. When the core runs the initiation routine for S2, DM marshals all cache lines in the marshal-set to S2's home core. Consequently, when S2 executes at its home core, it (very likely) will incur a cache hit for A.

7.2.3 Profiling Algorithm

Our profiling algorithm runs the application as a single thread and instruments all memory instructions.[1] The instrumentation code takes as input the PC-address of the instrumented instruction and the address of the cache line accessed by that instruction. Figure 7.3 shows the profiling algorithm. The algorithm requires three data structures: 1) A *generator-set* that stores the identified generators, 2) A *current-last-writer* table that stores the last-writer of each cache line modified in the current segment, and 3) A *previous-last-writer* table that stores the last-writer of each cache line modified in the previous segment.

[1]We also evaluated a thread-aware version of our profiling mechanism but its results did not differ from the single-threaded version.

For every memory access, the algorithm checks whether or not the line was modified in the previous segment by querying the previous-last-writer table. If the line was not modified in the previous segment, the line is ignored. If the cache line was modified in the previous segment, the last-writer of the line (an instruction in the previous segment) is added to the generator-set. When an instruction modifies a cache line, the profiling algorithm records the instruction as the last-writer of the destination cache line in the current-last-writer table. At the end of each segment, the lookup table of the previous segment is discarded, the current segment lookup table becomes the previous segment lookup table, and a new current segment lookup table is initialized. After the program finishes, the generator-set data structure contains all generators.

7.2.4 ISA Support

DM adds two features to the ISA: a GENERATOR prefix and a MARSHAL instruction. The compiler marks all generator instructions by prepending them with the GENERATOR prefix. The purpose of the MARSHAL instruction is to inform the hardware that a new segment is being initiated and provide the hardware with the ID of the home core of the next segment. The instruction takes the home core ID of the next-segment as its only argument. When the MARSHAL instruction executes, the hardware begins to marshal all cache lines in the marshal-set to the core specified by HOME-CORE-ID. We discuss the overhead of these changes in Section 7.2.9.

7.2.5 Library Support

DM requires the initiation routines to execute a `MARSHAL` instruction with the ID of the core to which the segment is being shipped. Since initiation routines are commonly a part of the library or run-time engine, the library or the run-time that decides which core to ship a task to is modified to execute the `MARSHAL` instruction with the correct HOME-CORE-ID.

7.2.6 Data Marshaling Unit

Each core is augmented with a Data Marshaling Unit (DMU), which is in-charge of tracking and marshaling the inter-segment data to the home core. Figure 7.4(a) shows how the core and the DMU are integrated. Figure 7.4(b) shows the microarchitecture of the DMU. Its main component is a *Marshal Buffer* to record the addresses of all cache lines to be marshaled. Figure 7.4(c) shows the operation of the DMU. When the core retires an instruction with the GENERATOR prefix, it sends the physical cache line address written by the instruction to the DMU. The DMU enqueues the address in the Marshal Buffer. The Marshal Buffer is combining, i.e. multiple accesses to the same cache line are combined into a single entry, and circular, i.e. if it becomes full its oldest entry is replaced with the incoming entry.

When the core executes the MARSHAL instruction, it asserts the MARSHAL signal and sends the HOME-CORE-ID to the DMU. The DMU starts marshaling data to the home core. For each line address in the Marshal Buffer, the DMU accesses the local L2 cache to read the coherence state and data. If the line is in shared state or if a cache miss occurs, the DMU skips that line. If the line is in exclusive or modified state, the DMU puts the line in a temporary state that makes it inaccessible (similar to [68]) and sends a *DM Transaction* (see Section 7.2.7) containing the cache line to the home core. The home core installs the marshaled line in its fill buffer and responds with an ACK, signaling the requesting core to invalidate the cache line. If the fill buffer is full, the home core responds with a NACK, signaling the requesting core to restore the original state of the cache line.

Note that we marshal the cache lines to the private L2 cache of the home core. Marshaling the cache lines into the L2, instead of L1, has the advantage that DM does not contend with the home core for cache bandwidth. Moreover, since L2 is bigger than L1, cache pollution is less likely. However, marshaling into L2 implies that the core will incur an L1 miss for inter-segment data cache lines. We found that the benefit of marshaling into the L2 outweighs the overhead

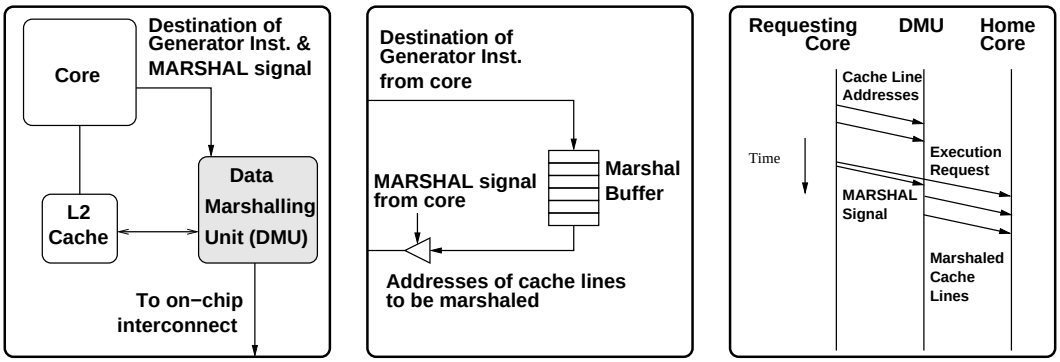


Figure 7.4: Data Marshaling Unit.

of marshaling into the L1.

7.2.7 Modifications to the On-Chip Interconnect

The interconnect supports one new *DM transaction*, which is sent by the DMU to a remote core. Each DM transaction is a regular point-to-point message containing the address and data of a marshaled cache line. Thus, it requires exactly the same interconnect support (wires, routing logic, etc.) of a cache line fill during a cache-to-cache transfer, which is supported by existing CMPs.

7.2.8 Handling Interrupts and Exceptions

The virtual-to-physical address mapping can change on an interrupt or an exception such as a page fault or expiration of the operating system time quantum. In such a case, the contents of the Marshal Buffer may become invalid. Since DM is used solely for performance and not for correctness, we simplify the design by flushing the contents of the Marshal Buffer every time an interrupt or exception occurs.

7.2.9 Overhead

DM has the potential to improve performance significantly by reducing inter-segment cache misses. However, it also incurs some overhead:

- DM adds a GENERATOR prefix to all generator instructions. This has the potential to increase I-cache footprint. However, we find that the generator-set of each application is small: 55 instructions on average. Thus, adding the GENERATOR prefixes increases the I-cache footprint only marginally.
- DM adds a MARSHAL instruction at the end of each segment. The overhead of the MARSHAL instruction is low as it does not read/write any data. The overhead is further amortized as it executes only once per segment, which often consists of thousands of instructions.
- The DMU can contend with the core on L2 cache accesses. However, this overhead is no different from the baseline, where the cache-to-cache transfer requests for the inter-segment data contend for the L2 cache.
- DM augments each core with a Marshal Buffer. The storage overhead of this buffer is small: only 96 bytes/core (16 entries of 6-bytes each to store the physical address of a cache line). We discuss the sensitivity of performance to Marshal Buffer size in Section 7.3.4.8.

In summary, DM's overhead is low, and is outweighed by its benefits. We now discuss how DM can be used by different execution paradigms.

7.3 DM for Accelerated Critical Sections (ACS)

Data Marshaling is beneficial in any execution paradigm that uses the basic principle of Staged Execution. In this section we describe the application of DM to significantly improve Accelerated Critical Sections (ACS).

7.3.1 Private Data in ACS

By executing all critical sections at the same core, ACS keeps the shared data (protected by the critical sections) and the lock variables (protecting the critical section) at the large core, thereby improving locality of shared data. However, every time the critical section accesses data generated outside the critical section (i.e. thread-private data), the large core must incur cache misses to transfer this data

from the small core. Since this data is generated in the non-critical-section segment and accessed in the critical-section segment, we classify it as inter-segment data. Marshaling this data from the small core to the large core can reduce cache misses at the large core. Note that marshaling the inter-segment data generated by the critical-section segment and used by the non-critical-section segment only speeds up the non-critical section code, which is not critical for overall performance. We thus study marshaling private data from the small core to the large core. Since private data and inter-segment data are synonymous in ACS, we use them interchangeably.

7.3.2 Data Marshaling in ACS

Employing DM in ACS requires two changes. First, the compiler must identify the generator instructions by running the profiling algorithm in Section 7.2.3, treating critical sections and the code outside critical sections as two segments. Once the generator-set is known, the compiler prepends the generator instructions with the GENERATOR prefix. Second, the compiler/library must insert the MARSHAL instruction into the initiation routine of every critical-section segment, right before the CSCALL instruction. The argument to the MARSHAL instruction, the core to marshal the data to, is set to the ID of the large core.

7.3.3 Evaluation Methodology

Table 7.1 shows the configuration of the simulated CMPs, using our in-house cycle-level x86 simulator. All cores, both large and small, include a state-of-the-art hardware prefetcher, similar to the one in [99].

Unless specified otherwise: 1) all comparisons are done at equal area budget, equivalent to 16 small cores, 2) the number of threads for each application is set to the number of available contexts.

Workloads: Our evaluation focuses on 12 critical-section-intensive workloads shown in Table 7.2. We define a workload to be critical-section-intensive if at least 1% of the instructions in the parallel portion are executed within critical

Table 7.1: Configuration of the simulated machines.

Small core	2-wide, 2GHz, 5-stage, in-order
Large core	4-wide, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, out-of-order; 4x the area of small core
Interconnect	64-byte wide bi-directional ring, all queuing delays modeled, minimum cache-to-cache latency of 5 cycles
Coherence	MESI on-chip distributed directory similar to SGI Origin [59], cache-to-cache transfers, # of banks = # of cores, 8K entries/bank
Prefetcher	Aggressive stream prefetcher [99] with 32 stream buffers, can stay 16-lines ahead, prefetches into cores' L2 caches
Caches	Private L1I and L1D: 32KB, write-through, 1-cycle, 4-way. Private L2: 256KB, write-back, 6-cycle, 8-way (1MB, 8-cycle, 16-way for large core). Shared L3: 8MB, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Precharge, activate, column access latencies are 25ns each
Memory bus	4:1 CPU/bus ratio, 64-bit wide, split-transaction
Area-equivalent CMPs. Area is equal to N small cores. We vary N from 1 to 64.	
ACMP	1 large core and N-4 small cores; large core runs serial part, 2-way SMT on large core and small cores run parallel part, conventional locking (Maximum number of concurrent threads = N-2)
ACS	1 large core and N-4 small cores; (N-4)-entry CSRB at the large core, large core runs the serial part, small cores run the parallel part, 2-way SMT on large core runs critical sections using ACS (Max. concurrent threads = N-4)
IdealACS	Same as ACS except all cache misses to private data on the large core are *ideally* turned into cache hits. Note that this is an *unrealistic* upper bound on DM.
DM	Same as ACS with support for Data Marshaling

Table 7.2: Simulated workloads.

Workload	Description	Source	Input set	# of disjoint critical sections	What is Protected by CS?
is	Integer sort	NAS suite [13]	n = 64K	1	buffer of keys to sort
pagemine	Data mining kernel	MineBench [70]	10Kpages	1	global histogram
puzzle	15-Puzzle game	[109]	3x3	2	work-heap, memoization table
qsort	Quicksort	[27]	20K elem.	1	global work stack
sqlite	Database [3]	SysBench [4]	OLTP-simple	5	database tables
tsp	Traveling salesman prob.	[55]	11 cities	2	termination cond., solution
maze	3D-maze solver		512x512 maze	2	Visited nodes
nqueen	N-queens problem	[48]	40x40 board	534	Task queue
iplookup	IP packet routing	[105]	2.5K queries	# of threads	routing tables
mysql-1	MySQL server [2]	SysBench [4]	OLTP-simple	20	meta data, tables
mysql-2	MySQL server [2]	SysBench [4]	OLTP-complex	29	meta data, tables
webcache	Cooperative web cache	[101]	100K queries	33	replacement policy

sections. The benchmark `maze` solves a 3-D puzzle using a branch-and-bound algorithm. Threads take different routes through the maze, insert new possible routes in a shared queue and update a global structure to indicate which routes have already been visited.

7.3.4 Evaluation

We evaluate DM on several metrics. First, we show that the generator-set stays stable throughout execution, Second, we show the coverage, accuracy, and timeliness of DM followed by an analysis of DM's effect on L2 cache miss rate inside critical sections. Third, we show the effect of DM on the IPC of the critical program paths. Fourth, we compare the performance of DM to that of the baseline ACS and idealACS (ACS with no misses for private data) at different number of cores.

7.3.4.1 Stability of the Generator-Set

DM assumes that the generator-set, the set of instructions which generate private data, is small and stays stable throughout execution. To test this assumption, we measure the stability and size of the generator set. Table 7.3 shows the size and variance of the generator-set in 12 workloads. Variance is the average number of differences between intermediate generator-sets (computed every 5M instructions) and the overall generator-set divided by the generator-set's size. In all cases, variance is less than 6% indicating that the generator-set is stable during execution. We also evaluated the stability of the generator-set on different input sets and found that the generator-set is constant across input sets.

Table 7.3: Size and variance of Generator-Set.

Workload	is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean
Size	3	10	24	23	34	49	111	23	27	114	277	7	58.5
Variance (%)	0.1	0.1	0.1	0.9	3.0	2.2	4.9	4.2	4.3	6.4	6.3	1.2	-

7.3.4.2 Coverage, Accuracy, and Timeliness of DM

We measure DM's effectiveness in reducing private data misses using three metrics: coverage, accuracy, and timeliness. **Coverage** is the percentage of private

data cache lines identified by DM. **Accuracy** is the percentage of the marshaled lines that are actually used at the large core. **Timeliness** is the percentage of useful marshaled cache lines that reach the large core before they are needed. Note that a marshaled cache line that is in transit when it is requested by the large core is not considered timely according to this definition, but it can provide performance benefit by reducing L2 miss latency.

Figure 7.5a shows the coverage of DM. DM is likely to detect all private lines because it optimistically assumes that every instruction that once generates private data always generates private data. We find that DM covers 99% of L2 cache misses to private data in all workloads except `is`. The private data in `is` is 117 cache lines, which fills up the Marshal Buffer, and thus several private lines are not marshalled (See Section 7.3.4.8).

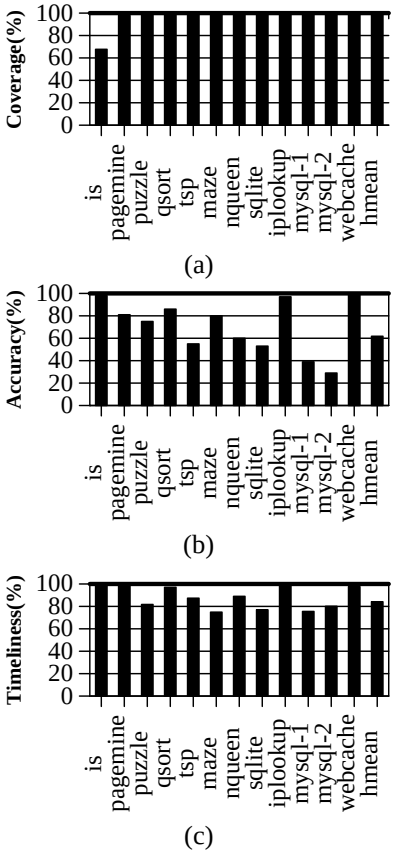


Figure 7.5: (a) Coverage, (b) Accuracy, and (c) Timeliness of DM.

Figure 7.5b shows the accuracy of DM. Recall our assumption that every cache line written by any of the generator instructions is private data. This assumption is optimistic since a generator's destination cache line may or may not be used by a critical section depending on control-flow inside the critical section. For example, the critical sections in the irregular database-workloads `mysql-1` and `mysql-2` have a larger amount of data-dependent control flow instructions which leads to a lower accuracy of DM. Despite our optimistic assumption, we find that a majority (on average 62%) of the cache lines marshaled are useful. Moreover, note that marshaling non-useful lines can cause cache pollution and/or interconnect contention, but only if the number of marshaled cache lines is high. We find this not to be the case. Table 7.4 shows the number of cache lines marshaled per critical section for every workload. In general, we found that transferring *only* an average of 5 cache lines, 62% of which are useful on average, causes a minimal amount of cache pollution and/or interconnect contention.

Table 7.4: Average number of cache lines marshaled per critical section

Workload	is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean
Lines Marshaled	**16**	8.95	5.64	1.78	3.62	1.82	2.78	8.51	2.89	9.83	15.39	1.83	4.99

Figure 7.5c shows DM's timeliness. We find that 84% of the useful cache lines marshaled by DM are timely. Since coverage is close to 100%, timeliness directly corresponds to the reduction in private data cache misses. DM reduces 99% of the cache misses for private data in `pagemine` where timeliness is the highest. In `pagemine`, the main critical section performs reduction of a temporary local histogram (private data) into a persistent global histogram (shared data). The private data is 8 cache lines: 128 buckets of 4-bytes each. Since the critical section is long (212 cycles on the large core) and contention at the large core is high, DM gets enough time to marshal all the needed cache lines before they are needed by the large core. DM's timeliness is more than 75% in all workloads. We show in Section 7.3.4.7 that timeliness increases with larger cache sizes.

7.3.4.3 Cache Miss Reduction Inside Critical Sections

Table 7.5 shows the L2 cache misses per kilo-instruction inside critical sections (CS-MPKI) for ACS and DM. When DM is employed, the arithmetic mean of CS-MPKI reduces by **92%** (from 8.92 to 0.78). The reduction is only 52% in `is` because `is` has low coverage. We conclude that DM (almost) eliminates L2 cache misses inside critical sections.

Table 7.5: MPKI inside critical sections.

Workload	is	pagemine	puzzle	qsort	tsp	maze	nqueen	sqlite	iplookup	mysql-1	mysql-2	webcache	amean	hmean
ACS	3.02	15.60	4.72	25.53	0.79	22.51	16.94	0.75	0.96	4.97	10.86	0.85	8.92	1.59
DM	1.43	0.22	0.48	0.23	0.89	2.15	1.33	0.35	0.41	0.47	1.18	0.13	0.78	0.42

7.3.4.4 Speedup in Critical Sections

DM's goal is to accelerate critical section execution by reducing cache misses to private data. Figure 7.6 shows the retired critical-section-instructions per-cycle (CS-IPC) of DM normalized to CS-IPC of baseline ACS. In workloads where CS-MPKI is low or where L2 misses can be serviced in parallel, DM's improvement in CS-IPC is not proportional to the reduction in CS-MPKI. For example, in `webcache`, DM reduces CS-MPKI by almost 6x but the increase in CS-IPC is only 5%. This is because CS-MPKI of `webcache` is only 0.85. Recall that these misses to private data are on-chip cache misses that are serviced by cache-to-cache transfers. Thus, an MPKI of 0.85, which would be significant if it were for off-chip misses, is less significant for these lower-latency on-chip misses.

In workloads where CS-MPKI is higher, such as in `pagemine`, `puzzle`, `qsort`, and `nqueen`, DM speeds up critical section execution by more than 10%. Most notably, `nqueen's` critical sections execute 48% faster with DM. Note that in none of the workloads do critical sections execute slower with DM than in ACS. On average, critical sections execute 11% faster with DM compared to ACS. Thus,

in benchmarks where there is high contention for critical sections, DM will provide a high overall speedup, as we show next.

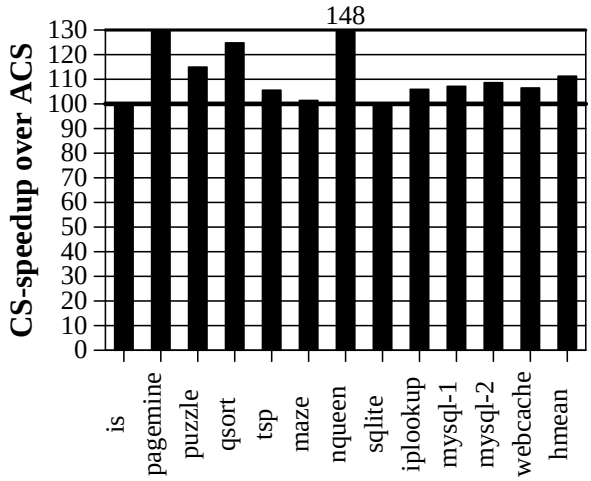


Figure 7.6: Increase in CS-IPC with DM.

7.3.4.5 Performance

DM increases the IPC of critical sections, thereby reducing the time spent inside critical sections. For highly-contended critical sections, reducing the time spent inside the critical section substantially increases overall performance. Moreover, as the number of available cores on a chip increases (which can increase the number of concurrent threads), contention for critical sections further increases and DM is expected to become more beneficial. We compare ACMP, ACS, and DM at area budgets of 16, 32 and 64 small cores.

Area budget of 16: Figure 7.7 shows the speedup of ACMP, DM, and IdealACS compared to the baseline ACS. DM outperforms ACS on all workloads, by 8.5% on average. Most prominently, in `pagemine`, `puzzle`, and `qsort`, DM outperforms ACS by more than 10% due to large increases in CS-IPC. In other benchmarks such as `tsp` and `nqueen`, DM performs 1-5% better than ACS. Note that DM's performance improvement strongly tracks the increase in critical section IPC shown in Figure 7.6. There is one exception, `nqueen`, where the main critical section updates a FIFO work-queue that uses very fine-grain locking. Thus, con-

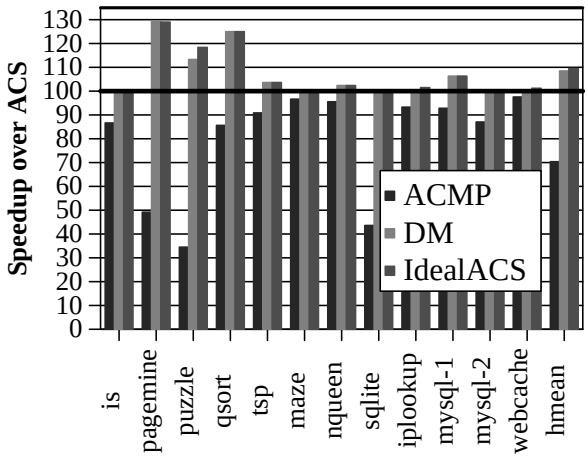


Figure 7.7: Speedup of DM with an area-budget of 16.

tention for the critical sections is low and, even though DM speeds up the critical sections by 48%, overall performance improvement is only 2%. In all other workloads, faster critical sections lead to higher overall performance as expected. DM's performance is within 1% of the IdealACS for all workloads. Thus, *DM achieves almost all the performance benefit available from eliminating cache misses to private data*.

It is worth mentioning that DM is accelerating execution of *only one* of the cores (the large core) by 11% (as shown in Figure 7.6) and providing an overall performance improvement of 8.5% (as shown in Figure 7.7). Since critical sections are usually on the critical path of the program, DM's acceleration of *just* the critical path by an amount provides almost-proportional overall speedup without requiring acceleration of all threads.

Larger area budgets (32 and 64): As the number of cores increases, so does the contention for critical sections, which increases ACS's benefit. Figure 7.8 shows that DM's average performance improvement over ACS increases to 9.8% at area budget 32 versus the 8.5% at area budget 16. Most prominently, in `pagemine` DM's improvement over ACS increases from 30% to 68%. This is because `pagemine` is completely critical-section-limited and any acceleration of critical sections greatly improves overall speedup. DM's performance is again within 1% of that of IdealACS, showing that DM achieves almost all potential ben-

efit.

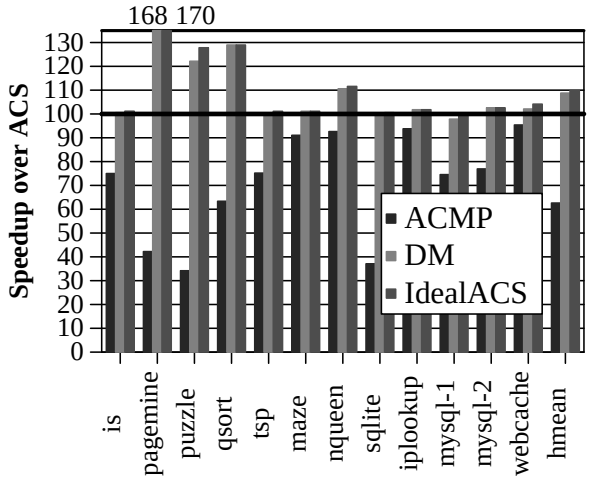


Figure 7.8: Speedup of DM with an area-budget of 32.

As the chip area further increases to 64, DM's improvement over ACS continues to increase (Figure 7.9). On average, DM provides 13.4% performance improvement over ACS and is within 2% of its upper-bound (IdealACS). We conclude that DM's benefits are likely to increase as systems scale to larger number of cores.

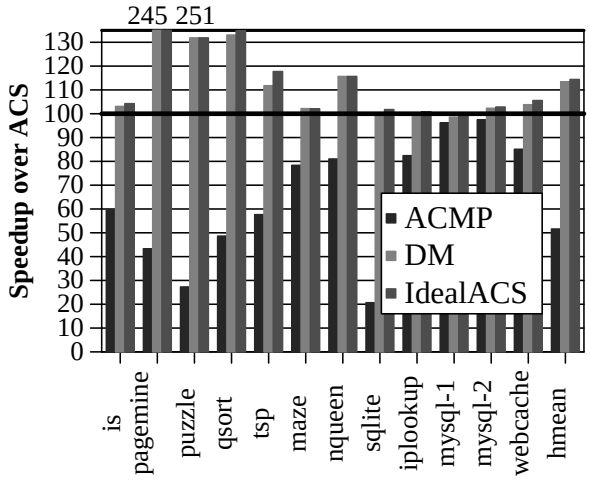


Figure 7.9: Speedup of DM with an area-budget of 64.

At best-threads: The best-threads of a workload is the minimum of the number of threads that minimizes its execution time and the number of processor

contexts. Determining the best-threads requires oracle information because this number depends on input data, which is only available at runtime. We also evaluated DM's speedup with best-threads normalized to ACS with best-threads for an area budget of 16. Our results show that even if we use oracle information to pick the best number of threads, which penalizes DM since DM performs better at a higher thread count, DM provides 5.3% performance over the baseline ACS.

7.3.4.6 Sensitivity to Interconnect Hop Latency

DM's performance improvement over ACS can change with the interconnect hop latency between caches for two reasons. First, increasing the hop latency increases the cost of each on-chip cache miss, increasing the performance impact of misses to private data and making DM more beneficial. Second, increasing the hop latency increases the time to marshal a cache line, which can reduce DM's timeliness, reducing its benefit. We evaluate ACS and DM using hop latencies of 2, 5, and 10 cycles. On average, the speed up of DM over ACS increases from 5.2% to 8.5% to 12.7% as hop latency increases from 2 to 5 to 10. We conclude that DM is even more effective in systems with longer hop latencies, e.g. higher frequency CMPs or SMPs.

7.3.4.7 Sensitivity to L2 Cache Sizes

Private data misses are communication misses. Such misses cannot be avoided by increasing cache capacity. Thus, DM, which reduces communication misses, stays equally beneficial when L2 cache size increases. In fact, DM's benefit might increase with larger caches due to three reasons: 1) enlarging the cache reduces capacity and conflict misses, increasing the relative performance impact of communication misses and techniques that reduce such misses; 2) increasing the L2 size of the large core increases the likelihood that a marshaled cache line will not be evicted before it is used, which increases DM's coverage and timeliness; 3) increasing the small core's L2 capacity increases the amount of private data that stays resident at the small cores' L2 caches and thus can be marshaled, which can

increase DM's coverage.

Table 7.6: Sensitivity of DM to L2 Cache Size.

L2 Cache Size (KB)	128	256	512	1024	2048
ACS vs. ACS 256KB (%)	-5.4	0.0	2.1	2.9	3.1
DM vs. ACS 256KB (%)	-11.4	8.5	10.6	11.3	12.0

Table 7.6 shows the average speedup across all benchmarks for ACS and DM for L2 cache sizes from 128KB to 2048KB. Note that the cache of the large core is always 4x larger than the cache of a small core. The performance benefit of DM over ACS slightly increases as cache size increases from 256KB to 2048KB. In fact, DM with a 256KB L2 cache outperforms ACS with a 2MB L2 cache. However, with a 128KB L2 cache, DM performs worse than ACS. This is because marshaling private data into a small L2 cache at the large core causes cache pollution, evicting shared data or marshaled data of other critical sections not yet executed, and leading to longer-latency L2 cache misses, serviced by the L3. We conclude that DM's performance benefit either increases or stays constant as L2 cache size increases.

7.3.4.8 Sensitivity to Size of the Marshal Buffer

The number of entries in the Marshal Buffer limits the number of cache lines DM can marshal for a critical section segment. We experimented with different Marshal Buffer sizes and found that 16 entries (which we use in our main evaluation) suffice for all workloads except `is`. Since `is` requires the marshaling of 117 cache lines on average, when we use a 128-entry Marshal Buffer, CS-MPKI in `is` is reduced by 22% and performance increases by 3.8% compared to a 16-entry Marshal Buffer.

7.4 DM for Pipeline Workloads

Pipeline parallelism is another instance of SE: each iteration of a loop is split into multiple code segments where each segment is one pipeline stage. Furthermore, segments run on different cores. As in SE, each core has a work-queue

Pipeline Stage S1: 1: .... 2: store X 3: Enqueue a request at S2's home core *Pipeline Stage S2:* 4: Y = ... X	*Pipeline Stage S1:* 1: .... 2: **GENERATOR store X** 3: Enqueue a request at S2's home core 4: **MARSHAL <S2's home core>** *Pipeline Stage S2:* 5: Y = ... X	 ;Compute X ;S2's initiation ;Compute Y
(a) Code of a pipeline.	(b) Modified code with DM.	

Figure 7.10: Code example of a pipeline.

and processes execution requests. Each pipeline stage first completes its computation and then executes the initiation routine for the next stage.

7.4.1 Inter-segment data in pipeline parallelism

Figure 7.10(a) shows a code example of two pipeline stages: S1 and S2, running on cores P1 and P2, respectively. S1 computes and stores the value of a variable X (line 1-2) and then enqueues a request to run S2 at core P2 (line 3). Note that X is used by S2 (line 4). P2 may process the computation in S2 immediately or later, depending on the entries in its work-queue.

7.4.2 DM in Pipelines

Processing of a pipeline stage often requires data that was generated in the previous pipeline stage. Since each stage executes at a different core, such inter-segment or inter-stage data must be transferred from core to core as the work-quantum is processed by successive pipeline stages. For example, in the pipeline code in Figure 7.10(a), variable X is inter-segment data as it is generated in S1 (line 2) and used by S2 (line 4). When S2 runs on P2, P2 incurs a cache miss to fetch X from P1.

DM requires two code changes. First, the compiler must identify the generator instructions and prepend them with a GENERATOR prefix. Second, the compiler/library must insert a MARSHAL instruction in the initiation routine. Figure 7.10(b) shows the same code with modifications for DM. Since X is inter-

segment data, the compiler identifies via profiling the store instruction on line 2 as a generator and prepends it with the GENERATOR prefix. Furthermore, the MARSHAL instruction is inserted in the initiation routine (line 4).

When P1 (the core assigned to S1) runs the store on line 2, the hardware inserts the physical address of the cache line being modified into P1's Marshal Buffer. When the MARSHAL instruction on line 5 executes, the Data Marshaling Unit (DMU) marshals the cache line containing X to P2's L2 cache. When S2 runs on P2, it incurs a cache hit for X, which likely reduces execution time.

7.4.3 Evaluation Methodology

We simulate three different configurations: Baseline (a baseline ACMP without DM), Ideal (an idealistic but impractical ACMP where all inter-segment misses are turned into hits), and DM (an ACMP with support for DM). The parameters of each core, the interconnect, caches, and the memory sub-system are shown in Table 7.1. The Ideal scheme, which unrealistically eliminates all inter-segment misses, is an upper bound of DM's performance.

Table 7.7 shows the simulated workloads. We developed a pipelined implementation of the dedup decoder based on the sequential code in PARSEC [15] and call it `dedupD`. A MARSHAL instruction was inserted in the initiation routine of each workload. Unless otherwise specialized, all comparisons are at equal-area and equal-number-of-threads. Results are for a 16-core CMP unless otherwise stated.

7.4.4 Evaluation

We evaluate DM's coverage, accuracy and timeliness, and its impact on the MPKI of inter-segment data, and overall performance. We also show DM's sensitivity to relevant architectural parameters. We also validated that the generator-sets are stable during execution and across input sets for pipeline workloads, but we do not show the results due to space constraints.

Table 7.7: Workload characteristics.

Workload	Description (No. of pipeline stages)	Major steps of computation	Input
black	BlackScholes Financial Kernel [74] **(6)**	Compute each option's call/put value	1M opts
compress	File compression using bzip2 algorithm **(5)**	Read file, compress, re-order, write	4MB text file
dedupE	De-duplication (Encoding) [15] **(7)**	Read, find anchors, chunk, compress, write	simlarge
dedupD	De-duplication (Decoding) [15] **(7)**	Read, decompress, check cache, reassemble/write	simlarge
ferret	Content based search [15] **(8)**	Load, segment, extract, vector, rank, out	simlarge
image	Image conversion from RGB to gray-scale **(5)**	Read file, convert, re-order, write	100M pixels
mtwist	Mersenne-Twister PRNG [74] **(5)**	Read seeds, generate PRNs, box-muller	path=200M
rank	Rank string similarity with an input string **(3)**	Read string, compare, rank	800K strings
sign	Compute the signature of a page of text **(7)**	Read page and compute signature	1M pages

7.4.4.1 Coverage, Accuracy, and Timeliness

Figure 7.11a shows DM's coverage, i.e., the percentage of inter-segment data cache lines identified by DM. Coverage is over 90% in all workloads except `dedupD`, `image`, and `mtwist`. In these workloads the inter-segment data needed per segment often exceeds the size of the Marshal Buffer (16). Since DM marshals only the cache lines in the Marshal Buffer, not all inter-segment data is marshaled.

Figure 7.11b shows the accuracy of DM, i.e., the percentage of marshaled lines that are actually used by the home core. DM's accuracy is low, between 40% and 50%, because stages contain control flow. However, the increase in interconnect transactions/cache pollution for DM is negligible because the number of cache lines marshaled per segment is small: the average is 6.8 and the maximum is 16 (the size of the Marshal Buffer).

Figure 7.11c shows DM's timeliness, i.e., the percentage of useful cache lines identified by DM that reach the remote home core before their use. Timeliness is high, more than 80% in all cases, for two reasons: (1) segments often wait in

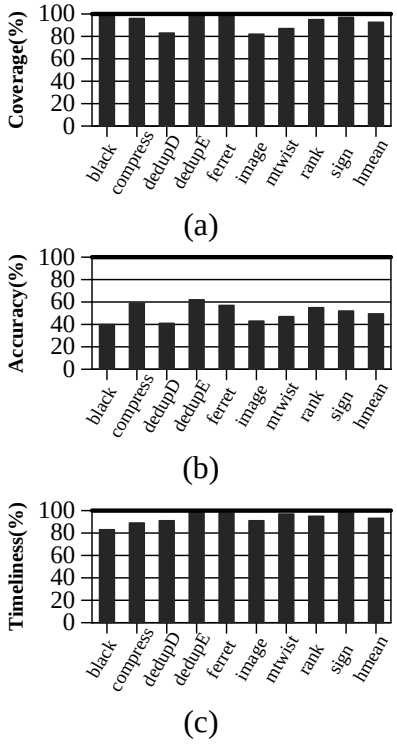


Figure 7.11: (a) Coverage, (b) Accuracy, and (c) Timeliness of DM.

the home core's work-queue before their processing, giving DM enough time to marshal the lines, (2) transferring the few lines that are marshaled per segment requires a small number of cycles.

7.4.4.2 Reduction in Inter-Segment Cache Misses

Table 7.8 shows the L2 MPKI of inter-segment data in the baseline and DM. DM significantly reduces the MPKI in all cases. The most noticeable is `sign` where DM reduces the MPKI from 30.3 to 0.9. `sign` computes the signature of an input page for indexing. The main inter-segment data in `sign` is the page's signature, a 256-character array (4 cache lines). Since DM's profiling algorithm marks the instruction that stores all array elements as a generator, DM saves all cache misses for the array. Similarly, DM completely eliminates inter-segment data misses in `ferret` and `dedupE`. DM reduces the harmonic mean of MPKI by 81% and the arithmetic mean by 69%.

Table 7.8: L2 Misses for Inter-Segment Data (MPKI). We show both amean and hmean because hmean is skewed due to `dedupE`. Note: MPKI of inter-segment data in Ideal is 0.

Workload	black	compress	dedupD	dedupE	ferret	image	mtwist	rank	sign	amean	hmean
baseline	14.2	7.7	47.5	0.4	4.4	55.6	51.4	4.1	30.3	23.95	2.76
DM	2.8	1.7	33.0	0.0	0.1	20.4	7.4	0.3	0.9	7.40	0.53

7.4.4.3 Performance

Execution time of a pipelined program is always dictated by its slowest stage. Thus, DM's impact on overall performance depends on how much it speeds up the slowest stage. Figure 7.12 shows the speedup of Ideal and DM over the baseline at 16 cores. On average, DM provides a 14% speedup over the baseline, which is 96% of the potential. DM improves performance in all workloads. DM's improvement is highest in `black` (34% speedup) because DM reduces inter-segment misses by 81% and as a result speeds up the slowest stage significantly. DM is within 5% of the Ideal speedup in `black` because accesses to inter-segment data are in the first few instructions of each stage and consequently the marshaled cache lines are not always timely. DM's speedup is lower in `dedupE` and `ferret` because these workloads only incur a small number of inter-segment misses and DM's potential is low (Ideal speedup is only 5% for ferret).

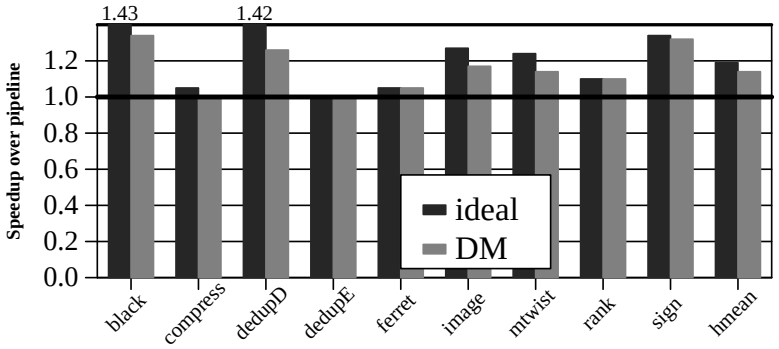


Figure 7.12: Speedup at 16 cores.

32-core results: Figure 7.13 shows the speedup of Ideal and DM over the baseline with 32 cores. DM's speedup increased for all workloads compared to 16

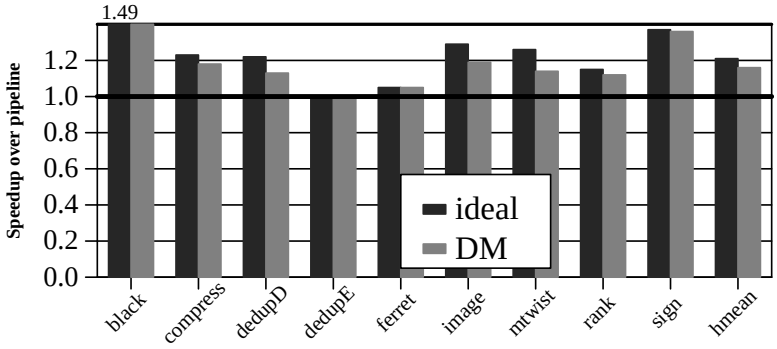


Figure 7.13: Speedup at 32 cores.

cores. Most significant is the change in `compress`, from 1% to 18%, because the slowest stage in `compress` changes between 16 and 32 cores. At 16 threads, `compress`'s slowest stage is the stage that compresses chunks of input data. This stage is compute-bound and does not offer a high potential for DM. However, the compression stage is scalable, i.e., its throughput increases with more cores. At 32 cores, the compression stage's throughput is more than the non-scalable re-order stage (the stage which re-orders chunks of compressed data before writing them to the output file). Unlike the compression stage which is compute-bound, the re-order stage is bounded by cache misses for inter-segment data, ergo, a higher potential for DM and thus the higher benefit. On average, at 32 cores, DM improves performance by 16%, which is higher than its speedup at 16 cores (14%). In summary, DM is an effective technique that successfully improves performance of pipelined workloads, with increasing benefit as the number of cores increases.

7.4.4.4 Sensitivity to Interconnect Hop Latency

We find that the speedup of DM increases with hop latency (refer to Section 7.3.4.6 for reasons). We evaluated DM with hop latencies of 2, 5, and 10 cycles and find that it provides speedups of 12.4%, 14%, and 15.1% over the baseline, respectively.

7.4.4.5 Sensitivity to L2 Cache Sizes

DM's benefit increases with cache size for pipeline workloads as well (see Section 7.3.4.7 for reasons). DM's speedup over the baseline is 4.6%, 14%, 14.8%, 15.3%, and 15.4% for cache sizes of 128KB, 256KB, 512KB, 1MB, and 2MB, respectively.

7.4.4.6 Sensitivity to size of Marshal Buffer

We find that a 16-entry Marshal Buffer is sufficient for all workloads except `dedupD`, `image`, and `mtwist`. The number of inter-segment cache lines per segment in these workloads is greater than 16. For example, the primary inter-segment data structure in image, an array of 150 32-bit RGB pixels, spans 32-cache lines. Table 7.9 shows the performance of DM with varying Marshal Buffer sizes in these three workloads. In each case, performance saturates once there are enough entries to fit all inter-segment cache lines (32, 32, and 128 for dedupD, image, and mtwist respectively). In summary, while there are a few workloads that benefit from a larger Marshal Buffer, a 16-entry buffer suffices for most workloads.

Table 7.9: Sensitivity to Marshal Buffer size: Speedup over baseline (%).

# of entries	16	32	64	128	256
dedupD	26	40	40	40	40
image	17	24	24	24	24
mtwist	14	18	21	22	22

7.5 DM on Other Paradigms

We have shown two concrete examples of how DM can be applied to different execution models: ACS and pipeline parallelism. DM can be applied to any paradigm that resembles SE, for example:

- Task-parallelism models such as Cilk [17], Intel TBB [47] and Apple's Grand Central Dispatch [12]. Any time a new task is scheduled at a remote core, DM can marshal the input arguments of the task to the core that will execute the

task.

- Computation Spreading [21], which improves locality by always running the operating system code on the same set of cores. DM can marshal the data to and from the cores dedicated for OS code.
- Thread Motion [80] migrates threads among cores to improve power-performance efficiency. DM can be extended to reduce cache misses due to thread migration.
- The CoreTime OS scheduler [18] assigns data objects to caches and migrates threads to the cache containing the majority of the data they access. DM can marshal any extra data required by the thread (e.g., portions of its stack).
- Remote special-purpose cores, e.g., encryption or video encoding engines, are often used to accelerate code segments. DM can be used to marshal data to such accelerators.

DM not only improves the existing paradigms, but can also enable new paradigms that follow SE. By significantly reducing the data-migration cost associated with shipping a segment to a remote core, DM can enable very fine-grain segments, which could allow more opportunity for core specialization. In summary, DM is applicable to widely-used current paradigms, makes proposed paradigms more effective, and can potentially enable new paradigms.

Chapter 8

Related Work

8.1 Related Work in Accelerating Non-Parallel Kernels

Morad et al. [69] used analytic models to show that an asymmetric architecture with one large core and multiple small cores can run non-parallel kernels quickly, thereby improving overall performance and scalability. We showed in [93] that an ACMP can be built using off-the-shelf cores and can improve performance of real workloads. Hill at al. [38] build on our ACMP model and present an analytic analysis which shows that ACMP can improve performance of non-parallel kernels. The work presented in this thesis is different from the work by Morad et. al. and Hill et. al. for four reasons: (1) we show a practical architecture for the ACMP, not just analytic models; (2) we propose an actual algorithm for accelerating non-parallel kernels; (3) we evaluate ACMP by simulating real workloads; and (4) we also show that the ACMP can also accelerate critical sections and limiter stages in addition to the non-parallel kernels.

Kumar et al. [56] propose heterogeneous cores to reduce power and increase throughput of a system running multiple single-threaded workloads. Their mechanism chooses the best core to run each application on. In contrast, we use the ACMP to reduce the execution time and improve scalability of a single multi-threaded program by accelerating the common critical paths in the program.

Annavaram et al. [11] propose that an ACMP can also be created using frequency throttling: they increase the frequency of the core that is executing the non-parallel kernel. However, they only deal with non-parallel kernels and do not accelerate critical sections or limiter pipeline stages, which are central to our proposal. The mechanisms we propose are general and can leverage the ACMP they

create via frequency throttling.

The IBM Cell processor [40] provides a PowerPC core in addition to the parallel processing engines on the chip. It is different from the ACMP because the cores have different ISAs and the PowerPC core is used primarily to run legacy code.

Previous research [49, 53, 65, 98, 112] also shows that multiple small cores can be fused to form a powerful core at runtime when non-parallel kernels are executed. If such a chip can be built, our techniques can be adapted to work with their architecture: multiple execution engines can be combined to form a powerful execution engine to accelerate the serial bottlenecks.

8.2 Related Work in Reducing Critical Section Penalty

8.2.1 Related Work in Improving Locality of Shared Data and Locks

Sridharan et al. [89] propose a thread scheduling algorithm for SMP machines to increase shared data locality in critical sections. When a thread encounters a critical section, the operating system migrates the thread to the processor that has the shared data. This scheme increases cache locality of shared data but incurs the substantial operating system overhead of migrating complete thread state on every critical section. Accelerated Critical Sections (ACS) does not migrate thread contexts and therefore does not need OS intervention. Instead, it sends a CSCALL request with minimal data to the core executing the critical sections. Moreover, ACS accelerates critical section execution, a benefit unavailable in [89]. Trancoso et al. [102] and Ranganathan et al. [81] improve locality in critical sections using software prefetching. These techniques can be combined with ACS for improved performance.

Several primitives like Test&Set, Test&Test&Set, Compare&Swap, fetch&add are commonly used to efficiently implement atomic operations such as increments, lock acquire, and lock release operations. Recent research has also studied hardware and software techniques to reduce the overhead of atomic op-

erations [9, 39, 61]. The Niagara-2 processor improves cache locality of atomic operations by executing the instructions [33] remotely at the cache bank where the data is resident. However, none of these techniques increase the speed/locality of general critical sections which read-modify-write multiple data, a feature provided by our mechanisms. As shown in Table 5.3 on page 49, many workloads contain critical sections which are hundreds of instructions long and these mechanisms are unable to shoten such critical sections.

8.2.2 Related Work in Hiding the Latency of Critical Sections

Several proposals try to hide the latency of a critical section by executing it speculatively with other instances of the same critical section *as long as they do not have data conflicts with each other*. Examples include transactional memory (TM) [37], speculative lock elision (SLE) [76], transactional lock removal (TLR) [77], and speculative synchronization (SS) [67]. SLE is a hardware technique that allows multiple threads to execute the critical sections speculatively without acquiring the lock. If a data conflict is detected, only one thread is allowed to complete the critical section while the remaining threads roll back to the beginning of the critical section and try again. TLR improves upon SLE by providing a timestamp-based conflict resolution scheme that enables lock-free execution. ACS is orthogonal to these approaches due to three major reasons:

1. TLR/SLE/SS/TM improve performance when the concurrently executed instances of the critical sections do not have data conflicts with each other. In contrast, ACS improves performance even for critical section instances that have data conflicts. If data conflicts are frequent, TLR/SLE/SS/TM can degrade performance by rolling back the speculative execution of all but one instance to the beginning of the critical section. In contrast, ACS's performance is not affected by data conflicts in critical sections.

2. TLR/SLE/SS/TM amortize critical section latency by concurrently executing non-conflicting critical sections, but they do not reduce the latency of each critical section. In contrast, ACS reduces the execution latency of critical sections.

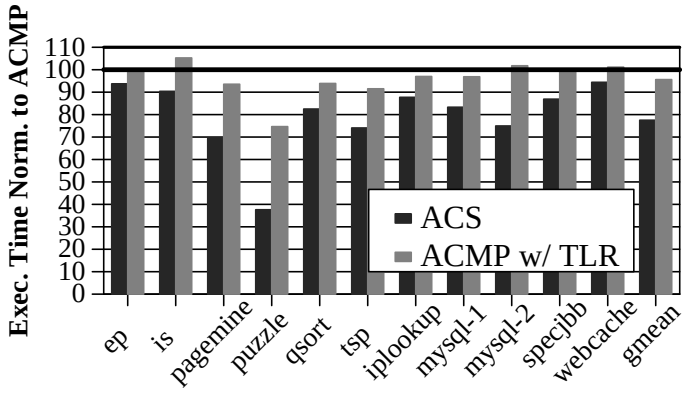


Figure 8.1: ACS vs. TLR performance.

3. TLR/SLE/SS/TM do not improve locality of lock and shared data. In contrast, ACS improves locality of lock and shared data by keeping them in a single cache.

We compare the performance of ACS and TLR. Figure 8.1 shows the execution time of an ACMP augmented with TLR[1] and the execution time of ACS normalized to ACMP (area budget is 32 and number of threads set to the optimal number for each system). TLR reduces average execution time by 6% while ACS reduces it by 23%. In applications where critical sections often access disjoint data (e.g., `puzzle`, where the critical section protects a heap to which accesses are disjoint), TLR provides large performance improvements. However, in workloads where critical sections conflict with each other (e.g., `is`, where each instance of the critical section updates all elements of a shared array), TLR degrades performance. ACS outperforms TLR on all benchmarks, and by 18% on average. This is because ACS accelerates many critical sections regardless of whether or not they have data conflicts, thereby reducing serialization.

As such, our approach is complementary to SLE, SS, TLR, and TM. These other approaches amortize critical section latency by allowing concurrent execution of critical sections but they do not improve the execution time spent in a critical section. In contrast, our work reduces the execution time of a critical section. Our

[1]TLR was implemented as described in [77]. We added a 128-entry buffer to each small core to handle speculative memory updates.

approach can be combined with SLE/SS/TM to further reduce the performance loss due to critical sections. For example, some "speculative critical sections" (as in SLE/SS) or transactions can be executed on the large core(s) using our mechanism. This allows not only faster but also concurrent execution of instances of a critical section and constitutes part of our future work.

8.2.3 Other Related Work in Remote Function Calls

The idea of executing critical sections remotely on a different processor resembles the *Remote Procedure Call (RPC)* [16] mechanism used in network programming to ease the construction of distributed, client-server based applications. RPC is used to execute (client) subroutines on remote (server) computers. In ACS, the small cores are analogous to the "client," and the large core is analogous to the "server" where the critical sections are remotely executed. ACS has two major differences from RPC. First, ACS executes "remote" critical section calls within the same address space and the same chip as the callee, thereby enabling the accelerated execution of shared-memory multi-threaded programs. Second, ACS's purpose is to accelerate shared-memory parallel programs, whereas RPC's purpose is to ease network programming.

Active messages [28] have been proposed to by Eicken et al. to allow for low overhead communication in large-scale systems. There are similarities between Active Messages and ACS's treatment of critical sections: both get sent as a message to another core for execution. However, Active Messages were proposed for very large-scale message passing systems while ACS is for shared memory machines. Furthermore, they did not accelerate the execution using a faster core, a key feature of ACS. Similar to the CSCALL instruction in ACS, the MIT J-machine [25] also provided instructions to send an execution request to a remote core. However, the J-machine was a symmetric system and did not accelerate critical paths using a high-performance engine.

8.3 Related Work in Increasing Pipeline Throughput

Several studies [15, 29, 34] have discussed the importance of using pipelined parallelism on CMP platforms. Our mechanism to Accelerate the Limiter Stages (ALS) provides automatic runtime tuning and accelration of this important paradigm and obtains improved performance and power-efficiency. To the best of our knowledge, ALS is the first scheme to improve pipeline throughput via asymmetric CMPs. However, there have been several schemes which choose the best core-to-stage allocation, attempting to speed up the limiter stage by assigning it more cores. Such schemes are more specifically related to FDP, the mechanism we proposed for choosing the best core-to-stage allocation.

Recently Hormati et al. proposed the Flextream compilation framework [41] which can dynamically recompile pipelined applications to adapt to the changes in the execution environment, e.g., changes in the number of cores assigned to an application. While FDP can also adapt to changes in the execution environment, its main goal is to maximize the performance of a single application. Flextream and FDP fundamentally differ for four reasons. First, Flextream does not consider the use of asymmetric cores. Second, unlike FDP, Flextream assumes that all stages are scalable and thus allocates cores based on the relative demands of each stage. This can reduce performance and waste power when a stage does not scale (see Section 6.6.1.2). Third, Flextream requires dynamic recompilation which restricts it to languages which support that feature, e.g., JAVA and C-sharp. In contrast, FDP is done via a library which can be used with any language. Fourth, Flextream cannot be used to choose the number of threads in work sharing programs because it will assume that the workload scales and allocate it all available cores. FDP, on the other hand, chooses the best number of threads taking scalability into account.

Other proposals in the operating system and web server domains have implemented feedback directed cores-to-work allocation [91, 106]. However, they do not use asymmetric cores and make several domain-specific assumptions which

makes their scheme applicable only to those domains.

Others have also tried to optimize pipelines through static core-to-stage allocation using profile information. The brute force search for finding the best mapping can be avoided by using analytical models. Other researchers have![31, 63, 71, 72] proposed analytic models for understanding and optimizing parallel pipelines. They do not account for asymmetry and while such models can help programmers design a pipeline, they are static and do not adapt to changes in input set and machine configuration. In contrast, FDP relieves the programmer from obtaining representative profile information for each input set and machine configuration and does automatic tuning using runtime information.

Languages and languages extensions [26, 47, 58, 100] can help with simplifying the development of pipelined programs. Raman et al. [78] propose to automatically identify pipeline parallelism in a program using intelligent compiler and programming techniques. Our work is orthogonal to their work in that our proposal optimizes at run-time an already written pipelined program.

Pipeline parallelism is also used in databases [36] where each database transaction is split into stages which can be run on multiple cores. Their work can also use FDP to choose the best core-to-stage allocation. Others have also proposed accelerating massively parallel computations in a kernel using special purpose accelerators such as GPUs [64, 74]. The focus of this thesis is to improve scalability by accelerating the serial bottlenecks, not the parallel parts.

8.4 Related Work in Data Marshaling

Data Marshaling (DM) has related work in the areas of hardware prefetching, cache coherence, and OS/compiler techniques to improve locality.

8.4.1 Hardware Prefetching

Hardware prefetchers can be broadly classified as prefetchers that can handle regular (stride/stream) memory access patterns (e.g., [51, 75, 99]) and those

that can handle irregular memory access patterns (e.g., [23, 50, 84, 88, 107, 108]). Prefetchers that handle only regular data cannot capture misses for inter-segment data because inter-segment cache lines do not follow a regular stream/stride pattern and are scattered in memory. Prefetchers that handle irregular data [50, 84, 88, 107](as well as stride/stream based prefetchers) are also not suited for prefetching inter-segment data because the number of cache misses required for training such prefetchers is often more than all of the inter-segment data (an average of 5 in ACS and 6.8 in pipeline workloads). Thus, by the time prefetching begins, a majority of the cache misses have already been incurred.

DM does not face any of these disadvantages. DM requires minimal on-chip storage, can marshal any arbitrary sequence of inter-segment cache lines, and starts marshaling as soon as the next code segment is shipped to its home core, without requiring any training. Thus, the likelihood of DM covering misses in a timely fashion is substantially higher than that of prefetching. Note that we already used an aggressive stream prefetcher [99] in our baseline and the improvements we report with DM are on **top** of this aggressive prefetcher.

8.4.2 Reducing Cache Misses using Hardware/Compiler/OS Support

Hossain et al. [42] propose DDCache, where the producer stores the sharers of every cache line and when one of the consumers requests a cache line, the producer sends it not only to the requester, but also to all the remaining sharers. DDCache is orthogonal to DM in that it improves locality of only shared data, while DM also improves locality of thread-private data. Thread Criticality Predictors [14] have been proposed to improve cache locality in task-stealing [47], another example of Staged Execution (SE) paradigm. They schedule tasks to maximize cache locality and DM can further help by eliminating the remaining cache misses.

Several proposals reduce cache misses for shared data using hardware/compiler support. Sridharan et al. [89] improve shared data locality by migrating threads wanting to execute a critical section to a single core. This increases

misses for private (inter-segment) data, the limitation addressed by DM. Other proposals improve shared data locality by inserting software prefetch instructions before the critical section [81, 102]. Such a scheme cannot work for inter-segment data because prefetch instructions must be executed at the home core, i.e. as part of the next code segment and very close to the actual use of the data, making the prefetches untimely. Recent cache hierarchy designs [22, 35] aim to provide fast access to both shared and private data and can further benefit from DM to minimize cache-to-cache transfers. Proposals that accelerate thread migration [19, 92] are orthogonal and can be combined with DM.

The Tempest and Typhoon framework [82] provides a mechanism which allows software to push cache lines from one core to the other. Data marshaling is different from their work for three reasons. First, transferring cache lines in their system requires invocation of send and receive handlers at the source and destination cores. Thus, their transfers interrupt normal thread execution. We perform our marshaling in hardware and do not interrupt the execution to run any handlers. Second, they require that the data must be packetized before it is transferred. We have no such restrictions. Third, they require that the compiler must be able to "fully analyze a program's communication pattern." We propose a simple profiling mechanism which does not require any complex compiler analysis.

8.4.3 Other Related Work

In the *Remote Procedure Call (RPC)* [16] mechanism used in networking, the programmer identifies the data that must be "marshaled" with the RPC request. This is similar to DM, which marshals inter-segment data to the home core of the next code segment. However, unlike RPC, marshaling in DM is solely for performance, does not require programmer input, and is applicable to instances of SE that do not resemble procedure calls.

Chapter 9

Conclusion

9.1 Summary

Performance and scalability of multi-threaded programs is severely limited due to program portions which are not parallelized. Major sources of serialization include non-parallel kernels, critical sections, and limiter stages in pipeline workloads. We show that all three reduce performance and limit the number of threads at which performance saturates. To overcome this limitation, we propose the Asymmetric Chip Multiprocessor (ACMP) paradigm and mechanisms to identify and accelerate the serial bottlenecks using the ACMP.

To accelerate the non-parallel kernels, we propose a thread scheduling algorithm which migrates the thread running a serial portion to the large core. This technique requires minimal hardware/software overheads and significantly reduces execution time by 17% over an area-equivalent symmetric CMP.

To accelerate the critical sections, we propose a combined hardware/software mechanism which accelerates critical sections by executing them at the large core of the ACMP. This mechanism significantly reduces execution time by 34% over an area-equivalent symmetric CMP.

To accelerate the limiter stages in pipeline workloads, we propose a software library to identify and accelerate the limiter stage using the ACMP's large core. The proposed mechanism significantly reduces execution time by 20% over an area-equivalent symmetric CMP.

We further identify a major performance limitation of ACMP: the cache misses at the large core for transferring data from the small cores. We propose *Data Marshaling (DM)* to improve the reduced locality of private data in Accelerated

Critical Section (ACS) mechanism and inter-segment data in pipeline workloads. We find that DM's performance is within 1% of an ideal mechanism that eliminates all private data misses using oracle information, showing that DM achieves almost all of its upper-bound performance potential.

We conclude that serial program portions can be efficiently accelerated using faster cores in an asymmetric chip multiprocessor, thereby improving performance and increasing scalability.

9.2 Limitations and Future Work

This thesis has proposed several brand new concepts which can be extended by future research. We envision future work in seven areas:

- *Analytic Models;* We proposed analytic models to analyze the effect of accelerating the serial bottlenecks. Since the intent was to show the importance of serial bottlenecks in the simplest possible manner, we made some simplifying assumptions. Future work can extend these models such that they take more run-time parameters into account.
- *Exploring the ACMP design space:* We proposed an ACMP architecture with one large core and many small cores. There are others ways to implement asymmetry among cores on a chip, e.g., by frequency scaling or managed memory scheduling. Future research can explore such options to create new asymmetric architectures, and/or increase or reduce the degree of asymmetry in the ACMP architecture we proposed. Note that the concepts we propose are applicable to any ACMP implementation.
- *Generalizing the ACMP:* This thesis analyzed an ACMP with a single large core running a single application in isolation. Future research can explore ways of leveraging multiple large cores. Future research can also explore different options for the small and large cores, e.g., use GPUs as the small cores.
- *Other bottlenecks:* This thesis showed how the ACMP can improve performance of three major bottlenecks. Future research can develop mechanisms

to identify and accelerate other bottlenecks, e.g., Reduction in Google MapReduce.

- *Alternate implementations of ACS:* We show a combined hardware/software implementation of ACS. ACS can also be implemented purely in hardware or purely in software. Future research can explore these directions.
- *New software algorithms:* ACS allows longer critical sections which can enable new parallel algorithms that were previously considered infeasible due to their data synchronization overhead. For example, priority-queue-based worklists, although known to be more work conserving, are not used solely because they introduce long critical sections. ACS makes their use possible, thereby enabling new software algorithms.
- *Alternate implementation of DM:* We show a combined hardware/software implementation of DM. Rigorous compiler algorithms to improve the accuracy of DM or alternate hardware-only implementations of DM is an open research topic.
- *Segment-to-core scheduling:* ACS and ALS show two examples of code segments which run "better" on a large core. There may be other code segments which could leverage the large core or a brand new type of core. Mechanisms to identify the best core for each segment is a wide open topic.

Bibliography

[1] Api - facebook developer wiki. http://developers.facebook.com/.

[2] MySQL database engine 5.0.1. http://www.mysql.com, 2008.

[3] SQLite database engine version 3.5.8. http:/www.sqlite.org, 2008.

[4] SysBench: a system performance benchmark version 0.4.8. http://sysbench.sourceforge.net, 2008.

[5] Opening Tables scalability in MySQL. MySQL Performance Blog, 2006. http://www.mysqlperformanceblog.com/2006/11/21/opening- tables-scalability.

[6] Mysql internals locking overview, 2008. http://forge.mysql.com/wiki/MySQL_Internals_Locking_Overview.

[7] Michael Abrash. *Michael Abrash's Graphics Programming Black Book.* Coriolis Group, 1997.

[8] Advanced Micro Devices, Inc. White Paper: Multi-Core Processors – The next evolution in computing. 2005.

[9] S. Adve et al. Replacing locks by higher-level primitives. Technical Report TR94-237, Rice University, 1994.

[10] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies*, 1967.

[11] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating amdahl's law through EPI throttling. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[12] Apple. Grand Central Dispatch - A better way to do multicore, 2009.

[13] D. H. Bailey et al. NAS parallel benchmarks. Technical Report Tech. Rep. RNR-94-007, NASA Ames Research Center, 1994.

[14] Abhishek Bhattacharjee and Margaret Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2009.

[15] Christian Bienia et al. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[16] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[17] Robert Blumofe et al. Cilk: an efficient multithreaded runtime system. In *Proceedings of the 7th ACM Sigplan Symposium on Principles and Practice of Parallel Programming*, 1995.

[18] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *HotOS-XII*, 2009.

[19] Jeffery A. Brown and Dean M. Tullsen. The shared-thread multiprocessor. In *Proceedings of the International Conference on Supercomputing*, 2008.

[20] Christian Brunschen et al. OdinMP/CCp - a portable implementation of OpenMP for C. *Concurrency: Prac. and Exp.*, 12(12), 2000.

[21] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[22] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2006.

[23] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[24] Intel Corporation. Intel turbo boost technology in intel core microarchitecture (nehalem) based processors. Whitepaper, 2008.

[25] William Dally et al. Architecture of a message-driven processor. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1998.

[26] Abhishek Das et al. Compiling for stream processing. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[27] Antonio J. Dorta et al. The OpenMP source code repository. In *Euromicro*, 2005.

[28] Thorsten Von Eicken et al. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1992.

[29] J. Giacomoni et al. Toward a toolchain for pipelineparallel programming on cmps. *Workshop on Software Tools for Multi-Core Systems*, 2007.

[30] Simha Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[31] D. González et al. Towards the automatic optimal mapping of pipeline algorithms. *Parallel Comput.*, 2003.

[32] Michael I. Gorden et al. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[33] Greg Grohoski. Distinguished Engineer, Sun Microsystems. Personal communication, November 2007.

[34] Jayanth Gummaraju et al. Streamware: programming general-purpose multicore processors using streams. *SIGARCH Comput. Archit. News*, 2008.

[35] Zvika Guz et al. Utilizing shared data in chip multiprocessors with the nahalal architecture. In *ACM Symposium on Parallelism in Algorithms and Architectures*, 2008.

[36] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *Biennial Conference on Innovative Data Systems Research*, 2003.

[37] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1993.

[38] Mark Hill and Michael Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7), 2008.

[39] Ralf Hoffmann et al. Using hardware operations to reduce the synchronization overhead of task pools. *International Conference on Parallel Processing*, 2004.

[40] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 258–262, 2005.

[41] Amir H. Hormati et al. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[42] Hemayet Hossain, Sandhya Dwarkadas, and Michael C. Huang. DDCache: Decoupled and delegable cache data and metadata. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[43] Intel. ICC 9.1 for Linux. `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.%htm`.

[44] Intel. *Pentium Processor User's Manual Volume 1: Pentium Processor Data Book*, 1993.

[45] Intel. Prescott New Instructions Software Development Guide, 2004.

[46] Intel. IA-32 Intel Architecture Software Dev. Guide, 2008.

[47] Intel. Source code for Intel threading building blocks, 2008. http://threadingbuildingblocks.org.

[48] Intel. Getting Started with Intel Par. Studio, 2009.

[49] Engin Ipek et al. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2007.

[50] Doug Joseph and Dirk Grunwald. Prefetching using Markov predictors. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1997.

[51] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1990.

[52] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, 2004.

[53] Changkyu Kim et al. Composable lightweight processors. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, 2007.

[54] Poonacha Kongetira et al. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[55] Heinz Kredel. Source code for traveling salesman problem (tsp). http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.

[56] Rakesh Kumar, Dean M. Tullsen, Norman P. Jouppi, and Parthasarathy Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11), 2005.

[57] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, August 1974.

[58] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference, 2002*.

[59] James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 241–251, 1997.

[60] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[61] Ulana Legedza and William E. Weihl. Reducing synchronization overhead in parallel simulation. In *Workshop on Parallel and Distributed Simulation*, 1996.

[62] Chunhua Liao et al. OpenUH: an optimizing, portable OpenMP compiler. *Concurr. Comput. : Pract. Exper.*, 19(18):2317–2332, 2007.

[63] Wei-Keng Liao. Performance evaluation of a parallel pipeline computational model for space-time adaptive processing. *Journal of Supercomputing*, 2005.

[64] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, New York, NY, USA, 2009. ACM.

[65] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J. Dally, and Mark Horowitz. Smart memories: a modular reconfigurable architecture. *SIGARCH Comput. Archit. News*, 28(2):161–171, 2000.

[66] Daniel Marino et al. Literace: effective sampling for lightweight data-race detection. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2009.

[67] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[68] Michael R. Marty. *Cache coherence techniques for multicore processors*. PhD thesis, 2008.

[69] Tomer Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Comp Arch Lttrs*, 2006.

[70] R. Narayanan et al. MineBench: A Benchmark Suite for Data Mining Workloads. In *IEEE International Symposium on Workload Characterization*, 2006.

[71] Angeles Navarro et al. Analytical modeling of pipeline parallelism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[72] Angeles Navarro et al. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *Proceedings of the International Conference on Supercomputing*, 2009.

[73] Yasunori Nishitani et al. Implementation and evaluation of OpenMP for Hitachi SR8000. In *International Symposium on High Performance Computing*, 2000.

[74] Nvidia. CUDA SDK Code Samples, 2007.

[75] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1994.

[76] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[77] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[78] Easwaran Raman. Parallel-stage decoupled software pipelining. In *International Symposium on Code Generation and Optimization*, 2008.

[79] R. M. Ramanathan. Intel multi-core processors: Making the move to quad-core and beyond. *Technology@Intel Magazine*, 4(1):2–4, December 2006.

[80] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2009.

[81] Parthasarathy Ranganathan et al. The interaction of software prefetching with ILP processors in shared-memory systems. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1997.

[82] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and typhoon: user-level shared memory. *SIGARCH Comput. Archit. News*, 22(2):325–336, 1994.

[83] Christopher Rossbach et al. TxLinux: using and managing hardware transactional memory in an operating system. In *Symposium on Operating Systems Principles*, 2007.

[84] Amir Roth and Gurindar S. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1999.

[85] Mitsuhisa Sato et al. Design of OpenMP compiler for an SMP cluster. In *European Workshop on OpenMP*, September 1999.

[86] Larry Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 2008.

[87] Jason Sobel. The facebook blog: Keeping up. http://blog.facebook.com/blog.php?post=7899307130.

[88] Stephen Somogyi et al. Spatio-temporal memory streaming. *Proceedings of the Annual International Symposium on Computer Architecture*, 2009.

[89] S. Sridharan et al. Thread migration to improve synchronization performance. In Workshop on OS Interference in High Performance Applications, 2006.

[90] The Standard Performance Evaluation Corporation. *Welcome to SPEC.* http://www.specbench.org/.

[91] David C. Steere et al. A feedback-driven proportion allocator for real-rate scheduling. In *USENIX Symposium on Operating Systems Design and Implementation*.

[92] Richard Strong et al. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, 43(2), 2009.

[93] M. Suleman et al. ACMP: Balancing Hardware Efficiency and Programmer Efficiency. Technical report, HPS, February 2007.

[94] M. Suleman et al. An Asymmetric Multi-core Architecture for Accelerating Critical Sections. Technical Report TR-HPS-2008-003, 2008.

[95] M. Suleman et al. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[96] M. Aater Suleman et al. Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[97] M. Aater Suleman, Onur Mutlu, Moinuddin Qureshi, and Yale Patt. An Asymmetric Multi-core Architecture for Accelerating Critical Sections. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[98] David Tarjan, Michael Boyer, and Kevin Skadron. Federation: repurposing scalar cores for out-of-order instruction issue. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, 2008.

[99] Joel M. Tendler et al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[100] William Thies et al. Streamit: A language for streaming applications. In *11th Intl. Conference on Compiler Construction*, 2002.

[101] Tornado Web Server. Source code. http://tornado.sourceforge.net/.

[102] Pedro Trancoso and Josep Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *International Conference on Parallel Processing*, 1996.

[103] Mark Tremblay et al. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *International Solid-State Circuits Conference*, 2008.

[104] Dean M. Tullsen et al. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1995.

[105] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed ip routing lookups. In *Special Interest Group on Data Communications (SIGCOMM)*, 1997.

[106] Matt Welsh et al. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.

[107] Thomas F. Wenisch. *Temporal memory streaming*. PhD thesis, 2007.

[108] Thomas F. Wenisch et al. Temporal streaming of shared memory. In *Proceedings of the Annual International Symposium on Computer Architecture*, 2005.

[109] Wikipedia. Fifteen puzzle. http://en.wikipedia.org/wiki/Fifteen_puzzle.

[110] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the Annual International Symposium on Computer Architecture*, 1995.

[111] Peng Zhao and José Nelson Amaral. Ablego: a function outlining and partial inlining framework. *Software – Practice and Experience*, 37(5):465–491, 2007.

[112] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, DC, USA, 2007. IEEE Computer Society.

Vita

M. Aater Suleman was born in Lahore, Pakistan on January 5, 1983, the son of Sheikh and Nusrat Suleman. He attended the Crescent Model Higher Secondary School in Lahore, Pakistan until 2000. He migrated to the United States and started his Bachelors in Electrical and Computer Engineering from University of Texas at Austin in 2000. He graduated in 2003. The following semester he joined graduate school, also at Texas. He received his Masters in 2005. He started working on his PhD with Prof. Yale Patt in 2004.

During his undergraduate years, Suleman worked for the UT Center for Space Research as a part of NASA's ICESat mission. He played a critical role in the development of the timing verification system. The software he developed is in use by NASA at the White Sands Missile Range in New Mexico. He published in the Journal of Measurement Science and Technology. He also interned at National Instruments. He was the Vice-President of Eta Kappa Nu in his senior year. He received a Distinguished Scholar medal, a medal for graduating with highest honors, a TxTEC scholarship, and two College Scholar awards.

While in graduate school, Suleman served as a teaching assistant for five semesters at The University of Texas at Austin. He did seven internships: four at Intel, two at AMD, and one at Oasis. He has published papers in International Symposium on Computer Architecture (ISCA), International Conference on Architectural Support for Programming Languages (ASPLOS), International Symposium on High-Performance Computer Architecture (HPCA), International Symposium on Code Generation and Optimization (CGO), and IEEE Micro Top Picks. His honors include a paper in IEEE Micro Top Picks, Intel PhD fellowship, and the Prestigious UT Graduate Dean Fellowship. He also served as the President of Pakistani Student Support Group in 2007.

Permanent address: 11932 Rosethorn Dr.,
Austin, Texas 78758

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.